MEDEA: Scheduling of Long Running Applications in Shared Production Clusters

Panagiotis Garefalakis*
Imperial College London

Konstantinos Karanasos
Microsoft

Peter Pietzuch
Imperial College London

Arun Suresh
Microsoft

Sriram Rao
Microsoft

ABSTRACT

The rise in popularity of machine learning, streaming, and latency-sensitive online applications in shared production clusters has raised new challenges for cluster schedulers. To optimize their performance and resilience, these applications require precise control of their placements, by means of complex constraints, e.g., to collocate or separate their long-running containers across groups of nodes. In the presence of these applications, the cluster scheduler must attain global optimization objectives, such as maximizing the number of deployed applications or minimizing the violated constraints and the resource fragmentation, but without affecting the scheduling latency of short-running containers.

We present MEDEA, a new cluster scheduler designed for the placement of long- and short-running containers. MEDEA introduces powerful placement constraints with formal semantics to capture interactions among containers within and across applications. It follows a novel two-scheduler design: (i) for long-running containers, it applies an optimization-based approach that accounts for constraints and global objectives; (ii) for short-running containers, it uses a traditional task-based scheduler for low placement latency. Evaluated on a 400-node cluster, our implementation of MEDEA on Apache Hadoop YARN achieves placement of long-running applications with significant performance and resilience benefits compared to state-of-the-art schedulers.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Software and its engineering** → **Scheduling**; *Cloud computing*; • **Theory of computation** → *Linear programming*;

ACM Reference Format:
Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. MEDEA: Scheduling of Long Running Applications in Shared Production Clusters. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3190508.3190549

*The bulk of the work was done while the author was an intern at Microsoft.

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190549

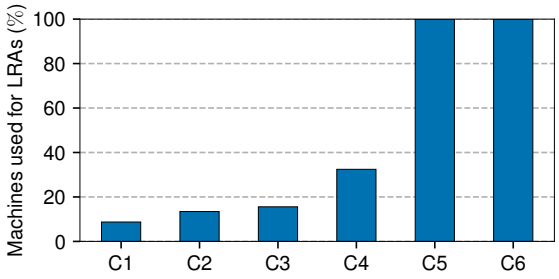


Figure 1: Machines used for long-running applications (LRAs) in six analytics clusters at Microsoft

1 INTRODUCTION

Modern organizations operate large clusters, which are typically shared across several users and applications. In this environment, cluster managers such as YARN [52], Mesos [27], and Borg [53] carry out the on-demand allocation of resources to applications. They employ schedulers that package resources (e.g., CPU and memory) as *containers* and allocate them to jobs. Since cluster managers are application-agnostic, they have enabled cluster operators to consolidate diverse workloads onto shared clusters.

Apart from traditional batch analytics jobs [4, 12, 17, 59], workloads in production clusters now include stream processing [11, 26], iterative computations [1], data-intensive interactive jobs [57], and latency-sensitive online applications [7, 40]. Unlike batch jobs that typically use short-lived containers (in the order of seconds), these applications benefit from *long-lived containers*. These containers are allocated and used for durations ranging from hours to months, thus avoiding repeated container initialization costs and reducing scheduling load. We refer to this class of applications as *long-running applications* (*LRAs*).

In fact, a substantial portion of production clusters today is dedicated to LRAs. As shown in Figure 1, across six analytics clusters at Microsoft, each comprised of tens of thousands of machines, at least 10% of each cluster's machines are used for LRAs. Two of them are used exclusively for LRAs. At the same time, placing LRAs, along with batch jobs, in shared clusters is appealing to reduce cluster operational costs, avoid unnecessary data movement, and enable pipelines involving both classes of applications.

Despite these observations, support for LRAs in existing schedulers is rudimentary [2, 31, 33, 53]. In particular, the bespoke scheduling requirements of LRAs (§2) remain largely unexplored (§8): (i) Precise control of container placement is key for optimizing the *performance* and *resilience* of LRAs. Simple affinity and anti-affinity constraints (e.g., collocate containers to reduce network costs or

separate them to minimize resource interference or chance of failure), which are already partially supported by a few schedulers, are necessary but not sufficient. Our experiments with various LRAs (e.g., HBase, TensorFlow, and Storm; see §2) reveal that **powerful constraints are required to capture interactions between containers and unlock the full potential of LRAs**.

(ii) When placing LRA containers, the cluster scheduler must achieve global optimization objectives, such as minimizing the violation of placement constraints, the resource fragmentation, any load imbalance, or the number of machines used. Due to their long lifetimes, LRAs can tolerate longer scheduling latencies than traditional batch jobs. The second requirement for LRA placement is therefore to allow cluster operators to **optimize for global cluster objectives, but without impacting the scheduling latency of short-lived containers**.

Motivated by these requirements, we describe the design and implementation of **Medea**[1], a new cluster scheduler that enables the placement of both long- and short-running containers. Our work makes the following contributions:

(i) Two-scheduler design. Medea uses a dedicated scheduler for the placement of LRAs, but task-based applications are scheduled directly by a traditional scheduler. This two-scheduler design ensures that the scheduling latency for task-based applications is not impacted, while enabling high-quality placements for LRAs. It also makes Medea compatible with existing task-based schedulers, reusing existing scheduler implementations and facilitating adoption in production settings (§3).

(ii) Expressive, high-level constraints. Medea enables application owners and cluster operators to specify powerful placement constraints across LRA containers with formal semantics. We show that a single generic constraint type is sufficient to express a wide variety of use cases related to application performance and resilience. Relying on the notions of *container tags* and *node groups*, Medea supports both intra- and inter-application constraints, without requiring knowledge of the cluster's configuration or of already-deployed LRAs (§4).

(iii) LRA scheduling algorithm. We formulate the placement of LRAs with constraints as an integer linear program (ILP), and solve it as an online optimization problem. Unlike existing approaches, our algorithm considers multiple LRA container requests at once to achieve higher-quality placements and global objectives (§5). Moreover, we investigate heuristics that trade placement quality for lower scheduling latency.

We implemented Medea (§6) as an extension of Apache Hadoop YARN [4], one of the most widely deployed cluster schedulers, used by companies such as Microsoft, Yahoo!, Twitter, LinkedIn, EBay, Cloudera, and Hortonworks. Moreover, we open-sourced our implementation, which will be included in the upcoming Apache Hadoop 3.1 release.[2] We are currently in the process of deploying Medea in production clusters.

Our experimental evaluation highlights the benefits of Medea when placing LRAs (§7). On a 400-node pre-production cluster, Medea reduces median runtime of HBase and TensorFlow workloads by up to 32% compared to our implementation of Kubernetes' scheduling algorithm (J-Kube) and by 2.1× compared to YARN, while significantly reducing runtime variability too. Moreover, it improves application unavailability by up to 24% compared to J-Kube. Medea leads to constraint violations of less than 10% even for complex inter-application constraints involving 10 LRAs, and to reasonable scheduling latencies. Finally, it does not affect neither the scheduling latency nor the performance of task-based jobs.

[1]From Greek Medeia, verb medomai: "to think, to plan".
[2]The open-sourcing effort can be tracked in [43].

2 LONG-RUNNING APPLICATIONS IN CLUSTERS

In this section, we explore the new challenges that cluster schedulers face due to LRAs. First, we describe practical use cases in production environments (§2.1). Then we motivate the importance of LRA placement for the application performance (§2.2) and resilience (§2.3), and for the global objectives imposed by cluster operators (§2.4). We conclude by listing the scheduling requirements for LRAs (§2.5).

2.1 Use cases

Based on discussions with cluster operators from several companies, we identify the following scenarios with LRAs:

- **Streaming systems** [3, 8, 9, 11, 21, 35] process data in near real-time via dataflows of operators deployed using containers.
- **Interactive data-intensive applications** [32, 37, 57] use long-standing workers (*executors*) to avoid container start-up costs and process in-memory data with low latency.
- **Latency-sensitive applications** [7, 28, 40] serve requests using long-standing containers to achieve low latency.
- **Machine learning frameworks** [1, 13, 47] employ executors to perform iterative computation efficiently.

