Les Cahiers du GERAD ISSN: 0711–2440

Low-power deployment of many-process applications on multiple clouds

B. Singh, R. Kaur,
M. Woodside, J.W. Chinneck

G–2019–101

December 2019

La collection *Les Cahiers du GERAD* est constituée des travaux de recherche menés par nos membres. La plupart de ces documents de travail a été soumis à des revues avec comité de révision. Lorsqu'un document est accepté et publié, le pdf original est retiré si c'est nécessaire et un lien vers l'article publié est ajouté.

Citation suggérée : B. Singh, R. Kaur, M. Woodside, J.W. Chinneck (Décembre 2019). Low-power deployment of many-process applications on multiple clouds, Rapport technique, Les Cahiers du GERAD G–2019–101, GERAD, HEC Montréal, Canada.

Avant de citer ce rapport technique, veuillez visiter notre site Web (https://www.gerad.ca/fr/papers/G-2019-101) afin de mettre à jour vos données de référence, s'il a été publié dans une revue scientifique.

The series *Les Cahiers du GERAD* consists of working papers carried out by our members. Most of these pre-prints have been submitted to peer-reviewed journals. When accepted and published, if necessary, the original pdf is removed and a link to the published article is added.

Suggested citation: B. Singh, R. Kaur, M. Woodside, J.W. Chinneck (December 2019). Low-power deployment of many-process applications on multiple clouds, Technical report, Les Cahiers du GERAD G–2019–101, GERAD, HEC Montréal, Canada.

Before citing this technical report, please visit our website (https://www.gerad.ca/en/papers/G-2019-101) to update your reference data, if it has been published in a scientific journal.

La publication de ces rapports de recherche est rendue possible grâce au soutien de HEC Montréal, Polytechnique Montréal, Université McGill, Université du Québec à Montréal, ainsi que du Fonds de recherche du Québec – Nature et technologies.

Dépôt légal – Bibliothèque et Archives nationales du Québec, 2019
– Bibliothèque et Archives Canada, 2019

The publication of these research reports is made possible thanks to the support of HEC Montréal, Polytechnique Montréal, McGill University, Université du Québec à Montréal, as well as the Fonds de recherche du Québec – Nature et technologies.

Legal deposit – Bibliothèque et Archives nationales du Québec, 2019
– Library and Archives Canada, 2019

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Low-power deployment of many-process applications on multiple clouds

Babneet Singh [a]

Ravneet Kaur [a]

Murray Woodside [a]

John W. Chinneck [a,b]

[a] *Systems and Computer Engineering, Carleton University, Ottawa (Ontario), Canada K1S 5B6*

[b] *GERAD, HEC Montréal, Montréal (Québec), Canada, H3T 2A7*

babneetsingh@cmail.carleton.ca
dhindsa.ravneet@gmail.com
cmw@sce.carleton.ca
chinneck@sce.carleton.ca

December 2019
Les Cahiers du GERAD
G–2019–101

Copyright © 2019 GERAD, Singh, Kaur, Woodside, Chinneck

Les textes publiés dans la série des rapports de recherche *Les Cahiers du GERAD* n'engagent que la responsabilité de leurs auteurs. Les auteurs conservent leur droit d'auteur et leurs droits moraux sur leurs publications et les utilisateurs s'engagent à reconnaître et respecter les exigences légales associées à ces droits. Ainsi, les utilisateurs:

- Peuvent télécharger et imprimer une copie de toute publication du portail public aux fins d'étude ou de recherche privée;
- Ne peuvent pas distribuer le matériel ou l'utiliser pour une activité à but lucratif ou pour un gain commercial;
- Peuvent distribuer gratuitement l'URL identifiant la publication.

Si vous pensez que ce document enfreint le droit d'auteur, contactez-nous en fournissant des détails. Nous supprimerons immédiatement l'accès au travail et enquêterons sur votre demande.

The authors are exclusively responsible for the content of their research papers published in the series *Les Cahiers du GERAD*. Copyright and moral rights for the publications are retained by the authors and the users must commit themselves to recognize and abide the legal requirements associated with these rights. Thus, users:

- May download and print one copy of any publication from the public portal for the purpose of private study or research;
- May not further distribute the material or use it for any profit-making activity or commercial gain;
- May freely distribute the URL identifying the publication.

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Abstract: Deploying applications having many processes in a multi-cloud environment is challenging. The cloud manager has goals such as minimizing power consumption, while the application manager must reserve processing and memory resources sufficient to satisfy user constraints on throughput and response time. These challenges must be addressed together. We describe an algorithm for this purpose, intended for use by cloud managers who must simultaneously manage hosts and application deployments, and plan capacity in order to offer services such as Serverless Computing. The algorithm combines queueing theory, clustering, graph partitioning and bin-packing strategies to solve the low-power many-process multi-cloud application deployment problem. It produces a solution within 20 seconds in 90% of the test scenarios, which is quick enough for use in practice. The algorithm is extremely effective: in 77% of the test scenarios the solution power consumption is within 10% of an unachievable theoretical lower bound.

Keywords: Multi-cloud, software, performance, layered queueing network, graph partitioning, bin packing, green computing

Acknowledgments: We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Grant numbers (RGPIN): 06274–2016 and 2014–04056.

1 Introduction

Many-process applications with several or many communicating concurrent services are increasingly common (e.g., systems with containers such as Docker may have dozens of these). While it may be preferable to deploy such an application in a single cloud, it may be necessary to use multiple clouds to obtain sufficient processing resources, or to access data sets located on different clouds.

Deploying across multiple clouds is challenging for the following reasons:

1. Network latencies: Inter-cloud network latencies are significant and the number of network hops in a single response depends on the deployment. This can severely impact the response time.
2. Heterogeneous clouds: Clouds have different total capacities and hosts with different power and speed characteristics.
3. Performance requirements: Throughput and response time have nonlinear dependencies on the allocation. For example, a network delay is part of the response time only if the sender and receiver are deployed in different clouds.

The goal of this work is an algorithm for generating power-efficient multi-cloud deployments that satisfy cloud capacity and space constraints and throughput and response time constraints that are typical of service level agreement (SLA) performance requirements. The Low Power Multi-Cloud Application Deployment (LPMCAD) algorithm takes an application in the form of a directed graph (Figure 1a) as input, and outputs a power-efficient deployment (Figure 1b) of application components to hosts on clouds. It is quick enough for use in practice and effective in finding a low power deployment.

LPMCAD is the first algorithm to solve the LPMCAD problem (see Section 1.2). This algorithm has a novel heuristic optimization procedure which combines a bin-packing and graph partitioning algorithm to satisfy the SLA and reduce power consumption in the same step (see Section 3.5). Another novelty is the response time approximation which allows the algorithm to quickly generate a solution (see Section 2.6).

(a) Application **(b) Deployment**

Figure 1: Example of an application and deployment

1.1 Background: Graph partitioning, bin packing and layered queueing networks (LQN)

LPMCAD uses K-way graph partitioning to reduce network hops, two-dimensional bin packing to ensure capacity constraints, and queueing models to ensure performance constraints.

In K-way graph partitioning, a graph is divided into K parts, where each part is a subset of the graph node set, and the parts are non-intersecting and cover the entire node set. A common objective is to minimize the sum of the edge weights that connect nodes in different parts. Graph partitioning is NP-hard [1], so it is difficult to find optimum solutions in a reasonable amount of time. Approximate solutions are provided more quickly by heuristics. Menegola et al. [1] survey K-way graph partitioning

techniques including integer programming, heuristic techniques, multilevel techniques and multi-start techniques.

Two-dimensional (2-D) bin packing tries to pack a set of 2-D items into a minimum number of 2-D bins. It is also NP-hard [2] and has applications in cutting stock, vehicle loading, pallet packing, memory allocation, and several other logistics and robotics related problems [2]. Surveys of 2-D bin packing algorithms are given in [2] and [3].

We model the software application via a layered queueing network (LQN) [4] which combines a high-level model of the software architecture with deployment and performance information. Its parameters capture the workload executed by the system in terms of CPU and I/O demands and network messaging. The LQN solution estimates the queuing delays due to congestion in both hardware and software resources, statistics that are vital to accurate prediction of system performance. LQN modeling allows the study of throughput, response times, wait times, resource utilization and other performance characteristics of software systems. Results are representative of real-world statistics.

