Lightweight Robust Optimizer for Distributed Application Deployment in Multi-Clouds

by

Ravneet Kaur

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Masters of Applied Science

in

Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

September 2015

The undersigned recommend to

the Faculty of Graduate and Postdoctoral Affairs

acceptance of this thesis

Lightweight Robust Optimizer for Distributed Application Deployment in Multi-Clouds

Submitted by

Ravneet Kaur B.Tech.

in partial fulfillment of the requirements for

the degree of Masters of Applied Science in Electrical and Computer Engineering

_____

Yvan Labiche, Chair, Department of Systems and Computer Engineering

_____

Thesis Supervisor, Murray Woodside

_____

Thesis Supervisor, John Chinneck

_____

External Examiner, Amiya Nayak, University of Ottawa

ABSTRACT

Cloud computing refers to the applications and services that run on a distributed network using virtualized resources and accessed by common Internet protocols and networking standards. In cloud computing, an edge cloud is close to some of the end users, to give faster service for very demanding applications. Transactions that require heavy processing capacity and longer processing times are better carried out at the core cloud. To deploy applications with many tasks across a cloud infrastructure, many goals must be satisfied, which poses a large and complex optimization problem. Meeting latency constraints is an important requirement in future cloud applications and is critical in task deployment.

This thesis creates a new approach for task assignment in an edge-core multi-cloud architecture to reduce power consumption in service centers using multilevel graph partitioning technique. Multilevel graph partitioning has three phases of coarsening, refinement and uncoarsening. For the refinement phase, a new algorithm based on a modified Kernighan–Lin algorithm is proposed which takes into account multiple constraints, and that mitigates the problem of stopping at a local minimum. Once tasks are assigned to the edge and core, multidimensional bin-packing is used to deploy tasks to individual hosts so that power consumption can be calculated. The approach is validated by comparing it to extended simulated annealing and an extended modified Kernighan–Lin algorithm. The experiments show that our approach is fast and produces better results. It is also less prone to failure in finding a feasible deployment for given constraints.

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Dr. John W. Chinneck and Dr. C. Murray Woodside, who provided the initial inspiration for this work and gave critical guidance throughout the process. Their solid understanding of the subject and mentorship helped me grow and mature as a serious graduate researcher and made this thesis an enjoyable experience. I thank members of SAVI group for their valuable comments and ideas, which were indispensable in producing this thesis.

I would like to acknowledge and thank my colleagues Adnan Faisal, Farhana Islam and Derek Hawker for their friendly, often philosophical, discussions which made me laugh and kept me engaged. I would also like to express my gratitude to the professors, staff and students at the department of Systems and Computer Engineering for making my work an enjoyable experience.

I would like to thank the most supportive, affectionate and loving husband Harjot Dhindsa for his encouragement, guidance and patience. Without his assistance and support this thesis would have been incomplete. I would love to dedicate this thesis to my darling daughter Hasrut Dhindsa who came in this world during my thesis work and was the inspiration for me to complete my thesis. She is a true blessing and I thank her for giving me time that was meant for her to finish this thesis.

Finally, I thank my parents Gurmel Singh and Harbans Kaur Dhillon, and my brother, Harpinder, for their encouragement and blessings from miles away and making me capable enough to finish this thesis. They inculcated in me, the determination and dedication to do whatever I undertake well. I am grateful to my in-laws for loving me and supporting me throughout this thesis.

Dedicated to my daughter

Hasrut Kaur Dhindsa

Table of contents

List of Figures

List of Tables

Notation

Acronyms

avg	Average
BPP	Bin Packing Problem
FFA	First Fit Allocation
FM	Fiduccia-Mattheyses algorithm
GA	Genetic Algorithm
GB	Giga Bytes
GHz	Giga Hertz
HEM	Heavy Edge Matching
IaaS	Infrastructure as a Service
KL	Kernighan-Lin
KL-SA	Kernighan-Lin-Simulated Annealing
LQN	Layered Queuing network
LTE	Long Term Evolution, a high-speed wireless network standard
MLKL	Multi-Level Kernighan Lin
msec	Milliseconds
PaaS	Platform as a Service
QoS	Quality of Service
RA	Random allocation
SA	Simulated Annealing
SaaS	Software as a Service
std dev	standard deviation
TS	Tabu Search
VNS	Variable Neighborhood Search
W	Watts

Symbols

A	Arcs in a graph
C	Capacity of a bin
C_{fh}	Fixed cost of host h
C_h	Power cost due to unit utilization of host h
c_{ij}	arc weight of the arc between vertex i and vertex j
DT_t	Service Demand of task t
f	Throughput
g_{ixy}	gain of vertex i when it is moved from partition x to partition y
h_i	Height of a bin
L	epoch length for SA
MC_h^j	Memory capacity of host h on j^{th}cloud
MR^i	Memory requirement of i^{th} task
m_t	number of replicas of task t
$P(j)$	Function that returns the partition where vertex j is deployed
PC_h^j	Processing capacity of host h on j^{th}cloud
P_h	Power of individual host h
PR^i	Processing requirement of i^{th} task

P_{total}	Total power of all hosts on edge and core cloud
PT_t	Processing time of task t
refTask	Reference Task for a system or LQN model
s	Source
t	sink
T_0	initial temperature for SA
U_h	Utilization of host h
U_{nom}	Nominal Utilization of hosts in cloud
UT_t	Total processor utilization of task t
V	Vertices in a graph
w_i	width of a bin
X_{ij}	Task cloud selection variable. Equal to 1 if task i is assigned to cloud j
Y_{ikj}	Task host cloud selection variable. Equal to 1 if task i is assigned to host k on cloud j
α	cooling coefficient for SA
Ψ	Response time threshold
Ω_h	Used capacity of host h

1. Introduction

Cloud Computing is the new cost-efficient computing paradigm in which information and computer power can be accessed by user applications, for instance phone apps. Users are able to access applications and data from a Cloud anywhere in the world. Cloud Computing shifts the computation from local, individual devices to distributed, virtual, and scalable resources, thereby enabling end-users to utilize the computation, storage, and other application resources, which form the Cloud, on-demand [1].

Cloud computing promises to deliver reliable services through next-generation data centers built on virtualized compute and storage technologies. This has resulted in reduced cost, higher scalability and improved performance for end users in comparison to maintaining their own private computer systems [2]. Power consumption is a major recurring cost that plays an important role in the economic viability of running and maintaining a profitable cloud computing network. Naturally, minimizing power consumption costs is of primary interest to cloud operators. In addition, it is extremely important to meet maximum response time constraints to provide an acceptable user experience in future applications such as real-time online multi-player games, musical interactions, etc. The latency introduced by inter-cloud data transfers can have a major impact on the response time; hence it is vital to allocate tasks in an application so that the latency of data transfers does not cause a violation of any latency constraints

1.1 The Characteristics of Future Applications

Future cloud applications will transmit large volumes of data (e.g. video streaming). Devices like smartphones and tablets which are equipped with cameras and

LTE make it easy and affordable for the user to download and transmit videos or large data files over the internet e.g. videos recorded at concerts transmitted by multiple users over the internet. As multi-person interactive applications are gaining popularity among the newer generation, real-time response time is another constraint for future applications. The entire application cannot be made to run on a cloud network that is geographically far away from the user as it will result in slow response for various user requests. The application cannot also be running on machines that are geographically closer to the user to provide low response time as maintaining a farm of such machines is not plausible at every user location.

