Affinity-Aware Resource Provisioning for Long-Running Applications in Shared Clusters

Clément Mommessin[a,*], Renyu Yang[a,*], Natalia V. Shakhlevich[a,**], Xiaoyang Sun[a,b], Satish Kumar[a], Junqing Xiao[b], Jie Xu[a]

[a] *School of Computing, University of Leeds, UK*
[b] *Alibaba Group, China*

Abstract

Resource provisioning plays a pivotal role in determining the right amount of infrastructure resource to run applications and reduce the monetary cost. A significant portion of production clusters is now dedicated to long-running applications (LRAs), which are typically in the form of microservices and executed in the order of hours or even months. It is therefore practically important to plan ahead the placement of LRAs in a shared cluster for the minimized number of compute nodes required by them. Existing works on LRA scheduling are often application-agnostic, without particularly addressing the constraining requirements imposed by LRAs, such as co-location affinity constraints and time-varying resource requirements. In this paper, we present an affinity-aware resource provisioning approach for deploying large-scale LRAs in a shared cluster subject to multiple constraints, with the objective of minimizing the number of compute nodes in use. We investigate a broad range of solution algorithms which fall into three main categories: Application-Centric, Node-Centric, and Multi-Node approaches, and tune them for typical large-scale real-world scenarios. Experimental studies driven by the Alibaba Tianchi dataset show that our algorithms can achieve competitive scheduling effectiveness and running time, as compared with the heuristics used by the latest work including Medea and LraSched. Best results are obtained by the Application-Centric algorithms, if the algorithm's running time is of primary concern, and by Multi-Node algorithms, if the solution quality is of primary concern.

Keywords: Resource Scheduling, Long-Running Applications, Vector Bin Packing

1. Introduction

Resource provisioning in large-scale compute clusters is of the utmost importance in IT infrastructure capacity management [1] and critical to the overall stability and performance of a cluster [2]. Virtualization and containerization offer a cost-effective solution for server and application consolidation [3, 4]. The consolidation typically has an objective of minimizing the number of occupied hosts (virtual machines or physical servers) needed to underpin the workloads. It must take into account the characteristics of workloads and use cases in order to correctly size a cluster and minimize the cost of workload deployment.

Traditional workloads in clusters are data analytic batch jobs [5, 6, 7] with short-lived tasks (in the order of seconds). However, long-running applications (LRAs) – such as latency-sensitive databases, user-facing services, streaming processing frameworks, etc. – have now become another main type of workloads supported by production clusters (Google [8], Microsoft [9], Alibaba [10]). In particular, across six analytics clusters at Microsoft, each comprising tens of thousands of machines, at least 10% of each cluster's machines are used for LRAs and two clusters are used exclusively for LRAs [9]. In Alibaba, 94.2% of the total CPU capacity in a cluster is allocated to LRAs [11]. In fact, microservice architecture has been the key

*Co-first authors
**Corresponding authors: Renyu Yang and Natalia V. Shakhlevich
Email addresses: r.yang1@leeds.ac.uk (Renyu Yang), n.shakhlevich@leeds.ac.uk (Natalia V. Shakhlevich)

enabler to build up large-scale IT infrastructures. Each individual microservice – practically instantiated as an LRA that can be independently implemented, built and maintained – is hosted in a long-lived container that usually executes for a long time frame (from hours to months) either for iterative computations in memory or for handling web requests. An LRA often makes use of multiple replicas of it to ensure low latency, fault tolerance, and high availability [8, 12, 13].

While it is appealing to build up complex enterprise IT systems consisting of a very large number of LRAs, there are many challenges associated with co-location, LRA multiplicity and heterogeneity. In reservation-based infrastructure, LRAs typically need to reserve multi-dimensional resources ahead of their execution, and their resource usage usually has strong temporal patterns. To optimize the performance and resilience, an LRA has application-specific placement preferences or exclusions when it is co-located with other LRAs. For instance, some LRAs are often required to be co-located to save network bandwidth and reduce latency or to be separately placed to reduce resource contention and performance interference. The ever-increasing scale of the number of new LRAs to be deployed (tens of thousands) and the corresponding affinity relationships further complicate resource reservation. In a nutshell, a robust and scalable resource provisioning scheme should tackle multi-dimensional temporal resource requests and LRA-level affinities, i.e., it should address placement of identical replicas incurred by each LRA, resolve replica conflicts stemming from the affinity constraints, and handle efficiently large-scale LRA deployment scenarios.

To the best of our knowledge, none of the existing studies to date addresses all these requirements at the same time, although every single requirement might have been considered. Most of the existing work (e.g., [7, 8, 14, 15, 16]) is application-agnostic and only focuses on node-related affinity, neglecting inter-application affinity constraints. Kubernetes [13] and Medea [9] address the application-related affinity, but do not address the requirement of scheduling all LRAs as a global optimization problem: Kubernetes schedules one LRA replica (pod) at a decision point, while Medea aims at runtime scheduling of relatively small batches of LRAs periodically. LraSched [17] only addresses the intra-application affinity constraints. Additionally, the capability of handling massive-scale scheduling problems of these three solutions has not been fully investigated.

The problem we study is to minimize the number of compute nodes required for accommodating LRAs in a shared cluster, subject to a set of strict resource and affinity constraints. We formulate the problem as an ILP and develop a new system model that can be considered as a generalization of the combinatorial optimization problems of Vector Bin Packing and Bin Packing with Conflicts [18]. Considering the diversity of real-world scenarios that gives rise to instances with a variety of characteristics, a fast heuristic, successful for one scenario, may perform poorly on another. This motivates us to develop an algorithm suite that can be used by practitioners for selecting the best performing heuristics that best fit the specific needs of a given scheduling scenario. To illustrate the capabilities of the suite, we perform experiments on instances generated from the Alibaba Tianchi dataset [19] and compare the winning approaches from our suite with the best performing published algorithms: two heuristics from Medea [9], namely *TagPopularity* and *NodeCandidate*, as well as a heuristic based on the *Fitness* measure introduced in LraSched [17]. A high-level summary of the most successful algorithms in our toolkit and those published in the literature is presented in Fig. 3, Section 5.

Our suite consists of three groups of algorithms: Application-Centric, Node-Centric and the Multi-Node approaches. The first two algorithm groups stem from the state-of-the-art research on Vector Bin Packing and Bin Packing with Conflicts [18]. The third algorithm group is particularly successful in the presence of LRA replicas and associated affinity restrictions. While Application-Centric algorithms are recommended when the computation time is required to be as small as possible, the Multi-Node algorithms deliver solutions of best quality (within only 0.3% deviation from the lower bound), with a larger running time. Node-Centric Algorithms place themselves in between, offering a trade-off between solution quality and time to find a solution.

To summarize, the main contributions of this paper are as follows:

- Formulating a resource provisioning problem to address temporal resource requests and application-level affinity constraints (§3);
- Devising an algorithm suite to provide adaptable solutions to a variety of real-world scenarios (§4);

- Selecting, via extensive computational experiments, a collection of best performing algorithms that can effectively handle large-scale LRA deployment (§5), focused on the use-case of the Alibaba Tianchi dataset [19];
- Elaborating algorithm recommendations providing a trade off between computation time and solution quality when confronted with different scenarios (§6).

Our findings can serve as the basis for practitioners and researchers for optimizing the resource provisioning and capacity planning to handle large-scale LRA placement in different scenarios.

2. Background and Motivation

2.1. Microservice and Long-running Applications

Cloud services and enterprise IT systems have been experiencing a major shift from monolithic applications that encompass the whole functionality within a software package (e.g., the full-stack LAMP application) to thousands of loosely-coupled microservices that can be independently built and maintained. According to Statista survey [20], in 2021, 85% of respondents from large organizations with $5,000$ or more employees stated that they had been using microservices in their software development environments.

As a key enabler, microservice architecture is particularly supportive to build extensible and loosely-coupled systems at scale. Enterprise microservices can be considered as an important and widely popular types of long-running applications (LRAs). They are typically hosted in long-lived containers that can run for hours, or even months, and consist of a diverse mix of applications from web servers to databases. Such applications are long-standing, user-facing and interactive services, working in "request-and-response" manner to serve user requests. Representative examples of LRAs include streaming processing frameworks (Storm [21], Flink [22], Kafka streams [23]), latency-sensitive database applications (HBase [24] and MongoDB [25]), and data-intensive in-memory computing frameworks (Spark [26], Tensorflow [27]).

2.2. Resource Provisioning

LRAs need to be deployed into on-premises or cloud infrastructure. Resource provisioning – one of the key elements in capacity management [1] – plays a pivotal role in determining the *initial* amount of infrastructure capacity (required resources) that can run a collection of applications. Particularly, for a homogeneous computing cluster where each node has the same hardware and the same operating system, infrastructure capacity can be regarded as the number of compute nodes (bare metal servers or virtual machines in a virtualized cloud cluster).

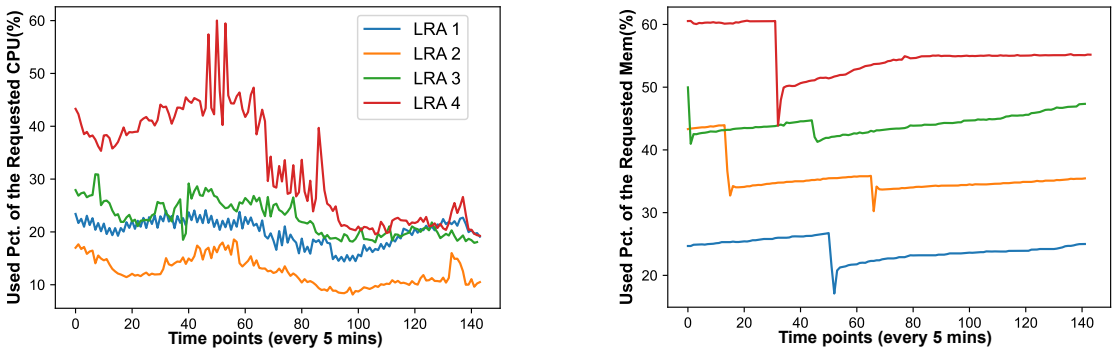
Most large-scale infrastructure managers [8, 10, 28] adopt *reservation*-based resource requests and resource allocations, i.e., application users or developers are required to specify the number of resources required (CPU cores, RAM, GPUs, etc.) at the submission of the applications. To lower the operational costs, one simple yet prevalent task of resource provisioning is to minimize the number of nodes capable of handling the reservation requests of a given set of LRAs. The plan-ahead before LRA deployment and execution is of major importance to IT administrators to facilitate a better understanding of resource requirement and to resize the infrastructure configuration in an economical and environmental-friendly manner.

2.3. Problem Scope and Challenges

While runtime LRA scheduling is well addressed by cluster schedulers [8, 16, 29, 30], this work focuses on addressing a planning problem for resource provisioning as we envisage the importance of pre-execution planning to the cost reduction of infrastructure management. The resource planner aims to work out the best option for deploying the LRAs ahead of their execution, given that all information of LRAs to be submitted is foreknown, to ensure a predictable LRA execution.

We highlight challenging requirements for the planning problem we address in this paper.