1.2 Related work

There is a host of work on cloud deployment, indicated by keywords such as resource allocation, host assignment, task scheduling, host consolidation, virtual machine (VM) placement and resource provisioning. Different approaches emphasize different concerns, but no existing work addresses all these concerns simultaneously:

1. Can it handle multiple clouds of different sizes?
2. Can it handle hosts with different processing and memory capacities?
3. Does it account for the network latencies between the clouds?
4. Does it handle distributed applications with communication between application components?
5. Does it satisfy the application's processing and memory requirements?
6. Does it enforce performance requirements (throughput or response time) as constraints?
7. Does it minimize power consumption?
8. Is the solution time small enough to be useful in practice? For example, can the algorithm find a solution in less than two minutes?

Verba [5] proposes a framework for large-scale fog computing which allocates highly connected applications to a set of "gateway", devices having computing and networking capabilities. It uses graph clustering methods to allocate while maximizing a utility function which has components for delay, reliability and constraint violations. It does not address power consumption.

Molka and Casale [6] allocate resources and hosts in a single cloud while satisfying the response time for a given set of in-memory databases using an efficient hybrid genetic algorithm, bin-packing and nonlinear optimization. They do not address multiple clouds, inter-cloud network delays or power.

Tunc et al [7] address most of the concerns, but only for a single cloud. They use a real-time control rule based on performance monitoring and a Value of Service metric that enforces the response time constraint as a deadline, while accounting for energy consumption.

Wang et al. [8] propose an energy-optimizing strategy for VM placement in the cloud. VMs are considered components of an application or independent tasks. A cubic power function is used to evaluate the power consumption of VMs: $P(f) = \alpha + \beta f^3$, where α is the static power consumption of the CPU, β is a constant coefficient, and f is the operating frequency of the CPU. In their solution, the processing and memory constraints are satisfied. The problem is modelled as a mixed integer program.

Panda et al. [9] propose three task scheduling algorithms for a heterogeneous multi-cloud environment. The task scheduling algorithms aim to minimize the makespan and maximize the average cloud utilization. Minimizing the makespan means minimizing the overall completion time needed to

execute all the application tasks on the available clouds. The solution depends on the cloud manager to handle the computation resources for the application.

Kaur [10] describes the HASRUT algorithm, which addresses all the concerns above but for only two clouds. It uses a combination of multi-level, multi-start and local search graph partitioning techniques. The work described here is a modification and extension of HASRUT.

Frincu et al. [11] propose a multi-cloud resource provisioning algorithm to maximize resource usage, minimize application run-time, and maximize application availability and fault-tolerance. The solution uses a genetic algorithm, which can be slow, to satisfy multiple objectives when finding a deployment in a multi-cloud environment. Network latencies and energy consumption are ignored.

Table 1 shows which concerns are addressed by each of these methods. No method covers all concerns.

Table 1: Summary of related work

Concern	Verba 2019	Molka 2017	Tunc 2016	Wang 2016	Panda 2015	Kaur 2015	Frincu 2011
1. Multiple (>2) heterogeneous clouds	Y	N	N	N	Y	N	Y
2. Heterogeneous hosts	Y	U	U	Y	U	N	N
3. Network latencies	Y	N	N	N	N	Y	N
4. Distributed applications	Y	N	N	N	Y	Y	Y
5. Processing and memoryrequirements	Y	Y	Y	Y	N	Y	Y
6.Performance requirements	Y	Y	Y	N	Y	Y	Y
7. Reduce power consumption	N	N	Y	Y	N	Y	N
8. Small solution time	Y	N	Y	Y	U	Y	N

"Y": covered. "N": not covered. "U": uncertain.

2 Modelling multi-cloud application deployments

2.1 Multi-cloud model

A cloud is a group of hosts with a connecting internal network and peripherals. Different clouds usually have different amounts and types of computing resources. In the multi-cloud model, there are multiple heterogeneous clouds having varying computing resources. Each cloud has different network latencies with respect to a group of users and to the other clouds. Figure 2 shows a multi-cloud environment. It consists of two types of entities: a cloud and user. It also specifies the cloud properties and the network latency between the entities. We ignore the intra-cloud networking delays and cloud disk storage capacity and delays. The labels on the cloud model are defined in Section 2.5.

2.2 AppModel: The application Model

Each application is represented by a model called (for brevity) an AppModel, which describes the components and their interactions using the notation of layered queueing models [12] as illustrated in Figure 3.

The AppModel in Figure 3 shows the components (including the group of users) as rectangles (e.g. t1) with their interface operations as attached rectangles (e.g. e2601). In LQN components are called tasks and operations are called entries. The entry is labelled by its mean CPU demand per invocation, for example 7.31 *msec* for entry e2604. Calls to entries are shown as arrows, labeled by the mean calls per invocation of the calling entry, for example 2.31 calls from entry e2601 to e2607. The CPU demand is the time in *msec* as calibrated on a particular reference host type and is scaled for a faster or slower host by a speed factor for the host. One or more replicas of each application task will be deployed, each with its own VM.

We attach two parameters to the users (*userTask*): a rate τ for the total requests per second, and a required maximum mean response time $R^{constraint}$.

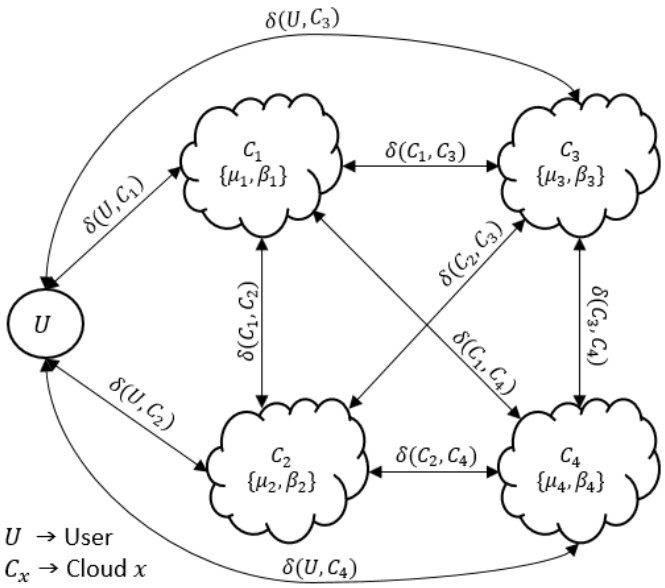

Figure 2: Multi-cloud model

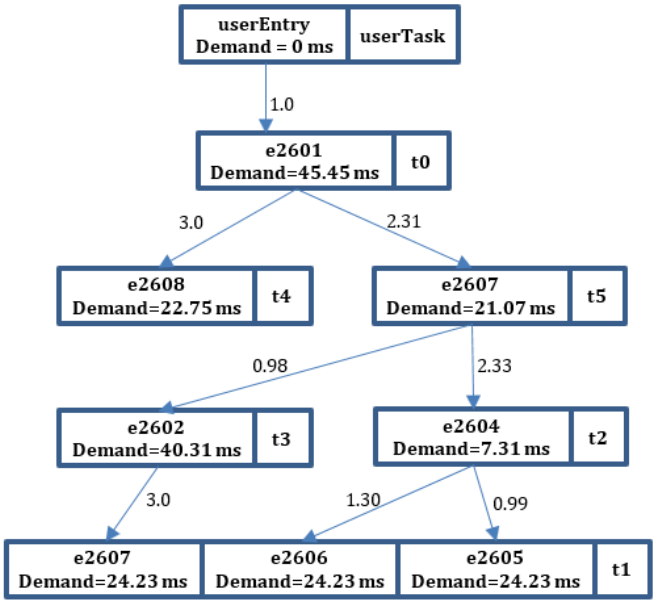

Figure 3: An example AppModel

2.3 Application graph

The deployment algorithm operates on a graph model G which can be derived from the AppModel, or directly from the system data. Each vertex in G represents a deployable unit (corresponding to a VM in the software and to a task x in the AppModel), and each edge in G represents all the interactions between a pair of tasks x and y (corresponding to the set of interactions between the task entries). Each vertex has the properties $p^{task}(x)$, the total average processing capacity requirement of that task converted to units of *ssj_ops* (java operations per second defined by the SPECPower benchmark, and $m(x)$, its memory requirement for deployment on a host, and each edge has a property $c(x, y)$ for the mean number of interactions between entries x and y, during one user response.

The graph G can be derived from an AppModel by a process described in Chapters 3 and 4 of [13]. For each entry the mean invocations per user response is found by analyzing the calling rates between tasks in the AppModel, and for each task the CPU time d is the weighted sum of the entry demands, as calibrated on the reference host type. Then, for user throughput τ,

$$p^{task}(x) = \tau \times (\text{CPU time } d \text{ for task } x) \times (ssj\_ops \text{ for the reference host type}) \quad (1)$$

In the same way $c(x, y)$ is the weighted sum of the calls from each entry of x to all the entries of y, weighted by the invocations of x per user response.