An **optimal deployment** of a given application on a cloud network will be defined as the deployment that has minimum power consumption while meeting response time, memory and processing capacity constraints. Therefore, optimal deployment is critical to ensuring an acceptable user experience at minimum cost to the cloud operator. Some examples of future-oriented applications include:

- *Multi-player interactive games*. Online gaming has gained popularity recently. Geographically distributed players interact and see each other in real time. Any lag due to latency in the complex simulated environment may degrade the gaming experience. Response time constraints must be met in order to provide an excellent gaming experience.

- *Battlefield situational awareness*. Military deployed overseas or locally over a battlefield has to transmit large amounts of real-time data between weapon systems, soldiers, and headquarters. Data must to be integrated in real-time to

provide situational awareness to commanders, weapons systems, and wirelessly-connected soldiers. Latency must be small as massive amounts of accurate information regarding whereabouts of enemy forces, their weapon repositories and also about their own assets is transmitted. Many of the relevant elements may be moving very swiftly (e.g. aircraft) [3] [4]

- *Sensor integration* [5]. Sensor based applications are sensitive as large data volumes are flowing from multiple distributed sensors in order to function correctly. Data has to be integrated in real time from various sensors to make sense of the system. Example: Aircraft and airports are equipped with multiple sensors. They must integrate all data in real-time correctly for flight control. For traffic control systems where multiple magnetic loops are embedded in roadways, data has to be integrated for correct traffic lights operation and traffic cameras. Latency in sensor based application cannot be tolerated.

- *Disaster response*. Effective response to disasters (tsunami, earthquake, forest fire, etc.) requires time-coordinated integration of multiple high-volume data streams as well as background data analysis, optimization, and scheduling.

- *Video integration*. While live streaming of an event is shot through multiple cameras during live streaming, data from all cameras has to be integrated with very little latency in order to broadcast the event in real-time. Example: Videos shot at a sports event or civil crisis. Again data volumes are large and only very small latencies can be tolerated.

1.2 The Characteristics of Future Cloud Architectures

Cloud architectures address key difficulties surrounding large-scale data processing. Current cloud computing architectures typically consist of users directly connected to a very large cloud computing center. Another important current architecture is the hybrid cloud, with a private cloud for secure data, connected to a public cloud for handling overloads and public data. The geographic placement of a central cloud and the user may introduce latency. If communication delays between the user and the central cloud are large, latency may become an issue. Future applications require massive data transmission and low latency.

One likely future cloud architecture is the "edge-core" arrangement. A small edge cloud is located near a population of users to reduce latency between user and cloud. It can connect to portable devices and it is further connected to larger core cloud. Depending on where tasks in an application are allocated, applications may require communication within the edge cloud or within the core cloud, which is assumed to have lower latency than communication between the edge and the core cloud. This situation is the motivation for the work in this thesis on the development of deployment algorithms that can make decisions about the distribution of applications or parts of applications across the cloud or within the cloud. A future cloud architecture is shown in Figure 1.1.

The multi-cloud: A multi-cloud is a heterogeneous collection of small, medium, and large cloud centers with a given set of communication links between them, where communication links themselves have different capacities and latencies. There are other cloud deployment models like private, public, hybrid cloud discussed in Chapter 2.

Deployment models play a major role in latency-sensitive applications. Using a multi-cloud is beneficial for scenarios where an application can be deployed at a distant lightly-loaded cloud if there are loose response time requirements to keep the local cloud available for tight real-time constraints applications. There are several architecture forms which may affect the quality of service provided to future applications. This work is however restricted to one edge cloud and one core cloud.

1.3 Future Deployment Issues

The main aspect of future deployments is that location matters to the cloud operator because response time, latency and data transfer are severely affected by communication links, and which communication links are utilized depends on the deployment location of processing tasks.

Currently, cloud operators do not know about the applications deployed on their infrastructure. Furthermore, users are also not aware of cloud infrastructure and other details of cloud. However, if the cloud operator is aware of a few details of the user application, much more efficient deployments are possible while still respecting response time, memory capacity and processing capacity constraints [6]. Such a deployment is referred to as an ***Application-Aware Deployment***. The Cloud operator can be made aware of application details through a performance model such as a layered queuing network. For instance, a cloud operator can minimize number of machines powered-up to minimize electricity consumption, and can make sure to deploy tasks to correct parts of the multi-cloud to satisfy response time constraints.

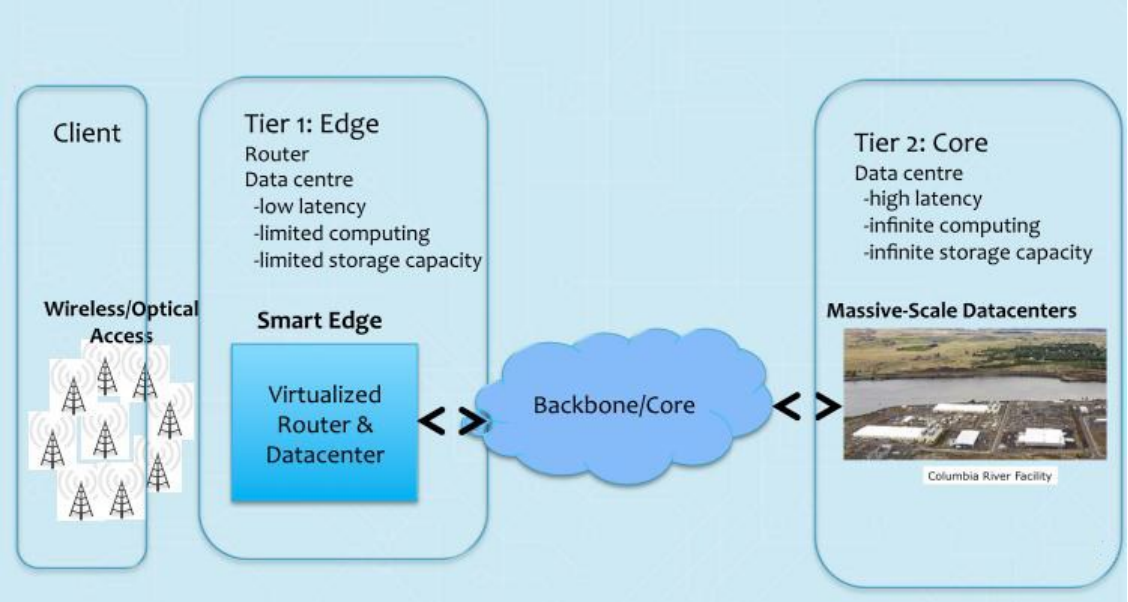

Figure 1.1: Future Cloud Architecture: Edge-Core cloud [7]

Application-Aware Deployment can be implemented for private clouds as applications are known to a cloud operator who is aware of the cloud's infrastructure and can deploy the application effectively based on this knowledge. Even though larger in size and more complex in terms of type of applications deployed, public cloud operators could conceive a model of the internal workings of a deployed application by detecting communication configurations and processing loads. This can be used to create application awareness to re-deploy application such that response time and other constraints are met while minimizing resource usage.