Within Microsoft's shared clusters, tens of unique application classes involve LRAs, falling into the above scenarios (see also Figure 1). At the same time, task-oriented batch jobs are also submitted to these clusters.

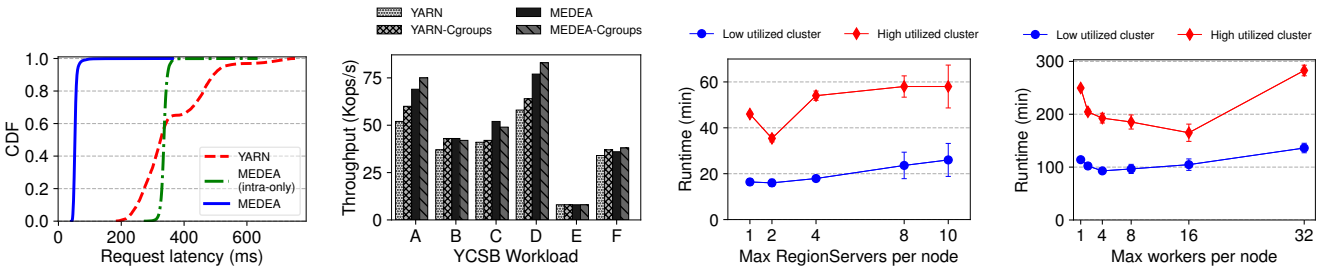
2.2 Application performance

Next we study the benefit of placement constraints (affinity, anti-affinity, and cardinality) on application performance.

Affinity. It is often beneficial to collocate the containers of an LRA on the same node or group of nodes, e.g., to reduce network traffic between containers of the same or different applications.

To observe the impact of this *intra-* and *inter-application affinity* on application performance, we deploy a Storm application on a 275-node cluster using YARN. The application identifies top-k trending hashtags on Twitter over a 60-second sliding window using an input stream of 6,000 tweets per second [51]. The resulting hashtags are combined with user profiles loaded from Memcached that stores a total of two million user profiles. We use five supervisors for Storm and a single instance for Memcached.

We compare three container placements: (i) with no constraints (no-constraints); (ii) all Storm containers on the same node (intra-only); and (iii) both Storm and Memcached containers on the same



(a) Memcached lookup latency with node affinity constraints

(b) HBase throughput with node anti-affinity constraints

(c) HBase total runtime with cardinality constraints

(d) TensorFlow runtime with cardinality constraints

Figure 2: Impact of placement constraints on application performance

node (intra-inter). Our results show that intra-only leads to an end-to-end latency improvement of 31% over no-constraints due to reduced network costs. However, this strategy cannot improve mean Memcached lookup latency (Figure 2a). On the other hand, using intra-inter, we can reduce mean Memcached latency by 4.6× and end-to-end latency by 5× over intra-only and by 7.6× over no-constraints (Figure 2a). Therefore, both intra- and inter-application affinity constraints are crucial to unlock full application performance.

Anti-affinity. To minimize resource interference between LRAs, it may be desirable to place containers on different machines through *intra-* and *inter-application anti-affinity*.

To validate the performance benefit of such constraints, we deploy 40 HBase instances with 30 region servers each, occupying 30% of the cluster's memory. We use the YCSB benchmark [15] with a dataset of 1 billion records (1 TB) and submit six YCSB workloads through multiple clients to generate load for the HBase instances. To emulate a shared production environment, we also submit GridMix batch jobs [24] that use 60% of the cluster's memory.

We first compare the following placements: (i) no-constraints; and (ii) with anti-affinity constraints to avoid collocating region servers of the same or different HBase instances on the same node (anti-affinity). As Figure 2b shows, no-constraints achieves 34% lower throughput compared to anti-affinity, as it can lead to collocated region servers, competing for CPU and I/O resources. Our experiment also reveals that no-constraints incurs increased tail latency compared to anti-affinity by up to 3.9× for the 99th percentile.

Furthermore, we repeat the above experiment using *cgroups* [38] to assess whether resource isolation mechanisms are sufficient to improve performance, instead of placement constraints. As shown in Figure 2b, although cgroups improve the throughput of no-constraints by 20%, they cannot match the performance of anti-affinity. The isolation offered by cgroups cannot prevent interference between resources not managed by the OS kernel, such as CPU caches and memory bandwidth. Hence, anti-affinity is required for optimizing LRA performance; combining it with resource isolation leads to the highest performance gains.

Cardinality. The affinity and anti-affinity constraints represent the two extremes of the collocation spectrum. To strike a balance between the two, we experiment with more flexible *cardinality* constraints, which set a limit on the number of collocated containers.

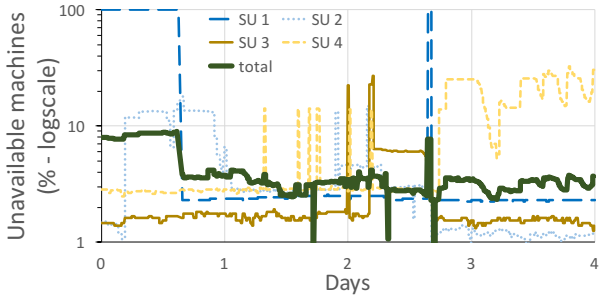


Figure 3: Unavailable machines in a Microsoft cluster (total is the percentage over all machines; SU1–SU4 are percentages over specific service units, i.e., logical node groups.)

Figure 2c reports the time required to run all YCSB workloads using 10 HBase region servers (RS) with full anti-affinity (i.e., 1 RS per node) to full affinity (i.e., all 10 RS on one node). Similarly, Figure 2d shows the time required for TensorFlow to complete a machine learning workflow with one million iterations using 32 workers, each time with a varying maximum workers per node. We use GridMix for additional cluster load (5% and 70% of cluster memory for the "low" and "high" utilized clusters, respectively).

Based on these results, we make the crucial observation that affinity and anti-affinity constraints, albeit beneficial, are not sufficient, and tighter placement control using cardinality constraints is required. In our experiments in the highly utilized cluster, we observe that collocating up to 16 TensorFlow workers on a node reduces runtime by 42% compared to the affinity placement (maximum cardinality of 32), and by 34% compared to the anti-affinity placement (maximum cardinality of 1). A second observation is that the cardinality that leads to optimal runtimes can vary based on the specific application and the current cluster load. Indeed, in the experiment of Figure 2d, the optimal cardinality value is 16 for the highly utilized cluster and 4 for the the less utilized one.

2.3 Application resilience

Unavailable machines in large clusters are common. This is due to machine failures, scheduled maintenance, OS and application upgrades, or machines being re-purposed. For administrative reasons, cluster operators typically split clusters into *fault domains*, i.e.,

System	[R1] Expressive constraints between containers					[R2] High-level constraints	[R3] Global objectives	[R4] Low-latency container allocation
	affinity	anti-affinity	cardinality	intra	inter			
YARN [52]	✧	–	–	✧	–	–	–	✓
Slider [10]	✧	✧	–	✧	–	–	–	–
Borg [53]	✧	✧	–	✧	✧	–	✻	✓
Kubernetes [53]	✓	✓	–	✓	✓	✓	✻	✓
Mesos [27]	✧	–	–	✧	–	–	–	–
Marathon [39]	✓	✓	✓	✓	–	–	–	–
Aurora [2]	✧	✓	✓	✓	–	–	–	–
TetriSched [50]	✧	✧	✧	✓	–	–	✻	✓
Medea	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Support for LRA requirements R1–R4 in existing schedulers (✧ indicates implicit support for constraints through static machine attributes and not by declaring explicit dependencies between containers; ✻ indicates a partially supported feature.)

machines with a higher likelihood of joint failure (e.g., racks), and *upgrade domains*, i.e., machines that are scheduled to be upgraded together. In some of Microsoft's production clusters, node groups called "service units" account for both upgrades and failures.

Figure 3 shows the percentage of machine unavailability in one of Microsoft's clusters with tens of thousands of machines (total) over four days. We also show the same percentage within four random service units of the cluster (SU1–SU4), comprising a couple of thousand machines each. We observe that: (i) unavailability in a service unit is usually below 3% but can spike to 25% or even 100%; (ii) there is a strong correlation of unavailability within a service unit; and (iii) service units tend to fail asynchronously (e.g., when SU1 is 100% unavailable, total is only 8%).

