Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

Low-power multi-cloud deployment of large distributed service applications with response-time constraints

Babneet Singh[1], Ravneet Kaur[2], Murray Woodside[3*] and John W. Chinneck[3]

Abstract

Distributed service applications make heavy use of clouds and multi-clouds, and must (i) meet service quality goals (e.g. response time) while (ii) satisfying cloud resource constraints and (iii) conserving power. Deployment algorithms must (iv) provide a solution meeting these requirements within a short time to be useful in practice. Very few existing deployment methods address the first three requirements, and those that do take too long to find a deployment. The *Low-Power Multi-Cloud Application Deployment* (LPD) algorithm fills this gap with a low-complexity heuristic combination of generalized graph partitioning between clouds, bin-packing within each cloud and queueing approximations to control the response time. LPD has no known competitor that quickly finds a solution that satisfies response time bounds. A host execution time approximation for contention is fundamental to achieving sufficient solution speed. LPD is intended for use by cloud managers who must simultaneously manage hosts and application deployments and plan capacity to offer services such as Serverless Computing.

On 104 test scenarios deploying up to 200 processes with up to 240 replicas (for scaling), LPD always produced a feasible solution within 100 s (within 20 seconds in over three-quarters of cases). Compared to the Mixed Integer Program solution by CPLEX (which took a lot longer and was sometimes not found) LPD solutions gave power consumption equal to MIP in a third of cases and within 6% of MIP in 95% of cases. In 93% of all 104 cases the power consumption is within 20% of an (unachievable) lower bound.

LPD is intended as a stand-alone heuristic to meet solution time restrictions, but could easily be adapted for use as a repair mechanism in a Genetic Algorithm.

Keywords: Multi-cloud, Services, Deployment, Power-minimization, Optimization, Graph-partitioning, Bin-packing

Introduction

Applications with many services are increasingly common (e.g., microservice systems) and use multiple clouds or data centres to obtain suitable resources, to access regional data sets, or for reliability. Deployment must respect cloud resources, power consumption, and user response time, and must provide a fast solution for re-computation when adapting to changes. Network delays between clouds are large enough that they must be considered explicitly in a useful deployment solution. It is clear from the literature that solution by Mixed Integer Programming (MIP) or by metaheuristics such as genetic algorithms take too long for larger problems, so heuristics must be used.

The service applications considered here have deployable units called "tasks" which may be virtual machines or containers. A user request results in tasks calling each other, possibly many times, in diverse patterns modeled as random interactions with average numbers of calls. They use the internet for calls between clouds, which

*Correspondence: cmw@sce.carleton.ca

[3] Department of Systems and Computer Engineering, Carleton University, Ottawa K1S 5B6, Canada
Full list of author information is available at the end of the article

introduces important delays which may add up to violate response time requirements.

The heuristic *Low Power Multi-Cloud Application Deployment* (LPD) algorithm takes an application in the form of a directed Application Graph with tasks as nodes and their interactions as edges (Fig. 1a), with a maximum anticipated load level, and finds a deployment of tasks to hosts in multiple clouds (Fig. 1b), including scaled-out replicas as necessary. Under autoscaling the deployment is then feasible (although perhaps not optimal) at lower load levels as well, using fewer replica tasks and hosts.

LPD has four goals. It should:

Goal 1. have low total power use by the hosts (ideally, minimum power),
Goal 2. satisfy arbitrary constraints on computing and memory capacity (heterogeneous clouds),
Goal 3. satisfy constraints on the response time, including accounting for intercloud network delay (here, the mean response time is constrained),
Goal 4. provide a solution quickly enough to be useful.

A suitable target maximum solution time depends on how often a deployment must be calculated. It could be a few seconds or minutes; for the examples considered here it is taken to be 1 minute on the grounds that cloud adaptation (such as deployment of a new virtual machine) occurs on a timescale of minutes.

Metaheurstics such as genetic algorithms, particle swarm optimization or stochastic annealing address goals 1–3 by combining them in utility functions that include weighted penalties for delay and other constraints. Soft constraints do not prevent constraint violation without

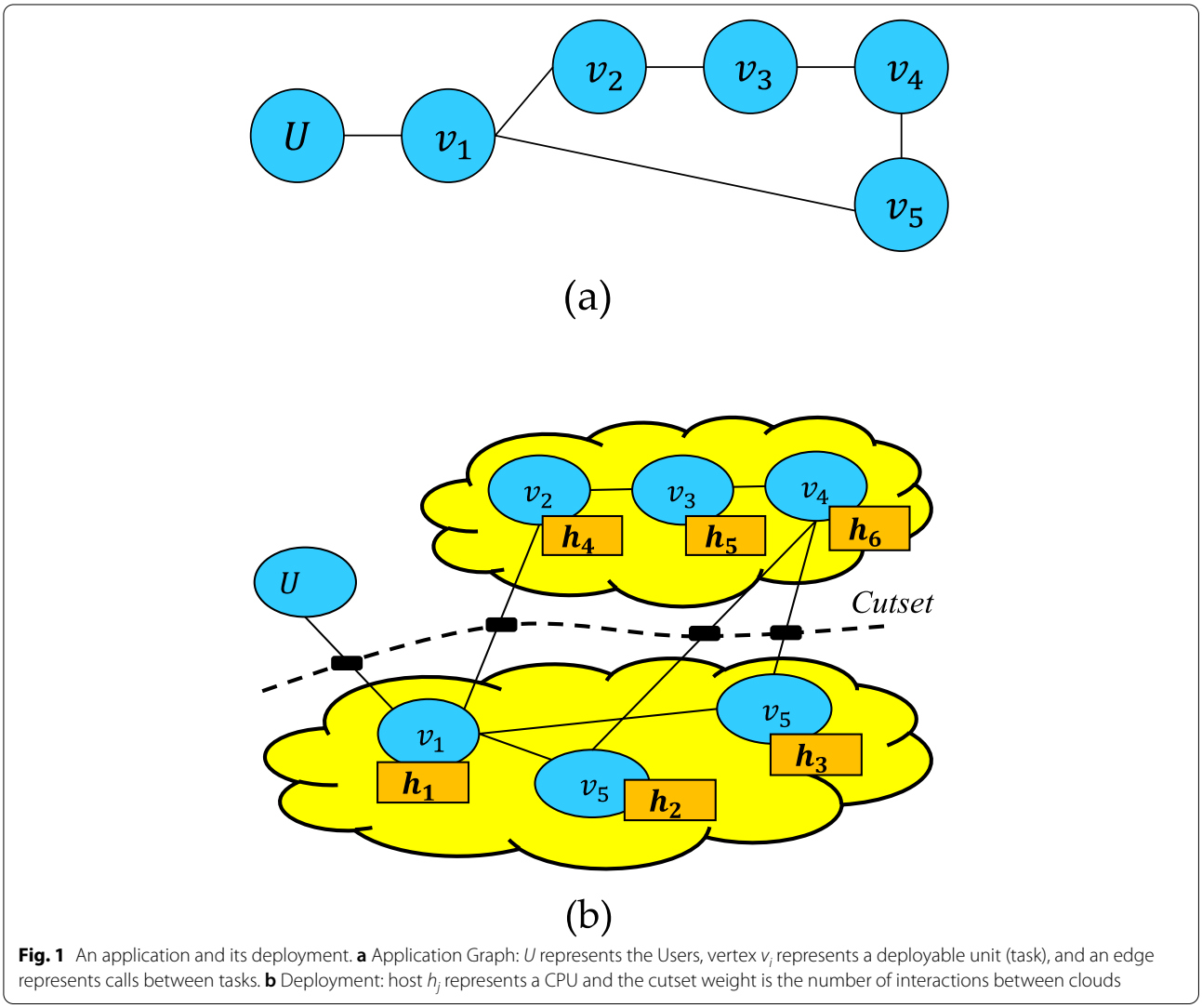

Fig. 1 An application and its deployment. **a** Application Graph: *U* represents the Users, vertex v_i represents a deployable unit (task), and an edge represents calls between tasks. **b** Deployment: host h_j represents a CPU and the cutset weight is the number of interactions between clouds

additional, possibly complex effort, so they are excluded here, and this excludes most metaheuristics. Additionally, their solution takes too long, as discussed below, and this applies also to versions which "repair" the solution to enforce actual constraints.

Excluding soft constraints the above combination of four goals is not resolved by existing methods, which motivates this work. Table 1 lists the combinations of the first three goals and scalable approaches that satisfy them, apart from MIP and metaheuristics. Some combinations have elementary solutions, indicated with italics.

The LPD approach adds power reduction to both graph partitioning (to allocate tasks to clouds) and bin-packing (to allocate tasks to hosts). Using an approximate bound in the response time constraint reduces the effort to compute it and brings the solution time within the desired range.

Related work

Optimal deployment is a classic problem. There have been recent surveys of optimal deployment in clouds by Zhang et al. [4] (emphasizing efficient algorithms), Helali et al. [5] (emphasizing adaptation by consolidation), Masdari and Zangakani [6] (emphasizing predictive adaptation), and Skaltsis et al. [7], oriented to deploying multi-agent systems. They list a variety of approaches, including

- Mixed Integer Programming (MIP),
- Genetic Algorithms (GA) and other metaheuristics such as ant colonies, particle swarms, and simulated annealing, that apply varieties of random search,
- Application Graph Partitioning (AGP), which focuses on the interactions between clouds,
- other approaches including bin-packing, hill-climbing and custom heuristics.