- **[R1] Multi-dimensional and time-varying resource requirements.** LRAs usually require resources of different types (CPU cores, memory, disk, etc.). Additionally, LRAs experience a noticeable temporal resource dynamicity over time. Fig. 1 illustrates the dynamicity of CPU and memory usage of multiple co-located LRAs over 12 hours, observed from the Alibaba Cluster Trace [31]. Such dynamicity can be captured through history-based profiling as most LRA workloads run in a recurring manner and have strong temporal pattern [32], which helps to unlock the potential of accurate



(a) CPU usage

(b) Memory usage

Figure 1: CPU and memory temporal usage over 12 hours of four anonymous LRAs in Alibaba cluster trace

requirement models and workload co-location in large-scale clusters [10, 11, 28]. The owner of an LRA typically needs to determine the resource demands (e.g., through extracting resource skyline based on the resource requirement model) and translate the temporal requirement into resource reservation.

- **[R2] Application-level affinity constraints.** Affinity constraints encompass placement preferences or exclusions between LRAs. While node-related affinity specifies which nodes an LRA is eligible to be placed on, application-level affinity specifies how many replicas of an LRA can be placed jointly given the co-located LRAs on a node. These constraints are completely application-specific. For example, data producing and data consuming applications could be co-located on the same node for sharing intermediate data to save network bandwidth and reduce network latency. To avoid excessive performance interference, latency-sensitive streaming applications should not be co-located on the same node. However, for some LRAs, it is reasonable to co-locate their replicas within the same available zone, which would help to ease service management, reduce the cost of synchronization or data communication between applications. Running applications without satisfying such constraints would lead to unexpected application slowdown or system turbulence. Such affinity requirements are usually specified in the configuration (e.g., in a YAML/JSON file) to flag LRA-specific performance preferences and QoS requirements, before the deployment requests are submitted to the infrastructure manager.

- **[R3] Large-scale LRA deployment.** Launching tens of thousands of LRAs has now become the norm rather than the exception for cloud service providers in the face of new cluster initialization. This increases the management complexity of deploying large-scale LRAs. Each LRA has its own specific deployment and resource requirements (e.g., CPU cores, RAM and persistent storage). Therefore, the infrastructure manager needs to be robust and scalable enough to make (near-)optimal decisions, incorporating in the planning a huge number of resource and affinity requirements, in the initial deployment stage.

The existing works only partially solve the above research challenges. Unlike runtime LRA scheduling, that aims to achieve low scheduling latency (in the order of seconds or milliseconds), the main task of pre-execution planning is to precisely place the LRAs and to determine the amount of required resources in the IT infrastructure while satisfying all sophisticated specific constraints of applications.

Obviously, for the resource planner, it is worth trading the planning time for solution quality. This trade-off in the planning procedure is particularly pivotal as low-quality LRA placement may incur excessive cost in LRA re-scheduling and container migration, which is expensive due to the huge amount of state and disk data to migrate over the network and unacceptable service downtime. We believe an optimization-based plan-ahead is a necessary and promising means for effective resource provisioning. Our work aims at integrating the above requirements into a holistic system model, and developing a suite of algorithms able to solve the resource provisioning problem and adapt to different scenarios.

3. System Model and Problem Formulation

3.1. System Model

Our system consists of *compute nodes*, which form the set $\mathcal{N}$, and *LRAs*, which form the set $\mathcal{L}$. Additionally, there are *affinity restrictions* for some pairs of LRAs from $\mathcal{L}$.

Compute nodes are identical and their resources are characterized by d dimensions. In our *basic* model, there are two types of resources, the number of CPU cores C_1 and the number of units of memory C_2. It can be extended to take into account such characteristics as the size of disk storage or Last-Level Cache, the memory bandwidth, the number of GPU, etc. In general, according to **[R1]** of the model, a node has d dimensions, with resource capacities $C_1, C_2, \ldots, C_d$.

LRAs differ in a number of parameters. In accordance with **[R2]**, each LRA consists of a given number of replicas that run from time 0 to infinity (or to a given time limit common for all LRAs). An LRA $i \in \mathcal{L}$ has a given size s_{ih} (i.e., resource requirement) in dimension h, $1 \leq h \leq d$, and that value is the same for all replicas of that LRA. For example, for the basic model, s_{i1} and s_{i2} are the number of CPU cores and the number of units of memory needed by each replica of LRA i.

In the *basic* model, we assume that the sizes of LRAs do not change over time. If several replicas of LRAs are allocated to the same node, then the total size of allocated replicas in each dimension cannot exceed the node capacity in that dimension. Thus d capacity constraints should be satisfied for each node.

In the *enhanced* model, the profiles of LRAs may change over time. They are approximated via piecewise constant functions. If the timeline is split into T time intervals, so that within one interval resource requirements of LRAs do not change, then the original d-dimensional problem, with d resource types, is converted into the problem of d' dimensions:

$$d' = T \times d.$$

Fig. 2 illustrates allocation of three LRAs to one compute node. Each LRA has specific resource requirements for $d = 3$ resource types: memory, CPU and disc space. If application requirements are static, then it is sufficient to consider only one fragment of Fig. 2: one three-dimensional cube for a node, with LRAs placed inside it without overlaps in each dimension. If application requirements

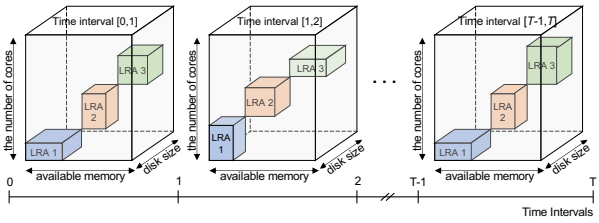


Figure 2: Allocation of three-dimensional LRAs to one node taking into account changing resource requirements over T time intervals

change in T time intervals, then the memory, CPU and disk constraints should be considered for each time interval. For the instance considered in Fig. 2, there is one node and T snapshots of that node, with the same three LRAs allocated to the node in each of the T snapshots. The resource requirements of the LRAs change, but the overall capacity of the node is not exceeded.

Affinity restrictions are defined for pairs of LRAs which replicas can be jointly co-located to the same node, but with some limits, or for pairs of incompatible LRAs, which cannot be co-located. If LRA i is restrictive to LRA j, then there is an integer *affinity value* a_{ij} which sets up an upper bound on the maximum number of replicas of j that can be co-located on a node where at least one replica of i is allocated. Thus **[R2]** of the model is characterized by the set of affinity restrictions, represented as a directed graph where vertices correspond to LRAs and arcs (i, j) correspond to affinity restrictions associated with the values a_{ij}.

3.2. Problem Formulation

In a feasible solution to the resource provisioning problem, all replicas of all LRAs in the given set $\mathcal{L}$ should be allocated to a subset of $\mathcal{N}$, without violating affinity restrictions and node capacities in each of the d dimensions (or, in general, d' dimensions). The objective is to minimize the total number of nodes in use.

We introduce an Integer Linear Programming (ILP) formulation for the resource provisioning problem with constant resource demands. Recall that for the time-varying resource demands, the d-dimensional problem is converted into d'-dimensional problem, $d' = Td$, which implies that d is replaced by d' in the ILP formulation.

We use the following notations:

$\mathcal{L}$ for the set of LRAs,
$\mathcal{R}_i$ for the set of replicas of an application $i \in \mathcal{L}$,

$\mathcal{N}$ for the set of nodes,
$\mathcal{A}$ for the set of pairs (i, j) of applications which have affinity restrictions a_{ij},
s_{ih} for the size of resource h required by a replica of application $i \in \mathcal{L}$, in dimension h, $1 \leq h \leq d$,
C_h for the capacity of each node in dimension h, $1 \leq h \leq d$,
a_{ij} for the affinity restriction imposed by application i (how many replicas of j can be co-located together with a replica of i).

The decision variables take 0 – 1 values:

x_{irn} is equal to 1 if the rth replica of application i is allocated to node n,
y_n is equal to 1 if node n is activated and accommodates some replica(s),
z_{in} is equal to 1 if at least one replica of application i is allocated to node n.

Additionally, we compute constants ν_i for the maximum number of replicas of application i which can be allocated to one node, regardless of affinity restrictions from other applications:

$$\nu_i = \min \left\{ \min_{1 \leq h \leq d} \left\{ \left\lfloor \frac{C_h}{s_{ih}} \right\rfloor \right\}, |\mathcal{R}_i| \right\}. \quad (1)$$

Here $|\mathcal{R}_i|$ is the total number of replicas of application i and $\lfloor C_h/s_{ih} \rfloor$ is the limitation associated with dimension h if replicas of application i are allocated to a node. For example, in the basic model with two resource types per node, the ratios $\lfloor C_1/s_{i1} \rfloor$ and $\lfloor C_2/s_{i2} \rfloor$ are related to the CPU and memory limitations for replicas of LRA i. In the enhanced model with time-varying profiles, each dimension $1 \leq h \leq Td$ gives rise to a resource restriction in the corresponding time interval.

The problem of allocating the replicas of all LRAs to the minimum number of compute nodes without exceeding node capacities and violating affinity restrictions of LRAs is modelled as the following ILP:

$$\min \quad \sum_{n \in \mathcal{N}} y_n \quad (2a)$$

$$\text{s.t.} \quad \sum_{n \in \mathcal{N}} x_{irn} = 1, \qquad i \in \mathcal{L}, r \in \mathcal{R}_i, \quad (2b)$$

$$\sum_{i \in \mathcal{L}} s_{ih} \sum_{r \in \mathcal{R}_i} x_{irn} \leq C_h y_n, \quad n \in \mathcal{N}, 1 \leq h \leq d, \quad (2c)$$

$$\sum_{r \in \mathcal{R}_i} x_{irn} \leq \nu_i z_{in}, \; i \in \mathcal{L}, n \in \mathcal{N}, \quad (2d)$$

$$z_{in} \leq \sum_{r \in \mathcal{R}_i} x_{irn}, \qquad i \in \mathcal{L}, n \in \mathcal{N}, \quad (2e)$$

$$\sum_{r \in \mathcal{R}_j} x_{jrn} \leq a_{ij} z_{in} + \nu_j (1 - z_{in}), \quad (i, j) \in \mathcal{A}, n \in \mathcal{N}, \quad (2f)$$

$$x_{irn}, \; y_n, \; z_{in} \in \{0, 1\}, \quad i \in \mathcal{L}, r \in \mathcal{R}_i, n \in \mathcal{N}. \quad (2g)$$

Objective function (2a) is the total number of activated nodes. Constraint (2b) ensures that all replicas of all applications are allocated, while constraint (2c) ensures that the capacity of each node is not exceeded in each dimension. The variables y_n and z_{in} are linked to x_{irn} by (2c)-(2e), and constraint (2f) guarantees that affinity restrictions are observed.

The resource provisioning problem is NP-hard, as it generalizes the combinatorial optimization problems of *Vector Bin Packing* and *Bin Packing with Conflicts* [18]: the Vector Bin Packing problem occurs when each LRA consists of a single replica and there are no affinity restrictions; the Bin Packing with Conflicts occurs when $d = 1$, each LRA consists of a single replica, and affinity values a_{ij} between two conflicting LRAs are restricted to 0.

As we will discuss in §5.1, the presented ILP is capable of solving medium size instances, with up to $2,000$ two-dimensional LRAs. In what follows we elaborate a broad range of heuristic methods capable of solving effectively and efficiently LRA scheduling problems typical for real-world massive-scale systems.

4. Our Algorithm Suite

This section presents an overview of the approaches (§4.1) and implementation details for a range of algorithms (§4.2 to §4.4). We then discuss

the worst-case time complexities of the algorithms (§4.5).