With a random placement, an application may lose multiple containers at once, which can in turn impact its recovery time or performance. Such a placement scheme hurts LRAs in particular because their containers are by definition long-lived and thus the failure probability increases over time. Hence, application owners must be able to spread containers across fault and upgrade domains (*anti-affinity* constraints).

However, it should not be required to explicitly refer to specific node domains when requesting containers: (i) these domains change over time, e.g., with node addition/removal; and (ii) it is cumbersome to enumerate all domains of a cluster with thousands of machines. In cloud environments, it may even not be feasible—the operator may not reveal the cluster configuration for security and business reasons.

2.4 Global cluster objectives

Cluster operators must also specify constraints that guarantee the smooth operation of the cluster. Besides local constraints, such as restricting the number of network-intensive containers per node, the scheduling of LRAs must meet *global objectives*:

- **Minimize constraint violations.** Satisfying the placement constraints of all applications may not be possible, especially in a heavily loaded cluster. In this case, the number and extent[3] of constraint violations should be minimized.
- **Minimize resource fragmentation.** It is undesirable to leave too few free resources on a node, as they might remain unutilized.
- **Balance node load.** By balancing the node load, applications can expand their allocated resources on a given node and better accommodate load spikes.
- **Minimize number of machines used.** A low number reduces the operating cost of a cluster in a cloud environment.

Note that some objectives may be conflicting, e.g., minimizing constraint violations and load imbalance, while others may be irrelevant in a specific scenario, e.g., minimizing the number of machines for an on-premises cluster. The cluster operator should be able to determine the objectives to be used and their relative importance. Supporting such global objectives should not affect the scheduling of traditional batch jobs, which are more sensitive to container allocation latencies due to their shorter container runtimes.

2.5 Scheduling Requirements

Summing up our observations from §2.2–§2.4, we list the requirements for the effective scheduling of LRAs:

[R1] Expressive placement constraints: We must support intra- and inter-application (anti-)affinity and cardinality to express dependencies between containers during placement.

[R2] High-level constraints: The constraints must be *high-level*, i.e., agnostic of the cluster organization, and capable of referring to both current and future LRA containers.

[R3] Global objectives: We must meet global optimization objectives imposed by the cluster operator.

[R4] Scheduling latency: Supporting LRAs, which can tolerate higher scheduling latencies, must not affect the scheduling latency for containers of task-based applications.

Table 1 highlights the support of existing schedulers for our requirements. Most schedulers support (only intra-application and non-high-level) constraints *implicitly* through machine attributes (e.g., place two containers on a node with a given hostname or on machines with GPUs). Only Kubernetes supports explicit intra- and inter-application high-level constraints between containers, but not cardinality, and lacks full support for global objectives by considering only one container request at a time. More details are provided in §8.

[3]Consider a constraint to place no more than 5 HBase containers on a rack. Placing instead 10 containers is a more extensive violation than placing 6.

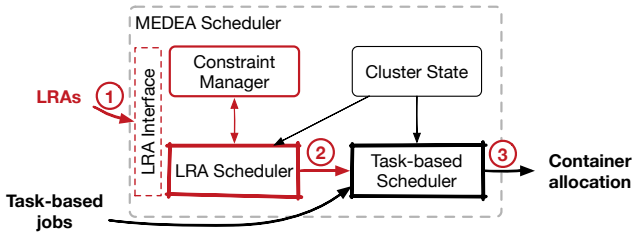


Figure 4: MEDEA scheduler design

3 MEDEA DESIGN

MEDEA supports the scheduling of both LRAs and "traditional" applications with short-running containers (referred to as *task-based* jobs). As shown in Figure 4, MEDEA uses a two-scheduler design for placing containers: (i) a dedicated *LRA scheduler* places long-running containers, accounting for constraints stored in the *constraint manager* component; and (ii) a *task-based scheduler* places task-based jobs. Next we discuss the key components of our design.

LRA interface. When an application owner submits a request to MEDEA, similar to other schedulers, they specify the required containers (e.g., "10 containers with 2 CPUs and 4 GB RAM each"). These resource demands, along with some simple constraints (e.g., data locality) are sufficient for task-based jobs. For LRAs, MEDEA introduces a rich API for expressive placement constraints that capture interactions between containers, satisfying requirements **R1**–**R2** (§2.5). Applications that use the constraints API are handled by the LRA scheduler, while the ones using the simpler container request API are handled by the task-based scheduler. A detailed description of our constraints is given in §4.

LRA scheduler. The LRA scheduler uses an online optimization-based algorithm that, given the current cluster condition, including already running LRAs and task-based jobs, determines the efficient placement of newly submitted LRAs. The scheduler is invoked at regular configurable intervals to place all LRAs submitted during the latest interval. Our scheduling algorithm, described in §5, takes into account multiple LRA container requests at once to satisfy their placement constraints and attain global optimization objectives (requirement **R3**).

Task-based scheduler. MEDEA's two-scheduler design removes the burden of handling complex placement decisions from the task-based scheduler, allowing the scheduling latency for task-based jobs to remain low (requirement **R4**). Moreover, this design allows the reuse of existing production-hardened task-based schedulers. This minimizes the changes required in the existing scheduling infrastructure, and is crucial for adopting MEDEA in production.

Constraint manager. We introduce a new central component in which all constraints—both from application owners and cluster operators—are stored. This allows MEDEA to have a global view of all active constraints and to easily add or remove constraints.

MEDEA operates as follows: it passes placement decisions made by the LRA scheduler (step 1 in Figure 4) to the task-based scheduler (step 2), which then performs the actual resource allocation (step 3). This approach avoids the challenge of conflicting placements, faced by existing multi-level [27, 45] and distributed [14, 30, 42, 44] schedulers: in these designs, different schedulers, operating on the same cluster state, may arrive at conflicting decisions, whereas in MEDEA the actual allocations are performed by a single scheduler. We further discuss placement conflicts in §5.4.

Performing all allocations through the task-based scheduler also allows to achieve (weighted) fairness and respect application priorities across both LRAs and task-based jobs.

4 DEFINING PLACEMENT CONSTRAINTS

In this section, we first introduce the notions of container tags and node groups (§4.1), which will allow us to present the syntax and semantics of our constraints (§4.2). We finish with a discussion of our constraint model (§4.3).

4.1 Tag model and node groups

Container tags. Unlike existing schedulers that attach attributes only to *machines* (see §8), MEDEA allows application owners to attach *tags* to *containers*.

Example: An HBase Master container can have the following tags: `appID:0023`[4] to denote the ID of the LRA; `hb` for the application type (HBase); `hb_m` for the container's role (master); and `memory_critical` as a resource specification.

Each container request r of an LRA submitted to MEDEA is associated with a set of tags $\mathcal{T}_r$. Tags are a simple yet powerful mechanism for constraints to refer to containers of the same or different, possibly not yet deployed, applications. For example, a constraint can use tag `hb` to refer to a current or future container of an HBase application.

Tag sets. We define the *node tag set* $\mathcal{T}_n$ to be the union of tags of the containers running on node n at a given moment. The node tag set is *dynamic*: when a container is allocated on node n, its tags are added to $\mathcal{T}_n$; when the container finishes execution, the associated tags are removed.

Given that each tag in $\mathcal{T}_n$ can be associated with multiple containers on node n, we define the *tag cardinality* function $\gamma_n : \mathcal{T}_n \rightarrow \mathbb{N}$, where $\gamma_n(t)$ is the number of occurrences of a tag $t \in \mathcal{T}_n$ on node n.

Example: Consider two HBase containers deployed on a node n_1: one master with tags {`hb`, `hb_m`} and one region server with tags {`hb`, `hb_rs`}. Then, $\mathcal{T}_{n1}$={`hb`, `hb_m`, `hb_rs`}, with γ_{n1}(`hb`)=2 and γ_{n1}(`hb_m`)=γ_{n1}(`hb_rs`)=1.