The use of MIP is evaluated in several papers. Malek et al. [8] also gives additional references to MIP. Li et al. [9] combine MIP with bin-packing, to optimize power. Ciavotta et al. [10] use MIP to partition a software architecture and deploy it. The latter study minimized cost while constraining response time (without network delays) as computed by a simple queueing model, and then used a more detailed layered queueing model and a customized local search. MIP can address Goals 1–3 of LPD, and is formulated for this purpose below. However, all these studies found that MIP scaled too poorly for practical use.

GA evolves deployments by modifying a set of initial candidates by random mutations and by combining existing candidates, to optimize a fitness function. The fitness weights together the objectives and constraints, making them soft constraints. GA has been applied for Goals 1–3 in [3, 8, 11]. The use of other metaheuristics is referenced in these papers and in the surveys. A strength of the fitness function approach is that it can be extended to include additional goals such as reliability, and to deal with multiple fitness functions. It can find sets of Pareto-optimal (i.e. non-dominated) solutions for multiple objectives, which goes beyond our goals. In Frincu et al. [12] the objectives are response time, resource usage, availability, and fault-tolerance; in Guerrero et al. [13] they include failure rates and total network delay as a proxy for response time. Frey et al. [14] optimize deployment over multiple clouds and multiple cloud vendors with three objectives of cost, mean response time, and SLA violation rate. Fitness is evaluated by a simulator, and constraints are enforced by rejecting infeasible candidates. Ye et al. [15] consider four objectives including energy minimization in the cloud and the network.

The inability of unmodified GA to directly enforce constraints leads us to exclude it as an approach. A recent method used in [15] enforces constraints by "repairing"

Table 1 Deployment problem cases and scalable solution approaches (excluding metaheuristics and MIP)

Goals	Approaches that Apply Hard Constraints (*elementary methods in italics*)
1 (power only)	*Deploy to the most power-efficient processor first, across all clouds (greedy on power).*
2 (capacity only)	2-D bin-packing (for memory and processing constraints) (e.g., [1])
3 (delay only)	*Deploy on one cloud to eliminate network delays, chosen based on delay to the user and speed of processing.*
1,2	2-D bin-packing (with dimensions of capacity and memory) augmented by considering power (e.g. [2]) (used in LPD).
1,3	Graph partitioning among clouds to control delay; for each cloud deploy to the most efficient processor first (as in LPD without capacity constraints).
2,3	Minimize response time (e.g. by graph partitioning accounting for delay and capacity as in [3]), and accept the solution if it meets the response time constraint.
1,2,3	**New in LPD:** Use graph partitioning methods among clouds to control delay, with two-dimensional bin-packing (capacity and memory) within clouds augmented by decisions based on power.

infeasibilities for each fitness evaluation. The repair changes the candidate solution to make it feasible. Similar changes are part of the partitioning algorithm in LPD. While it is not considered here, the potential use of part of LPD for GA repair is discussed later.

It is also clear from the experience in the literature that GA is too slow for our fourth goal, for large deployments. Malek et al. [8] compare MIP, GA and a heuristic and find that both MIP and GA are orders of magnitude too slow. Guerout et al. [16] compare MIP and GA for single-cloud deployment to optimize a utility function that combines four QoS objectives including energy, using a stagewise approach to improve MIP scalability, and find that MIP and GA are both too slow for practical use. Their calculated response times ignore network delays; including them would undoubtedly make both algorithms slower. Alizadeh and Nishi [17] carefully compare MIP solution times by CPLEX [18], and by GA, on a very large MIP unrelated to deployment. Their GA strictly enforces constraints by constraint repair. They find that the solution time of GA is at best an order of magnitude less than for MIP. Since MIP is several orders of magnitude too slow for Goal 4, this emphasizes that GA is unsuitable on the grounds of solution time.

Most papers consider only a single cloud or data centre. Multi-cloud systems have more serious response time concerns because of the network delay for messages that cross between clouds. Application Graph Partitioning (AGP) addresses these delays as costs on the arcs of an application graph as in Fig. 1(a). AGP algorithms such as in Fiduccia-Mattheyses [19] efficiently reduce or minimize the total cost, and can also include host constraints and costs. AGP is used for multi-cloud deployments in [20], and in [3] to minimize bandwidth (rather than delay), in [1] to minimize the sum of network and energy cost, and in [21] (via a metaheuristic) to minimize power in mobile clouds. The approach called Virtual Network Embedding [22] is also an adaptation of AGP. In summary, AGP deals efficiently with network delays but cannot resolve all the remaining goals; it needs to be augmented.

Bin-packing deals directly with capacity constraints (Goal 2) and is combined with performance models in [23] (ignoring power). It is adapted by Arroba et al. in [2] to reduce power (Goal 1) by applying CPU speed and voltage scaling, but they do not address response time.

Custom heuristics are described in [8] and other works; in [8] they are the only option that solved in a useful time, but this work does not address response time.

The response time includes delay due to processor contention and saturation, which are estimated by performance models. A survey of models for clouds is given by Ardagna et al. [24]. Some recent work on deployment that reflects resource contention (always on the mean response time): Aldhalaan and Menasce [25] use a queueing model to place VMs in a cloud, finding the number of replicas by a hill-climbing technique. Ciavotta et al. [10] use a simple queueing approximation in a first stage, and a detailed layered queueing model in a second stage of optimization. Molka and Casale [26] use queueing models with GA, bin-packing and non-linear optimization, to allocate resources for a set of in-memory databases. Wada et al. [27] use queuing models to predict multiple service level metrics and a multi-objective genetic algorithm to seek pareto-optimal deployments. Calheiros et al. [28] and Shu et al. [11] use a queueing model to scale a single task to meet response time requirements, and [11] also models response failures. None of these studies consider power.

Deployment to minimize power has been part of many studies, e.g. energy is included in the utility function in [8]. Wang and Xia [29] apply MIP to VM placement in the cloud to minimize power subject to processing and memory constraints (but ignoring response time and network delays). Tunc et al. [30] use a real-time control rule based on performance monitoring and a Value of Service metric that enforces the response time constraint while controlling for energy consumption. This is close to our stated problem, but they do not include network delays in the response time. Chen et al. [31] use a random search heuristic to minimize the money cost of power and network operations (but not delay).

For deployment tools, Arcangeli et al. [32] describe many tools from the viewpoint of a system operator, including MIP, GA, AGP and others.

To summarize, a heuristic solution for the minimum-power application deployment problem subject to constraints on response time and host capacities is needed because the solution times to find provable optima via methods such as MIP are too long. Metaheuristics like GA also take too long and have difficulty enforcing actual constraints. LPD, developed in the theses of Kaur [33] for a single cloud and Singh [34] for multi-clouds, uses instead a novel combination of graph partitioning and bin-packing to meet all the goals stated for LPD.

Background on techniques used in LPD

LPD uses customized algorithms derived from the following two families.

***K*-way graph partitioning (KGP)** [19] divides a graph into K parts, where each part is a subset of the graph node set, the parts are non-intersecting, and cover the entire node set. A common objective is to minimize the cut-weight, which is the sum of the edge weights that connect nodes in different parts. KGP is NP-hard [35] and so does not scale well, but approximate solutions are provided

more quickly using multi-level KGP schemes proposed by Karypis et al. [36, 37]. In LPD, the edge weights are defined so the cut-weight is the network delay and multi-level KGP is used to reduce the cut-weight until the response time constraint is satisfied, but not necessarily minimized. Graph partitioning algorithms often enforce a balance constraint to distribute the node weights equally among the partitions, but this is not required in the deployment problem.

2-D bin packing described in [1] is an NP-hard algorithm to pack a set of 2-dimensional items into a minimum number of 2-dimensional bins. In LPD, heuristic 2-D bin packing is incorporated into the KGP move evaluation, to reduce power consumption. After partitioning, a more rigorous bin packing is applied to each partition to further reduce power consumption.

The model for multi-cloud application deployments

We consider K heterogeneous clouds, with heterogeneous hosts, and different network latencies between clouds. Communication delays within the clouds are ignored. The set of users is treated as an additional cloud, which has only the users; there is no other deployment to the user cloud. Figure 2 shows the model, and Table 2 defines its parameters. The processing capacity of cloud hosts is defined in *ssj_ops* according to the SPECpower benchmark for processors [38].

Table 2 Cloud and Host Property Definitions

Parameter	Meaning	Units
$C=\{C_1,\ldots,C_K\}$	a set of K clouds; the set of users is C_0.	
C_k^{opsMax}	processing capacity of C_k.	*ssj_ops*
C_k^{memMax}	memory capacity of C_k.	MB
δ_{km}	mean delay between C_k and C_m	ms
h_{kl}	host l on cloud C_k	
$h_{kl}^{ops}(D)$	processing demand of host h_{kl}, for deployment D.	ms
h_{kl}^{opsMax}	processing capacity of host h_{kl}.	*ssj_ops*
h_{kl}^{memMax}	memory capacity of host h_{kl}.	MB

A deployment D of tasks to individual hosts in the clouds must provide a user throughput of λ responses/sec, and have a latency (response time) of no more than R^{req} msec.

The application model and application graph

The analysis begins from an Application Graph G of vertices (denoted as v_i) representing deployable components, and edges denoted as (v_i, v_j), representing their interactions and labelled by the number of messages c_{ij} exchanged, per user response. Optionally, G may be derived from an Application Model as in Fig. 3(a), which shows the components as "tasks" and the inter-task calls that invoke operations, labelled by resource demands.