1.4 Contributions

This thesis presents a novel algorithm to deploy applications or parts of applications across an edge-core cloud architecture. The proposed algorithm is an extension of the modified Kernighan-Lin (KL) algorithm presented in [8]. The main

contribution of the thesis is development of a heuristic algorithm to deploy applications in an edge-cloud architecture such that constraints on response time, memory capacity and processing capacity are met while heuristically minimizing power consumption. The developed algorithm is faster and more robust than competing alternatives. A second heuristic algorithm derived from this new algorithm is relatively faster while maintaining robustness and still finds a lower power consumption than competing alternative algorithms and thus may be better suited for applications where system load and requirements change dynamically.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 introduces background on performance management for service systems, including cloud computing, graph theory, Layered Queuing Network (LQN) models and optimization approaches. Chapter 3 reviews related works on deployment optimization. Chapter 4 discusses the edge-core cloud partitioning problem, and reviews the state of the art. Chapter 5 proposes the new edge-core partitioning algorithm and its graph representation. Chapter 6 presents the experimental setup. Chapter 7 evaluates the effectiveness and computational cost of the algorithms. Finally conclusions are presented in Chapter 8.

2. Background

2.1 Cloud Computing

Cloud computing, often referred to simply as “the cloud”, is a term employed to describe a computing style for next generation service centers where massively scalable service-oriented computer resources are dynamically delivered to multiple external customers over the Internet [2].

One of the major benefits of cloud computing is infrastructure management. A given application consists of tasks (processes). Each of these tasks has its own virtual machine. Each application in a cloud sees a virtual environment dedicated to itself, such as virtual machines for its deployable tasks (processes) and virtual disks for storage. Cloud management allocates real resources to virtual resources by giving a share of a real processor or memory to a virtual machine, or by deploying several virtual machines with replicas of application processes. Cloud computing provides high quality management as a service, allowing users to reduce time and skill requirements on lower-value activities and focus on strategic activities with greater impact on business.

There are three basic types of cloud computing based on the service offered (Figure 2.1):

1. Software as a Service (SaaS): Under this service, a user is provided with application(s) that are running on a cloud infrastructure. Application(s) can be accessed by users from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. A user is not responsible for managing or controlling the underlying cloud infrastructure such as network, servers, operating systems, storage, or even

individual application capabilities, with the possible exception of limited user-specific application configuration settings. Examples of SaaS are Google, Facebook, Twitter and Flickr.

2. Platform as a Service (PaaS): For this service user created application(s) is (are) deployed on a cloud infrastructure. A cloud service provider supplies programming languages, libraries, services, and supporting tools to enable deployment of User-created application on its cloud. Like SaaS, the user does not manage or control fundamental cloud infrastructure including network, servers, operating systems, or storage, but has control over deployed applications and possibly configuration settings for application-hosting environment. Examples of PaaS are Amazon AWS elastic beanstalk, Microsoft Azure, and Google AppEngine.
3. Infrastructure as a Service (IaaS): A cloud service provider offers user access to fundamental computing resources like storage, networks and servers such that a user is able to deploy and run arbitrary software, which can include operating systems and applications. The user again does not manage or control the underlying cloud infrastructure but can manage operating systems, storage, and deployed applications; and possibly has limited control of select networking components (e.g., host firewalls). Examples of IaaS are Amazon Elastic Compute Cloud (EC2), and Rackspace.

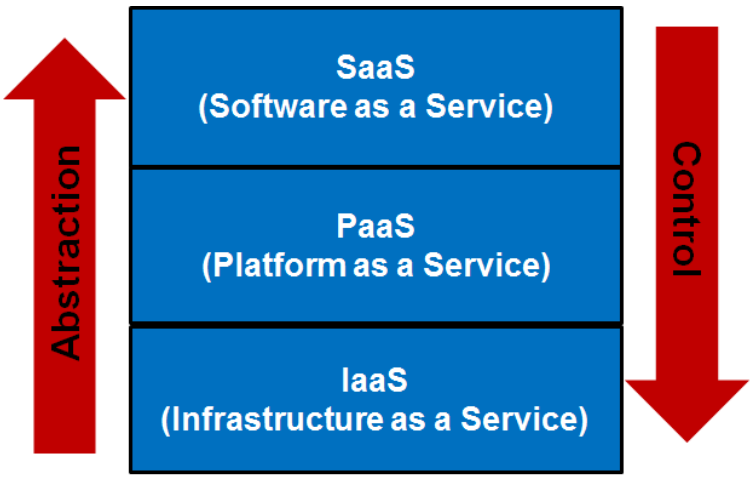

Figure 2.1: SaaS vs. PaaS vs. IaaS

2.1.1 Deployment Models

Resources available in a cloud can be connected via public and private networks. These networks provide infrastructure for application and data storage. Applications can be deployed on public, private or hybrid clouds [9]. Different deployment models are shown in Figure 2.2.

2.1.1.1 Private Cloud

A private cloud is a cloud computing infrastructure that is built solely for an individual enterprise within a firewall. This infrastructure allows applications to be hosted in a cloud, while addressing data control and security concerns which is often lacking in a public cloud environment. There are two types of private clouds:

- **On-Premise Private Cloud**: also known as an "internal cloud," is hosted within an organization's own data center. An internal cloud provides more standardized process and protection but is limited in size, scalability and incurs capital and operational costs for maintaining physical resources. These clouds are best used

for applications that contain critical data that requires high security, complete control and infrastructural configurability.

- **Externally-Hosted Private Cloud**: This type of private cloud model is hosted by an external cloud computing provider that facilitates exclusive cloud environment guaranteeing full privacy. This type of cloud network is suitable for organizations that prefer not to use a public cloud infrastructure due to risks related with sharing of physical resources.

2.1.1.2 Public Cloud

A Public cloud is one in which a service provider makes resources, such as applications, network and storage, available to general public over the Internet. Users benefit from economies of scale as all users share infrastructure costs allowing each individual user to operate on a low-cost, "pay-as-you-go" model. Typically public clouds are larger in scale than private clouds that provide clients with seamless and on-demand scalability.

The main disadvantage of a public cloud is that all users share the same infrastructure pool with limited configurations, security protections and availability variances, as these factors are solely managed and supported by the cloud service provider.