Note that a subset of a node tag set can also be defined statically, e.g., to identify nodes with specific hardware capabilities, such as GPUs or FPGAs. Therefore our tag model can also express the static machine attributes offered by existing schedulers (see §8).

Analogously, we define the *tag set* $\mathcal{T}_S$ of a *set of nodes* $\mathcal{S}$, such as a data center rack, to be the union of tag sets of the nodes that are part of S.

Example: Let nodes n_1 (from the above example) and n_2 belong to rack r_1, and assume $\mathcal{T}_{n2}$={`hb`, `hb_rs`} with γ_{n2}(`hb`)=γ_{n2}(`hb_rs`)=1.

[4] To avoid naming conflicts, namespaces can be used in tags, such as the predefined `appID` that defines the namespace of application IDs.

Then, $\mathcal{T}_{r1}$={hb, hb_m, hb_rs}, with γ_{r1}(hb)=3, γ_{r1}(hb_m)=1, and γ_{r1}(hb_rs)=2.

Node groups. Cluster operators can register *node groups*, which specify logical, possibly overlapping, categories of node sets. The simplest predefined node groups are node and rack. A node set belonging to the node group node includes a single element that corresponds to a cluster node; a rack node set contains all nodes of a physical rack. Other node group examples are fault and upgrade domains (see §2.3).

Node groups allow constraints to be expressed independently of the cluster's underlying organization. They thus allow operators to not reveal their cluster infrastructure. For example, a constraint requiring to "place hb containers of the same application in different upgrade domains" does not need to be aware of the cluster's upgrade domains or perform any actions when upgrade domains change. Node groups also allow for more succinct constraint definitions—in their absence, the above constraint would have to enumerate all possible node combinations through a disjunction.

Together with tags, node groups play a key role in enabling high-level constraints (requirement **R2** from §2.5).

4.2 Placement constraints

MEDEA allows application owners and cluster operators to specify *placement constraints* using *tags* to refer to containers in the same or different LRAs, and *node groups* to target specific node sets. In particular, it supports constraints of the following form:

$$C = \{\texttt{subject\_tag}, \texttt{tag\_constraint}, \texttt{node\_group}\}$$

where subject_tag is a tag (or conjunction of tags) that identifies the containers subject to the constraint; tag_constraint is a constraint of the form {c_tag, c_{min}, c_{max}} where c_tag is a container tag (or conjunction of tags), and c_{min}, c_{max} are positive integers; and node_group is a node group. The tag_constraint can also be a boolean expression of multiple tag constraints (we do not support negation yet).

The semantics of a constraint C is that *each* of the containers with subject_tag should be placed on a node belonging to a node set $\mathcal{S}$ ∈ node_group, such that $c_{min} \leq \gamma_{\mathcal{S}}(\texttt{c\_tag}) \leq c_{max}$ holds for the tag cardinality function of $\mathcal{S}$ (see §4.1).

This constraint type is sufficient to express the wide range of intra- and inter-application LRA constraints discussed in §2 (requirement **R1**):

(i) with $c_{min} = 1$ and $c_{max} = \infty$, we can express *affinity constraints*;

Example: Constraint C_{af} = {storm, {hb ∧ mem, 1, ∞}, node} requests each container with tag storm to be placed in the same node with at least one container with tags hb and mem. If we want to restrict the constraint to a specific application[5] with ID 0023, we would use $C_{af'}$ = {appID:0023 ∧ storm, {appID:0023 ∧ hb ∧ mem, 1, ∞}, node}.

(ii) with $c_{min} = 0$ and $c_{max} = 0$, we can express *anti-affinity constraints*;

Example: C_{aa} = {storm, {hb, 0, 0}, upgrade_domain} requests each storm container to be placed in a different upgrade domain from all hb containers.

(iii) other values for c_{min} and c_{max} allow us to express generic *cardinality constraints*.

Example: C_{ca} = {storm, {spark, 0, 5}, rack} requests each storm container to be placed in a rack that has no more than five spark containers deployed.

If the subject_tag and tag_constraint use the same tags, we specify constraints within a group of containers.

Example: C_{cg} = {spark, {spark, 3, 10}, rack} can be used by the cluster operator to allow no fewer than three and no more than five Spark containers in a rack.

Constraint dependencies. With the use of tag sets, which depend on already deployed containers and that do not have to be statically predefined, both intra- and inter-application constraints can be expressed. Note, however, that if a constraint must target a specific container of another deployed application, its application and container ID are required. To address this problem, application owners can submit the two applications together. Otherwise, a service that exposes the deployed applications and their containers can be used.

Compound constraints. Multiple constraints can be combined with boolean operators. Such constraints are specified in disjunctive normal form (DNF), allowing any combination of constraints.

Soft constraints and weights. Each constraint can be associated with a weight. An application can use different weights to determine the relative importance of its constraints, e.g., to request node or rack affinity, with a preference for the former. By default the constraints in MEDEA are soft, i.e., the scheduler will try to satisfy the constraint but not deny placement in case of constraint violations (see §5). In our encountered practical scenarios, soft constraints capture better the expected behavior by users; MEDEA can emulate hard constraints through the use of weight values.

4.3 Discussion

One limitation of the constraint model is that we cannot express constraints of the form "spread all spark containers across 3 racks", i.e., impose cardinality on the number of node sets rather than the number of containers per node set. Given that we can indirectly achieve similar placements using the cardinality constraints, we decided to not complicate our syntax to support these constraints.

MEDEA currently only allows constraints on LRAs. We discuss in §5.4 how we could allow task-based jobs to express constraints targeting LRAs, e.g., to have a map/reduce job be placed on the same rack as a Memcached application.

Note also that in this work we focus on building the scheduling infrastructure that enables placement constraints. Automatically inferring the most appropriate constraints for each application and cluster is the focus of future work.

[5]For convenience, we automatically attach some predefined tags to each container, e.g., the ID of the LRA that it belongs to.

Symbol	Description
k	Number of LRAs to be placed
N	Number of cluster nodes
T_i	Number of containers of LRA i
R_n^f, R_n^u	Free, used resources of node n[6]
m	Total number of constraints
w_i	Weights of components in objective function
B_n, D_n	Sufficiently large integers, used in inequalities
S_i	1 if all containers of LRA i are placed; 0 otherwise
X_{ijn}	1 if container j of LRA i placed at node n; 0 otherwise
r_{ij}	Resource demand of container j of LRA i
r_{min}	Minimum resource demand
$c_{min}^{l,v}$, $c_{max}^{l,v}$	Violation of cardinalities c_{min}, c_{max} for constraint C_l
v_c^l	Violation for constraint C_l
z_n	1 if free resources $\geq r_{min}$ after placement; 0 otherwise

Table 2: Notation used in ILP formulation (constants appear above the dashed line, variables below)

5 SCHEDULING LRAS

In this section, we give an overview of our scheduling approach (§5.1), present ILP-based (§5.2) and heuristic-based (§5.3) LRA scheduling algorithms, and discuss further potential improvements (§5.4).

5.1 Overview

When the LRA scheduler is invoked, it considers the following information: (i) the container requests and placement constraints of the newly submitted LRAs; (ii) the constraints of already deployed LRAs and the cluster operator constraints via the constraint manager (see Figure 4); and (iii) the available resources at each node. It then determines the cluster node at which each LRA container should be placed.

When determining the LRA container placement, our scheduling algorithm attempts to: (i) place all containers of an LRA; (ii) satisfy the placement constraints of the newly submitted LRAs, of the previously deployed ones, and of the cluster operator; (iii) respect resource capacities of all nodes; and (iv) optimize for global cluster objectives (see §2.4).

As mentioned in §3, the LRA scheduler is invoked at regular intervals, configured by the cluster operator. Longer intervals increase the scheduling latency for LRAs, but allow multiple LRAs to be considered together, improving placement quality. We experimentally study this trade-off in §7.

5.2 ILP-based scheduling algorithm

LRA placement is an optimization problem under a set of constraints, and can thus be expressed as an integer linear programming (ILP) problem. For completeness, Figure 5 provides the ILP formulation, relying on the notation from Table 2; below, we give an intuitive description.

Consider k LRAs that are submitted in the latest scheduling interval and must be scheduled on an N-node cluster.