4.1. Overview

The heuristics described in this section stem from the vast body of research on the Bin Packing Problem and its enhanced versions. The methods distinguish in how they order the set of applications $\mathcal{L}$, the set of nodes $\mathcal{N}$ or the set of application-node pairs. The choice of the most promising prioritization rules depends on the scenarios to which the method is applied and on the datasets.

All methods consider only *feasible* allocations of application replicas to the nodes, so that the node capacities are not exceeded for each of the d resource types in each of the T time intervals, and the affinity restrictions for the applications already allocated are observed. By allocating replicas to the nodes in a *feasible* fashion we guarantee that requirements **[R1]-[R2]** are satisfied. To handle requirement **[R3]** we strive to achieve fast running times for our heuristics.

Application-Centric approach. This approach considers the applications one by one in accordance with their ordering in $\mathcal{L}$. For a current application, it selects the first *feasible* node in the ordered list $\mathcal{N}$ and allocates the maximum number of replicas to that node. It then selects the next *feasible* node from $\mathcal{N}$ to continue allocation of the replicas of the current application. After all replicas of the current application are allocated, the algorithm proceeds with the next application in $\mathcal{L}$, etc. The rules for ordering $\mathcal{L}$ and $\mathcal{N}$ are formulated in §4.2, using the state-of-the-art findings in the body of research on Bin Packing and Vector Bin Packing [18, 33, 34]. Algorithm 1 outlines the pseudo-code of this approach.

Node-Centric approach. This approach considers the nodes one by one in accordance with their numbering in list $\mathcal{N}$. For a current node, the algorithm selects from the list of non-allocated applications the one which is *feasible* for the current node and has the largest *application-node score*. The maximum number of replicas of that application are allocated to the node. If the node is not fully packed, then the *application-node scores* are recalculated, taking into account the residual capacity of the current node, and the application delivering the highest score is used for loading the node. The process continues until no feasible application for the current node can be found on the list $\mathcal{L}$. The algorithm then proceeds with the next node in the list, etc. The scoring rules are formulated in §4.3, using the findings in the body of research on the Vector Bin Packing problem [34]. Algorithm 2 outlines the pseudo-code of this approach.

Algorithm 1 Application-Centric approach

```
1: Activate node n = 1 and set N ← {1}
2: while there are unallocated LRAs do
3:    Select i ∈ L using a predefined rule
4:    while not all replicas of i are allocated do
5:       if no node from N can accommodate i then
6:          Set n ← n+1, N ← N∪{n} and activate node n
7:       Select n* ∈ N, feasible for i, using a predefined rule
8:       Allocate the maximum number of replicas of i to n*
9:    Remove i from L
```

Algorithm 2 Node-Centric approach

```
1: Activate node n = 1 and set N ← {1}
2: while there are unallocated LRAs do
3:    if no i ∈ L is feasible for n then
4:       Set n ← n + 1, N ← N ∪ {n} and activate node n
5:    Select i* ∈ L which is feasible for n and delivers the maximum score
6:    Allocate the maximum number of replicas of i* to n
7:    if all replicas of i* are allocated then
8:       Remove i* from L
```

Multi-Node approach. This approach aims to overcome the myopic nature of the Application-Centric and Node-Centric algorithms. A large set of nodes is activated directly at start, and best allocation options are selected across the whole pool of nodes. The algorithm either finds a feasible solution or declares a failure, if the number of activated nodes is too small to accommodate all applications. The proposed approach requires that the desired number of nodes is specified as part of the input. The search for a feasible solution with the minimum number of nodes is arranged by calling the algorithm repeatedly with different trial values

Algorithm 3 Multi-Node approach with Replica Spreading

For a given n, activate nodes $\mathcal{N} = \{1, 2, \ldots, n\}$
while there are unallocated LRAs **do**
  Select $i \in \mathcal{L}$ using a predefined rule
  while not all replicas of i are allocated **do**
    if no node from $\mathcal{N}$ can accommodate i **then**
      declare a failure and **break**
    Select $n^* \in \mathcal{N}$, feasible for i, using a predefined rule
    Allocate one replica of i to n^*
  Remove i from $\mathcal{L}$

for the number of nodes, either via binary search, or with the trial value decremented in steps.

We distinguish between the following two version of the Multi-Node approach, whose pseudo-codes are given as Algorithms 3 and 4.

- The **Multi-Node approach with Replica Spreading** is the adaptation of the Application-Centric approach. LRAs are also considered one by one, but instead of allocating the *maximum* number of replicas of a current application to the highest priority node, only one replica is allocated. With replicas spread over a large pool of activated nodes, there is more flexibility for selecting compatible LRAs for future co-location: affinity constraints a_{ij} become less restrictive if a small number of replicas of i and j are allocated to the same node.

- The **Multi-Node approach with Application-Node Matching** is the adaptation of the Node-Centric approach. At each step the score for every *feasible* application-node pair is computed, and the pair with the highest score is selected for extending a partial solution. One replica of the selected application is allocated to the corresponding node, and the scores are recalculated to define the next most promising application-node pair. This approach is more flexible than the original Node-Centric approach: it benefits from a larger freedom for selecting the most promising application-node pairs, with potentially better utilized resources as a result.

Algorithm 4 Multi-Node approach with Application-Node Matching

For a given n, activate nodes $\mathcal{N} = \{1, 2, \ldots, n\}$
while there are unallocated LRAs **do**
  if no pair (i, n) is feasible ($i \in \mathcal{L}$, $n \in \mathcal{N}$) **then**
    declare a failure and **break**
  Select a feasible pair (i^*, n^*) which delivers the maximum score
  Allocate one replica of i^* to n^*
  if all replicas of i^* are allocated **then**
    Remove i^* from $\mathcal{L}$

4.2. Application-Centric Algorithms

At the core of the Application-Centric algorithms are the priority rules for ordering the list of applications $\mathcal{L}$ and the list of nodes $\mathcal{N}$. Based on the best performing algorithms known for Bin Packing, there are three widely accepted possible orderings for the nodes $\mathcal{N}$ and two orderings for the applications $\mathcal{L}$.

For $\mathcal{N}$, the nodes can be considered (a) in the activation order, (b) in the increasing order of a priority index, or (c) in the decreasing order of a priority index. For $\mathcal{L}$, the applications can be considered (1) in the order of their numbering, or (2) in the decreasing order of a priority index. The priority indices can be defined in multiple ways for the multi-dimensional problem. In the remainder of this section, we describe the rules for calculating the priority indices of applications, denoted by *size measures* and used for ordering (2) of list $\mathcal{L}$, and the rules for calculating the priority indices of nodes, denoted by *residual capacity measures* and used for ordering (b) or (c) of list $\mathcal{N}$.

Depending on how the rules for $\mathcal{N}$ are combined with the rules for $\mathcal{L}$, the resulting algorithms are classified as (1a) First Fit (FF), (1b) Best Fit (BF), (1c) Worst Fit (WF), (2a) First Fit Decreasing (FFD), (2b) Best Fit Decreasing (BFD) and (2c) Worst Fit Decreasing (WFD).

Applications' priority indices. In the presence of resource requirement in multiple dimensions, one most significant dimension can be used for prioritizing the applications. If no dominant dimension exists, as in the case of the Alibaba Tianchi dataset [19], there is a need to compute a *combined size measure* s_i for each application $i \in \mathcal{L}$ and to use it as a priority index. Introducing a single measure allows us to address efficiently the issues related to

Table 1: Application-Centric size measures s_i for applications $i \in \mathcal{L}$

Average		$s_i = \frac{1}{d}\sum_{h=1}^{d} s_{ih}$
Max		$s_i = \max_{1\le h\le d}\{s_{ih}\}$
Average with exponential weight	[34]	$s_i = \sum_{h=1}^{d} e^{\varepsilon D_h} \cdot s_{ih}$
Surrogate	[35]	$s_i = \sum_{h=1}^{d} \lambda_h s_{ih}$
Extended Sum	[17]	$s_i = \sum_{h=1}^{d} \frac{\lvert R_i\rvert}{W_h} s_{ih}$

requirement **[R1]**.

When dealing with non-comparable sizes s_{ih} of LRAs, such as the number of CPU cores and memory, the values should be normalized to satisfy $s'_{ih} \in [0,1]$, which is achieved by setting $s'_{ih} = \frac{s_{ih}}{C_h}$ in each dimension h, $1 \le h \le d$. With the normalized sizes s'_{ih} of LRAs, the node capacities are set to $C'_h = 1$. In what follows, we assume that the preprocessing has been done and the normalized values are calculated. For simplicity, we drop the prime in the notation.

The two natural combined measures are *Average* and *Max*, whose corresponding expressions are stated in the first two lines of Table 1.

The remaining measures use the following notations:

$W_h = \sum_{i\in\mathcal{L}} \lvert R_i\rvert\, s_{ih}$ for the total demand of all LRAs in dimension h,

$D_h = \frac{W_h}{\sum_{i\in\mathcal{L}} \lvert R_i\rvert}$ for the average demand of all LRAs in dimension h,

$\lambda_h = \frac{W_h}{\sum_{k=1}^{d} W_k}$ for the normalized demand of all LRAs in dimension h.

The *Average measure with exponential weight* is one of the best performing measures in experiments on Vector Bin Packing, performed by Panigrahy et al. [34]. It is computed as the weighted sum of s_{ih}-values, with exponential weights depending on average demands D_h. Parameter ε is a small number selected appropriately for scaling.

The *Surrogate* measure is a natural extension of the 2-dimensional measure of Caprara and Toth [35]. It is computed as the weighted sum of s_{ih}-values, with the normalized demands λ_h used for weights.

Finally, the *Extended Sum* is an adaptation of the measure used in LRASched [17]. For application i, it is defined as the sum, over all dimensions h, of the demands of all replicas of that application $\lvert R_i\rvert s_{ih}$ in dimension h normalized by the total demand W_h of all applications in that dimension.

Table 2: Application-Centric residual capacity measures $\overline{C}_n$ for node $n \in \mathcal{N}$

Average	$\overline{C}_n = \frac{1}{d}\sum_{h=1}^{d} \overline{C}_{nh}$
Max	$\overline{C}_n = \max_{1\le h\le d}\{\overline{C}_{nh}\}$
Average with exponential weight	$\overline{C}_n = \sum_{h=1}^{d} e^{\varepsilon D'_h} \cdot \overline{C}_{nh}$
Surrogate	$\overline{C}_n = \sum_{h=1}^{d} \lambda'_h \overline{C}_{nh}$
Extended Sum	$\overline{C}_n = \sum_{h=1}^{d} \frac{\overline{C}_{nh}}{W'_h}$

where $W'_h = \sum_{n\in\mathcal{N}} \overline{C}_{nh}$, $D'_h = \frac{W'_h}{\lvert\mathcal{N}\rvert}$, and $\lambda'_h = \frac{W'_h}{\sum_{k=1}^{d} W'_k}$

Prior research in the area of Bin Packing with Conflicts has discovered the benefits of combining the demand-based measure s_i with the conflict-based measure, which takes into account the criticality of an application in terms of interference [36]. Generalizing these ideas to affinity restrictions **[R2]** of our model, we define the *hybrid demand-affinity* measure as the weighted sum of the demand-based measure s_i and the affinity-based measure δ_i:

$$\tilde{s}_i = \alpha \frac{s_i}{\overline{s}} + (1-\alpha)\,\frac{\delta_i}{\overline{\delta}}. \tag{3}$$

Here s_i is computed via one of the expressions from Table 1, δ_i is the total number of applications linked with application i in the affinity graph, while $\alpha \in [0,1]$ is chosen to give a higher priority to application demands or to interference. Scaling is performed for handling incomparable parameters, dividing by $\overline{s}$ and $\overline{\delta}$, the average values of s_i and δ_i, respectively.