2.4 Power model

Power consumption is modelled as a function of the host's throughput using the SPECpower benchmark results. In the power model shown in Figure 4 [14], the y-axis represents the average active power in Watts (W) and the x-axis represents the SPECpower benchmark throughput in workload operations per second (*ssj ops*). A polynomial regression model of degree three is applied to the SPECpower benchmark results.

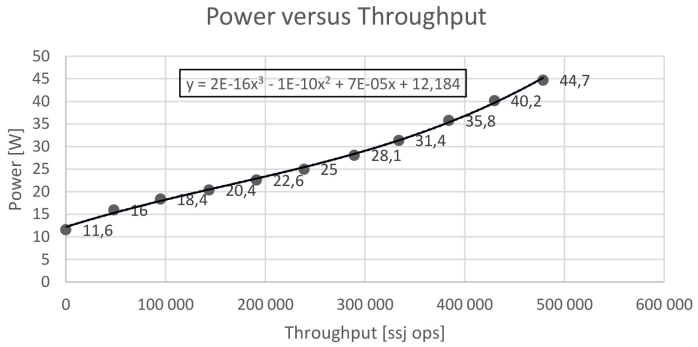

Figure 4: Power versus throughput for Fujitsu Server PRIMERGY TX1320 M2

Power Usage Effectiveness (PUE) relates host power to cloud power. It is the ratio between the total power consumed by a cloud and power consumed by the cloud's hosts [15]. We assume a constant PUE of 1 across all clouds for simplicity, so the power consumed by a cloud's hosts is equal to the total power consumed by a cloud.

2.5 Notation

2.5.1 Application graph

1. V: set of all application tasks (vertices in the application graph).
2. E: set of all calls between the application tasks (edges in the application graph).
3. $X = \{x_1,\ x_2,\ \ldots\}$: set of application tasks.
4. $p^{task}(x)$: processing requirement of application task x in operations/sec as defined for *ssj ops*, at user throughput τ responses/sec. (see Section 2.3).
5. $m^{task}(x)$: memory requirement of application task x in megabytes (MB).
6. $e(x)$: total processing (execution) time of application task x on a standard processor in *msec*, including waiting for the processor.
7. $c(x, y)$: mean number of calls of calling application task x to application task y, per user response.
8. $G = (V,\ E,\ p^{task},\ m^{task},\ c)$: application modelled as a directed graph.

2.5.2 Clouds and hosts

1. $C = \{C_0, C_1, C_2, \ldots, C_K\}$: set of K clouds in the multi-cloud environment, where C_0 represents the user(s).
2. h_{kl}: host l on cloud k. Unsubscripted h is an unspecified host.
3. $H_k = \{h_{k1}, h_{k2,} \ldots h_{kL_k}\}$: set of hosts in cloud C_k, which has L_k hosts.
4. $p^{used}(h)$ and $p^{total}(h)$: used and total processing capacity of host h in *ssj ops* respectively.
5. $m^{used}(h)$ and $m^{total}(h)$: used and total memory capacity of host h in *MB* respectively.
6. s_h: speed factor for host h, defined as the ratio of its speed in *ssj_ops*, relative to the reference host type. Then the processing requirement of task x deployed on host h is $p^{task}(x)/s_h$.
7. $\mu(k)$: total processing capacity of the cloud C_k in *ssj ops*.
8. $\beta(k)$: total memory capacity of the cloud C_k in *MB*.
9. $\delta(r, q)$: delay between two cloud-entities r and q in *msec*.

2.5.3 Deployment

1. X_{kl}: set of tasks deployed to host l on cloud k, a partition of X.
2. $D_k = \{X_{k1}, X_{k2}, \ldots, X_{k,L_k}\}$: deployment to cloud C_k .
3. $D = \{D_0, D_1, D_2, \ldots, D_K\}$: deployment to K clouds, $C_1, C_2, \ldots, C_K$, where D_0 contains only the pre-assigned *userTask*.
4. $pwr(h_{kl})$: power consumed by host h_{kl} when it is assigned a set of tasks X_{kl}.
5. ψ_D: cut-set of deployment D.
6. ω_D: total power consumption of deployment D in Watts (W).
7. $f(x)$: cloud-entity to which application task x is assigned. The cloud-entity can be a cloud or the set of users.
8. R_D^{pt}: total processing time of deployment D in *msec*.
9. R_D^{net}: total network delay due to inter-cloud communication of deployment D in msec.
10. R_D^{total}: response time of deployment D in *msec*.
11. τ: throughput constraint in requests per millisecond ($req/msec$).
12. $R^{constraint}$: response time constraint in milliseconds ($msec$).
13. U^{nom}: nominal resource utilization applied to all hosts for the response time approximation (see next).

2.6 Response time approximation

A simplifying assumption avoids solving the LQN model for every trial deployment. It has two parts:

1. It is assumed that the deployment has a target utilization for all hosts, set at U^{nom}, equal to the fraction of the resource that is utilized. This is a reasonable assumption since one goal of deployment is efficient use of the processors, and their power efficiency is usually better at higher utilization, thus the optimization will tend to pack the processors to give this utilization.
2. The queueing delay at each host processor h is approximated by a simple processor-sharing calculation for a single processor with the total capacity of the host and with this utilization. The execution time (waiting plus execution) per user response is then [16]:

$$e(x) = \left(\frac{d}{1 - U^{nom}}\right) \tag{2}$$

Service time of task x per user response,

$$d = \left(\frac{p^{task}(x)}{\tau \times Speed\ of\ host\ in\ ssj\_ops}\right) \tag{3}$$

This quick calculation is used during optimization to approximately enforce the response time constraint. It tends to understate the delay, so a solution may give a response time (found by LQNS) greater than $R^{constraint}$. This can be corrected by setting a factor α^{wait} to their ratio, reducing $R^{constraint}$ by the factor α^{wait}, and re-optimizing. Alternatively, our experience with response-time accuracy, reported in Section 4.7.4 below, suggests that a fixed value $\alpha^{wait} = 1.10$ applied to all cases would eliminate this problem in most cases.

2.7 Optimization model

A mathematical model of the power-efficient multi-cloud application deployment problem follows:

1. The tasks assigned to a host should not exceed a host's processing and memory capacities, so for each host h:

$$\sum_{x \in X_h} p^{task}(x) \leq p^{total}(h) \tag{4}$$

$$\sum_{x \in X_h} m^{task}(x) \leq m^{total}(h) \tag{5}$$

2. A deployment is a partition of G, that is every task is assigned to exactly one cloud. Equation 6 requires that all the tasks have been assigned to a host in the deployment. Equation 7 requires that a task is assigned to only one cloud i.e. there is no overlap between the deployment subsets.

$$\bigcup_{1 \leq k \leq K} D_k = V \tag{6}$$

$$D_i \cap D_j = \varnothing,\ \forall i \neq j \tag{7}$$

3. A deployment D to the multi-cloud C is a collection of K subsets of $V : D = \{D_0, D_1,\ D_2,\ \ldots,\ D_K\}$, where each subset is the set of tasks assigned to a cloud. The cut-set of a deployment D is the set of edges that connect between clouds or users via a network:

$$\psi_D = \{\{x,\ y\} \in E \mid x \in D_i,\ y \in D_j,\ i \neq j,\ \forall i \in [0, K],\ \forall j \in [0, K]\} \tag{8}$$

4. The total network delay (ignoring delays within each cloud) is the cut-weight or cut-size [1] for a deployment D:

$$R_D^{net} = \sum_{\{x,y\} \in \psi_D} c(x, y)\delta\big(f(x), f(y)\big) \tag{9}$$

5. The total power consumption for a deployment D is:

$$\omega_D = \sum_{k=1}^{K} \sum_{l=1}^{L_k} pwr(h_{kl}) \tag{10}$$

6. The total processing time for a deployment D is (using $e(x)$ from Equation 2):

$$R_D^{pt} = \alpha^{wait} \sum_{x \in V} e(x) \tag{11}$$

7. The total response time for a deployment D is taken to be:

$$R_D^{total} = R_D^{pt} + R_D^{net} \tag{12}$$

The optimization problem follows below. Equation 13 specifies the minimum power objective function for the multi-cloud application deployment problem. Equation 14 ensures that the deployment satisfies

the response time constraint. Equations 4 and 5 indirectly ensure that the total processing and memory capacities of a cloud are not exceeded.

$$\text{Find } D \text{ to minimize: } \omega_D \tag{13}$$

$$\text{Subject to: } R_D^{\text{total}} \leq R^{\text{constraint}} \tag{14}$$

Equations (2) through (12)

3 Low-power multi-cloud application deployment (LPMCAD) algorithm

Algorithm 1 provides an overview of the LPMCAD algorithm. It uses a combination of clustering, multi-level graph partitioning, multi-start graph partitioning, graph partitioning local search and bin-packing techniques. It has six core stages and two enhancements, described next. Figure 5 provides a visual aid to illustrate each stage and the transition between stages.