2.1.1.3 Hybrid Cloud

Hybrid clouds combine best features of both public and private cloud models. A hybrid cloud can leverage third-party cloud providers in either a full or partial manner,

which increases computing flexibility. The hybrid cloud environment is also capable of providing on-demand, externally-provisioned scalability. Augmenting a private cloud with resources of a public cloud can help manage unexpected surges in workload. An edge-core cloud architecture falls under the hybrid cloud scheme where the edge is represented by the private cloud and core by the public cloud.

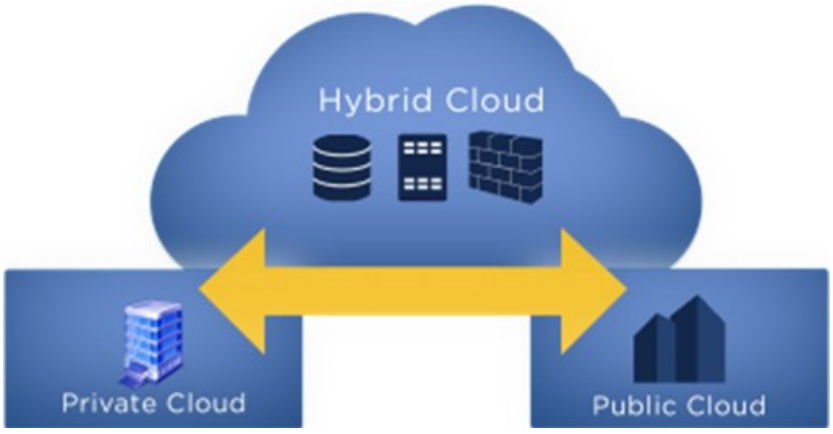

Figure 2.2: Public vs. Private vs. Hybrid Cloud [10]

2.2 Graph Theory

This sub-section explains a few basic concepts of graph theory.

A graph is a pair of sets *(V, A)*, where:

- *V* is a nonempty set whose elements are called vertices (or nodes).
- *A* is a collection of two-element subsets of *V* called arcs.

Graphs can be directed or undirected. A directed graph is one where arcs have a direction associated with them whereas an undirected graph is a graph where arcs are directionless. A weighted graph has arcs with associated arc weight whereas an unweighted graph has arcs with no associated arc weight. Figure 2.3 shows a simple undirected weighted graph with vertices *V = [n1, n2, n3, n4, n5, n6]*, arcs *A = [(n1,n2), (n1,n3), (n3,n4), (n3,n5), (n2,n6)]* .

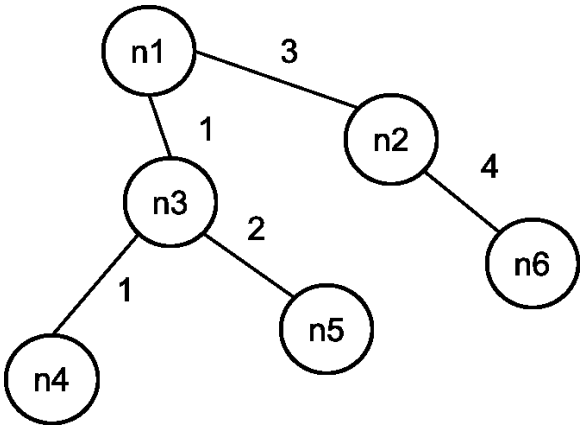

Figure 2.3 : A simple undirected weighted graph

In this research, we assume our network to be an undirected weighted graph. Each task (process) of an application is represented by a vertex in the graph and communication between different tasks is represented by an arc with an associated arc weight.

2.2.1 Graph Partitioning Problem

A graph partition problem is defined as partitioning of a graph $G = (N,A)$, with N nodes/vertices and A arcs/links, into smaller graphs with particular properties. For example, an m-way partition divides a given graph G into m smaller subgraphs. A 2-way partition of an undirected weighted graph is shown in Figure 2.4. The total weight of all arcs that have one end in one partition and the other end in a different partition is called the "cut weight" of the partition. A common graph partitioning problem is to find a partition that, in addition to satisfying various constraints, has the minimum cut weight.

If all arc weights are same, then a good partition is one in which small number of arcs connect different subgraphs. A uniform graph partition is a type of graph partitioning that divides a graph into smaller subgraphs having identical number of nodes, while

having few arcs between the subgraphs. Non uniform graph partitioning divides graph into smaller subgraphs of different sizes with few connections between the subgraphs.

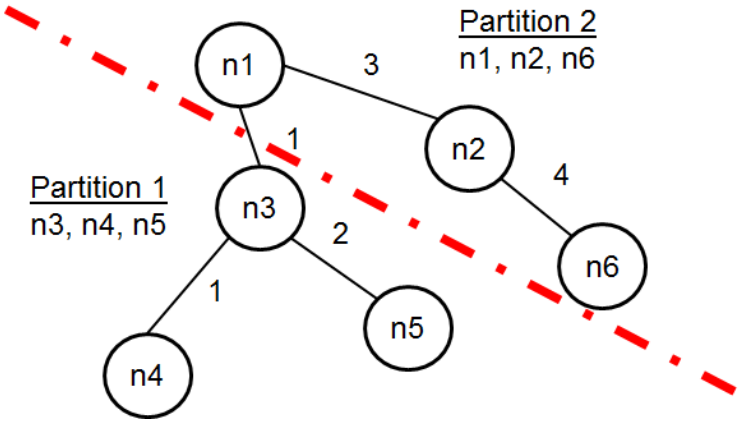

Figure 2.4: 2-way graph partitioning

Graph partitioning falls under the category of NP-hard problems which means that these problems cannot be solved deterministically for graphs with a large number of nodes and solution to these problems is achieved via heuristics and/or approximation. A survey of graph partitioning algorithms is given in [11].

2.3 Layered Queuing Network Model

A layered queuing network (LQN) is a performance model for a hardware/software system. LQN models are capable of representing the software components and their deployment, capturing inter-component communications, and capturing resource interactions between layers of an application**.**

The main elements in an LQN are *tasks* (representing software processes), which are represented as rectangles or parallelograms. Tasks are deployable units of software and are interacting entities in a model. Tasks that do not receive any requests are called

reference tasks and represent load generators or users of the system, that cycle endlessly and create requests to other tasks [12]. Every task other than reference task, has a host processor that is represented by an oval (see Figure 2.5), which models the physical entity that carries out the operations.

A task has one or more *entries*, representing different operations it may perform and that are labeled with host (CPU) demand for one invocation of the entry. *Demands* are total average amounts of host processing and average number of calls for service operations required to complete an entry. A *call* from one entry to another is represented by an arrow labeled with the number of calls. Calls are requests for service from one entry to an entry of another task.

Every task and host is a queuing server, so requests to an entry first go into the queue for the task owning that entry, and is served when a task thread becomes available. The task thread then queries for the host server, to be executed. An infinite server provides a pure time delay (without any waiting for the server). It is modeled as an infinite task running on an infinite host. Infinite servers are used in this work to represent network latencies.