Nodes' residual capacities. The key characteristics of a partly loaded node $n \in \mathcal{N}$ are residual capacities $\overline{C}_{nh}$, maintained for all dimensions $h = 1, 2, \ldots, d$. They are computed as original node capacities C_h minus the total size of allocated replicas for the same dimension h. In the presence of residual capacities in multiple dimensions, there is a need to compute a single *residual capacity measure* $\overline{C}_n$ for each node $n \in \mathcal{N}$ and to use it as a priority index.

For each application size measure s_i from Table 1, we similarly define the corresponding node

Table 3: Bin-Centric scores ξ_{in} for applications $i \in \mathcal{L}$ and nodes $n \in \mathcal{N}$

DotProduct	[34]	$\xi_{in} = \sum_{h=1}^{d} s_{ih} \overline{C}_{nh}$
L2Norm	[34]	$\xi_{in} = -\sum_{h=1}^{d} \left(\overline{C}_{nh} - s_{ih}\right)^2$
Fitness	[17]	$\xi_{in} = \sum_{h=1}^{d} \frac{s_{ih}}{W_h} \cdot \frac{\overline{C}_{nh}}{\sum_{k \in \mathcal{N}} \overline{C}_{kh}}$
TightFill		$\xi_{in} = \sum_{h=1}^{d} \frac{s_{ih}}{\overline{C}_{nh}}$

residual capacity measure $\overline{C}_n$ (see Table 2).

4.3. Node-Centric Algorithms

For the Node-Centric approach, the *application-node score* for application i and node n, denoted by ξ_{in}, is computed only for a *feasible* application node pair. The higher the score, the more beneficial it is to allocate replicas of application i to node n, which is currently being packed.

We explore in our algorithms the known best-performing scores, together with a newly proposed score, denoted by *TightFill*, as shown in Table 3.

All four scores select for a current node n the application which uses the d resources of the node to the highest extent.

- In the *DotProduct* score this is achieved by prioritizing the dimensions for which node n has the largest capacity. An application with highest demands in those dimensions is considered as the best choice.

- In the *L2Norm* score, the expression is negative so that the smallest positive value indicates the best application for node n. The preferred application minimizes the difference between its size and residual capacity of the node measured via the L2 norm.

- In the *Fitness* score, the application demands s_{ih} are normalized with respect to W_h, the total demand of all applications in dimension h, and the node capacities $\overline{C}_{nh}$ are normalized with respect to the total free capacity of all nodes in dimension h, $1 \leq h \leq n$.

- The *TightFill* score is a counterpart of *DotProduct* which ensures the tightest usage of the node residual capacity.

Table 4: Algorithms' time complexity. L is the number of applications, R is the total number of all replicas of all applications, n is the given (target) number of nodes

Application-Centric	$O(R^2L)$
Node-Centric	$O(RL^2)$
Multi-Node with Replica Spreading and n nodes	$O(RLn)$
Multi-Node with Application-Node Matching and n nodes	$O(RL^2n)$

4.4. Multi-Node Algorithms

Recall that multi-node algorithms require a target number of nodes as part of the input. The search for a feasible solution with the minimum number of nodes is arranged by calling the algorithm repeatedly with different trial values for the number of nodes, either via binary search or with the trial value decremented in steps.

Multi-Node Algorithms with Replica Spreading. These algorithms use the same principles as the Application-Centric algorithms, but with the aim of replica spreading across the whole pool of activated nodes, reducing this way the restrictions imposed by the affinity constraints a_{ij}. Among the six Application-Centric algorithms discussed in Section 4.2, only Worst Fit and Worst Fit Decreasing produce different solutions if n nodes are activated at start rather than being activated one by one on the fly. The remaining Application-Centric algorithms, First Fit, First Fit Decreasing, Best Fit and Best Fit Decreasing, do not change their behavior if a pool of nodes is activated at start. For this reason, we create only two algorithms by combining the Multi-Node and the Application-Centric approaches, with the shortcut names *SpreadWF* and *SpreadWFD*.

Multi-Node Algorithms with Application-Node Matching. These algorithms use the same principles as the Node-Centric algorithms, but on a pool of n activated nodes rather than on single nodes considered one by one. Each time, the most appropriate application-node pair is selected among all possible pairs of unallocated applications and non-fully packed nodes by using the scores defined in §4.3 for the Node-Centric approach, and a single replica is allocated. It is expected that the replicas of an application are spread broadly across the nodes pool, with less restrictions caused by the affinity constraints.

4.5. Time Complexity of Algorithms

The three introduced approaches, Application-Centric, Node-Centric and Multi-Node, provide the foundation to build a wide range of heuristics. The choice of a specific method, together with the most appropriate measures or scores, depends on special features of scenarios and datasets, and on limitations on algorithms' running times. Analytical estimates of running times are provided in Table 4. Note that the actual performance of the algorithms may differ from the theoretical estimates since the worst-case analysis takes into account very rare scenarios. It is also noted that the running time estimates for the Multi-Node approach are made for a single call with a fixed n given as the trial number of nodes. These estimates have to be multiplied by the total number of calls made by the decrementing method, or by the binary search, to get the time complexity of the overall procedure.

The choice of the size measure (Table 1) for s_i should take into account not only its impact on the running time, but also the nature of the dataset. In the presence of a dominating (bottleneck) resource type h^*, $1 \leq h^* \leq d$, which plays the critical role in application allocation, computing of the measure s_i can be simplified by using $s_i = s_{ih^*}$ instead. For Application-Centric approaches, incorporating the hybrid size measure of Eq. (3) on top of one of the standard measures from Table 1 can be beneficial if affinity constraints are very restrictive, so that many pairs of applications are in conflict. Note that Eq. (3) does not affect the asymptotic worst-case time complexity, but may slow down the algorithms' performance on large datasets.

5. Performance Evaluation

All algorithm codes, scripts for generating the instances, as well as additional figures, are publicly available at `https://github.com/DSSGroup-Leeds/LRA-binpacking-expe`.

5.1. Experimental Settings

Simulation configuration and instance generation. As the pre-execution planning is independent from the runtime execution of LRAs, we adopt simulation-based evaluation to validate the efficacy of different algorithms on a single machine equipped with one Intel Xeon Gold 6138 CPU and 64 GB of memory. We simulate different scales of LRA submission and evaluate how our algorithms succeed in LRA allocation onto a mocked compute cluster with identical nodes comprising 64 CPU cores and 128 GB of memory.

Our aim is to examine several sets of instances, each set with common features and related to a specific scenario, and to select the winning algorithms from our suite. The instances stem from the dataset published by the Alibaba Tianchi Platform [19]. The original dataset contains the data for $9,338$ LRAs with a total of $68,224$ replicas and $24,078$ affinity restrictions. Each LRA has resource requests in two dimensions: CPU cores and memory. LRA resource profiles change over time, with recordings known for 98 time sampling points.

We study two scenarios: one with different densities of affinity restrictions and another one with different numbers of LRAs. Our aim is to evaluate the impact of these characteristics on the solution quality and the running times of the proposed algorithms. Each scenario is subdivided into two sets of instances depending on whether LRA resource requests are constant or change over time. Each set contains a total of 90 instances:

- three types of affinity graphs (arbitrary, normal, threshold),
- three values of one of the varied parameters (affinity density or the number of LRAs),
- 10 instances for each combination.

A summary of the generated instances is presented in Table 5.

In the instances with *varied affinity density*, represented in the second column of Table 5, the number of LRAs $|\mathcal{L}|$ is the same as in the original Alibaba dataset [19], while the number of affinity restrictions, measured as *affinity density*, is different. The affinity density Δ is defined as the average number of affinity restrictions per LRA divided by the total number of LRAs. For example, affinity density of 10% means that each LRA has affinity restrictions with 10% of other LRAs on average. Note that in the original Alibaba dataset, the affinity density is lower than 0.05%. However, in practice, the real graph is system-specific – for those cluster systems with sufficient resources, the affinity graph could be sparse due to less restrictions on application co-location. In comparison, there could be complex dependencies or placement constraints among applications in some systems, which lead to denser affinity relationships in the graph. We therefore select diverse higher density values for

Table 5: Summary of generated instances

Scenario	Varied affinity density	Varied number of LRAs
$\|\mathcal{L}\|$	$9,338$	$10,000$ $50,000$ $100,000$
$\|\mathcal{R}_i\|, s_{ih}$	same as Alibaba [19]	similar to Alibaba [19]
affinity density Δ	1% 5% 10%	0.5%
affinity graph type	arbitrary threshold normal	arbitrary threshold normal
$d = 2$ (CPU, memory)	90 instances without temporal changes	90 instances without temporal changes
$d = 98 \times 2$ (CPU, memory, 98 time steps)	90 instances with temporal changes	90 instances with temporal changes

experiments to investigate the impact of affinity restrictions on the solution quality and algorithms' running times. For each LRA, the number of replicas per application $|\mathcal{R}_i|$ and resource requirements s_{ih} are kept unchanged, as in the original Alibaba dataset.

In the instances with *varied number of LRAs*, represented in the third column of Table 5, the affinity density is fixed to the same value (0.5%), while the number of LRAs $|\mathcal{L}|$ is different. We select larger instances compared to the Alibaba dataset [19] to explore the capabilities of the algorithms for optimizing the performance of massive scale systems. The values for the number of replicas $|\mathcal{R}_i|$ and resource requirements s_{ih} are defined using the same probability distributions as in the original Alibaba dataset.

For any type of instance, affinity values a_{ij} were generated following the same probability distribution as in the original Alibaba dataset.

Consider now the three approaches to graph generation, given number $|\mathcal{L}|$ of vertices and expected density Δ. One method for generating *arbitrary* graphs is described by Sadykov and Vanderbeck [37]. The key idea is as follows: starting with a graph with no arcs, pairs of nodes are selected at random (with uniform distribution) and connected by arcs. Arc generation stops when the desired graph density is achieved. Another method generates *threshold* graphs. It is described by Gendreau et al. [38] and elaborated further by Bacci and Nicoloso [39] for parameter correction. The produced graphs fall into the category of interval graphs and they are characterized by a given expected edge density. Note that forsome optimization problems on graphs, their versions with interval graphs are sometimes easier to solve. We propose the third approach to generate so called *normal* graphs. The resulting graphs differ from arbitrary random graphs by the presence of clustered nodes and sparingly connected nodes. They also differ from threshold graphs as they generally do not satisfy the strict restrictions of interval graphs. The method starts with a graph of $|\mathcal{L}|$ vertices and no arcs, and then for each vertex i it randomly picks a value p_i following the normal distribution of mean $\Delta|\mathcal{L}|$ and standard deviation $\Delta|\mathcal{L}|/2$, restricting the value between 0 and $|\mathcal{L}| - 1$. Then p_i vertices are selected at random using uniform distribution and they are used as end-nodes for arcs originating from vertex i.

The resource requirements of each LRA are copied from the Alibaba dataset for all 98 sampling points, if considering the class *with temporal changes* (last row of Table 5), or they are extrapolated if considering the class *without temporal*

changes (penultimate row of Table 5): for each LRA i we select the maximum values s_{i1}, s_{i2} among those provided for the 98 sampling points and round them to the next integer.