Algorithm 1 Overview of the low-power multi-cloud application deployment (LPMCAD) algorithm

INPUTS: clouds, hosts, users, application, control parameters, delays (δ).
Stage 1: Select a subset of clouds (C) which could host the application.
Stage 2: Generate a graph for the application (G) from the architecture model.
Stage 3: Scale the application to the workload and generate a graph for the scaled application (G^{scaled}).
Enhancement 1: Coarsen G^{scaled} and generate a graph of the coarsened application (G^{coarse}).
for $i \leftarrow 1$ to number of initial deployments (N^{init_depl}), **do**
Stage 4: Generate an initial deployment (D^{init}).
Stage 5: Partition G^{coarse} using D^{init}.
If the partitioned deployment ($D^{partition}$) does not satisfy $R^{constraint}$, then record $D^{partition}$ and go to the next iteration in the loop.
Enhancement 1: Perform uncoarsening on $D^{partition}$.
for each bin packing algorithm {HBF-Proc, HBF-Mem}, **do**
Enhancement 2: Perform bin packing on the uncoarsened partitioned deployment ($D^{partition'}$) and record the deployment ($D^{HBF\text{-}Proc}$ or $D^{HBF\text{-}Mem}$).
end for
end for
OUTPUTS: (*Stage 6*): D that satisfies $R^{constraint}$ and consumes the least power, or null on failure.

3.1 Stage 1: Cloud selection

When there are many clouds, a preliminary choice of which clouds to consider for deployment must be made. It is straightforward to eliminate from consideration clouds that are at capacity or that have latencies to the user that are beyond the permitted maximum. We defer the details of the cloud selection process to future work. Without loss of generality, the algorithm is tested against a four-cloud configuration of an edge, small, medium and large cloud, which vary in computing resources. The clouds are positioned such that the largest cloud has the greatest network delay to the user and smallest cloud has the least network delay to the user. This is similar to a fog computing environment since it employs a multi-cloud architecture where the clouds are tiered based on size.

3.2 Stage 2: Generating the application graph

Refer to Section 2.3.

3.3 Stage 3: Scaling the application

The application is scaled by adding enough replicas to satisfy the throughput requirement using processors whose utilization is bounded at U^{nom}. For each task x the number of replicas, $N^{replicas}$, is calculated so that the task can be executed on the slowest host found across all clouds (with speed $s^{\min}$). This gives for task x:

$$N^{replicas} = \left\lceil \frac{\tau Y_x p(x)}{U^{nom} s^{\min}} \right\rceil \tag{15}$$

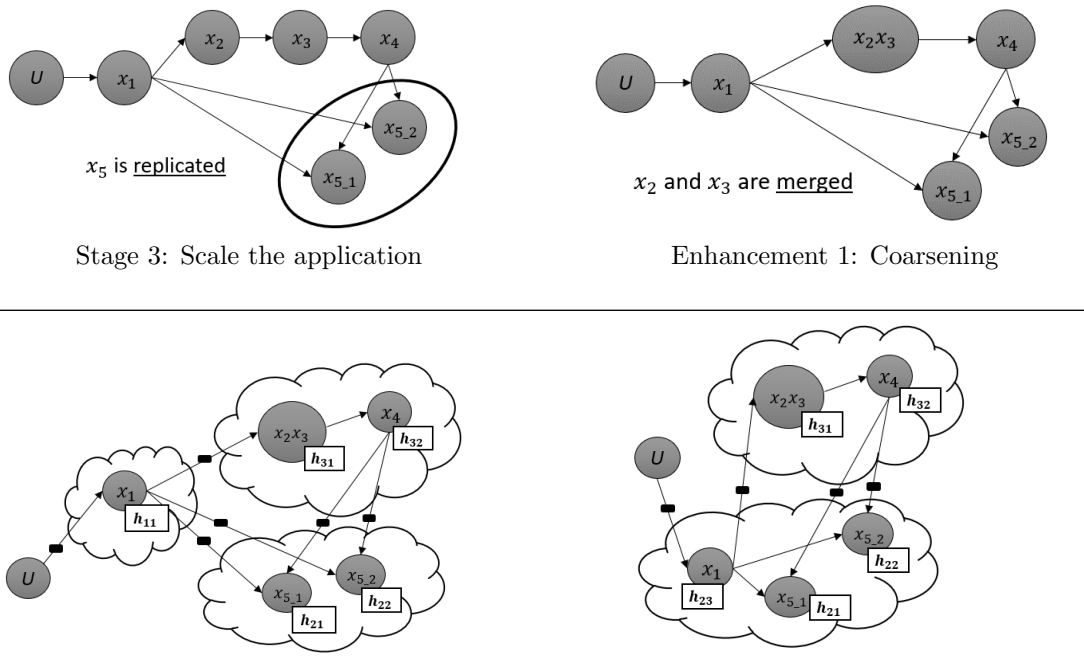

Figure 5: Visual summary of each stage of the LPMCAD algorithm

The mean number of calls to x per user request (Y_x) is distributed equally among the replicas (this is done in practice by a load balancer). In the model the call from any task to a set of replicas is replaced by a set of calls to replicas with the number of calls divided equally among them. A new application graph G^{scaled} is created for the replicas. This graph is then optionally coarsened (see Section 3.6.1) to reduce the number of nodes.

3.4 Stage 4: Initial deployments

An initial deployment assigns the tasks to the hosts on the clouds as a starting point for the partitioning stage. It is not required to be power efficient nor to satisfy the response time constraint, but it must satisfy the processing and memory limits on each host. Three initial deployment strategies are employed: deploy on the power efficient hosts first (*power-sensitive*, Algorithm 2), deploy on the next closest cloud first (*delay-sensitive*, Algorithm 3) and deploy randomly on clouds (*random*, Algorithm 4). The first two strategies generate one initial deployment each while the third strategy generates multiple initial deployments. All three strategies take a directed graph and a list of clouds as the input, and deploy to the most power-efficient hosts first. The output initial deployment is a list of entries for each cloud; each entry lists the task, host, and power consumed. When a task is assigned to a cloud, a host is allocated, and the power consumption is calculated for the task.

3.4.1 Find the most power efficient host for a task in a cloud

The hosts in each cloud are pre-sorted in decreasing order of the *overall score* of the SPECPower benchmark, which is the ratio of throughput and power, representing the power efficiency [17]. First-Fit-Decreasing-Height (FFDH) bin packing [3] is used to find a host for a task in a cloud, where the

Algorithm 2 Power-sensitive initial deployment

INPUTS: G, C.
  Instantiate D with no elements.
  Sort the tasks in G in decreasing order of processing requirement.
  for each task in the sorted set, **do**
    Find the most power efficient host that can accept the task and its associated cloud.
    if a host is found, **then**
      add the task to X_{host}, X_{host} to D_{cloud} and D_{cloud} to D if not already present.
    else
      fail since no host with enough resources was found to allocate the task.
    end if
  end for
OUTPUTS: D on success and null on failure.

Algorithm 3 Delay-sensitive initial deployment

INPUTS: G, C, δ.
  Instantiate D with no entries.
  $C^{to} \leftarrow \emptyset$. // next cloud to use
  Sort the tasks in G in decreasing order of processing requirement.
  for each task in the sorted set, **do**
    if C^{to} is $\emptyset$, then: **then**
      Find the cloud with least delay to the user that can accept the task; set as C^{to}.
    else
      if C^{to} cannot accept the task **then**
        Find the cloud with least delay to C^{to} that can accept the task, and update C^{to}.
        if C^{to} is $\emptyset$, **then**
          fail (no host has enough resources to accept the task).
        end if
      end if
    end if
    Find the most power efficient host in C^{to} that can accept the task.
    Add the task to X_{host}, X_{host} to $D_{C^{to}}$ and $D_{C^{to}}$ to D if not already present.
  end for
OUTPUTS: D on success and null on failure.