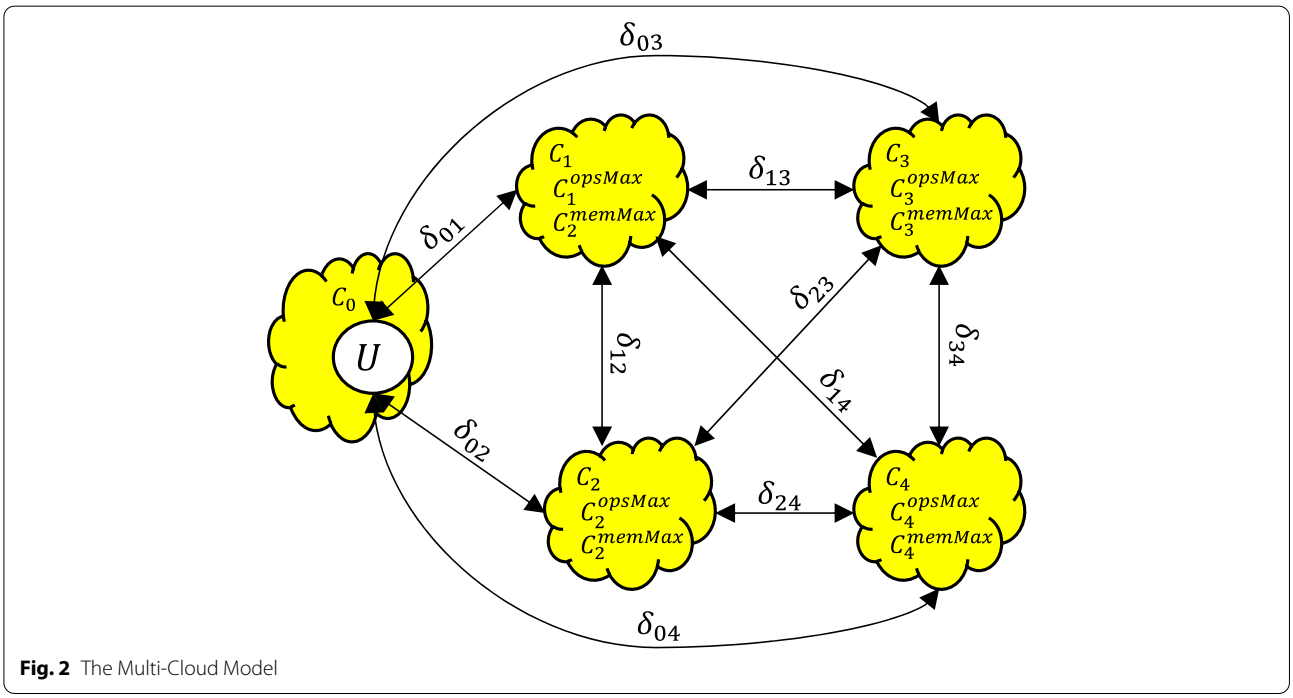

Fig. 2 The Multi-Cloud Model

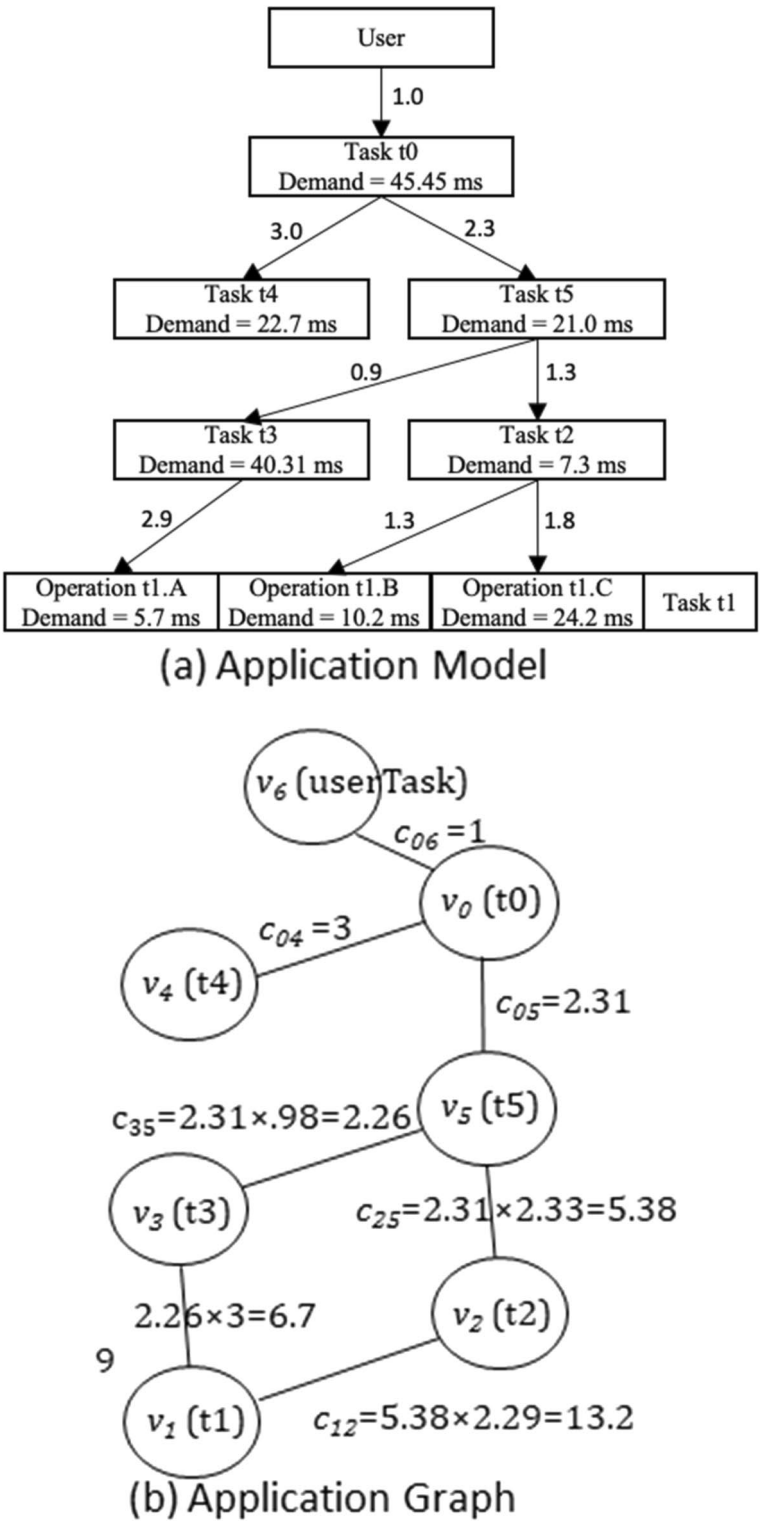

Fig. 3 An Application Model and its Graph. **a** Example Application Model in LQN Notation. **b** Corresponding Application Graph, Labelled with Mean Interaction Counts

The Application Model uses the notation of layered queueing models [39, 40] in which the components are shown as rectangles (e.g. t0) and their operations as attached rectangles (e.g. e2601). Each invocation of an operation has a mean CPU demand and makes calls as shown by arrows, labeled by the mean calls. Derivation of the graph parameters from a layered queueing model is described in [41].

Service operations may have highly variable resource demands, depending on the nature of the user request, and the data that it accesses. This is modeled by random values for CPU demands and calls, with average values as parameters.

Figure 3(b) gives an example of an Application Graph derived from the Application Model of Fig. 3(a). Alternatively, the Application Graph could be derived directly from measurements on the deployment units. Table 3 defines the properties of the Application Graph.

The task processing demand v_i^{ops} is measured in operations/s in units of *ssj_ops* as defined in the SPECPower benchmark [38], executed on a reference host h_{ref} at a nominal user throughput of λ responses/s. To obtain the demand on any other host type it is scaled by the host speed factor to give v_i^{ops}/h_{kl}^{sf}.

Power model

Power consumption h_{kl}^{pow} for each host h_{kl} is related to the work it is doing, modeled by a linear least-squares fit to its SPECpower [38] benchmark results, stated as:

$$h_{kl}^{pow} = h_{kl}^{base} + h_{kl}^{ops} * h_{kl}^{pRate} \tag{1}$$

Table 3 Application Model and Application Graph Property Definitions

Parameter	Meaning
$G = (V, E, v_i^{ops}, v_i^{mem}, c_{ij}, \forall i, j)$	the graph representing the application
$V = \{v_1, v_2, \ldots, v_n\}$	the set of application tasks (vertices of *G*).
v_i^{ops}	processing requirement of task v_i
v_i^{mem}	memory requirement of v_i
$E = \{(v_i, v_j)\}$	edges of *G*.: the set of all calls between tasks v_i and v_j
c_{ij}	mean calls between v_i and v_j, per user response
h_{kl}	host *l* in cloud C_k
h_{kl}^{opsMax}	processing capacity of h_{kl}
h_{ref}	reference host for processing demands
$h_{kl}^{sf} = h_{kl}^{opsMax}/h_{ref}^{opsMax}$	the speed factor of h_{kl}, relative to h_{ref}.

where h_{kl}^{base} is the base turn-on power in W, h_{kl}^{pRate} in W/*ssj_ops* is the slope of the fitted line, and h_{kl}^{ops} is the load.

Power Usage Effectiveness, defined as the ratio of cloud power consumption to total host power consumption [42], is assumed for simplicity to be unity for all clouds, so cloud power equals total host power. If it is less (but still the same for all clouds), power can be scaled.

Bound on Total execution delay

The response time is the network delay plus the execution delay. The total execution delay is the time (per user response) spent executing and waiting to be scheduled on some processor. LPD applies a capacity constraint on each host that restricts its CPU utilization below a nominal value U^{nom}. Since execution delay increases with utilization, the delay for utilization $= U^{nom}$ is a bound.