Evaluation methodology and metrics. We evaluate the effectiveness and time efficiency of each algorithm.

The *effectiveness* is measured by recording the number of nodes found in a feasible solution and calculating the *deviation from the lower bound*, a "lower-the-better" indicator. Since the total number of nodes cannot be smaller than the total demand W_h of all LRAs in dimension h divided by the node capacity C_h in that dimension, where $h = 1, \ldots, n$, the lower bound is defined as

$$LB = \max_{1 \le h \le d} \left\{ \left\lceil \frac{W_h}{C_h} \right\rceil \right\}. \quad (4)$$

The *time efficiency* is measured as the algorithm's computation time, averaged over the 10 instances of a given configuration of graph class and density value, or graph class and LRA number.

Baseline Methods. We mainly have three baselines in the experiments: two heuristics of Medea [9] and one heuristic of LraSched [17].

The *TagPopularity* (Medea-TP) heuristic is Application Centric. It allocates applications one by one, starting with those having the highest interference. This heuristic can be classified as FFD with size measure $s_i = \delta_i$, the special case of Eq. (3) with $\alpha = 0$. *NodeCandidates* (Medea-NC) is another version of the Application-Centric approach, with s_i-parameters representing the total number of available nodes in the system which can accommodate a replica of i, observing capacity and affinity restrictions:

$$s_i = \sum_{n \in \mathcal{N}} \zeta_{in}. \quad (5)$$

Here $\zeta_{in} = 1$ if a replica of application i can be allocated to node n without violating affinity restrictions, and $\zeta_{in} = 0$, otherwise. Applications are allocated one by one, starting with the most restrictive ones, i.e., those having the lowest sizes s_i computed by Eq. (5), and sizes of the remaining applications are re-computed after each step.

LraSched [17] uses a two-phase approach. The first phase aims at maximizing the number of fully allocated LRAs and resource utilization of the given restricted pool of available nodes. The second phase aims at minimizing the number of new nodes used to allocate remaining LRAs. The second phase employs a Node-Centric algorithm with the *Fitness* score. We denote the algorithm of this second phase by *LRASched-Fitness*.

Algorithm naming. We implemented our algorithms and the three baseline algorithms in C++.

The shortcut names of *Application-Centric* algorithms include the ordering rule (§4.2) and the size measure (Table 1). For example, *WFD-AvgExp* denotes the WFD algorithm with the size measure "average with exponential weight".

Node-Centric Algorithms with Decreasing Scores are denoted by *NCD* followed by the scoring name (Table 3). "Decreasing score" indicates the choice of the largest application-node score in each step. For example, *NCD-DotProduct* denotes the Node-Centric algorithm with decreasing dot-product score.

Considering *Multi-Node algorithms*, we focus on the versions with *replica spreading* and exclude the versions with *application-node matching* from our experiments, as their running times were observably too long even for the instances with 9,338 LRAs.

For the replica spreading versions we use prefix *Spread* in the notation, and postfix *BinSearch* or *Decr*, depending on the search strategy used for multiple calls with different values of the target number of nodes.

Binary search strategy narrows down the interval which estimates the minimum number of nodes. It uses Eq. (4) for the initial lower bound, and the output of the *First Fit (FF)* algorithm for the initial upper bound. For example, *SpreadWFD-Avg-BinSearch* denotes the spreading version of WFD (with "average" size measure) in combination with binary search.

The alternative, *Decrementing* approach arranges the search by decreasing the target number of nodes in steps. For the starting point, it uses the same value for the upper bound as binary search. In the notation, postfix *Decr* is followed by the step value. For example, *SpreadWFD-Avg-Decr2* denotes the spreading version of WFD (with "average" size measure) in combination with the decrementing approach, which decreases the target number of nodes from the best value found so far, in decrements computed as 2% of the lower bound.

5.2. Capabilities of the ILP Model

The instances introduced in Table 5 appeared to be too hard for the ILP model formulated in §3.2.

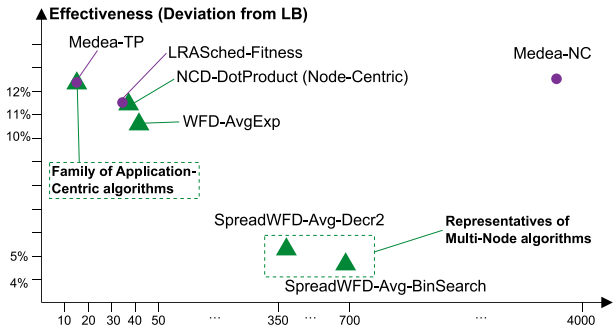


Figure 3: Performance summary of algorithms for instances with 9,338 LRAs, different affinity densities and without temporal changes

Considering smaller instances, we have found that solutions can be obtained for medium size instances, with up to 2,000 LRAs having about 15,500 replicas in total. In those instances, LRAs have resource requirements in CPU and memory, which do not change over time. This is the two-dimensional case of the problem under study. Allowing sufficiently large computation time, of up to 4 hours, Gurobi solver can find solutions within 0.2% from lower bounds.

Clearly, for instances with more than 2,000 LRAs, heuristics should be preferred due to their scalability and flexibility of integrating with real-life schedulers.

5.3. Results for Instances without Temporal Changes

In this section we discuss the performance of the algorithms on two-dimensional instances, which correspond to the penultimate row of Table 5. A high-level overview of the results, averaged over all 90 instances with different affinity densities, is illustrated in Fig. 3, and the overall shape of the trade-off does not change essentially in experiments with temporal changes. The trade-off between effectiveness and computation time can help practitioners in selecting the algorithm that best fits their requirements.

In the following, we analyze in depth the algorithms' performance on instances with varied affinity density (described in column 2 of Table 5) and on instances with varied number of LRAs (described in column 3 of Table 5). As no major differences were observed between the results obtained for the three types of affinity graphs, we report the results for the graphs of *arbitrary* type, unless specified.

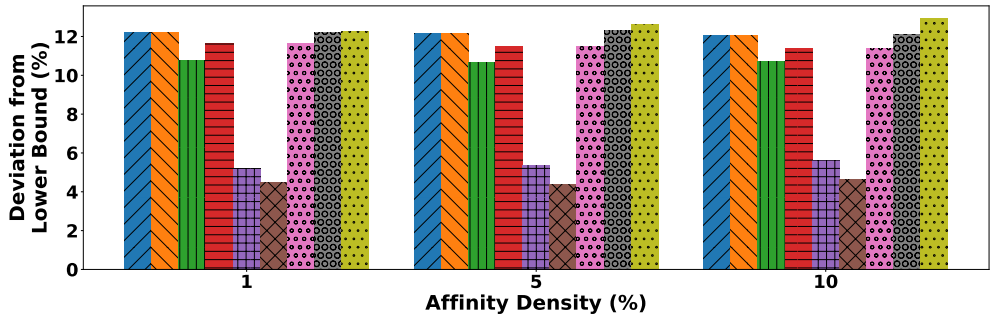
Effectiveness. *Instances with varied density.* In general, all Application-Centric algorithms (FF and various versions of FFD, BFD and WFD with different size measures) perform similarly, with approximately 12.1% deviation from the lower bound on average, with two exceptions. First, algorithms FFD, BFD and WFD with the "Extended Sum" measure are consistently worst-performing, with 15.6% deviation on average. Second, *WFD-AvgExp* has 10.7% deviation on average and consistently outperforms all others. The advantage of *WFD-AvgExp* stems from the focus on the most demanding dimensions when selecting the next LRA to be allocated.

Node-Centric algorithms place themselves between *WFD-AvgExp* and the other Application-Centric algorithms, with 11.5% deviation on average.

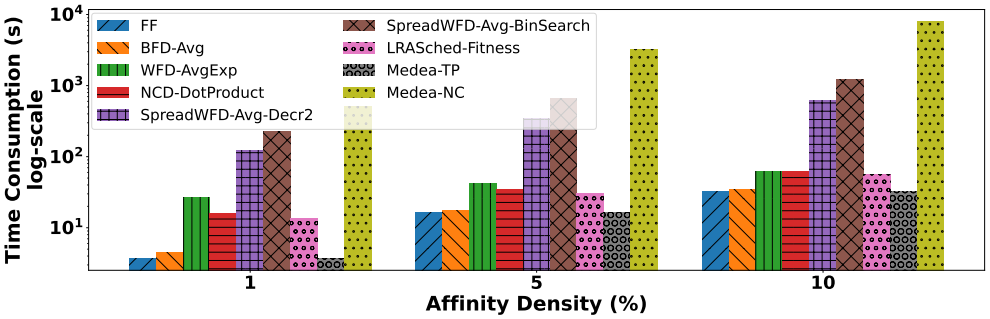
The spreading versions of the Multi-Node algorithms are particularly successful. For example, *SpreadWFD-Avg-BinSearch* and *SpreadWFD-Avg-Decr2* achieve 4.5% and 5.4% deviation from the lower bound, respectively. Solutions of similar quality are obtained by the versions of *SpreadWFD-AvgExp*, but at the cost of larger computation time (a consequence of computing a more elaborate size measure).

We visualize the results of the representatives of each algorithm family in Fig 4, where we also include the summary of the baseline algorithms. We observe that *Medea-NC*, with 12.6% deviation, is outperformed by all other algorithms (except for algorithms with the "ExtendedSum" measure not included in Fig 4), while *Medea-TP* performs similar to the Application-Centric algorithms, with 12.2% deviation. *LRASched-Fitness* works similar to other Node-Centric algorithms, with a slightly smaller execution time. Compared with these baselines, our algorithms of type *SpreadWFD-Avg* are 7% closer to the lower bound. This marginal number implies about 350 nodes saving, which is of significance for cost-effective and energy-efficient datacenters.

Comparing the results for different affinity densities we do not observe noticeable differences in the algorithms' effectiveness. The exceptions are *SpreadWFD-Avg-BinSearch* and *SpreadWFD-Avg-Decr2* applied to the instances with threshold graphs, where the deviation from the lower bound



(a) Effectiveness



(b) Time consumption - log-scale

Figure 4: Different affinity densities under fixed resource requests, $|\mathcal{L}| = 9,338$

increases from 3.6% to 10.5% as the graph density increases.

Instances with varied LRA number. As shown in Fig. 5(a), the algorithms' effectiveness generally improves when the LRA scale increases. With 100,000 LRAs, *FF*, *BFD-Avg* and *Medea-TP* achieve 2.5% deviation from the lower bound on average, *NCD-DotProduct* and *LRASched-Fitness* achieve 2.4% deviation, and *WFD-AvgExp* reaches 2% deviation.