An example LQN model is shown in Figure 2.5. In this example, tasks are *t0, t1, t2, t3, t4* and *t5*. Corresponding entries are *e2601, e2602, e2603, e2604, e2605, e2606, e2607,* and *e2608*. The reference task is shown as *refTask* and its corresponding entry is *refEntry*. Processors represented by ovals for each task are *refP, e0P, e1P, e2P, e3P, e4P, e5P, e6P, e7P* and *e8P*. Entry *e2601* makes 2.31 calls to *e2607*, this is shown as an arrow between entries *e2601* and *e2607*. Similarly all other calls between different entries are shown with respective arrows.

2.4 Optimization

An optimization algorithm is a procedure for seeking an extreme value (maximum or minimum) for an objective function, possibly subject to constraints and limits on the ranges of the variables. With the advent of novel computing styles, superior quality optimization methodologies are needed to satisfy new requirements in nearly every commercial enterprise product to achieve required system performance. Various optimization techniques are discussed next.

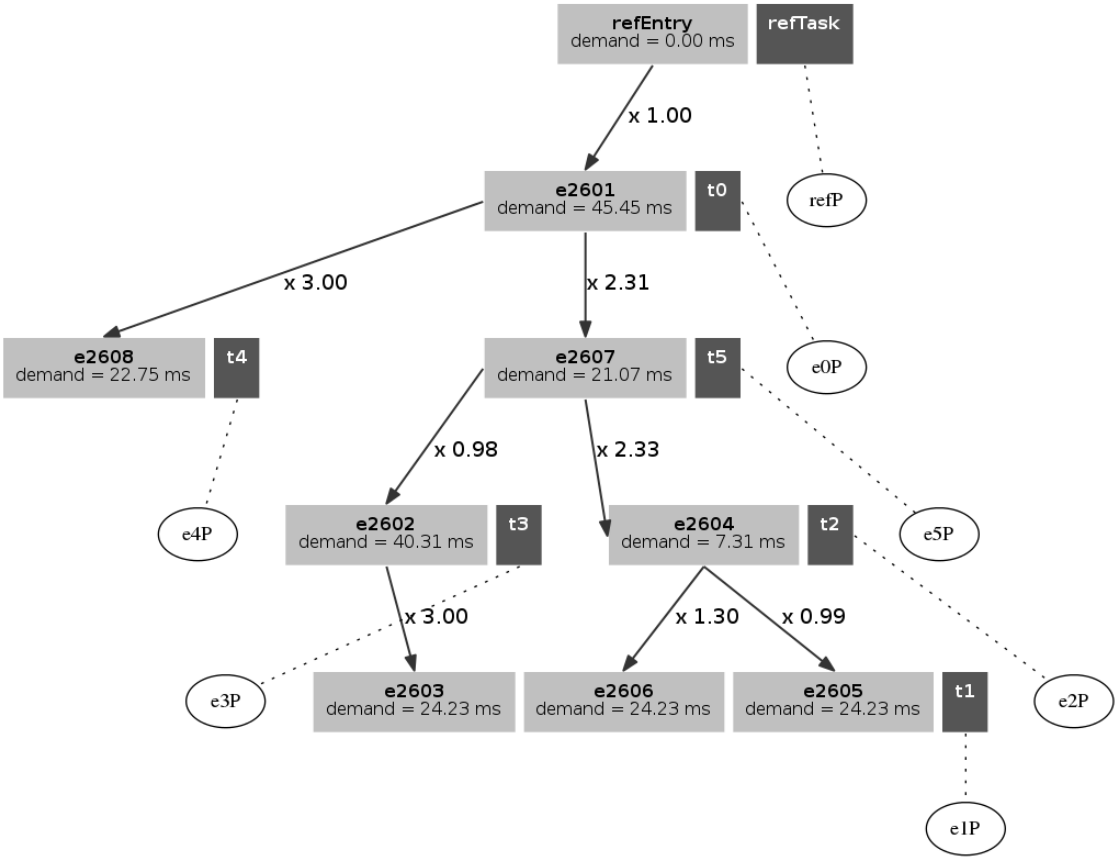

Figure 2.5: Layered Queuing Network (LQN) model example

2.4.1 Bin Packing

The bin packing problem (BPP) solves for the minimum number n of identical bins with capacity C that are needed to pack objects of individual requirements $C_1, C_2, C_3, \ldots C_m$ such that sum of the individual requirements of objects packed in a bin is less than the capacity (C) of that bin.

We can describe BPP mathematically as follows:

Objective function: minimize $\sum_{i=1}^{n} p_i$

$$\text{subject to } \sum_{j=1}^{m} q_{ij} C_j \leq p_i C \,, i \in \{1,2,3 \ldots n\} \,,$$

$$\sum_{j=1}^{m} q_{ij} = 1 \,, j \in \{1,2,3 \ldots m\} \,,$$

$$\text{where } p_i = \begin{cases} 1 & if\ bin\ i\ is\ used, \\ 0 & otherwise \end{cases}$$

$$q_{ij} = \begin{cases} 1 & if\ object\ j\ is\ assigned\ to\ bin\ i, \\ 0 & otherwise \end{cases}$$

BPP is known to be NP-hard [13]; hence it is usually solved with heuristics that provide an approximate solution.

In the above example we only consider one dimension (or constraint), i.e. capacity C, however BPP can be extended to take into account multiple dimensions (or constraints). This is called *multi-dimensional BPP*. Bin packing can also be classified as online and offline. For *online bin packing* items arrive one at a time in unknown order, each must be put in a bin before considering the next item. In *offline bin packing*, all items are given upfront.

In the two-dimensional bin packing problem (2D-BPP), a set of n rectangular items is given, each having height h_i and width w_i where $i \in \{1,2,3, \ldots, n\}$, and an unlimited number of finite identical rectangular bins, having height H and width W. The

problem is to assign all items to the minimum number of bins, with their edges parallel to those of the bins and without overlapping with each other. It is assumed that items cannot be rotated to fit a given bin i.e. length of the item should align with length of given bin not the height.

There are many possible ways to do this, like next-fit, best-fit and first-fit approach. First-fit approach allocates next item *i* to the first bin where it fits. If no bin can accommodate *i*, a new bin is created. Time complexity of this approach is O(*n*log*n*) [14].

2.4.2 Simulated Annealing

Simulated Annealing (SA) is a combinatorial optimization technique to find a good approximation to the global optimum of a given function in a large search space. It was introduced by Cerny [15] and Kirkpatrick [16]. For certain problems, simulated annealing may be more effective than exhaustive enumeration if the goal is merely to find reasonably good solution in a fixed amount of time instead of the best possible solution. More than that, exhaustive enumeration is not even possible for problems of even moderate scale. Simulated annealing belongs to a class of optimization algorithms that operate by generating a random initial solution and "exploring" the area nearby. If a neighboring solution is better than the current one, then it moves to it. If not, then the algorithm keeps the current solution. This is perfectly logical, but it can lead to a sub-optimal local solution. SA avoids getting stuck in local optima by allowing infeasible moves randomly. The probability of accepting a worsening move decreases gradually over time, which allows SA to converge to a best solution that is frequently close to optimal solution. Because of this it also takes a long time to converge to a solution and is

one of the major drawbacks of SA. Pseudocode of SA is shown in Figure 2.6. In the pseudocode, the following values are used:

- ***L*** is the epoch length.
- ***α*** is the cooling coefficient. It is the value by which temperature is diminished at each iteration.
- ***stopcondition***: Johnson's stopping rule is used which increments a counter by one at the end of an epoch when the fraction of accepted moves is less than a predefined limit F_{min}(=0.02). The counter is reset to 0 when a better solution is found. SA is also terminated when temperature drops below final temperature.
- ***Initial temperature*** $\boldsymbol{T_0}$ is the starting temperature for the algorithm.