[6]For simplicity, we use a single scalar value for the cluster resources. However, our model can be extended to use a vector of resources instead, having separate equations for each resource type.

$$\text{maximize } \frac{w_1}{k}\sum_{i=1}^{k} S_i + \frac{w_2}{m}\sum_{l=1}^{m} v_c^l + \frac{w_3}{N}\sum_{n=1}^{N} z_n \tag{1}$$

subject to:

$$\forall i, j: \sum_{n=1}^{N} X_{ijn} \leq 1 \tag{2}$$

$$\forall n: \sum_{i=1}^{k}\sum_{j=1}^{T_i} r_{ij} \cdot X_{ijn} \leq R_n^f \tag{3}$$

$$\forall i: \sum_{n=1}^{N}\sum_{j=1}^{T_i} X_{ijn} - T_i S_i = 0 \tag{4}$$

$$\forall n: \sum_{i=1}^{k}\sum_{j=1}^{T_i} r_{ij} \cdot X_{ijn} - B_n(1 - z_n) \leq R_n^f - r_{min} \tag{5}$$

For each constraint $C_l = \{\texttt{s\_tag}, \{\texttt{c\_tag}, c_{min}^l, c_{max}^l\}, \texttt{G}\}$,
$\forall$ container $t_{i_s j_s} \in \texttt{s\_tag}$, $\forall$ node set $\mathcal{S} \in \texttt{G}$:

$$\sum_{n\in\mathcal{S}}\left(\sum_{\substack{i,j:\texttt{tag}\in t_{ij}\\ t_{ij}\neq t_{i_s j_s}}} X_{ijn} + D_n(1 - X_{i_s j_s n})\right) - c_{min}^l + c_{min}^{l,v} \geq 0 \tag{6}$$

$$\sum_{n\in\mathcal{S}}\left(\sum_{\substack{i,j:\texttt{tag}\in t_{ij}\\ t_{ij}\neq t_{i_s j_s}}} X_{ijn} - D_n(1 - X_{i_s j_s n})\right) - c_{max}^l - c_{max}^{l,v} \geq 0 \tag{7}$$

$$v_c^l = \frac{c_{min}^{l,v}}{c_{min}^l} + \frac{c_{max}^{l,v}}{c_{max}^l} \tag{8}$$

Figure 5: ILP formulation

Objective. Our objective function (Equation 1) has three components: (1) it aims to place as many of the k LRAs as possible; (2) it minimizes the number of constraint violations (more on this below); and (3) it avoids resource fragmentation by minimizing the number of nodes left with few resources (Equation 5).

Note that in order to combine these components linearly, independently of their range or units, we normalize each component to take values from 0 to 1. We also use weights (w_1–w_3) to assign different priorities to components. The cluster administrator is responsible for setting these weights based on the desired cluster behavior. In Equation 1, we include the components that we use in our clusters, but additional ones can be easily added, such as load imbalance or minimizing the number of nodes used for placement.

We also make sure that each container is placed at most once, respecting node capacities, and that we place either all or none of an LRA's containers (Equations 2 to 4).

Placement constraints and violations. For each placement constraint that belongs to a newly submitted or already deployed LRA or to the cluster operator, we use two inequalities to impose the constraint semantics (see §4.2): one for the minimum c_{min} and one for the maximum c_{max} cardinality (Equations 6 to 7). In case of a constraint violation, which is more common in heavily utilized clusters or when dealing with very restrictive constraints, we quantify the extend of the violation relative to the values of c_{min} and c_{max} using Equation 8.

Resolution of constraint conflicts. The set of constraints considered when placing an LRA may include conflicts, e.g., one constraint requesting no more than three Spark containers in a rack, and another one at least four. In such cases, cluster operator constraints override the application constraints, as long as they are

more restrictive. In case of conflicting application constraints, our ILP formulation favors the placement that minimizes the number of violations.

Compound constraints. To support compound constraints expressed in DNF (§4.2), MEDEA treats each DNF conjunct as a separate constraint, and adds an extra inequality to guarantee that at least one of these constraints is met.

5.3 Heuristic-based scheduling

To examine whether simpler approaches are sufficient to achieve high quality LRA placements, we experiment with various heuristics. Next we discuss the ones that gave us best results in practice; an experimental comparison with the ILP algorithm is given in §7.

Tag popularity prioritizes the placement of containers that are associated with tags appearing in the largest number of constraints. The intuition is that such containers often have complex constraints and are thus harder to place.

Node candidates heuristic first calculates the number of nodes N_c on which a given container is allowed to be placed, subject to all placement constraints. It then places the container with the smallest N_c, as that container has the least placement flexibility. Note that N_c values need to be recalculated after each container placement. We avoid this by recalculating N_c only for containers whose placement opportunities were affected in the previous iteration.

5.4 Discussion

Placement conflicts. Given MEDEA's two-scheduler design (§3), the cluster state may have changed from the moment an LRA is submitted until its containers are allocated, due to allocations for task-based jobs. Possible solutions to this problem are: (i) kill containers of task-based jobs to free up resources for LRAs, (ii) relax some LRA constraints, e.g., by placing an LRA container on the corresponding rack instead of the specified node; (iii) allow LRAs to reserve cluster resources with constraints in advance. Given that these approaches could significantly impact the execution of task-based jobs and increase the scheduler's complexity, in the current version of MEDEA, we opted for the simpler approach of resubmitting the LRA in case of conflict.

Constraints for task-based jobs. Our focus in this paper is on placement constraints for LRAs. However, task-based jobs might also require placement constraints among them or towards LRAs (see also §4.3). We can address this by extending the task-based scheduler to support constraints in a heuristic fashion, in order to not affect the scheduling latency of task-based jobs nor overload the LRA scheduler.

Container migration. Under high cluster load, when LRAs enter and leave the system at high rates or when their resource demands change over time, it might be beneficial to combine MEDEA's proactive approach in achieving high quality LRA placements with reactive approaches, such as container migration. As part of our future work, we can extend our ILP formulation to enable migration, also accounting for migration cost in our objective function.

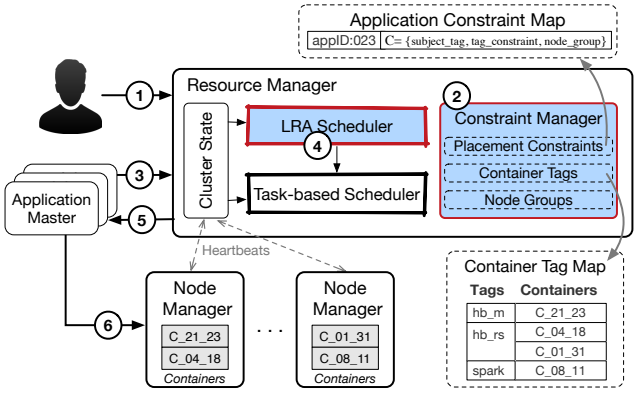


Figure 6: MEDEA architecture

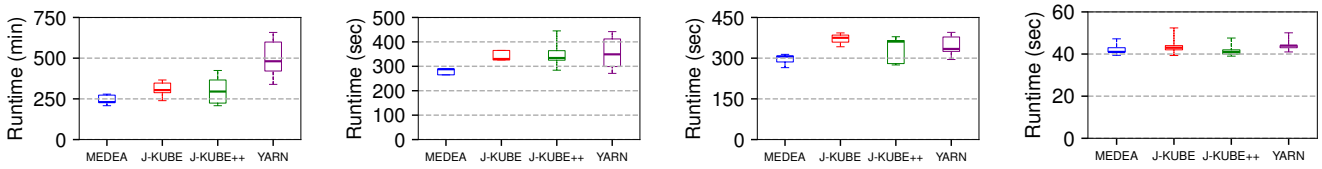
6 IMPLEMENTATION

Our implementation of MEDEA is built as an extension to Apache Hadoop YARN [4], following the design from §3. Figure 6 shows (in blue) the new components added to YARN's *resource manager* (RM), namely the *constraint manager* (CM) and the *LRA scheduler*. The CM is responsible for storing container tags and node groups (§4.1), along with application-specific and cluster-wide placement constraints (§4.2). The LRA scheduler determines the placement of LRA containers to nodes, given the placement constraints and the cluster condition. To this end, it uses either the ILP-based (relying on the CPLEX solver [16]) or the heuristic-based scheduling algorithms (§5). For the task-based scheduler that handles applications with no placement constraints, we use YARN's Capacity Scheduler [5], but Fair Scheduler [6] can be used instead, simply by changing a configuration parameter.