Algorithm 4 Random initial deployment

INPUTS: G, C.
  Instantiate D with no entries.
  Shuffle list of tasks in G to generate a randomly ordered list.
  for each task in the list, **do**
    Find the most power efficient host in a randomly selected cloud that can accept the task.
    if a host is found, **then**
      add the task to X_{host}, X_{host} to D_{cloud} and D_{cloud} to D if not already present.
    else
      fail since no host with enough resources was found to accept the task.
    end if
  end for
OUTPUTS: D on success and null on failure.

"height" is the processing requirement of the task. The combination of pre-sorting the hosts in a cloud based upon power efficiency and utilizing the FFDH bin packing to fit tasks to hosts is a fast method for host allocation that improves the power efficiency of the deployment while satisfying the task processing and memory requirements.

3.5 Stage 5: Partition to satisfy the response time and bin-pack to minimize the power consumption

The goal in Stage 5 is to find a more power efficient deployment that satisfies $R^{constraint}$ by moving tasks from one cloud to another. Moves are evaluated by the amount of reduction in R_D^{net} and ω_D. The K-way variant of the Fiduccia-Mattheyses (FM) graph partitioning refinement heuristic [1] is employed for satisfying $R^{constraint}$ since Kaur [10] achieved the best results with it while solving the edge-core (two-way) cloud deployment problem. The FM heuristic primarily reduces the cut-weight of

a graph, which is equivalent to R_D^{net} in this context. Ultimately, R_D^{total} is lowered due to the reduction in R_D^{net}. Further, the techniques in section 3.4.1 reduce ω_D by selecting the most power efficient hosts on a cloud during a move. The move selection process initially focusses on satisfying $R^{constraint}$ while ignoring the changes to ω_D. After $R^{constraint}$ is satisfied (Equation 14), then it prioritizes reducing ω_D while not violating $R^{consraint}$. The detailed process for stage 5 is shown in Algorithm 5. Key symbols and phrases used in this stage:

1. *Latency reduction*: the difference between R^{net} before a move and R^{net} after a move.
2. *Power reduction*: the difference between ω_D before a move and ω_D after a move.
3. *Stopping condition*: stop partitioning if no inter-cloud moves with positive latency or power reductions are found after evaluating all the tasks in the graph.

Algorithm 5 Partition to satisfy the response time and bin-pack to minimize the power consumption

INPUTS: G, D, C, δ, R^{pt}, $R^{constraint}$, user, root task.
  Accepted moves $\leftarrow \emptyset$, Attempted moves $\leftarrow \emptyset$.
  Calculate R_D^{net} and ω^D for D.
  Sort the tasks in G in decreasing order of processing requirement.
  repeat
    Best power reduction move $\{C^{from}, C^{to}$, task, host, power reduction, latency reduction$\} \leftarrow \emptyset$.
    Best latency reduction move $\{C^{from}, C^{to}$, task, host, power reduction, latency reduction$\} \leftarrow \emptyset$.
    for each task in the sorted task list **do**
      $C^{from} \leftarrow$ the cloud where the task is currently assigned.
      for each cloud (C^{to}) in clouds, where C^{to} is not equal to C^{from}, **do**
        Find the most power efficient host in C^{to} to allocate the task (see Section 3.4.1).
        Calculate the power and latency reductions for moving the task from C^{from} to C^{to}.
        if this move has the largest power reduction that does not violate $R^{constraint}$, **then**
          record it as the best power reduction move.
        end if
        if this move has the largest latency reduction, **then**
          record is as the best latency reduction move.
        end if
      end for
    end for
    Increment attempted moves by 1.
    if a best power reduction move exists, **then**
      Increment accepted moves by 1, and subtract the latency reduction from R_D^{net} and power reduction from ω_{D}.
      Accept the best power reduction move. Task is moved from D_{Cfrom} to $D_{C^{to}}$ in D.
    else if $R^{constraint}$ is not met and a best latency reduction move exists **then**
      Increment accepted moves by 1, and subtract the latency reduction from R_D^{net} and power reduction from ω_{D}.
      Accept the best latency reduction move. Task is moved from D_{Cfrom} to $D_{C^{to}}$ in D.
    else
      no move is found.
    end if
    if the stopping condition is true, **then**
      exit.
    end if
  end repeat
OUTPUTS: D if $R^{constraint}$ is met or null on failure.

3.6 Enhancements

3.6.1 Enhancement 1: Coarsening and uncoarsening

Coarsening is the process of merging tasks that have a large amount of communication and is introduced between stages 3 and 4. The edges are sorted in decreasing order of edge weight and the tasks they connect are then merged recursively. The edge between two merged tasks disappears, so the merged tasks are deployed together on the same host. This avoids the delay that would be incurred if the tasks were deployed on hosts in different clouds. Coarsening is controlled by the following parameters:

1. B^{cpu} and B^{mem} represent the upper bounds on the processing and memory requirements of a merged task, respectively. Merged tasks with a process capacity higher than B^{cpu} or a memory capacity higher than B^{mem} are not allowed.

2. B^{edge_factor}: only edges with an edge weight higher than $\left(B^{edge\_factor}\frac{\sum_{x \in V} Y_x}{|E|}\right)$ are considered for coarsening.

A record is created for each merge operation that contains the information about the two merged tasks. These records are used to restore the merged tasks into the corresponding original tasks in the uncoarsening stage.

Uncoarsening restores the merged tasks to their original state. Where a single coarse node comprises several tasks, they are all deployed to the same cloud.

3.6.2 Enhancement 2: Apply 2D bin-packing

After uncoarsening, two bin packing strategies further reduce the power consumption by reducing the number of active hosts. They reassign tasks to different hosts in the same cloud using variants of the Hybrid-Best-Fit (HBF) bin packing approach [3]. HBF sorts the tasks in decreasing order of "height", and then assigns each task in order to the host with the least available "height" and "width" that can accommodate it. Equation 16 describes the bin packing score for host x. A task is assigned to the host with the highest bin packing score.

$$\text{Bin packing score for host } h \ = \frac{\left(p^{used}(h)/p^{total}\,(h)\right) \ + \left(m^{used}(h)/m^{total}\,(h)\right)}{2} \tag{16}$$

In the HBF-Mem variant, the height is $m^{task}(x)$, and the width is $p^{task}(x)$, and in the HBF-Proc variant, the height is $p^{task}(x)$, and the width is $m^{task}(x)$.

3.7 Stage 6: Selecting the deployment solution

Stages 4 and 5 and enhancement 2 are repeated for one power-sensitive initial deployment, one delay-sensitive initial deployment, and 50 random initial deployments. Each initial deployment yields three deployment candidates: Stage 5 provides one candidate, and enhancement 2 provides the other two. With 52 initial deployments, there are $52 \times 3 = 156$ deployment candidates for the final solution. In stage 6, the deployment that consumes the least power while satisfying the response time constraint is chosen as the solution. R^{LPMCAD} and P^{LPMCAD} represent the response time per user request and power consumption of the solution respectively.

4 Experiments

4.1 Experimental setup

4.1.1 Hardware details

The experiments are run on an Intel i5 3570 machine with 3.4 GHz processor, and 16 GB memory (RAM).

4.1.2 Software details

1. The algorithm is implemented in Java.
2. The Java Universal Graph Framework (JUNG) v2.0.1 [18] is used for graph data structures.
3. JSON.simple v1.1.1 [19] is used for JSON processing.
4. Guava v19.0 [20], Google Core Libraries for Java, provide the Multimap data structure that is used to represent the deployment. Multimap maps a key to multiple values. In the implementation, the key is the cloud, and the values are the {task, host, power} entries associated to the cloud.

5. The Apache Commons Mathematics Library v3.6.1 [21] generates the power versus throughput regression model for the hosts.
6. The Eclipse IDE v4.6.3 [22] is used for development.
7. OpenJDK8 v1.8.0_222 [23], powered with OpenJ9 Java Virtual machine, runs the Eclipse IDE and the algorithm.
8. Ubuntu v16.04 [24], a Linux operating system, is used.

4.1.3 Algorithm control parameters

1. $U^{nom} = 0.8$.
2. $B^{edge\_factor} = 1$.
3. Refer to the next section for the values of B^{mem} and B^{cpu}.

4.1.4 Host parameters

1. Hosts have either 16 GB, 64 GB, 128 GB or 192 GB of memory capacity. For a merged task to fit on all hosts, B^{mem} is chosen to be 16 GB.
2. Hosts have either 8 CPUs, 72 CPUs, 88 CPUs or 112 CPUs. For a merged task to fit on all hosts, B^{cpu} is chosen to be the average throughput of 8 CPUs. Its unit are *ssj ops*.
3. Host speedup information (see Table 2) is derived from the SPECpower results.