2.4.3 Local Search Optimization Methods

Local search optimization methods attempt to find an optimal solution by moving from one solution to other in the search space of potential solutions by applying local changes within an acceptable time bound [17].

A local search method starts from a candidate solution and moves iteratively to a neighboring solution defined by neighborhood relation on the search space. Usually, every candidate solution has one or more neighboring solutions; the choice of which one to move to is taken using only information about solutions in neighborhood of the current one, hence the name local search. *Hill climbing* is a special case of local search methods in which the algorithm chooses to move to a solution that locally maximizes a given criterion.

Local search methods can be terminated either based on elapsed time or when successive new solutions are worse than the best solution found so far by the algorithm. In such circumstances, local search methods can converge to a non-optimal solution and become stuck at a *local optimal point* while an optimum solution is far from neighborhood of the solutions considered by the algorithm. This local-optima issue can be resolved by using different initial conditions to initiate local search, or more complex schemes like iterated local search, reactive search optimization, or simulated annealing.

INPUTS: initial solution S, initial temperature T_0, final temperature T_{final}, epoch length L, cooling coefficient α, stopping condition *StopCondition*

- $T \leftarrow T_0$
- **While** $T \geq T_{final}$ and *StopCondition* not met:
    - Perform the following loop L times:
        - Pick a random neighbor S' of S
        - $\Delta \leftarrow cost(S') - cost(S)$
        - **IF** $\Delta \leq 0$ (downhill move):
            - $S \leftarrow S'$
        - **IF** $\Delta > 0$ (uphill move):
            - $S \leftarrow S'$ with probability $e^{-\Delta/T}$
    - $T \leftarrow r \times T$ (reduce temperature).

OUTPUT: S

Figure 2.6: Pseudocode for Simulated Annealing (SA) algorithm

2.4.4 Kernighan-Lin (KL) Algorithm

The KL algorithm [18] is a move based algorithm that is used in graph partitioning problems where a graph is split into two by removing the minimal number of arcs in the graph. The KL method starts with an initial solution (or partition) and iteratively tries to improve this solution by moving vertices from one partition to another such that the total number of connecting arcs between the partitions decreases.

Fiduccia-Mattheyses [19] presented a modified KL algorithm (also called the FM algorithm) that improves the run time of the KL algorithm and can handle unbalanced partitions. It works by moving a vertex from one partition to another based on value of *gain* associated with those nodes. Gain represents the net change in arc cut weight that would result from moving a vertex from one partition to another. The gain g_{ixy} introduced by moving vertex *i* from partition *x* to partition *y* can be expressed mathematically as:

$$g_{ixy} = \sum_{j \in V; P(j)=y} c_{ij} - \sum_{j \in V; P(j)=x} c_{ij} \quad (1)$$

where *P(j)* is a function that returns the partition where vertex *j* is deployed. Figure 2.7 shows pseudo code of generic modified KL algorithm.

2.4.5 Flow Based Optimization: Maximum Flow Minimum Cut Problem

Graph connectivity is one of the classical subjects in graph theory, and has many practical applications, for example, in chip and circuit design, reliability of communication networks, transportation planning, and cluster analysis. Finding a minimum cut of an undirected weighted graph is a fundamental problem. Precisely, it consists in finding a nontrivial partition of the graphs vertex set V into two parts such that the cut weight, the sum of the weights of the arcs connecting the two parts, is minimum [20]. The Max-Flow-Min-Cut-Theorem by Ford-Fulkerson [21] states that when network flows are suitably defined from a single source node to a single sink node, with flows on arcs constrained by the arc weights, then the maximum value of flow equals the minimum cut-weight of any partition of the graph [22].

INPUTS: Initial current partition *current_partition*, partition *constraints*

- *best_partition* ← *current_partition*
- Compute initial gains for all moves of vertices from one partition to the other based on equation (1)
- Initialize *moved_list* to empty list
- Try all possible moves with positive gain. **Repeat** :
  - Find a vertex to move, not in *moved_list* with highest gain that satisfies given *constraints*
  - Update *current_partition* by performing the move and add vertex to *move_list*
  - Update gains of neighbors of moved node
  - **IF** *cut_weight(current_ partition)* ≤ *cut_weight(best_ partition)*:
    - *best_partition* ← *current_ partition*
  - **IF** there are no possible moves with positive gain:
    - Break out of loop

OUTPUT: *best_partition*

Figure 2.7: Pseudocode for generic modified KL algorithm

3. Optimal Deployment: Review of state of the art

This chapter reviews the state of the art of methodologies for finding the optimal deployment of an application on a cloud network, and discusses their strengths and weaknesses.

As explained in Chapter 1, it is very important to find an optimal deployment of given application on a cloud network to ensure that it consumes minimum power and meets various constraints that provide acceptable user experience. In the literature, there has been a lot of research to solve this problem. Since this is an NP-hard problem [23], approximate solutions have been proposed. These solutions can be broadly classified under:

1. Bin packing methods
2. Metaheuristic methods
3. Graph partitioning methods

3.1 Bin packing methods

Bin packing is one of the most commonly used approaches to find optimal deployment [24]. Coffman [25] presented the MULTIFIT algorithm based on a first-fit decreasing scheme to bin pack processing capacity. This was extended to include execution and inter-processor communication to create the MULTIFIT-COM algorithm [26]. A memory requirement was added that created an online bin-packing problem solved in [27]. Bin packing has also been used in conjunction with other techniques like max-flow algorithms by Tang et al. [28] to improve solution quality. Krave et al. [29] used bin packing based heuristics to evenly balance workloads across a virtual computing

environment. Recently, Paul et al. [24] used bin packing (greedy algorithms) based on random allocation (RA), next fit allocation, first fit allocation (FFA), best fit allocation and worst fit allocation to minimize total energy/power cost of deployment on a cloud network while considering assignment, capacity and placement constraints. Paul et al. [24] showed that best fit first allocation (BFA) is the best approach and using greedy algorithms might be faster for larger problems.

Bin packing algorithms are suitable for any problem size and can handle multiple constraints. They are also extremely fast to find a potential solution, however they suffer from frequent convergence to non-optimal solutions. In addition to this by itself, bin packing algorithms cannot handle latency constraints.