LRA life-cycle. Figure 6 also shows the life-cycle of an LRA in MEDEA: (1) the client submits an LRA to the cluster, including a set of tags for its containers and a set of placement constraints; (2) when the RM receives the LRA, the CM validates and stores the associated constraints, and then the job-specific application manager (AM) is initialized; (3) the AM petitions the RM for cluster resources based on the resource requirements of its containers; (4) the LRA scheduler takes into account the relevant constraints, the cluster's node groups and the current container tags stored in the CM, as well as the available resources in the cluster, and finds the best placement for the LRA containers, based on our objective function (see §5.2). This placement is passed to the task-based scheduler that performs the allocation (see §3); (5) the RM notifies the AM; (6) the AM dispatches the containers for execution to the node managers.

7 EVALUATION

We now experimentally validate how MEDEA meets our requirements (§2.5). After describing our setup (§7.1), we study: (i) the benefit of MEDEA on LRA performance (§7.2) and resilience (§7.2); (ii) the achieved global objectives (§7.4); and (iii) the scheduling latency (§7.5).



(a) TensorFlow **(b)** HBase insert **(c)** HBase workloadA **(d)** GridMix workload

Figure 7: Application performance (lower is better)

7.1 Experimental setup

Cluster setup. To evaluate MEDEA on a real deployment, we use a pre-production cluster of 400 machines grouped into 10 racks. Each machine has a dual quad-core Intel Xeon E5-2660 CPU with HT, 128 GB of RAM, and ten 3 TB data drives, configured as a JBOD. The network supports 10 Gbps within and 6 Gbps across racks.

Simulation. To experiment with multiple configurations, we use a simulator that executes MEDEA with simulated machines, merely ignoring RPCs and task execution. To drive the simulations, we extend the synthetic workload generator GridMix [24] to produce LRAs with custom constraints, along with short-running batch Tez [12] jobs.

Workload. To emulate a real shared cluster workload, we use the following applications:

- HBase [7] instances, each with ten workers. We use the YCSB benchmark [15] with 1 TB of data (one billion records), and submit six YCSB workloads, A–F, using one YCSB client per HBase instance;
- TensorFlow [1] instances, each with eight workers and two parameter servers. We run machine learning workloads that involve one million iterations; and
- Batch Tez [12] jobs, generated using the GridMix workload generator [24], resembling some of our production workloads, similar to the ones used in Yaq [44].

To deploy the above applications, we use <2 GB, 1 CPU> containers for the HBase and TensorFlow workers, <4 GB, 1 CPU> containers for the TensorFlow chiefworkers, and <1 GB, 1 CPU> containers for the rest.

Placement constraints. Unless otherwise specified, we use the following placement constraints when deploying the HBase and TensorFlow instances: (i) to minimize network traffic, we use the intra-application affinity constraint that all workers of the same HBase or TensorFlow instance should be on the same rack; (ii) to minimize resource interference, we impose the inter-application cardinality constraint that no more than two (four) HBase (TensorFlow) workers are placed on the same node;[7] and (iii) for HBase, we also request node affinity between the Master and the Thrift Server, and node anti-affinity between the Master and Secondary.

Comparisons. We compare the following systems:

- MEDEA-ILP (or MEDEA): This is our ILP-based algorithm (§5.2). For the objective function (Equation 1), we use weights w_1=1, w_2=0.5 and w_3=0.25, i.e., we give higher priority to maximizing the number of scheduled LRAs and then to minimizing constraint violations and resource fragmentation. We use a scheduling interval of 10 sec.
- MEDEA-NC and MEDEA-TP: These are our heuristic-based algorithms (§5.3). We also use a 10 sec scheduling interval.
- SERIAL: This is another heuristic-based algorithm that, unlike MEDEA-NC and MEDEA-TP, does not order the container requests within a scheduling interval.
- YARN: We use YARN as a production-ready constraint-unaware scheduler, as MEDEA is also built on YARN.
- J-KUBE: Kubernetes [33] is the most complete system to date for supporting placement constraints (see Table 1 and §8). As it follows a different architecture, to have a fair comparison, we implement its scheduling algorithm in MEDEA's LRA scheduler. Kubernetes considers one container request at a time during scheduling, and supports (anti-)affinity but no cardinality constraints.
- J-KUBE++: This is the same as J-KUBE after extending it to support cardinality constraints.

Our implementation is based on Apache Hadoop 2.7.2 [4], and we use Apache Slider 0.92.0 [10] to deploy LRAs.

[7] The maximum cardinality used for each application was decided empirically after experimenting with different values. See also the discussion in §4.3 on automatically inferring constraints.

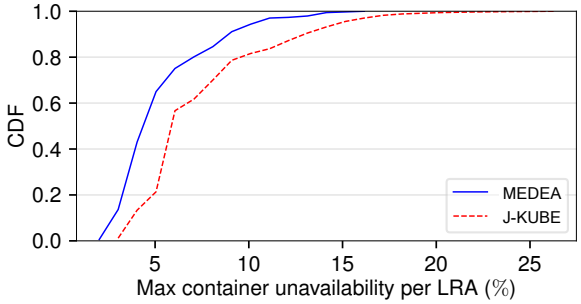
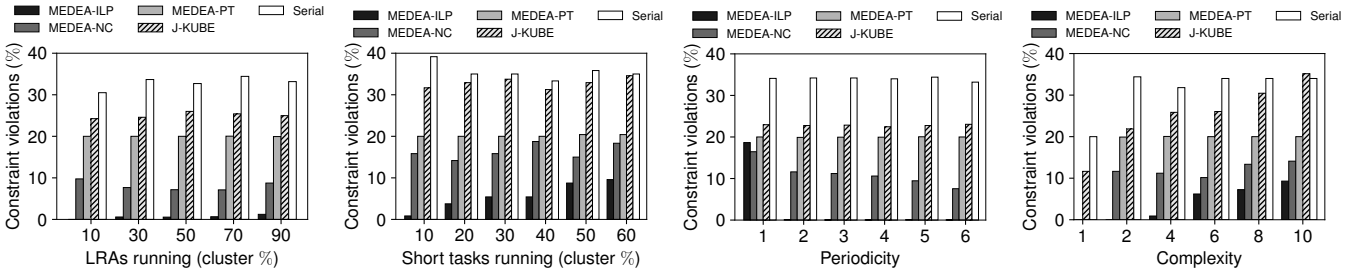


Figure 8: Application resilience over 15 days

7.2 Application performance

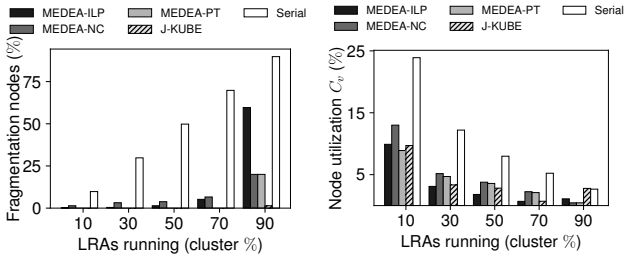
To study the impact of MEDEA on application performance in a real environment, we use the 400-node pre-production cluster described above and deploy 45 TensorFlow and 50 HBase instances. We also submit GridMix jobs that account for 50% of the cluster's memory.

Figure 7a illustrates the runtimes for our machine learning workflows on TensorFlow; Figure 7b and Figure 7c show the runtimes for data insertion and Workload A on HBase, respectively. We use box plots in which the lower/upper part of the box represents the



(a) Varying cluster LRA utilization **(b)** Varying task-based utilization **(c)** Varying scheduling intervals **(d)** Varying constraint complexity

Figure 9: Constraint violations



(a) Nodes with resource fragmentation **(b)** Coefficient of variation for node memory utilization

Figure 10: Load balance with varying LRA utilization

25th and 75th percentiles, respectively; the middle line is the median; and the whiskers are the 5th and 99th percentiles, respectively.