Table 2: Host details sorted in decreasing order of speed and power efficiency [14]

Host Name (h)	Number of Logical CPUs, N^{cpu}	SPECpower Throughput at 100% Host Load, p^{total} [ssj ops]	SPECpower Throughput Per Logical CPU, $\frac{p^{total}}{N^{cpu}}$ [ssj ops]	$s_h = \frac{p^{total}}{s^{min} N^{cpu}}$
Inspur Corporation NF5280 M4	88	3,561,599	40,473 ($\boldsymbol{s^{min}}$)	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-1U	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

4.2 Solution bounds

There is no existing exact solution method for the K-cloud deployment problem defined in this paper with which to compare the LPMCAD algorithm. Instead we looked for lower bounds on the power consumption and response time per user request, the two most important characteristics of a deployment.

4.2.1 Power consumption lower bound

A lower bound on an application's power consumption (P^{bound}) was estimated by allocating the total workload to the most power-efficient hosts. The most power efficient hosts are assumed to operate at the optimal utilization where the ratio of the throughput to power is the highest. The processing and memory requirements are pooled and allocated in a fluid manner, to fill the most power efficient host first and then overflow to the next most power efficient host and so on. Network delays are ignored. For example, 40% of the resources for an application component can be allocated on one host and the remaining 60% on another, so fractional resources for an application component can be allocated on two different hosts, which can be located on two different clouds. This lower bound on power consumption is unlikely to be attainable in practice.

4.2.2 Response time per user request: No lower bound

A lower bound on the response time per user request for an application could not be found. Two approaches were considered. The first assigns tasks to the cloud with the least latency to the current cloud if no resources are available on the current cloud. The first cloud chosen has the least latency to the user. The second approach is similar to the partitioning stage, but the goal is to only minimize R_D^{net} for the deployment. Neither approach could guarantee a lower bound on the response time per user request. We instead verified the response time using the Layered Queueing Network Solver software [4], as described next.

4.3 Response time verification by layered queueing network solver (LQNS)

The goal of this step is to verify the response time per user request of the LPMCAD deployment solution (R^{LPMCAD}). As shown in Section 3.3, LPMCAD approximates the wait times and resource contention in R^{pt} by using a nominal target utilization U^{nom}=0.8 for the hosts. The LQNS evaluates the wait times and resource contention more accurately, so its response time per user request (R^{LQNS}) is closer to the actual value that would be experienced by an application deployed in a real multi-cloud environment. Comparing R^{LPMCAD} and R^{LQNS} evaluates the quality of the LPMCAD approximation for the wait times and resource contention.

A new LQN model representing the LPMCAD deployment is generated using the details about the replicas, inter-cloud delays and host assignments from the LPMCAD solution. The processor sharing (PS) discipline is used in the new LQN model since it runs all the tasks simultaneously on the processor. The LQNS solution provides a value for R^{LQNS}. The use of this value is discussed in Section 4.7.4 below.

4.4 Evaluation criteria

We compare two values using the relative error measure defined in Equation 17:

$$\Delta_{\%}(A, B) = \left(\frac{A - B}{A}\right) \times 100\% \tag{17}$$

These criteria are used to evaluate a solution:

1. Is the response time constraint satisfied ($R^{LPMCAD} \leq R^{constraint}$)?
2. How close is P^{LPMCAD} to P^{bound}, as measured by $\Delta_{\%}\left(P^{LPMCAD}, P^{bound}\right)$?
3. LPMCAD algorithm runtime, T^{LPMCAD}.
4. Does the deployment satisfy the application's resource requirements?
5. How does the LQNS response time per user request compare to the LPMCAD response time per user request, as measured by $\Delta_{\%}\left(R^{LPMCAD}, R^{LQNS}\right)$?

4.5 Tuning and test models

Test models were generated as LQN AppModels via the random model generator *lqngen*, which is part of the LQNS software package [4]. *lqngen* generated models with a random structure and parameters, using a size parameter (option –*A*[size]) which governs the number of layers, clients, processors and tasks.

4.6 Multi-cloud test environment

Figure 2 illustrates the abstract multi-cloud environment used for tuning (see Section 4.7) and evaluation (see Section 4.8). Table 3 shows the individual clouds total processing, memory and CPU capacities. Table 4 shows the network delays in the test environment.

Table 3: Cloud capacities in the multi-cloud test environment

	Total memory capacity, β [GB]	Total processing capacity, μ [$ssj\ ops$]	Total number of logical CPUs
Cloud Edge (C_1)	64.0	1,669,696.80	32
Cloud Small (C_2)	80.0	2,028,905.60	40
Cloud Medium (C_3)	176.0	4,240,154.40	88
Cloud Large (C_4)	272.0	7,467,443.20	200

Table 4: Network delays (in msec) in the multi-cloud test environment

$\delta(r,q)$ [msec]		r				
		U	C_1	C_2	C_3	C_4
q	U	N/A	25	100	175	250
	C_1	25	N/A	75	150	225
	C_2	100	75	N/A	75	150
	C_3	175	150	75	N/A	75
	C_4	250	225	150	75	N/A

4.7 Tuning the LPMCAD algorithm

4.7.1 Tuning experiments

The tuning experiments study the impact of τ and $R^{constraint}$ on R^{LPMCAD}, P^{LPMCAD} and T^{LPMCAD}. Five test models are generated for tuning using the *lqngen* tool with these values for -*A*: {4, 8, 18, 24, 30}. These models have different characteristics in terms of number of application components, processing and memory requirements, number of calls per user request, and processing time [See Section 4.8.1]. Two experiments are performed for each of the tuning test models with and without coarsening:

1. *Experiment 1*: The throughput requirement (τ) is varied with an easily achievable response time constraint ($R^{constraint}$).
2. *Experiment 2*: The response time constraint ($R^{constraint}$) is varied from an easy target to a hard target. A use-case is taken from experiment 1 where $R^{constraint}$ is tightened until none of the initial deployments can satisfy $R^{constraint}$.

The two experiments provide 94 test cases, which are used to derive a good configuration for the LPMCAD algorithm. Evaluation criteria 1–4 (see Section 4.3) are used to assess the LPMCAD algorithm.

4.7.2 Summary of results

The tuning experiments show that:

1. Enabling coarsening helps in satisfying tighter response time constraints, achieving lower response times, delivering deployments with lower power consumption, and reducing the LPMCAD algorithm runtime.
2. In the 154 test cases, the power-sensitive initial deployment produced a deployment solution nine times, delay-sensitive initial deployment produced a deployment solution one time, and random initial deployment produced a deployment solution in the remaining successful test cases.
3. There is a negligible difference in the power consumption when the bin packing strategies in enhancement 2 (see Section 3.6.2) are performed with or without uncoarsening.
4. In 85% of the test cases, the deployment solution from stage 5 has the least power consumption. In 10% of the test cases, the deployment solution from the HBF-Mem strategy in enhancement 2 yields the lowest power consumption. In the remaining 5% of the test cases, the deployment solution from the HBF-Proc strategy in enhancement 2 yields the lowest power consumption.

4.7.3 Tuned algorithm

In the tuned LPMCAD algorithm:

1. Coarsening (enhancement 1) is enabled due to its benefits.
2. All three initial deployment strategies are used since each produced a deployment solution at least once.
3. The bin packing strategies in enhancement 2 are invoked once after uncoarsening.
4. Both HBF-Mem and HBF-Proc strategies are enabled since they yield a deployment solution with least power consumption at least once.

4.8 Evaluation of the tuned LPMCAD algorithm

4.8.1 Test models and experiments

104 random AppModels were generated using the *lqngen* tool, with 20 or 21 models for each -A value: {4, 8, 18, 24, 30}. The random generation process produced model parameters with these ranges:

1. Number of application components: 20 – 240.
2. Total processing requirement: 5 – 84% of the total multi-cloud capacity.
3. Total memory requirement: 5 – 87% of the total multi-cloud capacity.
4. Total number of calls per user request: 3 – 11050.
5. Total processing time: 8 $msec$ – 65000 msec.

The tuned LPMCAD algorithm was invoked once for each of the 104 test models. The percentage difference (see Equation 17) is used for comparison. All five evaluation criteria (see Section 4.4) are used to assess the tuned LPMCAD algorithm.

Tuning experiments 1 and 2 (see Section 4.7) provide the boundary limits or failure points for the throughput and response time constraints respectively. The 104 evaluation test models are related to the five tuning test models due to the common -A value set. The boundary limits for the constraints from the tuning experiments are used as a starting point to set the throughput and response time constraints for the evaluation tests. If the boundary limits for the constraints were infeasible for an evaluation test model, then the constraints were incrementally relaxed in order to obtain feasible evaluation test cases. This section studies the LPMCAD algorithm only for feasible cases. Table 5 shows an example failure point of the response time constraint for tuning model 2.