3.2 Metaheuristic methods

Hansen et al. [30] discussed various *Variable Neighborhood Search* (VNS) methods and applications. VNS involves iterative exploration of larger and larger neighborhoods for a given local optimum until an improvement is located after which time the search across expanding neighborhoods is repeated. It is based on three principles:

- The local minimum for one neighborhood may not be a local minimum for a different neighborhood,
- The global minimum is a local minimum for all possible neighborhoods, and
- Local minima are relatively close in value to the global minimum for many problems.

VNS is an efficient and robust method that is able to find optimal or near optimal solutions for a wide variety of large and small problems. Various local search methods can be used in conjunction with VNS to enhance the accuracy of the overall method. A genetic algorithm proposed by Ahmad and M. K. Dhodhi [31] assumes communication time to be negligible to perform scheduling on parallel processors. The genetic algorithm is combined with a list-scheduling heuristic where population members (chromosomes) are represented by arrays of task priorities. The first member in the initial population is determined by using a critical path method in calculating task priorities. The remainder of the chromosomes in the initial population are generated by randomly perturbing priorities (genes) of the first chromosome. In each generation, a list-scheduling heuristic is applied to all of the chromosomes to obtain corresponding cost function values. In E. S. H. Hou, N. Ansari, and H. Ren [32] members of the population are represented by lists containing tasks associated with one of the processors. The crossover and mutation operations take into account precedence relations between tasks. GA based algorithms can solve large multi-dimensional, non-differential, non-continuous, and even non-parametrical problems [33].

A. Thesen [34] presents Tabu search (TS) metaheuristics to schedule a set of *n* independent tasks on a network of *m* processors. Tabu search augments local search by allowing worsening moves if no improving move is available (this can happen when search is stuck at a local minimum). Furthermore, it keeps track of previously visited searches and prohibits the search from coming back to them. It can also be used with other local search methods

Johnson et al. [35] presented the effectiveness of the simulated annealing (SA) algorithm for the graph bi-partitioning problem. Performance of SA is highly dependent on the choice of different annealing parameters like initial temperature, cooling schedule, epoch length, and stopping condition. Park et al. [36] proposed a better set of annealing parameters to improve the results of SA. These methods can deal with arbitrary systems and cost functions and will find good solutions for many problems. This is because SA is less likely to get stuck at a local minimum because it makes random steps that gradually decrease in probability according to the cooling schedule. It is also very easy to code for large complex problems.

3.3 Graph partitioning methods

Flow networks can be employed to find an optimal task deployment for given application on multiple processors. Stone [37] presented a mathematical model for static assignment of modules/programs for a dual processor system to minimize computation costs on the system. Bokhari [38] extended Stone's work to allow dynamic re-assignment of modules during execution of the program by taking into account changes due to local reference patterns of the program. Bokhari [23] proposed a shortest tree algorithm minimizing execution and communication costs for an arbitrarily connected distributed system with an arbitrary number of processors that form a tree structure.

Another extension of Stone's work is a static assignment heuristic by Lo [39] that aims to reduce total execution and communication costs associated with the assignment by introducing the concept of interference cost representing incompatibility between two tasks thus causing tasks to be scheduled on different processors.

Move-based approaches try to iteratively improve graph partition by moving a vertex between partitions such as the Kernighan-Lin (KL) algorithm [18]. This algorithm converges to a local optimum by choosing moves that reduce the cost of the graph cut. Fiduccia and Mattheyses improved the KL algorithm to present the modified KL algorithm that has linear time complexity for graph partitioning [19]. These were described in Section 2.4.4 above.

To avoid the problem of local optima, move based algorithms can be combined with stochastic methods such as ant colony optimization [40], simulated annealing [41], or particle swarm optimization [42]. A multilevel approach [40], [41], [42], [43], [44] can be used for partitioning larger graphs where the original large graph is iteratively coarsened by merging vertices creating a smaller graph with similar structure. The multilevel scheme is used in state of-the-art graph partitioning libraries such as METIS [45], and SCOTCH [46].

Verbelen, et al. [8] try to find a feasible deployment of application(s) on multi-cloud network by introducing the modified Kernighan-Lin-Simulated Annealing (KL-SA) based hybrid algorithm. In contrast to traditional graph partitioning algorithm(s) that are restricted to balanced partitions, the algorithm takes infrastructure heterogeneity into account when calculating the best partition while minimizing bandwidth between modules; hence minimizing network usage instead of minimizing execution time for deployment optimization. For efficient partitioning, the multilevel KL based algorithm is utilized that permits calculations for real-time deployment. Another algorithm uses simulated annealing to improve the quality of solution for small graph sizes. Finally, a hybrid algorithm is introduced that creates slightly better partitions than KL with an

intermediate execution time. In the next section weaknesses and limitations of these methods in literature are highlighted.

3.4 Deficiencies of the State of the Art

With the advent of large-scale computing, new challenges have emerged. Scalability is the biggest challenge as with increasing number of users, more and larger applications need to be supported. Also, there is an increase in complexity in terms of number of simultaneous objectives, like minimizing latency and minimizing power consumption, and constraints, like limited memory and processing capacity. This has created a major burden for any allocation algorithm employed in cloud computing. Any new approach must also be able to cope with dynamic changes in requirements that can considerably change deployment leading to different allocations of application tasks in the cloud frequently and quickly.

Bin packing is very easy to implement and scales well with problem size. It can also be extended to add multiple constraints for partitioning. Furthermore, it can adapt well to dynamic systems where constraints might change frequently and fast re-calculation of a partition is required. However, like other greedy algorithms [24] it fails to converge to an optimal solution for larger problem sets. For the context of this thesis, bin packing algorithms alone cannot handle latency constraint. This makes bin packing a less attractive solution.

VNS methods, though efficient and robust, often suffer from the difficulty of creating neighborhoods to search within. Sometimes it is not possible to find a viable search neighborhood which renders VNS unusable. In other cases, search neighborhoods

are so large that the local search method is not able to find a suitable solution within these neighborhoods [30]. GA offers no assurance of finding a global optimum especially when populations are large. Genetic algorithms are generally slow and cannot assure constant optimization response times which limit their use in real time applications. The computation cost of genetic algorithms is also an issue and depends on the problem at hand [47].

Tabu search has been widely and successfully used as a combinatorial optimization method for finding good graph partitioning solutions. The main issue when extending Tabu search to handle multiple objectives is how to keep diversity so that points not necessarily within the neighborhood of a candidate solution can be generated. Also, it is imperative to develop effective procedures to build tabu lists and to select the next state to which the algorithm should move. Using Tabu search for exploring the neighborhood of solutions produced by another method seems a natural choice but this adds extra computational cost associated with local search [48].

One of the main disadvantages of SA is computation time that grows exponentially with problem size. Hence, for certain problems SA may end up requiring more iterations than exhaustive search [49]. SA gradually converges to optimal (or near optimal) solution by using a cooling function. Defining an effective cooling schedule is also one of the weaknesses associated with SA.