For a given task v_i on host h_{kl} the maximum delay is calculated by the well-known processor-sharing approximation to host queueing (see e.g. [43]) as:

$$time(v_i, h_{kl}) = \left(v_i^{ops}/h_{kl}^{sf}\right)/\left(1 - U^{nom}\right)\ ms. \tag{2}$$

The total delay bound D^{Tproc} is the sum of $time(v_i, h_{kl})$ over all tasks, which simplifies to

$$D^{Tproc} = \sum_{k,l}\left(h_{kl}^{ops}/h_{kl}^{sf}\right)/\left(1 - U^{nom}\right) \tag{3}$$

Adding this value to the total network delay gives an upper bound on response time, defined as D^{RT} below.

Deployment model

Using the concepts and notation defined above, a deployment is described as follows:

- $d_{ikl} = 1$ if task v_i is deployed to host h_{kl} (the *l*th host in cloud C_k), or 0 otherwise.
- D: a deployment, a setting of all variables d_{ikl}. D_q is the *q*th deployment in a set of deployments.
- D^{pow}, D^{Tnet}, D^{Tproc}, D^{RT}: respectively the total power (W), total network delay, and bounds on the total execution delay and the response time of deployment D in ms.
- $y_{ik}(D) = 1$ if task i is assigned to a host on cloud C_k, or 0 otherwise.
- $z_{kl}(D) = 1$ if host h_{kl} is used, or 0 otherwise.

A feasible deployment must not exceed the processing capacity (in *ssj_ops*) and memory capacity (in MB) of any host and must satisfy the overall response time requirement R^{req} (in msec).

Mixed-integer quadratic programming optimization

The deployment optimization gives a mixed-integer quadratic programming (MIQP) model (quadratic because the network delay depends on the deployment of pairs of tasks). MIQP solutions found using CPLEX [18] were used to evaluate the accuracy of the LPD heuristic on small examples, though MIQP does not scale up well.

In the formulation, a large positive constant M is introduced for calculating z_{kl}, of value $M = \max_{kl}\left(h_{kl}^{opsMax}\right) + offset$, where *offset* is a large value, e.g. 1000.

The resulting deployment is designated D_{MIQP} with optimal power D_{MIQP}^{pow} and response time D_{MIQP}^{RT}.

MIQP Constraints:

Host processing capacity:

$$h_{kl}^{ops} = \sum_i \left[v_i^{ops}/h_{kl}^{sf} \times d_{ikl}\right] \le \left[h_{kl}^{opsMax} \times U_{nom}\right] \forall k, l \tag{4a}$$

Host memory capacity:

$$\sum_i \left[v_i^{mem} \times d_{ikl}\right] \le h_{kl}^{memMax} \forall k, l \tag{4b}$$

Each task is assigned exactly once:

$$\sum_{k,\,l} d_{ikl} = 1 \forall i \tag{4c}$$

Indicator variable to show that task v_i is assigned to cloud C_k:

$$y_{ik} = \sum_l d_{ikl} \forall i, k \tag{4d}$$

Network delay for deployment D_{MIQP} (quadratic in y):

$$D_{MIQP}^{Tnet} = \sum_{m,n \text{ in clouds}} c_{ij} \times y_{im} \times y_{jn} \times \delta_{mn} \forall (v_i, v_j) \text{ in } E \text{ and } (m \ne n) \tag{4e}$$

Total execution time:

$$D_{MIQP}^{Tproc} = \sum_{k,l} \left(h_{kl}^{ops}/h_{kl}^{sf}\right)/\left(1 - U^{nom}\right) \tag{4f}$$

Response time constraint:

$$D_{MIQP}^{RT} = D_{MIQP}^{Tproc} + D_{MIQP}^{Tnet} \le R^{req} \tag{4g}$$

Indicator variable z_{kl} to show whether host h_{kl} is used:

$$h_{kl}^{ops} - M \times z_{kl} \le 0 \forall k, l \tag{4h}$$

Power consumption of host h_{kl}:

$$h_{kl}^{pow} = \sum_i \left(h_{kl}^{base} \times z_{kl}\right) + h_{kl}^{ops} \times h_{kl}^{pRate} \forall k, l \tag{4i}$$

MIQP Objective function (minimize total power in W):

$$\text{minimize } D_{MIQP}^{pow} = \sum_{k,l} h_{kl}^{pow} \tag{4j}$$

CPLEX solutions and the time it took to obtain them are described in Section 6.1, in comparison with LPD.

The low-power multi-cloud application deployment (LPD) algorithm

LPD first scales the application by creating replicas of each task to provide sufficient processing power. Then it deploys tasks to clouds by a graph-partitioning algorithm with a first phase that iteratively reduces D^{RT} until $D^{RT} \le R^{req}$, and a second phase that minimizes D^{pow} while maintaining $D^{RT} \le R^{req}$. This is carried out for a set of initial partitions, including random partitions. Finally, the host deployments d_{ikl} within each cloud are determined by a power-aware bin-packing stage which satisfies the capacity constraints for each host and reduces power further.

There are six stages of LPD as described in the Algorithm summary and illustrated in Fig. 4.

LPD algorithm summary

INPUTS: clouds, hosts, users, application, control parameters, network delays.

Stage 1: Select a subset of clouds C to host the application.

Stage 2: Generate the application graph G from the application model.

Stage 3: Scale the application to the workload, generate a graph for the scaled application (G^{scaled}), and coarsen G^{scaled} to get G^{coarse}.

Stage 4: Generate a set of initial deployments.

Stage 5: For each initial deployment D_q^{init} do:

5a $D_q^{partition}$ = Partition (D_q^{init}, G^{coarse}, C, δ, R^{req}) (for the *Partition* algorithm, see below),

If $D_q^{partition}$ is null (no feasible deployment found), then go to next initial deployment.

5b Uncoarsen $D_q^{partition}$.

5c For each cloud, perform bin packing on its deployed tasks by both HBF-Proc and HBF-Mem and record the deployments as $D_q^{HBF-Proc}$ and $D_q^{HBF-Mem}$.

Stage 6: Set D^* to the deployment with the smallest D^{pow} out of all $D_q^{partition}$, $D_q^{HBF-Proc}$, and $D_q^{HBF-Mem}$ or to null if there is no feasible deployment.

OUTPUT: Deployment D^*.

A major part of LPD is the *Partition* sub-algorithm defined below along with the details of the six stages.

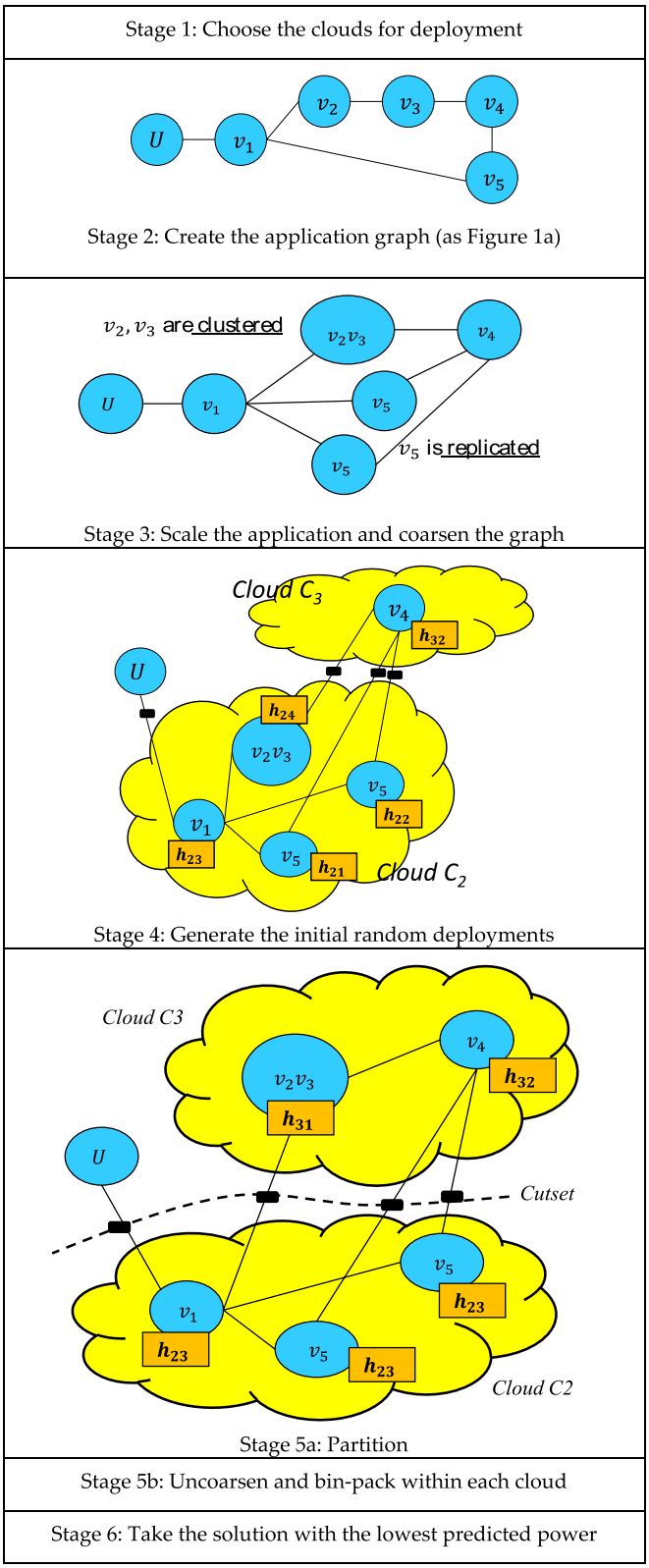

Fig. 4 Stages of the LPD algorithm. Stage 1: Choose the clouds for deployment. Stage 2: Create the application graph (as Fig. 1a). Stage 3: Scale the application and coarsen the graph. Stage 4: Generate the initial random deployments. Stage 5a: Partition. Stage 5b: Uncoarsen and bin-pack within each cloud. Stage 6: Take the solution with the lowest predicted power

LPD stages 1–4: preparation and initial deployment

The first stage selects the clouds for deployment, and stage 2 creates the application graph G. Stage 3 adds replicas to provide sufficient capacity for each task, even if deployed on the slowest host in the system (call its speed factor s^{min}). For task v_i this gives:

$$replicas = v_i^{ops} / \left(h_{ref}^{ops} \times s^{min} \right) \quad (5)$$

Each replica receives an equal share of the calls to v_i and duplicates the calls made by v_i.