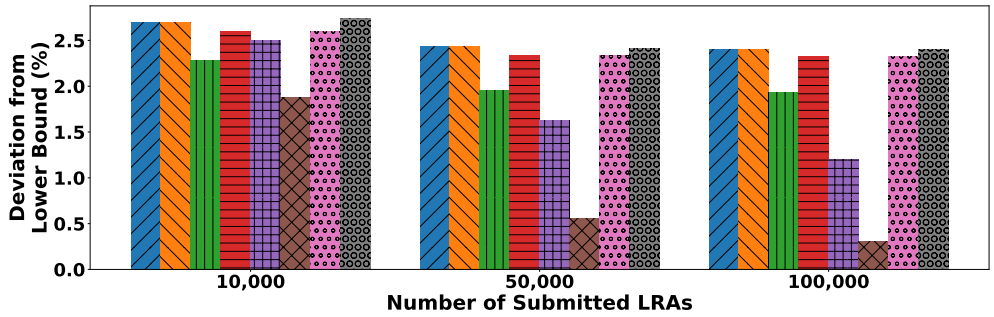
SpreadWFD-Avg-BinSearch and *SpreadWFD-Avg-Decr2* are particularly successful, achieving 0.9% and 1.8% deviation on average, with figures as low as 0.3% for *SpreadWFD-Avg-BinSearch* when applied to instances with 100,000 LRAs. However, an interesting anomaly was observed for smaller instances, with 10,000 LRAs: there were several instances with *arbitrary* and *normal* affinity graphs for which two *SpreadWFD* algorithms could not find better solutions than *FF*. Still the performance of *SpreadWFD* is the best even on small instances, if averaging the results of multiple experiments.

Execution time. *Instances with varied density.* Fig. 4(b) shows the average execution times of the algorithms when applied to the instances with different affinity densities. FFD-based algorithms are among the fastest, along with *FF* and *Medea-TP*, while BFD-based algorithms are slightly slower. All these algorithms merely take less than 5s, 18s, and 33s for densities 1%, 5% and 10%, respectively. In contrast, WFD-based algorithms are much slower, taking 26s, 41s and 61s, respectively.

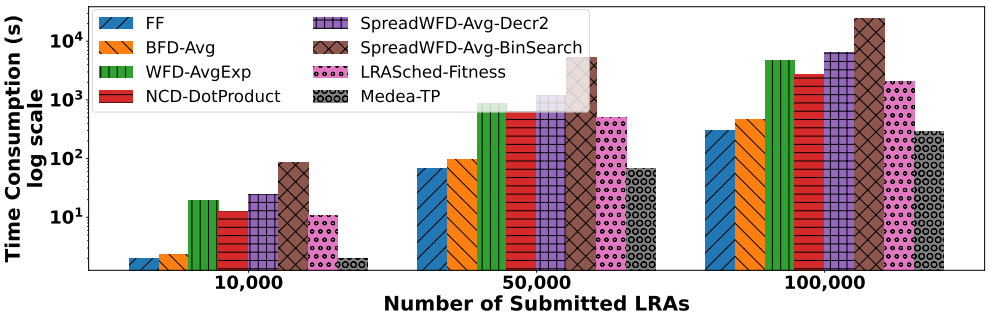
Node-Centric algorithms and *LRASched-Fitness* are in-between: *NCD-DotProduct* takes 16s, 34s and 62s on average for the three densities, while *LRASched-Fitness* runs a few seconds faster.

Overall, the relative difference in running times between these algorithms tends to decrease when the affinity density increases. With 10% density, the running times for the WFD-based algorithms are similar to *LRASched-Fitness* and *NCD-DotProduct*.

The best-performing algorithm *SpreadWFD-Avg-BinSearch* is unsurprisingly among the slowest algorithms, taking 225s, 653s and 1214s on average, when the affinity density grows. This is because binary search needs iterative calls of the replica spreading version of *WFD* to find the appropriate number of nodes. Replacing binary search by iteratively decreasing the number of target nodes enables *SpreadWFD-Avg-Decr2* to achieve a two-fold



(a) Effectiveness



(b) Time consumption - log-scale

Figure 5: Different LRA numbers under fixed resource requests, affinity graph density is 0.5%

speedup, compared with the binary search version.

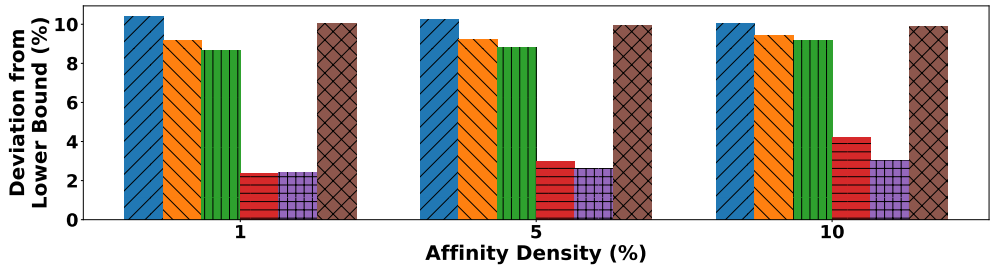
Medea-NC is the slowest algorithm observed. It takes on average 512s, 3,200s and 8,005s when handling the instances with 1%, 5% and 10% density. It is worth noticing that, while using fine-tuned data structure may reduce the running time of *Medea-NC*, its effectiveness would not change and remain inferior to other algorithms.

Instances with varied LRA number. Fig. 5(b) shows the average execution times of the algorithms applied to the instances with different numbers of LRAs, and obviously there is an increasing trend when there are more LRAs to be scheduled. The fastest algorithms include *FF*, *Medea-TP* and *BFD-Avg* that can solve instances with 100,000 LRAs within 8 minutes. In contrast, *LRASched-Fitness*, *NCD-DotProduct* and *WFD-AvgExp* are much slower, taking about 35, 45 and 78 minutes on average to do the same task. Spreading approaches take even longer time: 2 and 7 hours, respectively. *Medea-NC* was excluded from this series of experiments due to overly excessive execution time even for 10,000 LRAs. Aligned with Fig. 3, the results indicate that datacenter operators need to thoroughly strike a balance between the targeted solution quality and the permitted planning time to pinpoint the bespoke option.

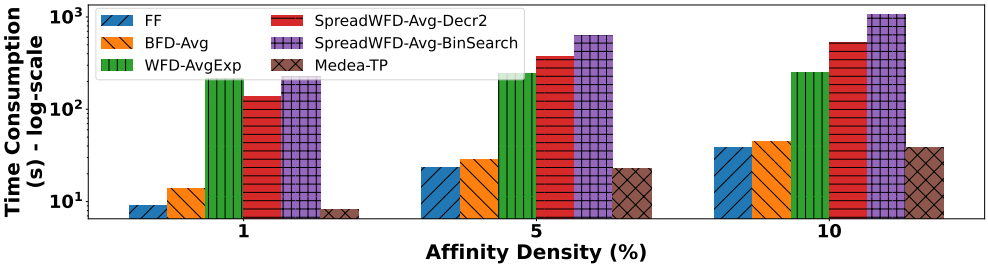
5.4. Results for Instances with Temporal Changes

The instances with time-varying resource requests of applications are modeled as the problem with $d = 98 \times 2$ dimensions, as described in the last row of Table 5. This dimension increase leads to a substantial growth of execution time. *Medea-NC*, *LRASched-Fitness* and Node-Centric algorithms such as *NCD-DotProduct* were discarded from the performance comparison for being too computationally expensive. Again, as no major differences were observed between results of the three different affinity graphs, we only report the results for the graphs of *arbitrary* type, unless specified.

Effectiveness. *Instances with varied density.* For the majority of the algorithms, the change in the affinity density does not significantly affect the accuracy of the solutions found, as demonstrated in Fig 6(a). The exceptions occur for the threshold graphs, similar to the instances without temporal changes: there is a substantial degradation in the performance of the two *SpreadWFD* algorithms, from 2.2% to 10.1% when the affinity den-



(a) Effectiveness



(b) Time consumption - log-scale

Figure 6: Different affinity densities under time-varying resource requests, $|\mathcal{L}| = 9,338$

sity changes from 1% to 10%. Again, this is because the *SpreadWFD* algorithms could not find better solutions than the given upper bound on several instances with 5% or 10% density, and the solutions from *FF* were used instead.

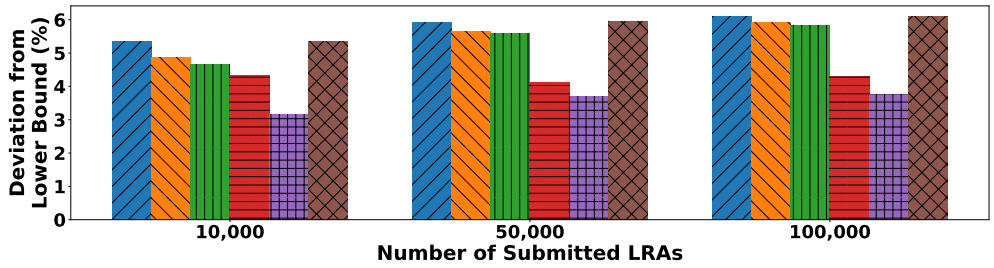
Instances with varied LRA number. As shown in Fig. 7(a), there is a negligible discrepancy among the performance of each algorithm with different numbers of LRAs, when handling time-varying resource requests. For example, with *SpreadWFD-Avg-BinSearch*, the deviation from the lower bound only increases from 3.2% to 3.8% when the LRA number grows from 10,000 to 100,000. Similar observations are valid for other algorithms, indicating that the proposed algorithms are successful in large-scale scenarios.

Execution time. *Instances with varied density.* As shown in Fig. 6(b), *FF*, *BFD-Avg* and *Medea-TP* can solve any instance within 45 seconds on average, while *WFD-AvgExp* finishes within 4 minutes and *SpreadWFD-Avg-Decr2* within 9 minutes. *SpreadWFD-Avg-BinSearch* takes about 18 minutes to solve high density instances with 9,338 LRAs, which seems to be the best choice of algorithm considering its achieved effectiveness of less than 3% deviation from the lower bound, on average. It is also worth noticing that, for instances with 1% density, the running times of *SpreadWFD-Avg-BinSearch* and *WFD-AvgExp* are similar and almost double the running time of *SpreadWFD-Avg-Decr2*. This is particularly unexpected for *WFD-AvgExp*, which involves one call of the application-centric WFD-algorithm, compared to multiple calls of *SpreadWFD-Avg-Decr2*.

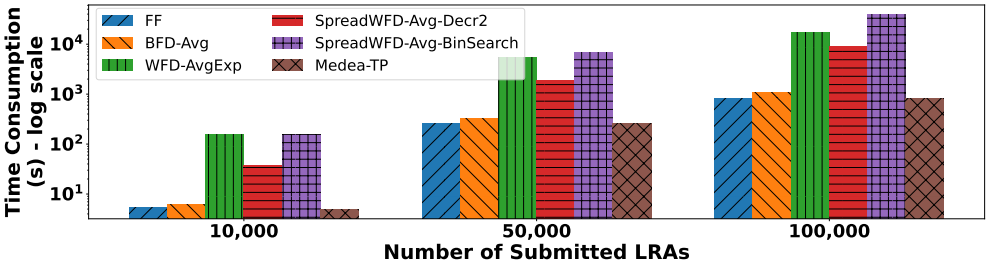
Instances with varied LRA number. As shown in Fig. 7(b), similar but smaller differences in the execution times can be observed under different submission scales, compared with the observations in Fig. 6(b). The disparity is due to the computation time of the size measures of LRAs with 196 dimensions. Numerically, *FF* and *Medea-TP* can solve any instance with 100,000 LRAs in 14 minutes on average and *BFD-Avg* takes 18 minutes. *SpreadWFD-Avg-Decr2*, *WFD-AvgExp* and *SpreadWFD-Avg-BinSearch* take 2.5, 5 and 11 hours, respectively, to solve the largest instances. Interestingly, *SpreadWFD-Avg-Decr2* appears to be the best choice for instances with time-varying resource requests, as it achieves effectiveness close to the best algorithm, *SpreadWFD-Avg-BinSearch*, with a 4-fold speedup in terms of the running time.