Medea consistently outperforms J-Kube across all percentiles due to J-Kube's lack of support for cardinality constraints—anti-affinity constraints alone are not sufficient. In particular, median runtime is 32% longer on J-Kube for TensorFlow and 23% longer for HBase Workload A.

We observe similar improvements of up to 28% when comparing Medea with J-Kube++. Importantly, J-Kube++ leads to significant variability in runtimes, which is detrimental in production clusters [29]: while the 5th percentile is similar, the 99th percentile is up to 54% higher than Medea. By considering only one container request at a time, J-Kube++ often leads to placements with many constraint violations (see §7.4), which are also not consistent across application instances. For instances that happen to have many violations, the benefit of Medea is higher.

Medea's benefits are even more pronounced when compared with YARN that does not support constraints: median runtime is up to 2.1× shorter and the 99th percentile up to 2.4×. YARN also leads to high runtime unpredictability, as some constraints are randomly satisfied for some LRAs.

Finally, Figure 7d shows that Medea's benefits do not come at the expense of task-based jobs, whose runtimes are consistently similar across all schedulers.

Overall, compared to state-of-the-art schedulers, Medea significantly improves LRA performance and predictability, without affecting the performance of task-based jobs.

7.3 Application resilience

To assess the effectiveness of Medea on improving application resilience, we use unavailability data from one of our production clusters, which comprises a few tens of thousands of machines grouped into 25 service units (see §2.3).

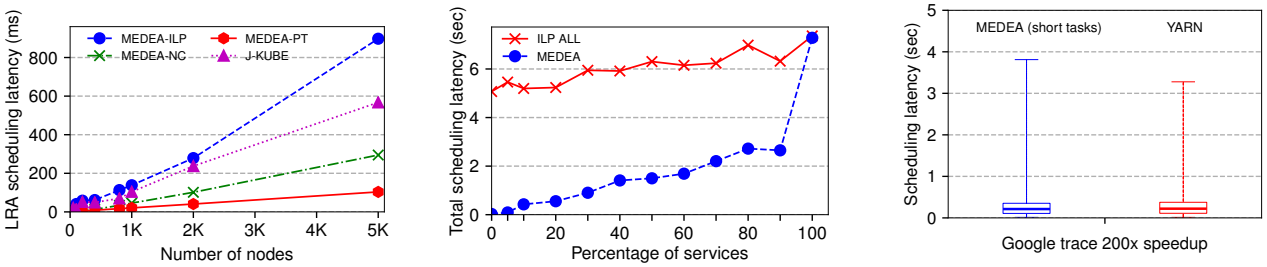
We collect the number of unavailable machines per service unit (due to failures, machine upgrades, maintenance, etc.) for each hour over a period of 15 days. Then we generate LRAs, with 100 containers each, and place them with the intra-application anti-affinity constraint that containers of the same LRA should be spread across service units, using Medea and J-Kube. Given the machine unavailability and the container placements achieved, we pick for each hour the LRA with the highest percentage of unavailable containers.

Medea's placement leads to fewer anti-affinity constraint violations compared to J-Kube. As shown in Figure 8, this leads to lower container unavailability across all percentiles. It improves the median and maximum unavailability percentage by 16% and 24%, respectively, which is crucial for a production environment.

7.4 Global cluster objectives

We now focus on Medea's ability to achieve global objectives (see §2.4 and §5). We use a simulated cluster of 500 machines (8 CPU cores and 16 GB RAM each) and 10 racks. We generate HBase instances using the constraints mentioned in §7.1. We conduct four experiments and observe for each scheduling algorithm: (i) the percentage of containers that violate constraints; (ii) the percentage of nodes that are fragmented (i.e., have less than 1 core/2 GB RAM and are not fully utilized); and (iii) the coefficient of variation of nodes' memory utilization (as a proxy for load imbalance).

First, we vary the number of submitted LRAs, for a cluster memory utilization of 10% to 90%. Our results are shown in Figs. 9a, 10a, and 10b. The scheduling interval for our Medea algorithms (i.e., Medea-ILP, -NC, -TP) is such that two LRAs are considered at every scheduling cycle. Medea-ILP, thanks to its objective function and consideration of multiple container requests at a time, leads to almost no constraint violations, even for 90% utilization (Figure 9a). In contrast, our heuristic algorithms result in 10%–20% violations, even for low utilization, despite also considering multiple requests. J-Kube, handling one request at a time, exhibits even more violations. Note that for all schedulers, constraint violations do not significantly increase with utilization, as most of our constraints in this experiment are intra-application.



(a) Varying cluster size **(b)** Two-scheduler benefit **(c)** Task-based scheduling latency

Figure 11: Scheduling latency

All MEDEA algorithms, as well as J-KUBE, lead to few fragmented nodes except for high utilizations (Figure 10a). Moreover, all algorithms, apart from SERIAL, perform similarly in terms of load imbalance (Figure 10b). Notice that load imbalance is more pronounced for low utilizations, but the load evens out for higher utilized clusters.

Next, we use LRAs with a stable utilization of 10% and task-based jobs with varying utilization from 10% to 60%. We observe similar trends for constraint violations as above: MEDEA-ILP yields less than 10% violations, while the other algorithms lead to more than 15% and up to 40% violations.

We then vary the scheduling interval with 10% LRA utilization, affecting the number of LRAs considered at each scheduler invocation (periodicity). With periodicity 1, MEDEA-ILP also exhibits violations (Figure 9c). Increasing periodicity reduces violations for MEDEA-ILP and -NC. This observation highlights the importance of considering multiple container requests at a time for satisfying inter-application constraints.

Finally, Figure 9d reports violations for constraints of varying complexity (complexity X means that we have affinity or cardinality inter-application constraints involving up to X LRAs). Even with constraints involving 10 LRAs, MEDEA-ILP results in less than 10% violations; MEDEA-NC and -TP also perform relatively well with less than <20% violations. In contrast, J-KUBE has more than 20% violations; considering only one request at a time makes it difficult to satisfy inter-application constraints.

7.5 Scheduling latency

We investigate the scheduling latency of different LRA scheduling algorithms, while increasing the simulated cluster size from 50 to 5000 machines. Each time, we generate LRAs to consume 20% of the cluster resources.

Figure 11a shows the average scheduling latency for placing all containers of an LRA. Our heuristic algorithms achieve the lowest latencies, with MEDEA-NC being more expensive. J-KUBE leads to higher latencies, due to the frequent scoring of nodes, however, we believe we can further optimize our implementation via smart caching of node scores. Although MEDEA-ILP has the highest latency, even with 5000 machines, the average latency is 850 ms, which is low compared to the typical execution times of LRAs (hours, days or months). For larger clusters, MEDEA-NC and J-KUBE can be reasonable compromises of latency and quality. In practical scenarios, MEDEA-ILP is the better choice, as it combines a relatively low latency with better placement quality.

To assess the benefit of our two-scheduler design, we compare MEDEA-ILP with a modified version that uses the solver for both long- and short-running containers (ILP-ALL). We simulate a 256-machine cluster, and generate LRAs and task-based jobs, resulting in a fully utilized cluster.

Figure 11b shows the total scheduling latency for LRAs, as we vary the fraction of cluster resources used by LRAs. The single-scheduler design results in an increased scheduling latency (e.g., 9.5× worse latency for 20% LRA utilization). This justifies the rationale behind MEDEA's use of the more expensive ILP solver to schedule only LRAs.

Finally, we study the impact of MEDEA on scheduling short-running containers, using the Google cluster trace [54], which we speed up by a factor of 200×. In Figure 11c, we report the scheduling latency achieved by MEDEA-ILP and YARN when placing the trace tasks (we use box plots as in Figure 7). For MEDEA, we add an extra 10% scheduling load coming from LRAs. Despite the additional LRA load, MEDEA achieves scheduling latencies similar to those of YARN, which shows that MEDEA's LRA scheduler does not impact the operation of the underlying task-based scheduler.

8 RELATED WORK

An overview of the most related cluster schedulers was given in Table 1 and §2.5. Below we provide a detailed analysis.