Stage 3 coarsens the resulting graph to G^{coarse} by repeatedly merging the nodes (tasks) joined by the edge with the largest weight, subject to maximum nodal processing and memory requirements (which ensure that the merged node will fit on any host). It terminates when the largest edge weight is less than the original average edge weight.

Experience showed that the partitioning sometimes found local optima. To improve the solution, Stage 4 creates a set of initial deployments using three strategies:

1. *Random*: N^{random} initial deployments.
2. *Greedy on Power/Delay*: with two variants: repeatedly deploy the task with the largest CPU demand on the feasible cloud that has space and (is most power-efficient)/(adds the least delay) (2 deployments),
3. *Bin-packing*: all combinations of two bin-packing strategies, packing on memory or power, and sorting bins in increasing or decreasing order (8 deployments).

LPD stage 5: partitioning

Partitioning among clouds uses the sub-algorithm described below with a simplified set of constraints. The response time constraint was simplified by computing D^{Tproc} just once based on the slowest host, and then constraining D^{Tnet} by

$$D^{Tnet} \leq R^{req} - D^{Tproc} \quad (6)$$

which is the cut weight of the partition. Simplified capacity constraints for processing and memory in each cloud are computed based on the most restricted hosts in the cloud. These simplifications are conservative, in that any feasible solution satisfies the original constraints.

Partitioning uses the K-way variant of the Fiduccia-Mattheyses (FM) graph partitioning refinement heuristic [19, 35]. It recursively identifies clusters to move from one cloud to another, based on the improvement in a fitness function calculated for each potential move. In phase 1 it obtains feasibility by making moves that reduce D^{Tnet} until Eq. (6) is satisfied, and then in phase 2 it minimizes power while maintaining feasibility. The *Partition* sub-algorithm uses this additional notation:

$move(v_i, h_{to,\,l})$ moves task v_i to host l on cloud C_{to},

- $\Delta_{RT}(move)$ = amount of response time reduction,
- $\Delta_{pow}(move)$ = amount of power reduction,

and partitions G^{coarse} among clouds to minimize power subject to response time and capacity constraints.

Partition sub-algorithm

INPUTS: D_{init}, G, C, δ, R^{req}

$D \leftarrow D_{init}$. Calculate D^{RT} and D^{pow} and sort the tasks in G in decreasing order of v_i^{ops}.

Repeat:

Initialize best power reduction move: $move_{pow} \leftarrow \phi$, $\Delta_{pow}^{max} \leftarrow 0$.
Initialize best latency reduction move: $move_{RT} \leftarrow \phi$, $\Delta_{RT}^{max} \leftarrow 0$.
For each task v_i in the sorted task list:

C_{from} = cloud containing v_i in deployment D.
For each cloud C_{to} other than C_0 or C_{from}:

Find the most power efficient available host $h_{to,\,l}$ in C_{to} that can accept v_i.
Calculate the power and latency reductions Δ_{pow} and Δ_{RT} for moving v_i to $h_{to,\,l}$.
If $\Delta_{pow} > \Delta_{pow}^{max}$, then update Δ_{pow}^{max} to Δ_{pow} and $move_{pow} \leftarrow move(v_i, h_{to,\,l})$.
If $\Delta_{RT} > \Delta_{RT}^{max}$, then update Δ_{R}^{max} to Δ_{RT} and set $move_{RT} \leftarrow move(v_i, h_{to,\,l})$.

If $move_{pow} \neq \phi$ and does not violate R^{req}, then accept $move_{pow}$.
Else if R^{req} not met and $move_{RT} \neq \phi$, then accept $move_{RT}$.
If a move has been accepted, make it and update D, D^{RT},and D^{pow}.
Else (no move is found): exit.

OUTPUT: D.

After partitioning among clouds, Stage 5 uncoarsens G and restores the original constraints. It applies a final 2-D bin-packing step within each cloud (tasks deployed on hosts) to reduce power consumption and satisfy the constraints of individual hosts. It uses two variants of Hybrid-Best-Fit (HBF) two-dimensional bin-packing [44] with dimensions of processing (v_i^{ops}) and memory (v_i^{mem}).

HBF-Proc sorts the tasks in descending order by v_i^{ops} and HBF-Mem sorts them by v_i^{mem}.

Stage 6: selecting the final deployment

Among all the deployments found from all the initial deployments, stage 6 returns the one with the lowest power consumption D^{pow}.

LPD design choices

Preliminary experiments described in [34] refined the choice of algorithm steps and parameters, showing that:

- Most solutions arise from the random initial deployments. $N^{random} = 60$ assured a solution.
- Coarsening using the stated termination condition reduced runtime and gave better quality solutions for both power and delay.
- The final bin-packing using HBF improved the solution in 15% of cases, mostly due to HBF-Mem. Other approaches to bin-packing did not improve on this.

Time complexity of LPD

There are n tasks and $|E|$ edges in G^{scaled}, K clouds each with O(h) hosts, and d initial deployments. The important operations have these time complexities (see [37] for related complexity analyses of the partitioning):

(i) sorting hosts in each cloud by power efficiency is O($Kh \log(h)$)
(ii) application scaling is O($|V|$) (tasks before scaling).
(iii) coarsening and uncoarsening are both O($d\,|E|$),
(iv) partitioning is O($dK\,|E|$) for the K-way Fiduccia Mattheyses algorithm,
(v) bin packing for K clouds is O($dKn \log(n)$),
(vi) selecting the best deployment is O(d).

Assuming that the number of interactions of each task is bounded, then $|E|$ is O(n), operation (v) is dominant and the time complexity is O($dKn \log(n)$).

LPD for genetic algorithm solution repair

Stage 4 of the LPD generates numerous infeasible initial deployments. The partitioning in stage 5a first works to render the initial solution feasible, and then to improve the objective function value. It is simple to terminate the partitioning at the first feasible solution. This truncated stage 5a partitioning step can be used as a quick repair mechanism for infeasible intermediate solutions generated by a Genetic Algorithm. This possibility will be investigated in future research.

Table 4 Cloud parameters (Definitions in Tables 2 and 3)

Cloud	Total memory capacity C^{memMax} (GB)	Total processing capacity C^{opsMax} (ssj_ops)	Total logical CPUs
Edge (C_1)	64.0	1,669,696.80	32
Small (C_2)	80.0	2,028,905.60	40
Medium (C_3)	176.0	4,240,154.40	88
Large (C_4)	272.0	7,467,443.20	200

Evaluation experiments: setup

The quality of LPD deployments was compared to (i) MIQP solutions when they could be obtained, and (ii) a loose power bound. Response time feasibility was verified against the results from the LQNS solver, which gives a more refined calculation of delay.

Comparisons of measure m found by two methods A and B were made using the relative error measure $\Delta_{A:B}$ defined as:

$$\Delta^m_{A:B} = \left(\frac{A-B}{B}\right) \times 100\% \tag{7}$$

Application test cases

Random test models were generated, structured to provide many variations of the layered architecture typical of cloud service applications such as e-commerce and social media, and of microservices generally. Application models consisted of 4, 8, 18, 24, or 30 tasks and had a wide range of workload characteristics. One hundred four test models were generated by the random model generator *lqngen* [45]. Model properties ranged widely, to represent a wide range of applications and of deployment difficulty, giving these properties:

- Number of application components, including scaled-out replicas: 20–240,
- Total processing requirement: 5–84% of the available total multi-cloud capacity,
- Total memory requirement: 5–87% of the total multi-cloud capacity,
- Total number of calls per user request: 3–11,050.

Six comparisons were made:

1. Power: LPD vs MIQP (Section 6.1), measured by $\Delta^{pow}_{LPD:MIQP}$.
2. Power: LPD vs the bound, measured by $\Delta^{pow}_{LPD:bnd}$.
3. Response time: LPD vs the requirement, measured by $\Delta^{RT}_{LPD:req}$. Should be zero or negative.

Table 5 Network delays δ_{mn} (msec)

Cloud *m*	Cloud *n*				
	0	**1**	**2**	**3**	**4**
0 (Users)	N/A	25	100	175	250
1	25	N/A	75	150	225
2	100	75	N/A	75	150
3	175	150	75	N/A	75
4	250	225	150	75	N/A

4. Response time: LPD vs more accurate LQNS calculation, measured by $\Delta^{RT}_{LPD:LQNS}$.
5. Response time: LQNS on LPD deployment vs the requirement, measured by $\Delta^{RT}_{LQNS:req}$.
6. LPD runtime, denoted by T_{LPD}.