Table 5: Tuning experiment 2 results for tuning test model 2

$R_D^{pt} = 608$ $msec$ $P^{bound} = 1107.1$ W	$R^{constraint}$ $[msec]$	R^{LPMCAD} $[msec]$	P^{LPMCAD} $[W]$
Run 1	12500	12454	1155.19
Run 2	10000	9954	1158.57
Run 3	7500	7446	1151.83
Run 4	5000	4988	1160.44
Run 5	2500	Failed	Failed

4.8.2 P^{LPMCAD} versus P^{bound}

In the 104 test cases, P^{LPMCAD} is within 10% of the unachievable lower bound P^{bound} in 77% of the test cases. In 21% of the test cases, P^{LPMCAD} is within 10-20%; and for 2% of the test cases, it is within 25-27% of P^{bound}. These results are very satisfactory considering P^{LPMCAD} is within 10% of P^{bound} for most test cases. P^{LPMCAD} does poorly relative to P^{bound} in two situations. First, application components are unable to reach the optimal power efficient host state (the operating point corresponding to the host's highest throughput to power ratio) because the application has small

resource requirements that are insufficient to reach the power efficient state. Second, tighter response time constraints can lead to moves which primarily focus on reducing the total network delay (R_D^{net}), which prevents components from being deployed on the most power efficient machines across all the clouds.

4.8.3 R^{LPMCAD} versus $R^{constraint}$

In the 104 test cases, the LPMCAD algorithm always satisfies the response time constraint: $\Delta_{\%}\left(R^{constraint}, R^{LPMCAD}\right)$ is always positive i.e. $R^{constraint} > R^{LPMCAD}$. R^{LPMCAD} has two components: R_D^{pt} and R_D^{net}. R_D^{pt} is static, and it is not influenced in the LPMCAD algorithm, whereas R_D^{net} is reduced in stage 5. If R_D^{pt} does not violate $R^{constraint}$ by itself, then R_D^{net} determines the quality of R^{LPMCAD} relative to $R^{constraint}$. R_D^{net} depends on the application graph properties such as the number of edges and edge weights, which represent the communication between the application components. It is difficult to satisfy $R^{constraint}$ by reducing R_D^{net} for relatively large values of total number of calls per user request. In such cases, $\Delta_{\%}\left(R^{constraint}, R^{LPMCAD}\right)$ is closer to 0%. In contrast, it is easier to reduce R_D^{net} for relatively small values of total number of calls per user request. In such cases, R^{LPMCAD} can be much lower than $R^{constraint}$.

4.8.4 R^{LPMCAD} versus R^{LQNS}

To assess the response time approximation for the wait delays (see Section 2.5), R^{LPMCAD} is compared with R^{LQNS} because the LQNS approximates the wait delays more accurately using nonlinear models. Figure 6 summarizes the comparison between R^{LPMCAD} and R^{LQNS}. It only accounts for 82 test cases. In 22 test cases, the LQNS failed to solve the LQN model. In 50% of the 82 test cases, the wait delays were underestimated and in the other 50% of the 82 test scenarios, the wait delays were overestimated i.e. $\Delta_{\%}\left(R^{LPMCAD}, R^{LQNS}\right)$ is positive. Overestimation of wait delays is acceptable since $R^{constraint}$ will not be violated. $R^{constraint}$ may be violated in case of underestimation i.e. $\Delta_{\%}\left(R^{LPMCAD}, R^{LQNS}\right)$ is negative. So, a deflation factor (α^{wait}) is proposed, which would scale down $R^{constraint}$ in order to handle the underestimation errors.

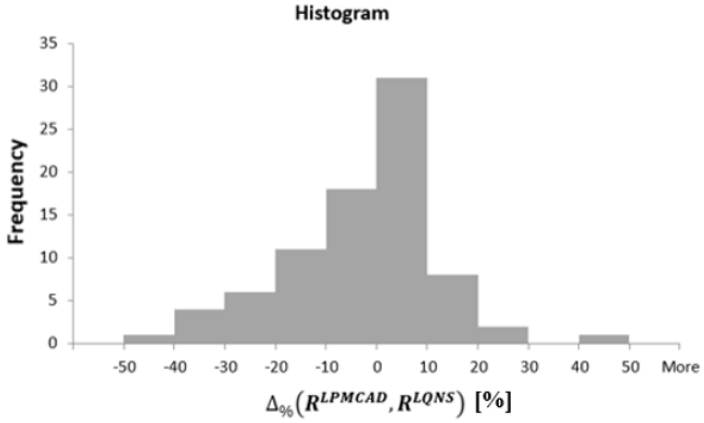

Figure 6: LPMCAD's response time vs. response time from the LQNS

The application's resource requirements were compared with the cloud resources to explain the $\Delta_{\%}\left(R^{LPMCAD}, R^{LQNS}\right)$ values, but no patterns were found. The utilization of processors in the LQNS solution did not exceed $U^{nom} = 0.8$, so no hosts were overloaded. The differences between R^{LPMCAD} and R^{LQNS} are approximation errors in R^{pt}. However, even when R^{LPMCAD} is underestimated, it is often still sufficiently smaller than $R^{\ constraint}$ that using the larger R^{LQNS} still satisfies $R^{constraint}$.

$R^{constraint}$ is compared to R^{LQNS} to find the number of test cases that violate $R^{constraint}$ due to underestimation of the wait delays, as shown in Figure 7. Just 16 of the 40 test cases with underestimated wait delays violate $R^{constraint}$ with the largest error being 35 sec.

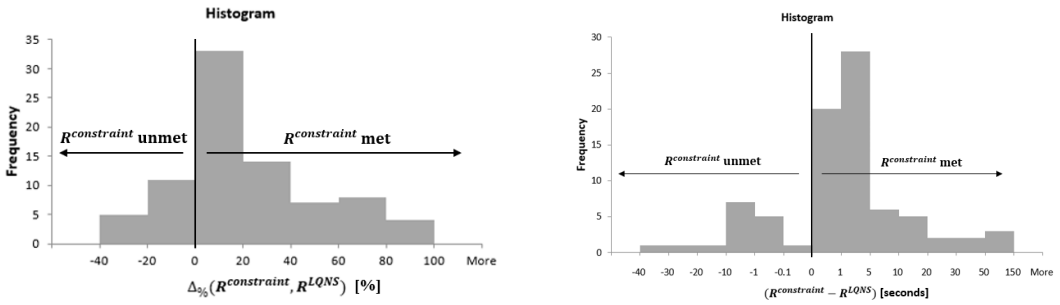

Figure 7: Response time constraint vs. response time from the LQNS

4.8.5 Algorithm runtime

Algorithm runtime determines how well the algorithm scales. Table 6 shows the impact of tightening $R^{constraint}$ on T^{LPMCAD}. T^{LPMCAD} increases and reaches an upper threshold as $R^{constraint}$ is tightened. This happens because the number of initial deployments that do not converge to a solution (blind alley deployments), increases as the throughput and response time constraints are set closer to the boundary limits or failure points.

Table 6: Tuning experiment 2: results for tuning model 2

$R_D^{pt} = 608\ msec$	$R^{constraint}$ [msec]	T^{LPMCAD} [msec]	Good Deployments	Bad Deployments
Run 1	12500	6574	51	1
Run 2	10000	8655	51	1
Run 3	7500	16686	51	1
Run 4	5000	24718	14	38
Run 5	2500	25523	0	52

LPMCAD runtime depends on multiple factors such as the total number of tasks and calls in the graph, number of clouds, number of hosts in each cloud, network latencies, constraints and capacities of the hosts. It is challenging to study the algorithm runtime while accounting for all these factors. For simplicity, runtime is studied as the number of tasks in the graph (problem size) is increased. Figure 8 summarizes the data for T^{LPMCAD} (crosses) and the HASRUT algorithm runtime, T^{HASRUT} (circles). In 77% of the test cases, T^{LPMCAD} is below 10 seconds. In another 11% of the test cases it is 10 to 20 seconds. For the remaining test cases, T^{LPMCAD} is between 20 and 70 seconds. LPMCAD, which can support K-clouds, is faster and more scalable than HASRUT, which can support just two clouds. These results are very good: T^{LPMCAD} is below 20 seconds in 90% of the test cases and can be further reduced by handling the many initial deployments concurrently.