Schedulers with LRA support. YARN [52] was initially designed for task-based jobs and currently supports only affinity to specific nodes/racks [31]. Slider [10] and ongoing extension efforts [46] enable LRAs in YARN. Our work on MEDEA, which will be included in the upcoming 3.1 release of Apache Hadoop [43], adds support to YARN for expressive placement constraints.

Mesos [27] follows a push-based model offering resources to "frameworks". Aurora [2] and Marathon [39] add support for LRAs to Mesos. They enable simple, only intra-application, (anti-)affinity and cardinality constraints (e.g., limit number of containers per node or rack), but with no support for container tags (§4.1). Being external to Mesos, they cannot optimize for global cluster objectives. In principle, native support for constraints could be added to Mesos, following a design similar to MEDEA, but given Mesos' offer-based model, it would be harder to do this efficiently.

Google's Borg [53] schedules both LRAs and task-based jobs, and supports scoring of nodes during scheduling, which can emulate a restricted version of our global objectives. Only support for affinity to machines with specific attributes is mentioned. Kubernetes [33] uses similar node scoring, accounting for constraints and load balance. Unlike our ILP-based algorithm, it considers only one container request at a time, which leads to significant constraint violations (see §7.4). There is no scheduler dedicated to LRAs, although a user could plug one similar to ours. Kubernetes is the only scheduler other than Medea that exploits container tags, called pod labels, with the feature being currently listed as beta [34], but does not support cardinality constraints.

Prophet [56] improves LRA scheduling based on historical data, but requires jobs to be recurring.

Constraints in general-purpose schedulers. TetriSched [50] supports various placement *and* time constraints, but exact machines/racks have to be specified. Unlike Medea, it uses its ILP scheduling algorithm to place all requests and not just the ones with constraints. This is problematic under high load, because the scheduling latency or the placement quality gets compromised. Firmament [23] uses a graph-based approach, supporting simple constraints, such as affinity to specific machines. Adding more constraints would require (an exponential number of) additional vertices in the scheduling graph, increasing scheduling latency. A few other schedulers offer rudimentary support for constraints, based again on machine attributes [22, 48, 49].

Multi-scheduler designs. Multi-level and distributed schedulers also rely on multiple schedulers. They resolve scheduling conflicts by pessimistic [27] or optimistic [45] concurrency control, or by queuing tasks at worker nodes [14, 30, 42, 44]. Medea bypasses this problem by allowing only one of its two schedulers to perform actual allocations.

Resource isolation. Some schedulers mitigate interference between collocated workloads by throttling low priority jobs in favor of LRAs [58], detecting interference though profiling [18–20], or using hardware-based isolation mechanisms [36]. Our goals are broader, focusing on both performance and resilience. However, several of the above techniques can be combined with Medea.

Virtual machine (VM) placement has some commonalities with LRA placement, as VMs can also be long-running. These approaches address resource overload and interference through VM migration [41, 55]. BtrPlace [25] is related to Medea's LRA scheduler in that it supports placement constraints defined by applications and admins. Unlike Medea, it follows a control-loop that mitigates non-viable placements through VM migration, while minimizing the number of machines used. Medea targets instead high-quality placements but could also benefit from container migration.

9 CONCLUSION

We presented Medea, a system for efficiently scheduling applications with long-running containers (LRAs). Medea is the first system to fully support complex high-level constraints both within and across LRAs, which are crucial for the performance and resilience of LRAs. It follows a two-scheduler design, using an optimization-based algorithm for high-quality placement of LRAs with constraints, and a traditional scheduler for placing task-based jobs with low scheduling latency. We evaluated our YARN-based implementation of Medea on a 400-node cluster and showed that it achieves significant benefits over existing schedulers for applications such as TensorFlow and HBase.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Sonia Ben Mokhtar, and the anonymous reviewers for their valuable comments. We also thank Carlo Curino, Chris Douglas, Alexandros Koliousis, Virajith Jalaparti, and Subru Krishnan for their insightful feedback throughout this work. This work was supported by the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) (EP/L016796/1).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. *OSDI*, 2016.
[2] Apache Aurora. http://aurora.apache.org. 2018.
[3] Apache Flink. http://flink.apache.org. 2018.
[4] Apache Hadoop. http://hadoop.apache.org. 2018.
[5] Apache Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html. 2018.
[6] Apache Hadoop Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html. 2018.
[7] Apache HBase. http://hbase.apache.org. 2018.
[8] Apache Kafka. http://kafka.apache.org. 2018.
[9] Apache Samza. http://samza.apache.org. 2018.
[10] Apache Slider (incubating). http://slider.incubator.apache.org. 2018.
[11] Apache Storm. http://storm.apache.org. 2018.
[12] Apache Tez. https://tez.apache.org. 2018.
[13] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 2016.
[14] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. *OSDI*, 2014.
[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. *SoCC*, 2010.
[16] CPLEX Optimizer. http://www.ibm.com/software/integration/optimization/cplex-optimizer. 2018.
[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *OSDI*, 2004.
[18] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ASPLOS*, 2013.
[19] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. *ASPLOS*, 2014.
[20] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. *SoCC*, 2015.
[21] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Making State Explicit for Imperative Big Data Processing. *USENIX ATC*, 2014.
[22] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: max-min fair sharing for datacenter jobs with constraints. *EuroSys*, 2013.
[23] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. *OSDI*, 2016.
[24] Hadoop GridMix. http://hadoop.apache.org/docs/r1.2.1/gridmix.html. 2017.
[25] Fabien Hermenier, Julia Lawall, and Gilles Muller. BtrPlace: A flexible consolidation manager for highly available applications. *IEEE TDSC*, 2013.
[26] Heron. http://twitter.github.io/heron. 2018.
[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *NSDI*, 2011.

[28] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. *USENIX ATC*, 2010.

[29] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Iñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. *OSDI*, 2016.

[30] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. *USENIX ATC*, 2015.

[31] Konstantinos Karanasos, Arun Suresh, and Chris Douglas. Advancements in YARN Resource Manager. *Encyclopedia of Big Data Technologies*, 2018.

[32] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. *CIDR*, 2015.

[33] Kubernetes. http://kubernetes.io. 2018.

[34] Kubernetes: Assigning pods to nodes. http://kubernetes.io/docs/concepts/configuration/assign-pod-node. 2018.

[35] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. *SIGMOD*, 2015.

[36] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. *ISCA*, 2015.

[37] I/O Long-lived daemons for query fragment execution and caching. https://issues.apache.org/jira/browse/HIVE-7926. 2018.

[38] LXC: LinuX Container. http://linuxcontainers.org. 2018.

[39] Marathon. http://mesosphere.github.io/marathon. 2018.

[40] Memcached. http://memcached.org. 2018.

[41] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. *EuroSys*, 2010.

[42] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. *SOSP*, 2013.

[43] Rich placement constraints in YARN. https://issues.apache.org/jira/browse/YARN-6592. 2018.

[44] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. *EuroSys*, 2016.

[45] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. *EuroSys*, 2013.

[46] Simplified and first-class support for services in YARN. https://issues.apache.org/jira/browse/YARN-4692. 2018.

[47] Spark MLLib. http://spark.apache.org/docs/latest/ml-guide.html. 2018.

[48] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor: a distributed job scheduler. *Beowulf cluster computing with Linux*, 2001.

[49] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. *SoCC*, 2012.

[50] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. *EuroSys*, 2016.

[51] Twitter Record. Twitter blog. 2013. http://blog.twitter.com/2013/new-tweets-per-second-record-and-how.

[52] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. *SoCC*, 2013.

[53] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. *EuroSys*, 2015.

[54] John Wilkes. More Google cluster data. Google research blog. Nov. 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[55] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. *NSDI*, 2007.

[56] Guoyao Xu, Cheng-Zhong Xu, and Song Jiang. Prophet: Scheduling Executors with Time-Varying Resource Demands on Data-Parallel Computation Frameworks. *ICAC*, 2016.

[57] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 2010.

[58] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. *EuroSys*, 2013.

[59] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: parallel databases meet MapReduce. *VLDB*, 2012.