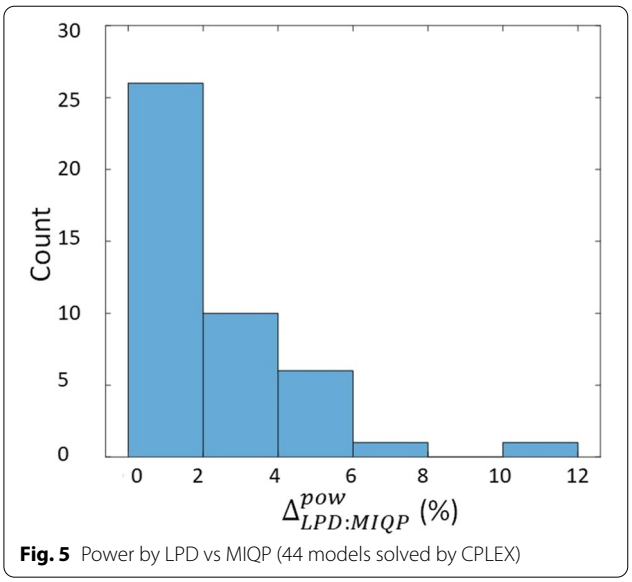

Fig. 5 Power by LPD vs MIQP (44 models solved by CPLEX)

Cloud environment

The test cases considered four clouds as shown in Fig. 2, with parameters in Table 4 below. Table 5 shows the network delays, and Table 6 describes the hosts. The experiments were run with $U^{mom} = 0.8$, which is a typical target value for maximum utilization, and with 60 initial deployments.

Software and hardware

The algorithm was implemented in Java, using the Java Universal Graph Framework (JUNG) v2.0.1 and Guava v19.0, with Google Core Libraries for Java for the Multimap data structure that represents the deployment. LPD was run on an Intel i5 3570 PC with 3.4 GHz processor, 16 GB memory and with Ubuntu v16.04 Linux, while the MIQP solutions were run on a faster Intel Core i9-9980HK running at 2.40GHz.

Experimental results

The experiments evaluate the effectiveness of LPD and explore its properties on applications of different sizes and requirements. The principal concerns are (1) closeness of the power to the bound on the optimal value, (2) meeting the response time constraint, and (3) algorithm solution time versus the target of 1 minute for practical use in deploying models in this size range. Also examined were (4) the effectiveness of the strategy used by LPD versus two simple alternatives, and (5) the impact of tighter response time constraints on solution effort and quality.

Power consumption: LPD vs. MIQP

MIQP solutions by the CPLEX optimizer [18] give the minimum power, which was compared with LPD. Forty four cases were solved by CPLEX within a time limit of 300 seconds per solution. Fig. 5 shows a histogram of

Table 6 Host parameters, in decreasing order of speed factor (Definitions in Tables 2 and 3)

Host Type	Number of Logical CPUs, h^{ncpu}	SPECpower Throughput at 100% Load, h^{opsMax} (*ssj_ops*)	h^{opsMax}/h^{ncpu} (*ssj_ops*)	Speed Factor h^{sf}
Inspur Corporation NF5280 M4	88	3,561,599	40,473	1.00 (slowest)
Dell Inc. PowerEdge R630	72	3,240,418	45,006	1.11
Dell Inc. PowerEdge R740	112	5,727,798	51,141	1.26
QuantaGrid S31A-1 U	8	474,667	59,333	1.47
Fujitsu Server PRIMERGY TX1320 M2	8	478,512	59,814	1.48
Fujitsu Server PRIMERGY TX1330 M2	8	484,122	60,515	1.50
Fujitsu Server PRIMERGY RX1330 M1	8	508,013	63,502	1.57
Fujitsu Server PRIMERGY RX1330 M3	8	586,973	73,372	1.81

$\Delta^{pow}_{LPD:MIQP}$. LPD power is equal to MIQP power (within a very small tolerance) in 15 of the 44 cases (34%) and is within 6% in 42 of 44 cases (95%). The largest difference is 10.8%. The overall success rate of LPD in finding near-optimum low-power solutions for these 44 cases is excellent. For the remaining 60 larger cases a comparison was made to a bound.

The 300-second MIQP limit is well beyond the assumed useful time limit of 60 s and the limited solvable cases demonstrate the poor scalability of the MIQP approach.

Power consumption: LPD vs. a lower bound

A simple lower bound P^{bound} on total power consumption is calculated by assigning fractions of the pooled total workload to the most power-efficient processors across all clouds and taking the sum of their power consumptions. The construction of P^{bound} ignores the granularity of processing by the tasks, and the response time constraints, which will make it optimistic and unattainable, but it provides a comparison for the cases in which the MIQP solver exceeds its time limit.

The histogram of the difference $\Delta^{pow}_{LPD:bnd}$ in Fig. 6 shows that the power found by LPD is within 10% of P^{bound} for 22 of the 104 cases and within 10–20% in a further 75 cases. Of the 7 cases with a larger difference, 3 had MIQP solutions with identical power to LPD, leaving just four cases with (possibly) unsatisfactory power solutions.

For the 60 cases where the MIQP timed out, the LPD power averages 14.3% greater than P^{bound}, with 56 of the 60 cases (93%) closer than 20%.

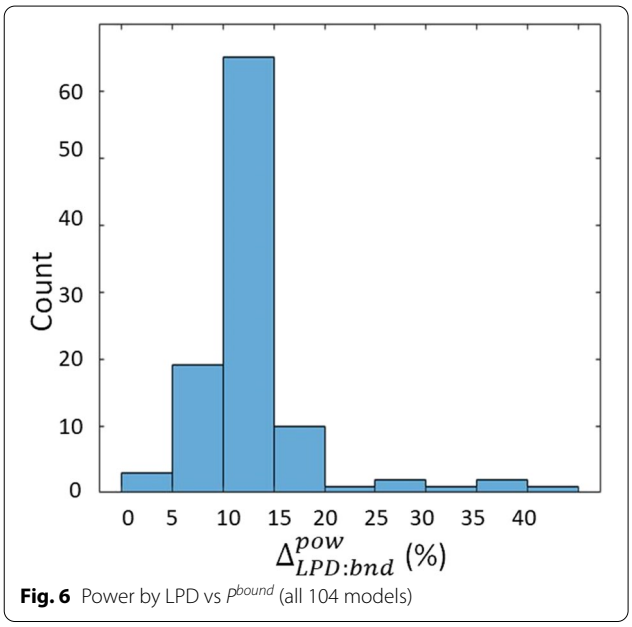

Fig. 6 Power by LPD vs P^{bound} (all 104 models)

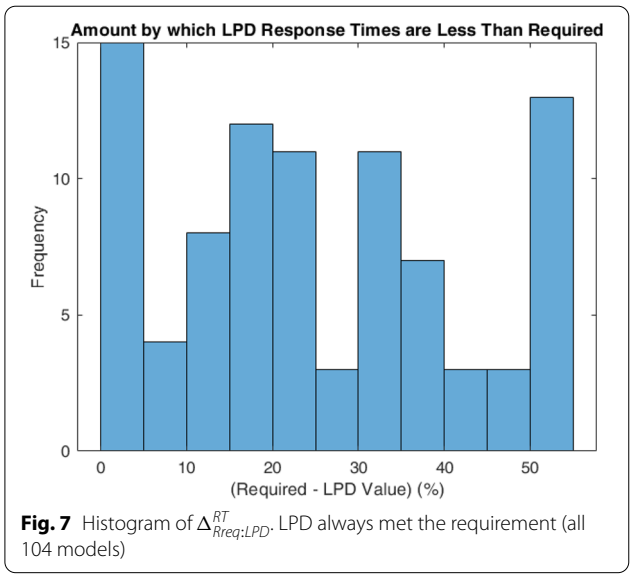

Fig. 7 Histogram of $\Delta^{RT}_{Rreq:LPD}$. LPD always met the requirement (all 104 models)

Verification of the response time constraint

In the 104 LPD solutions, the simplified response time approximation D^{RT}_{LPD} always satisfies the response time limit R^{req}. If there are many calls between processes then network delay reduction may take precedence over power reduction and the response time may be close to the constraint. If the response time constraint is too tight a feasible solution may not be found (or even be possible). Figure 7 shows a histogram of the relative amount by which LPD met the response constraint; the solution was always feasible and in some cases the structure of the application messaging, combined with power optimization, gave response times much smaller than the constraint.

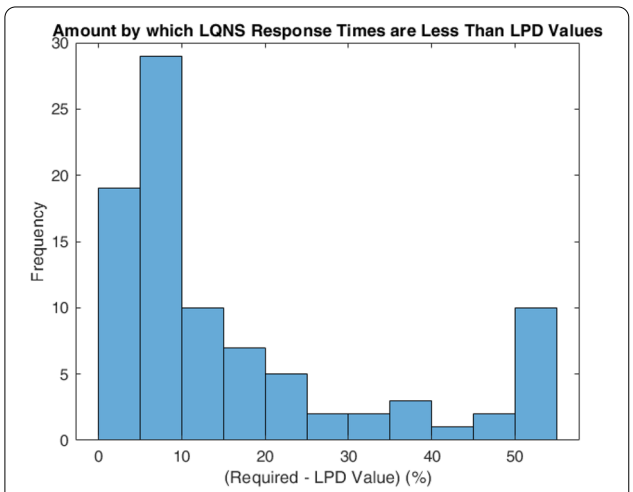

Fig. 8 Histogram of $\Delta^{RT}_{LPD:LQNS}$. It is always positive (90 models solved by LQNS)

A more accurate estimate of the response time of the LPD deployment, denoted D^{RT}_{LQNS}, can be found using the Layered Queueing Network Solver LQNS [39]. In the LQNS solution the host utilizations depend on the deployment (rather than assuming the value U^{nom}) and the solver exploits more sophisticated queueing approximations. Because the actual utilizations can never be higher, the LQNS values are always feasible for an LPD solution. This was verified in the 84 cases for which LQNS could find a solution; in the remaining cases the model was too large for the LQNS solver. The histogram in Fig. 8 shows the relative margin by which RT found by LQNS was smaller than by LPD.