5 Conclusions

The LPMCAD algorithm is very effective and is suitable for use in practice. Over the 104 test cases with four clouds and up to 240 deployable units (tasks):

1. Power consumption is very low: it is within 10% of a theoretical lower bound (which is too low to be feasible) in 77% of the cases.
2. Solution times are small: less than 20 sec in 82% of cases and less than 10 sec in 76% of cases. These times make LPMCAD suitable for practical use.
3. The algorithm scales well: solution times increase roughly linearly with problem size (deployable units, including replicas introduced by scaling out to handle high system loads), as shown in Figure 8.

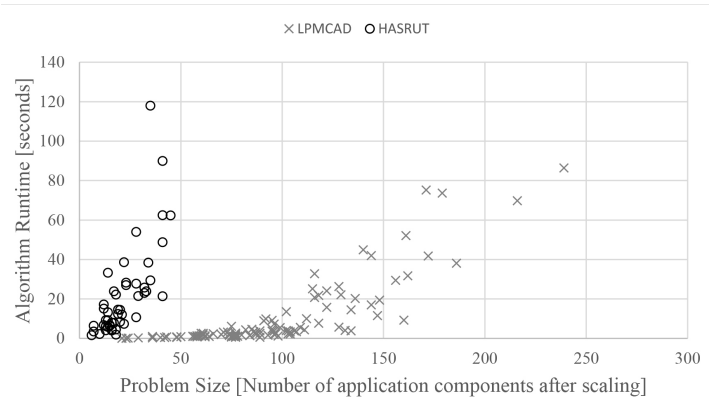

Figure 8: Algorithm runtime versus problem size for the LPMCAD and HASRUT algorithms

Coarsening/uncoarsening of the application graph considerably improves the quality of the solutions and reduces the time to obtain them. The two non-random initial deployment strategies (power-sensitive and delay-sensitive) led to the final solution in relatively few cases.

The response time approximation used in the deployment calculations is essential to obtaining small solution times, but it does lead to underestimated response times in some cases. In the 82 test cases where the LQNS solved the model, 40 test cases (50%) returned a response time that exceeded the time estimated by LQNS, but 66 test cases (80%) still satisfied the response time constraint and another 10 test cases (93% in total) came within 20% of the response time limit. The violators can be corrected by repeating the solution with an adjusted, tighter response time constraint.

Our application model does not describe components whose workload increases when they are replicated, as often happens when replicated data must be synchronized. This limitation could be addressed by iterating the solution and introducing the overhead once the number of replicas is known at least roughly.

This work can be generalized by: 1) selecting the clouds and hosts in real-time using active learning, 2) employing a more accurate power model, 3) using an adaptive technique in the coarsening algorithm instead of enforcing constant upper bounds on the processing and memory requirements of a merged task, 4) accounting for cloud outages, 5) supporting models with multiple groups of users with different locations and requirements, 6) modelling I/O delays and storage requirements, and 7) incorporating a more real-time network model.

References

[1] Bruno Menegola. 2012. A study of the k-way graph partitioning problem. Master's Thesis. Federal University of Rio Grande do Sul, Rio Grande do Sul, Brazil.

[2] Henrik I. Christensena, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. 2017. Approximation and online algorithms for multidimensional bin packing: A survey. Computer Science Review, 24 (May 2017), 63–79. DOI: https://doi.org/10.1016/j.cosrev.2016.12.001

[3] British German Academic Research Collaboration Programme. 2006. Survey on two-dimensional packing. (September 2006). Retrieved December 5, 2018 from http://cgi.csc.liv.ac.uk/~epa/surveyhtml.html

[4] Murray Woodside. 2013. Tutorial Introduction to Layered Modeling of Software Performance. (February 2013). Retrieved December 22, 2018 from http://www.sce.carleton.ca/rads/lqns/lqn-documentation/tutorialh.pdf

[5] Nandor Verba. 2019. Application Deployment Framework for large-scale Fog Computing Environments. Ph.D. Dissertation. Coventry University, Coventry, England. DOI: https://doi.org/10.13140/RG.2.2.27166.79684

[6] Karsten Molka and Giuliano Casale. 2017. Energy-efficient resource allocation and provisioning for in-memory database clusters. IFIP/IEEE Symposium on Integrated Network and Service Management (IM) (May 2017), 19–27. DOI: https://doi.org/10.23919/INM.2017.7987260

[7] Cihan Tunc, Nirmal Kumbhare, Ali Akoglu, Salim Hariri, Dylan Machovec, and Howard J. Siegel. 2016. Value of Service Based Task Scheduling for Cloud Computing Systems. International Conference on Cloud and Autonomic Computing (ICCAC) (Sept. 2016), 1–11. DOI: https://doi.org/10.1109/ICCAC.2016.22

[8] Yi Wang and Ye Xia. Energy Optimal VM Placement in the Cloud. IEEE 9th International Conference on Cloud Computing (CLOUD) (June-July 2016), 84–91. DOI: https://doi.org/10.1109/CLOUD.2016.0021

[9] Sanjaya K. Panda and Prasanta K. Jana. 2015. Efficient task scheduling algorithms for heterogeneous multi-cloud environment. The Journal of Supercomputing, 71, 4 (April 2015), 1505–1533. DOI: https://doi.org/10.1007/s11227-014-1376-6

[10] Ravneet Kaur. 2015. Lightweight Robust Optimizer for Distributed Application Deployment in Multi-Clouds. Master's Thesis. Carleton University, Ottawa, Canada.

[11] Marc E. Frincu and Ciprian Craciun. 2011. Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments. Fourth IEEE International Conference on Utility and Cloud Computing, Victoria, NSW (Dec. 2011), 267–274. DOI: https://doi.org/10.1109/UCC.2011.43

[12] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. 2009. Enhanced Modeling and Solution of Layered Queueing Networks. IEEE Transactions on Software Engineering, 35, 2 (March-April 2008), 148–161. DOI: https://doi.org/10.1109/TSE.2008.74

[13] Farhana Islam, Dorina Petriu, and Murray Woodside. 2015. Simplifying Layered Queuing Network Models. Beltrán M., Knottenbelt W., Bradley J. (eds) Computer Performance Engineering (EPEW 2015), Springer LNCS, 9272 (Aug. 2015), 65–79. DOI: https://doi.org/10.1007/978-3-319-23267-6_5

[14] Standard Performance Evaluation Corporation. 2015. SPECpower_ssj2008: Fujitsu FUJITSU Server PRIMERGY TX1320 M2. (Nov. 25, 2015). Retrieved Dec. 7, 2018 from https://www.spec.org/power_ssj2008/results/res2015q4/power_ssj2008-20151110-00704.html

[15] Jumie Yuventi and Roshan Mehdizadeh. 2013. A critical analysis of Power Usage Effectiveness and its use in communicating data center energy consumption. Energy and Buildings, 64 (Sept. 2013), 90–94. DOI: https://doi.org/10.1016/j.enbuild.2013.04.015

[16] André B. Bondi. 2014. Foundations of Software and System Performance Engineering. Addison-Wesley Professional, Boston, MA.

[17] Standard Performance Evaluation Corporation. 2007. SPECpower_ssj2008 Result File Fields. (Nov. 2017). Retrieved Dec. 6, 2018 from https://www.spec.org/power/docs/SPECpower_ssj2008-Result_File_Fields.html

[18] Danyel Fisher, Joshua O'Madadhain, and Scott White. 2003. JUNG: Java Universal Network/Graph Framework. (Aug. 2003). Retrieved Dec. 16, 2018 from https://github.com/jrtom/jung

[19] Yidong Fang, Chris Nokleberg and Dave Hughes. 2008. json-simple. (Nov. 2008). Retrieved Dec. 16, 2018 from https://code.google.com/archive/p/json-simple

[20] Google. 2010. Guava. (April 2010). Retrieved Dec. 15, 2018 from https://github.com/google/guava

[21] Apache Commons. 2012. Commons Math: The Apache Commons Mathematics Library. (Mar. 2012). Retrieved Dec. 17, 2018 from http://commons.apache.org/proper/commons-math

[22] Eclipse Foundation. 2001. Eclipse IDE. Retrieved Dec. 16, 2018 from https://www.eclipse.org/ide

[23] AdoptOpenJDK. 2017. Prebuilt OpenJDK Binaries. Retrieved Dec. 17, 2018 from https://adoptopenjdk.net/?variant=openjdk8&jvmVariant=openj9

[24] Canonical. 2016. Ubuntu 16.04.5 LTS. April 2016. Retrieved Dec. 17, 2018 from http://releases.ubuntu.com/16.04