6. Algorithm Recommendations

We recommend Application-Centric algorithms if the computation time is required to be as small as possible. In that group of algorithms, the version of the traditional Bin Packing algorithm *First Fit*



(a) Effectiveness



(b) Time consumption - log-scale

Figure 7: Different LRA numbers under time-varying resource requests, affinity graph density is 0.5%

(FF), adjusted to handle the problem with replicas and affinities, is among the fastest approaches. Its solution quality is either similar or just slightly worse than the quality of solutions found by other Application-Centric algorithms. Only one published algorithm, *Medea-TP*, achieves comparable computation time and solution quality. As we show in §4, *Medea-TP* belongs to the same group of Application-Centric algorithms and differs from *FF* by an additional ordering of LRAs. It appears that, on the instances generated from the Alibaba Tianchi dataset, special ordering does not have a significant impact on the quality of the solution and on computation time.

We recommend Multi-Node algorithms if the primary aim is to find solutions of the best quality, possibly with longer but still acceptable computation time (say, up to 30 minutes to allocate $10,000$ LRAs). An ultimate winner in our experiments is *Spread-WFD-Avg-BinSearch*. It uses a special spreading mechanism to allocate replicas of the same LRA across different nodes. The spreading mechanism substantially increases the range of nodes suitable for co-location of a current application with a broader set of compatible LRAs. Additionally, it adopts binary search to identify the smallest, but feasible, number of nodes in the solution. None of the algorithms, either in our suite or among the published ones, achieves the same solution quality, namely 0.3% deviation from the lower bound, when handling instances with $100,000$ LRAs.

Finally, in-between the two extremes of fastest but less accurate algorithms, and slowest but most accurate ones, there are those of intermediate running time and intermediate solution quality. All Node-Centric algorithms fall into this category, with *LRASched-Fitness* and *NCD-DotProduct* being best performing. Both algorithms produce solutions of comparable quality and differ slightly in their running times: *LRASched-Fitness* is faster on instances with affinities, while *DotProduct* is faster and superior in terms of the solution quality on instances without affinities.

There is one outlier in the Application-Centric group, *WFD-AvgExp*: it performs slower than the majority of Application-Centric algorithms and slower than the Node-Centric algorithms but outperforms all of them in terms of the solution quality. We would like to observe that overall the Application-Centric algorithm WFD is often overlooked by practitioners and not included in their trials.

Note that we observe that all algorithms become much slower for instances with time-varying profiles, and the Node-Centric algorithms become prohibitively slow. Therefore, we narrow down our recommendations to *Medea-TP* and *FF* (the fastest),

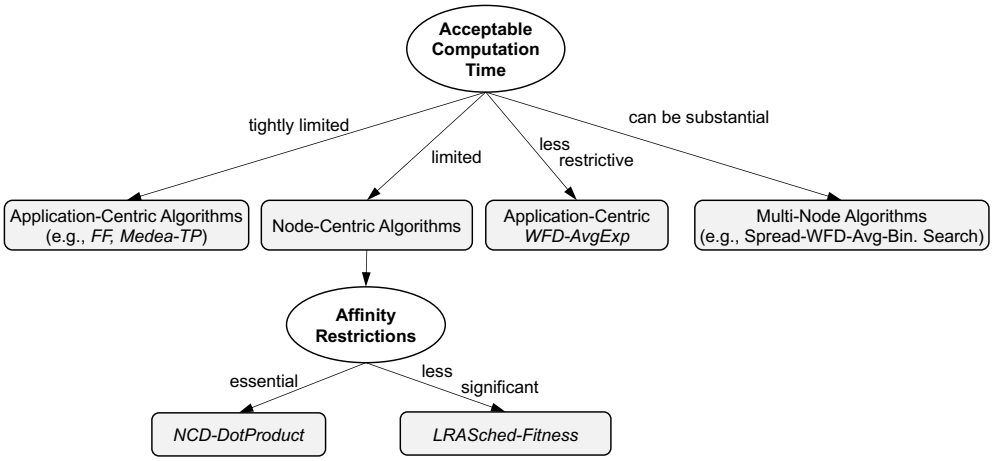


Figure 8: Algorithm selection dependent on practitioners' needs

Spread-WFD-Avg-Decr2 (of intermediate running time and solution quality) and *Spread-WFD-Avg-BinSearch* (the best solution quality).

As a summary we present Fig. 8 which highlights our recommendations for practitioners on selecting the most promising solution approach depending on the application scenario and practitioners' needs. The main question to consider is the acceptable computation time. For most of the scenarios the limitations related to affinity restrictions do not affect the algorithm choice: generally speaking, the density of the affinity restrictions does not change the performance of our algorithms essentially. The only exception is the group of the Node-Centric algorithms, in which the Dot-Product version is better suited for solving problems with multiple affinity constraints, and the LRASched-Fitness [17] might have advantages for less restricted problems.

While we provide generic algorithm recommendations, one common practice in large-scale system engineering is to further conduct trade-off analysis on a case-by-case basis given the requirements of algorithm quality and execution time. Scheduler system administrators or developers can first run profile-based testing based on sampled data to pick up competitive candidate algorithms and validate in a small-scale test system. This procedure can significantly help to understand system behaviors in a controllable manner – Unseen instances are rare as the workload of a production system is supposed to be stable, and profiling and small scale tests can usually capture most of the workloads and learn their patterns. Then larger and diverse instances could be used to tune the performance of algorithms and figure out the best performer before deploying the algorithms into production grade systems.

7. Practical Considerations

Integration into multi-stage cluster management. While this paper focuses on the algorithmic support for resource provisioning, the proposed algorithm suite can be more widely integrated in a multi-stage cluster management that consists of cluster initialization and runtime scheduling.

At the initialization stage, given that the scheduling system foreknows all information of LRAs to be submitted, the resource planner that runs the algorithm suite can work out the best option for scheduling the LRAs with the minimal required nodes. Horizontal scaling will be consequently used to match the planning outcome, through elastically sizing the number of bare metal servers or virtual machines in the resource pool. Once the cluster is initialized for hosting the LRAs, the cluster management will shift into the runtime scheduling stage that responds to the new LRA submissions and available resource release. Cluster schedulers can accept any incoming LRA in the regular round of resource allocation [8, 16, 29, 30]. Consequently, the admitted LRAs will gradually consolidate the nodes in the cluster until there is no room for new LRAs and a long waiting queue manifests. Cluster auto-scaling will be performed to mitigate the long starvation of the waiting LRAs and handle dynamic load spikes. The resource provisioning algorithm will be re-triggered accordingly.

Runtime management considerations. While our algorithm suite can provide competitive solu-

tions that minimize the number of required computing nodes, the resource provisioning in practice usually comes with some resource slack or over-provisioning to increase reliability for the unknown and prevent degradation in user experience. Based upon the calculation of initial resource provisioning as a guidance, additional resource reservation by system operators allows to mitigate uncertainties at runtime such as an excessive increase in LRA's tail latency, out-of-memory problems when the LRA's resource usage fluctuates, failures or stragglers due to unexpected data stream coming into the LRA, etc. The reserved yet idle resources can be harvested by using a series of system optimization techniques including hypervisor or kernel level oversubscription [16, 29, 40] and core reassignment mechanism [41].

Other objectives considerations. The scheduling problem formulated in the paper is an attempt to find the minimum number of nodes that accommodate different LRA scales and affinity restriction densities. However, in real scenarios, the compute capability is sometimes limited compared to the increasing number of LRAs. The Multi-Node algorithms are well suited to address these types of problems. They operate with a fixed value n for the number of nodes, given as part of the input. In the implementation described in §4, a Multi-Node algorithm declares a failure if not all LRAs are allocated to the pool of n nodes. However, the LRA allocation, available after the algorithm terminates, is an appropriate solution for the problem with a given node value n. Depending on the optimization criterion, one may decide to adopt the Multi-Node Algorithms with Replica Spreading, if the number of accepted LRAs is to be maximized, or the Multi-Node Algorithm with Application-Node Matching if the node utilization is to be maximized.

8. Related Work

Cluster management. Resource management systems in shared clusters can be divided into two categories: centralized and decentralized systems. Centralized approaches assign resources based on user requests [7, 8, 14] or framework offers [42]. Multiple resources are negotiated among diverse applications through a central resource manager. To make the procedure fair and avoid resource starvation, Dominant Resource Fairness [43], capacity or fair scheduling are adopted for resource sharing among multiple jobs. Decentralized approaches [15, 29, 44, 45] are developed for clusters that expect a high throughput or high cluster utilization. However, the goal of these works is to enable sub-second resource allocation and task scheduling at runtime without solving a global optimization problem with complex placement constraints.

LRA scheduling. YARN [14] mainly supports the affinity constraints related to nodes/racks. Borg [8] and ROSE [15] use machine scoring mechanism for matching a specific collection of nodes to the requirements of the applications. Graph-based approaches [46, 47] model the scheduling problem as a min-cost max-flow optimization over a network. However, they merely consider one dimension in the capacity constraint, and affinities to specific machines constraints. An attempt to incorporate those additional features in Aladdin [48] makes it prohibitive for applying powerful min-cost max-flow methods.

Application-level affinity is increasingly important. Kubernetes scheduler [13] is responsible for selecting the best node for each incoming pod. A pod is referred to as an independent execution unit and is equivalent to one replica of an LRA in this paper. A *ReplicaSet* parameter ensures that a specified number of pods are running anytime. However, it considers one pod at each scheduling round and implements the node selection in a filtering phase. The nodes that cannot run the pod are ruled out considering the specifications in the node/pod affinity. This design leads to one-shot resource allocation to a pod rather than considering it as a global optimization problem.

Medea [9] formulates the placement problem as an ILP and employs heuristics periodically to consider multiple LRAs at once at a lower scheduling latency. However, the focus of the authors is on scheduling a small batch of LRAs. By contrast, our work addresses pre-execution resource planning for the whole set of LRAs.

We also refer the reader to thorough surveys on wide-ranging Bin Packing algorithm design [18, 33, 34].

In addition, a huge body of machine learning and reinforcement learning based scheduling techniques offer alternatives for scheduling LRAs to mitigate the limitations of manual specification and resource estimation – which usually require expert knowledge and operational experience – in the process of requirement engineering. LraSched [17] em-

ploys online machine learning for auto-estimating the size of LRAs' containers and the degree of affinity. Metis [49] and George [50] adopt deep reinforcement learning (DRL) to automatically learn to place LRAs based on observing the incurred reward and iteratively improving the scheduling policy. However, these works heavily depend on a huge number of high-quality workload logs, which are feasible for big companies but will place a huge obstacle on small businesses and academic organizations. Due to the exponential space of actions, DRL-based solutions are also limited to small-scale optimization problem, and thus only applicable to on-the-fly decision making.

Interference-aware LRA runtime management. There is a substantial body of research on interference-aware LRA scheduling and runtime management. Paragon [51] and Quasar [52] use multi-variable statistical classifiers to predict the expected interference among co-located LRAs. ROSEQ [16] and Toposch [30, 53] devise performance-aware scheduling mechanisms that can safely co-locate batch jobs together with LRAs through elaborately monitoring the runtime performance of the LRAs. Horus [54] and Mendoza et al. [55] propose interference-aware schedulers for inference serving or model training, reducing the latency degradation from co-location interference or holistic training time. However, kernel/application-level counters are leveraged to track the runtime performance of the LRAs as a whole, without discussing the replicas and their impact on the scheduling quality. Overall, the focus of these research works prioritize the performance guarantee through effective container isolation and low-cost preemption. They are orthogonal to the resource provisioning scheme developed in this paper and offer supplementary mechanisms in the runtime execution stage.