Variants on LPD

Two variants on LPD were created to consider alternative strategies:

> *Variant1*: Stop at the first solution that satisfies the response time constraint. This pays most attention to reducing the latency until the response time constraint is met.
>
> *Variant2*: Continue from the Variant1 solution to a final solution that minimizes response time. This gives absolute priority to a short response time.

The two variants were applied to 16 test cases. All the solutions met the response time constraint, with Variant2 providing the lowest values. However LPD gave the lowest power consumption in every case, exceeding P^{bound} by just 6.5% on average (vs 18.4% for Variant1 and 21.1% for Variant2), confirming that it provides a suitable balance between power and response time.

Algorithm runtime vs constraint tightness

Algorithm runtime is influenced by the difficulty of the deployment due to the tightness of the constraints. To examine this relationship, the response-time constraint of one model was gradually tightened. The response time includes an execution delay of $R^{pt}_D = 608\ msec$. Table 7 shows the impact of the reduced values of R^{req} on the LPD runtime T_{LPD} with 52 starting points. At first almost all trials find feasible solutions with increasing T_{LPD}, then T_{LPD} levels off and there are increasing failures. At some

Table 7 LPD Runtime Dependence on the Response Time Requirement

Run	R^{Req} [msec]	T_{LPD} [msec]	Deployment Successes	Deployment Failures	Comment
1	12,500	6574	51	1	Loose time requirement makes feasibility easy and allows power optimization
2	10,000	8655	51	1	
3	7500	16,686	51	1	
4	5000	24,718	14	38	Time tightness is a factor
5	2500	25,523	0	52	R^{Req} is infeasible

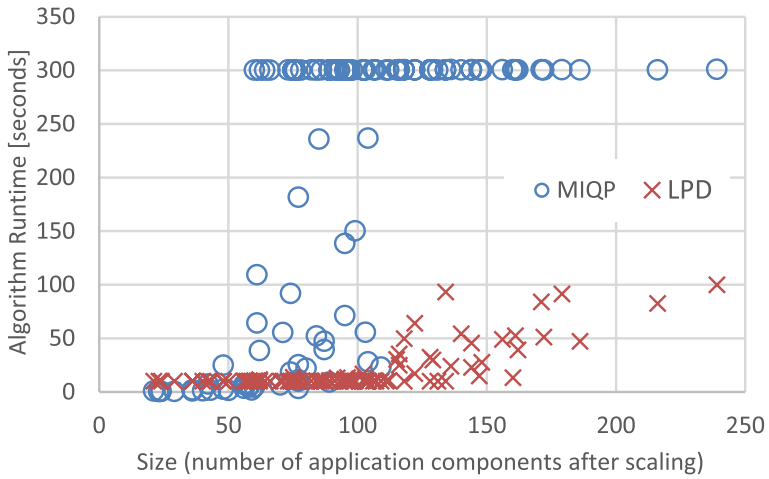

Fig. 9 Algorithm runtime versus problem size for the LPD and MIQP

point the capacity and runtime constraints together make the problem infeasible, shown by the final row.

Scalability of CPLEX MIQP vs LPD

Besides constraint tightness, the runtime is affected by the size of the task graph, the number of clouds and hosts, the capacities of the hosts, and network latencies. For a simple evaluation of algorithm scalability, runtime was related to problem "size" expressed as the number of tasks in the graph. Figure 9 summarizes runtime vs. problem size for LPD and the CPLEX MIQP solution on the 104 test cases. For technical reasons, CPLEX was run on a faster machine with an Intel Core i9-9980HK running at 2.40GHz (average CPU mark 15,089), or roughly 3 times faster. Thus the runtime difference between CPLEX and LPD is three times as great as the figure indicates.

CPLEX converges to a solution in less than 300 seconds in just 44 cases. For half of these it has a runtime of less than 10 seconds and is faster than LPD; for the other 22 cases LPD is faster than CPLEX. CPLEX is able to provide a first feasible solution (not optimal) within 60 sec of scaled time (time multiplied by three) for 85 of 104 models (81.7%), but with more power consumption than LPD in 64 (75%) of them.

LPD always runs the complete set of 60 initial deployments, taking as long as needed. If that takes less than 10 seconds, then it runs additional random initial deployments until 10 seconds have elapsed. The runtime breaks out from the minimum of 10 seconds at about 120 tasks. LPD succeeds in all cases, with a largest runtime of 100 seconds, and a runtime greater than 60 seconds in just 6 cases. The time of the first feasible solution was not recorded, but in all 6 cases, multiple feasible solutions were reached by 100 seconds, so at least one feasible solution was likely available at the 60 second mark and could be used for deployment.

Overall, Fig. 9 shows that LPD scales well with problem size as compared to the MIQP.

Conclusions

The deployments found for the 104 test cases with four clouds and up to 240 deployable units show that:

1. *LPD finds low power deployments*: it matches the minimum found by MIQP for 34% of the comparable cases (where the MIQP solution succeeded within 5 minutes) and is within 5% in another 55% of the cases. Compared to a lower bound which is too low to ever be feasible, it is within 20% in 93% of all the cases.
2. *Solution times are practical*: all are under 100 sec, and less than 20 sec in 79% of the cases. Only 6 cases exceeded 1 minute of solution time. This meets the goal to be fast enough for practical use.
3. *The algorithm scales well*: solution times increase roughly linearly with problem size (in terms of deployable units, including replicas introduced by scaling out to handle high system loads), as shown in Fig. 9.

It was also found that (1) coarsening/uncoarsening of the application graph considerably improves the quality of the LPD solutions and reduces the time to obtain them; (2) random initial deployments provide almost all the final solutions; (3) the execution-time bound of Section 2.4 is essential to obtaining small solution times. Since the bound overestimates the response time, it may eliminate some feasible deployments.

LPD finds a deployment for the maximum expected level of the workload. When the load is lighter, simple autoscaling (which adapts the number of replicas of a task for varying loads) will always be able to meet the SLA requirement, although power optimality is not addressed.

Overall, LPD finds a balance between problem detail, solution accuracy, and run-time.

Threats, issues and extensions

Some aspects of LPD and its evaluation may be, or appear to be, limitations to its usefulness.

An evaluation on an industrial case study might be preferred for evaluation, over the set of randomly created examples used here. However, actual service systems take on a huge variety of sizes, structures and patterns of interaction. The difficulty of obtaining data on real cases would limit their number severely, and a large number of randomly created models spans the range of possibilities better, and gives more confidence that LPD can handle anything that is thrown at it. An even larger number of cases would be even better.

An LQN Application Model is assumed here and could be a burden to create. However it is not actually required, since the input to LPD is the Application Graph, which could be constructed from data obtained by monitoring (that is, from the mean total execution time for each deployable unit, and the mean number of interactions between pairs of units, per User response).

The Users are modeled as a single group. However to model different kinds of users, this group can make a variety of request types to the system.

The response time constraint applies to predicted mean delays, while SLAs often constrain percentiles of delay. To accommodate a target percentile, the measured shape of the response distribution can provide the ratio of the

percentile value to the mean, and the requirement can be translated into a requirement on the mean.

The Power Usage Effectiveness of the clouds could easily be included in the power calculation, to give total power rather than just power used by the application.

Cloud internal network delays are ignored, and may be significant. They can be included in two ways: either as a fixed mean allowance for every interaction, which introduces a constant additional delay for the application, or by modeling the internal cloud network and performing the *Partition* algorithm on it for the final deployment, with bin-packing only on individual hosts. This would give one additional partitioning per cloud.

Storage operations are not mentioned, however they can be modeled as services and deployed by LPD.

The choice of 1 minute as a maximum execution time is arbitrary and could be larger (say, 10 minutes) without changing the determination of the non-scalability of MIQP or meta-heuristics. One minute is an attractive target time for practical use, given that scaling-out operations may take only a few minutes.

Extensions

Extensions that could improve LPD are (1) improved heuristics for the initial deployments, and (2) concurrent execution of Stage 5 of LPD. Also the LPD *Partition* algorithm could be exploited for feasibility repair in a Genetic Algorithm. The *Partition* algorithm determines a sequence of moves that respect capacity constraints and decrease response time and power, with initial priority to response time. It would be terminated at the first solution that is feasible for response time.

Limitations of the approach that would require substantial extensions are: (1) the Users all have the same network delays to access any cloud, and the same response requirement; (2) execution of each operation is sequential within a single response (parallelism is ignored); (3) limited concurrency within a task is not modeled; (4) host and network failures are ignored.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Grant numbers (RGPIN): 06274-2016 and 2014-04056.

Authors' contributions

Babneet Singh: LPD design and evaluation experiments. Ravneet Kaur: design of an earlier version of LPD called Hasrut. Murray Woodside: co-supervision and discussion. John Chinneck: co-supervision and discussion. All authors: writing and reviewing the manuscript. The authors read and approved the final manuscript.

Authors' information

Babneet Singh received his MASc in Computer Engineering from Carleton University in 2019. Currently, he is employed by IBM and works on Java runtime development and cloud optimizations. He has contributed to open-source projects such as Eclipse OpenJ9 (Java Virtual Machine) and Eclipse OMR (language runtime framework).

Ravneet Kaur received her M.A.Sc. from Carleton University in 2016. Currently, she is employed by Phreesia Inc. and leads the development teams working on API based integration of electronic medical records(EMR) systems and Phreesia's SAAS based in-house patient intake management system.

Murray Woodside received his PhD in Control Systems from Cambridge University in 1964 and has done research on control systems, system optimization, queueing theory and performance modeling at the National Research Council of Canada and at Carleton University. He is currently active in retirement at Carleton with research interests in cloud system performance model extraction, estimation, and optimization.

John Chinneck received his PhD in Systems Design from the University of Waterloo in 1983. His research focus is optimization, including improved algorithms and analysis of issues such as infeasibility. His algorithms are in use in most commercial and open-source optimization solvers. He continues to research the application of optimization techniques in a variety of areas including data classification, data compression, task assignment in cloud computing, etc.

Funding

Natural Sciences and Engineering Research Council of Canada (NSERC) grant numbers (RGPIN): 06274–2016 and 2014–04056.

Availability of data and materials

The models, code and analysis are found in https://github.com/JChinneck/LPD.

Declarations

Competing interests

The authors declare that they have no competing interests.

Author details

[1]IBM, Ottawa, Canada. [2]Phreesia, Inc., Brampton, Canada. [3]Department of Systems and Computer Engineering, Carleton University, Ottawa K1S 5B6, Canada.

Received: 18 February 2022 Accepted: 1 October 2022
Published online: 03 January 2023

References

1. Christensen HI, Khan A, Pokutta S, Tetali P (2017) Approximation and online algorithms for multidimensional bin packing: a survey. Comp Sci Rev 24:63–79
2. Arroba P, Moya J, Ayala J, Buyya R (2017) Dynamic voltage and frequency scaling-aware dynamic consolidation of virtual machines for energy efficient cloud data centers. Concur Comput Pract Exper 29 (10), https://doi.org/10.1002/cpe.4067
3. Verba N (2019) Application deployment framework for large-scale fog computing environments, PhD Thesis. Coventry University, Coventry, England
4. Zhang J, Huang H, Wang X (2016) Resource provision algorithms in cloud computing: a survey. J Netw Comput Appl 64:23–42
5. Helali L, Omri MN (2021) A survey of data center consolidation in cloud computing systems. Comp Sci Rev 39, 100366
6. Masdari M, Zangakani M (2020) Green cloud computing using proactive virtual machine placement: challenges and issues. Springer J. Grid Computing
7. Skaltsis GM, Shin H-S, Tsourdos A (2021) A survey of task allocation techniques in MAS, pp 488–497 Int. Conf. on Unmanned Aircraft Systems (ICUAS)

8. Malek S, Medvidovic N, Mikic-Rakic M (2012) An extensible framework for improving a distributed software System's deployment architecture. IEEE Trans Softw Eng 38(1):73–100
9. Li J, Woodside M, Chinneck J, Litiou M (2017) Adaptive cloud deployment using persistence strategies and application awareness. IEEE Trans Cloud Comput 5(2):277–290
10. Ciavotta M, Ardagna D, Gibilisco GP (2017) A mixed integer linear programming optimization approach for multi-cloud capacity allocation. J Syst Softw 123:64–78
11. Shu W, Cai K, Xiong NN (2021) Research on strong agile response task scheduling optimization enhancement with optimal resource usage in green cloud computing. Elsevier Future Generation Computer Systems
12. Frincu ME, Craciun C (2011) Multi-objective meta-heuristics for scheduling applications with high availability requirements and cost constraints in multi-cloud environments, pp 267–274 4th IEEE Int Conf on Utility and Cloud Computing
13. Guerrero C, Lera I, Juiz C (2018) Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. J Grid Computing 16:113–135
14. Frey S, Fittkau F, Hasselbring W (2013) Search based genetic optimization for deployment and reconfiguration of software in the cloud, pp 512–521 Proc Int. Conf. on Software Engineering (ICSE '13)
15. Ye X, Yin Y, Lan L (2017) Energy-efficient many-objective virtual machine placement optimization in a cloud computing environment. IEEE Access 5:16006–16020
16. Geurout T, Gaoua Y, Artigues C, Da Costa G (2017) Mixed integer linear programming for quality of service optimization in clouds. Futur Gener Comput Syst 71:1–17
17. Alizadeh R, Nishi T (2019) A genetic algorithm for multi-period location problem with modular emergency facilities and backup services. Trans Inst Syst Contr Inform Eng 32(10):370–377
18. IBM, "IBM CPLEX Optimizer", https://www.ibm.com/analytics/cplex-optimizer, (Accessed 11 May 2021)
19. Fiduccia CM, Mattheyses RM (1982) A linear-time heuristic for improving network partitions, pp 175–181 19th Design Automation Conference
20. Alicherry M, Lakshman T (2012) Network aware resource allocation in distributed clouds, pp 963–971 Proc. IEEE INFOCOM
21. Sachdeva S, Kaur K (2015) ACO based graph partitioning algorithm for optimistic deployment of software in MCC. Proc. Int. Conf. Innovations in information, pp 1–5 Embedded and Communication Systems (ICIIECS 2015)
22. Leivadeas A, Papagianni C, Papavassiliou S (2013) Efficient resource mapping framework over networked clouds via iterated local search-based request partitioning. IEEE Trans Parallel Distrib Syst: vol. 24, no. 6, pp. 1077–1086.
23. Machida F, Kawato M, Maeno Y (2010) Redundant virtual machine placement for fault-tolerant consolidated server clusters, pp 32–39 Network Operations and Management Symposium (NOMS 2010)
24. Ardagna D, Barbierato E, Evangelinou A, Gianniti E, Gribaudo M, Pinto TB, Guimarães A, Couto da Silva AP, Almeida JM (2018) Performance prediction of cloud-based big data applications, pp 192–199 Proc ACM/SPEC Int Conf on Performance Engineering
25. Aldhalaan A, Menascé DA (2015) Near-optimal allocation of VMs from IaaS providers by SaaS providers. Proc. Int. Conf. on Cloud and Autonomic Computing
26. Molka K, Casale G (2017) Energy-efficient resource allocation and provisioning for in-memory database clusters, IFIP/IEEE Symp on integrated network and service management (IM), pp 19–27
27. Wada H, Suzuki J, Oba K, Theoretic Q (2009) Evolutionary deployment optimization with probabilistic SLAs for service oriented clouds, pp 661–669 Proc 2009 Congress on Services
28. Calheiros RN, Ranjan R, Buyya R (2011) Virtual machine provisioning based on analytical performance and QoS in cloud computing environments, pp 295–304 Int. Conf. on Parallel processing (ICPP 2011)
29. Wang Y, Xia Y (2016) Energy optimal VM placement in the cloud, pp 84–91 IEEE 9th Int Conf on Cloud Computing
30. Tunc C, Kumbhare N, Akoglu A, Hariri S, Machovec D, Siegel HJ (2016) Value of service based task scheduling for cloud computing systems, pp 1–11 Int Conf on Cloud and Autonomic Computing (ICCAC)
31. Chen K-Y, Xu Y, Xi K, Chao HJ (2013) Intelligent virtual machine placement for cost efficiency in geo-distributed cloud systems. IEEE Int Conf Commun:3498–3503
32. Arcangeli J-P, Boujbel R, S. (2015) Leriche automatic deployment of distributed software systems. J Syst Softw 103(C):198–218
33. Kaur R (2015) Lightweight robust optimizer for distributed application deployment in multi-clouds, MASc thesis. Carleton Unversity, Ottawa, Canada
34. Singh B (2019) Multi-cloud application deployment, MASc thesis. Carleton University, Ottawa, Canada
35. Menegola B (2012) A study of the k-way graph partitioning problem. Federal University of Rio Grande do Sul, Rio Grande do Sul
36. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
37. Karypis G, Kumar V (1998) Multilevel k-way partitioning scheme for irregular graphs. J Parallel Distributed Comput 48(1):96–129
38. Standard Performance Evaluation Corporation, SPECpower_ssj2008 Result File Fields, https://www.spec.org/power/docs/SPECpower_ssj2008-Result_File_Fields.html. (Accessed 6 Dec 2018)
39. Franks G, Al-Omari T, Woodside M (2009) Enhanced modeling and solution of layered Queueing networks. IEEE Trans Software Eng 35(2):148–161
40. Woodside M, Tutorial Introduction to Layered Modeling of Software Performance, http://www.sce.carleton.ca/rads/lqns/lqn-documentation/tutorialh.pdf (Accessed Sept 2021)
41. Islam F, Petriu D, Woodside M (2015) Simplifying layered queuing network models. In: Beltrán M, Knottenbelt W, Bradley J (eds) Computer performance engineering (EPEW 2015), vol 9272. Springer LNCS
42. Yuventi J, Mehdizadeh R (2013) A critical analysis of power usage effectiveness and its use in communicating data center energy consumption. Energy Buildings 64:90–94
43. Bondi AB (2014) Foundations of software and system performance engineering. Addison-Wesley
44. Han G, Que W, Jia G, Zhang W (2018) Resource-utilization-aware energy efficient server consolidation algorithm for green computing in IIOT. J Netw Comput Appl 103:205–214
45. Franks G, lqngen — generate layered queueing network models. https://github.com/layeredqueuing/V5 (Accessed 10 Feb 2022)
46. Verbelen T, Stevens T, De Turck F, Dhoedt B (2013) Graph partitioning algorithms for optimizing software deployment in mobile cloud computing. Futur Gener Comput Syst 29(2):451–459

Publisher's Note