9. Conclusions and Future Work

Resource provisioning of shared clusters is extremely important for minimizing the operating cost and ensuring that the scheduling systems meet both current and future demands. LRA workloads add further complexity to resource provisioning since they run from hours to months, typically having time-varying resource requirements and co-location affinity constraints. Careless or no planning often leads to poor utilization and performance of a cluster system.

This paper develops an affinity-aware resource provisioning scheme for LRA placement in shared clusters, supported by a new system model and an adjustable algorithmic toolkit. The main benefits of that toolkit are as follows.

- Consisting of dozens of algorithms with multiple parameters, there are three major approaches which complement each other. Their implementation can be streamlined as algorithms' building blocks are of similar nature.
- Application-Centric approach is the most popular one with researchers and practitioners. However, one of its algorithms, Worst Fit Decreasing, is broadly overlooked in the literature and in practice. Our experiments show that it often outperforms all other Application-Centric algorithms in terms of solution quality, and its execution time is comparable to the execution times of widely used First Fit Decreasing and Best Fit Decreasing algorithms from the same approach. Worst Fit Decreasing also outperforms the Node-Centric algorithms but at the cost of a slightly longer execution time.
- The third and novel approach is Multi-Bin activation. While it involves multiple calls to one of the LRA allocation functions of Application-Centric and Node-Centric approaches, individual calls are relatively fast. If needed, the algorithm can be terminated earlier, still achieving improved solutions compared to the first two approaches.
- The proposed toolkit is comprehensive and, together with new approaches, it encompasses a variety of the published algorithms, which can be classified as special cases of the Application-Centric and Node-Centric approaches. A systematic summary of size measures and score functions, provided in this paper, makes the toolkit tunable to fit specific features of real-world scenarios. We have illustrated how the tuning works based on an Alibaba public dataset and similar work could be conducted for any required scenario.

In the future, we plan to investigate automatic algorithm selection from our algorithm pool and automatic tuning of the selected algorithm. We also plan to integrate the proposed heuristics into Kubernetes to evaluate how theoretical study can navigate the runtime execution.

Acknowledgments

This work was supported by UK EPSRC Grant (EP/T01461X/1), Turing Pilot Project and Turing PDEA Scheme funded by UK Alan Turing Institute. Experiments were undertaken on ARC4, part of the High Performance Computing facilities at the University of Leeds, UK. Clément Mommessin and Renyu Yang are co-first authors with equal contribution.

References

[1] L. Q. Torres, D. Colish, SRE Best Practices for Capacity Management, Proc. of USENIX PATRONS (2020) 49.

[2] L. Cherkasova, W. Tang, S. Singhal, An SLA oriented capacity planning tool for streaming media services, in: Proc. of IEEE DSN, 2004, pp. 743–752.

[3] L. Helali, M. N. Omri, A survey of data center consolidation in cloud computing systems, Computer Science Review 39 (2021) 100366.

[4] R. Yang, J. Xu, Computing at massive scale: Scalability and dependability challenges, in: 2016 IEEE symposium on service-oriented system engineering (SOSE), IEEE, 2016, pp. 386–397.

[5] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[6] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, C. Curino, Apache Tez: a unifying framework for modeling and building data processing applications, in: Proc. of ACM SIGMOD, 2015, pp. 1357–1369.

[7] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, J. Xu, Fuxi: a fault-tolerant resource management and job scheduling system at internet scale, in: Proc. of VLDB Endowment, 2014, pp. 1393–1404.

[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at Google with Borg, in: Proc. of ACM Eurosys, 2015, pp. 1–17.

[9] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, S. Rao, Medea: scheduling of long running applications in shared production clusters, in: Proc. of EuroSys, 2018, pp. 1–13.

[10] Q. Liu, Z. Yu, The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from Alibaba trace, in: Proc. of ACM SoCC, 2018, pp. 347–360.

[11] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, Y. Bao, Who limits the resource efficiency of my datacenter: an analysis of Alibaba datacenter traces, in: Proc. of ACM IWQoS, 2019, pp. 1–10.

[12] L. Cherkasova, W. Tang, S. Singhal, Providing high availability using lazy replication, ACM TOCS 10 (4) (1992) 360–391.

[13] Kubernetes.
URL `kubernetes.io/docs/concepts`

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache Hadoop YARN: yet another resource negotiator, in: Proc. of ACM SoCC, 2013, pp. 1–16.

[15] X. Sun, C. Hu, R. Yang, P. Garraghan, T. Wo, J. Xu, J. Zhu, C. Li, ROSE: cluster resource scheduling via speculative over-subscription, in: Proc. of IEEE ICDCS, 2018, pp. 949–960.

[16] R. Yang, C. Hu, X. Sun, P. Garraghan, T. Wo, Z. Wen, H. Peng, J. Xu, C. Li, Performance-aware speculative resource oversubscription for large-scale clusters, IEEE TPDS 31 (7) (2020) 1499–1517.

[17] B. Cai, Q. Guo, J. Yu, LraSched: Admitting More Long-Running Applications via Auto-Estimating Container Size and Affinity, The Computer Journal.

[18] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, D. Vigo, Bin Packing Approximation Algorithms: Survey and Classification, in: Handbook of Combinatorial Optimization, 2013, pp. 455–531.

[19] Alibaba Tianchi Dataset.
URL `https://tianchi.aliyun.com/dataset/dataDetail?dataId=6287&lang=en-us`

[20] Statista survey.
URL `https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/`

[21] Storm.
URL `storm.apache.org`

[22] Flink.
URL `flink.apache.org`

[23] Kafka stream.
URL `kafka.apache.org`

[24] HBase.
URL `hbase.apache.org`

[25] MongoDB.
URL `www.mongodb.com`

[26] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al., Apache Spark: a unified engine for big data processing, Communications of the ACM 59 (11) (2016) 56–65.

[27] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: a system for large-scale machine learning, in: Proc. of USENIX OSDI, 2016, pp. 265–283.

[28] Y. Cheng, Z. Chai, A. Anwar, Characterizing co-located datacenter workloads: an Alibaba case study, in: Proc. of ACM APSys, 2018, pp. 1–3.

[29] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, C. Kozyrakis, Heracles: improving resource efficiency at scale, in: Proc. of ACM ISCA, 2015, pp. 450–462.

[30] C. Hu, J. Zhu, R. Yang, H. Peng, T. Wo, S. Xue, X. Yu, J. Xu, R. Ranjan, Toposch: Latency-Aware Scheduling Based on Critical Path Analysis on Shared YARN Clusters, in: Proc. of IEEE CLOUD, 2020, pp. 619–627.

[31] Alibaba cluster trace.
URL `github.com/alibaba/clusterdata`

[32] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, R. Bianchini, Resource central: understanding and predicting workloads for improved resource management in large cloud platforms, in: Proc. of ACM SOSP, 2017, pp. 153–167.

[33] S. Martello, P. Toth, Bin-Packing Problem, in: Knapsack Problems: Algorithms and Computer Implementations, Wiley, 1990, pp. 221–245.

[34] R. Panigrahy, K. Talwar, L. Uyeda, U. Wieder, Heuristics for Vector Bin Packing, Microsoft Research.

[35] A. Caprara, P. Toth, Lower bounds and algorithms for

the 2-dimensional vector packing problem, Discrete Applied Mathematics 111 (3) (2001) 231–262.
[36] A. E. F. Muritiba, M. Iori, E. Malaguti, P. Toth, Algorithms for the Bin Packing Problem with Conflicts, INFORMS Journal on Computing 22 (3) (2010) 401–415.
[37] R. Sadykov, F. Vanderbeck, Bin Packing with conflicts: a generic branch-and-price algorithm, INFORMS Journal on Computing 25 (2) (2013) 244–255.
[38] M. Gendreau, G. Laporte, F. Semet, Heuristics and lower bounds for the bin packing problem with conflicts, Computers and Operations Research 31 (3) (2004) 347–358.
[39] T. Bacci, S. Nicoloso, On the benchmark instances for the Bin Packing with Conflicts, arXiv preprint arXiv:1706.03526.
[40] S. Kim, H. Kim, J. Lee, J. Jeong, Group-based memory oversubscription for virtualized clouds, Journal of Parallel and Distributed Computing 74 (4) (2014) 2241–2256.
[41] A. Ousterhout, J. Fried, J. Behrens, A. Belay, H. Balakrishnan, Shenango: achieving high CPU efficiency for latency-sensitive datacenter workloads, in: Proc. of NSDI, 2019, pp. 361–378.
[42] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, I. Stoica, Mesos: a platform for fine-grained resource sharing in the data center, in: Proc. of USENIX NSDI, 2011.
[43] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica, Dominant Resource Fairness: Fair Allocation of Multiple Resource Types, in: Proc. of USENIX NSDI, 2011.
[44] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, L. Zhou, Apollo: scalable and coordinated scheduling for cloud-scale computing, in: Proc. of USENIX OSDI, 2014, pp. 285–300.
[45] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, S. Sakalanaga, Mercury: hybrid centralized and distributed scheduling in large shared clusters, in: Proc. of USENIX ATC, 2015, pp. 485–497.
[46] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: fair scheduling for distributed computing clusters, in: Proc. of ACM SOSP, 2009, pp. 261–276.
[47] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, S. Hand, Firmament: fast, centralized cluster scheduling at scale, in: Proc. of USENIX OSDI, 2016, pp. 99–115.
[48] H. Wu, W. Zhang, Y. Xu, H. Xiang, T. Huang, H. Ding, Z. Zhang, Aladdin: optimized maximum flow management for shared production clusters, in: Proc. of IEEE IPDPS, 2019, pp. 696–707.
[49] L. Wang, Q. Weng, W. Wang, C. Chen, B. Li, Metis: Learning to schedule long-running applications in shared container clusters at scale, in: SC20, IEEE, 2020, pp. 1–17.
[50] S. Li, L. Wang, W. Wang, Y. Yu, B. Li, George: learning to place long-lived containers in large clusters with operation constraints, in: Proc. of ACM SoCC, 2021, pp. 258–272.
[51] C. Delimitrou, C. Kozyrakis, Paragon: QoS-aware scheduling for heterogeneous datacenters, ACM SIGPLAN Notices (2013) 77–88.
[52] C. Delimitrou, C. Kozyrakis, Quasar: resource-efficient and QoS-aware cluster management, ACM SIGPLAN Notices (2014) 127–144.
[53] J. Zhu, R. Yang, X. Sun, T. Wo, C. Hu, H. Peng, J. Xiao, A. Y. Zomaya, J. Xu, Qos-aware co-scheduling for distributed long-running applications on shared clusters, IEEE TPDS 33 (12) (2022) 4818–4834.
[54] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, P. Garraghan, Horus: Interference-aware and prediction-based scheduling in deep learning systems, IEEE TPDS 33 (1) (2022) 88–100.
[55] D. Mendoza, F. Romero, Q. Li, N. J. Yadwadkar, C. Kozyrakis, Interference-aware scheduling for inference serving, in: Proc. of MLSys, 2021, pp. 80–88.