Flow networks oversimplify the problem to find an optimal solution that might not be useful for more complex problems. Pure flow networks ignore complex performance issues, such as blocking delays due to resource contention, which are in general nonlinear and NP-hard. Move based approaches like Kernighan-Lin (KL) and

modified KL algorithm, though very fast, often converge to a local optimal solution that leads to sub-optimal deployment. The primary disadvantage of iterative improvement methods is that their performance deteriorates for larger graphs. Verbelen et al. [8] take into account only the processing constraints while minimizing response time but they do not try to solve the problem of minimizing electrical energy cost.

4. Edge-Core Allocation Problem

In this chapter, the edge-core allocation problem is described first, followed by evaluation of state of the art algorithms in light of the described problem.

The problem at hand is to find a deployment for an application on a cloud network such that the power consumed is minimized. The final deployment should meet response time constraints so that quality of service is not compromised. Memory and processing capacity constraints on individual cloud networks have to be met so that application can be feasibly deployed on hosts in the network. Furthermore, the memory and processing capacities on individual host machines must be respected to deploy tasks. In addition, the final solution to this problem should be quick (i.e. it should return a solution within some realistic time limit) and robust (i.e. it should always return a feasible solution within a time limit that can be practically deployed on cloud network). In the remainder of this chapter, this problem is formulated for an edge-core cloud system with same objective and constraints starting with representation of an application with a graph structure derived from its LQN model.

4.1 Graph Representation of Application

Layered Queuing Network [12] models are capable of representing software components (tasks) and their deployment, to capture inter-component communications, and to analyze resource interactions between layers of the application. These LQN models of applications can be mapped onto a graph gathering all the essential information required to address our problem.

An application can be modeled as an undirected weighted graph derived from an LQN model. Let *G= [N,A]* be the derived weighted undirected graph where *N* are the nodes (tasks) in the graph *N*=($n_1, n_{2,}, n_3, .., n_n$) and *A* are arcs between communicating tasks. Edge and core clouds are represented as nodes with memory and processing capacity equal to the sums of the capacities in their host machines. Arcs represent communication between different software components. Every arc has certain weight associated i.e. communication latency introduced into each user response, by all calls between two tasks that correspond to nodes at ends of the arc. Arc weight is derived from the LQN model and is defined as network delay (in msec) times the number of calls. It is assumed that, should two tasks be co-resident on the same cloud (both on edge or both on core) the cost of communication between them is zero. Network delay is introduced if there is interaction between tasks across the clouds. There is one distinguished node called the Reference node, denoted *refNode,* pinned to the edge which interacts directly with the end user as it is usually closer geographically.

Each node n_i has a processing requirement (service demand) and a memory requirement to execute on allocated host. An edge-core cloud is assumed to have homogeneous hosts. Every host has a memory capacity representing the total memory available to run allocated task(s) from the application on that host and a processing capacity representing the total computing power available to run allocated task(s) from the application on that host. There are arcs between every task of an application and edge and core cloud nodes. These arcs represent requests for service by this task, on this cloud, in executing one user response. Processing time is computed from arc weights and demands.

Each task t has a processing demand DT_t in milliseconds (ms) per user response, computed from its entries and requests rates (calls). Let

- s_e = demand of entry e per invocation
- YE_e = invocations of e per user request
- DE_e = total demand of entry e per user request = $YE_e \times s_e$
- DT_t = sum over entry e belonging to task t of DE_e

For the optimization algorithm, a nominal maximum processing resource utilization U_{nom} will be assumed for all processing on a host machine or a cloud (e.g. $U_{nom} = 0.8$). This nominal resource utilization is a target utilization for the deployment solution, which allows us to approximate the contention delay by assuming a processor-sharing queuing discipline at the hosts. As a result, the processing time PT_t to satisfy 1 sec of service demand is assumed to be $1/(1 - U_{nom})$ sec, giving:

$$PT_t = \frac{DT_t}{(1 - U_{nom})} \tag{2}$$

An example of an LQN model is shown in Figure 4.1. Its derived graph is shown in Figure 4.2.

4.2 Edge Cloud vs. Core Cloud

For highly demanding and dynamic applications an edge cloud which is relatively small in terms of total processing power, relative to core cloud, is introduced close to some of the end users, to offer faster service. Transactions requiring heavy processing capacity, a larger memory requirement and longer processing times may be suitable to be executed at larger core cloud. The communication between application tasks deployed on

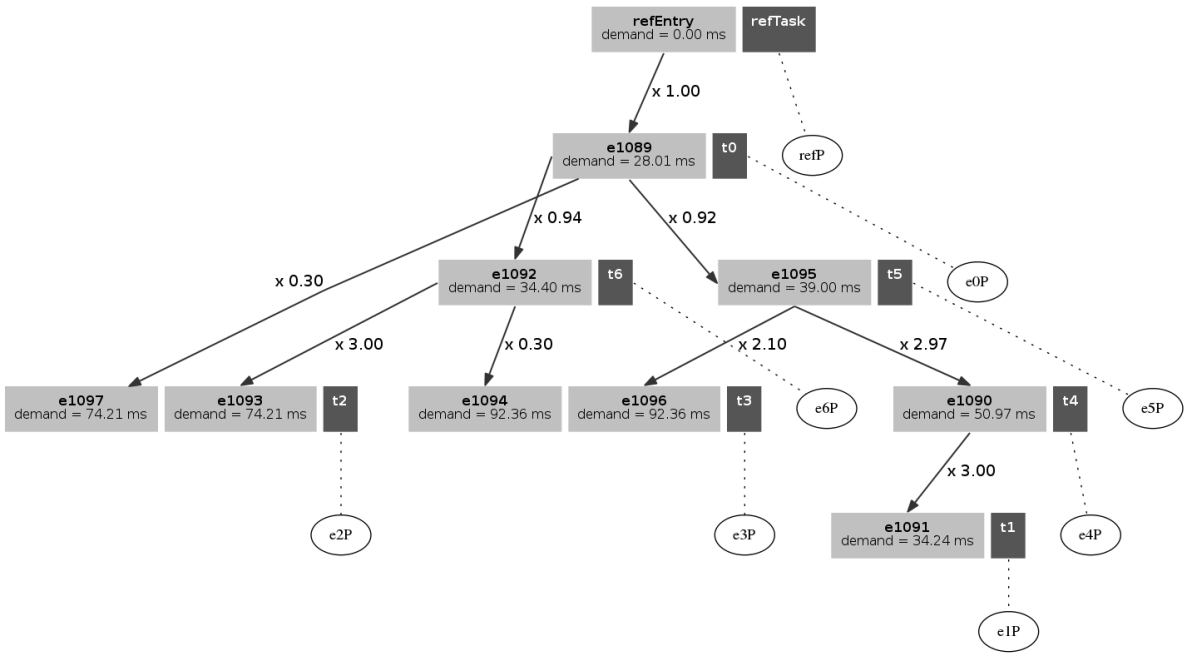

Figure 4.1: LQN model example

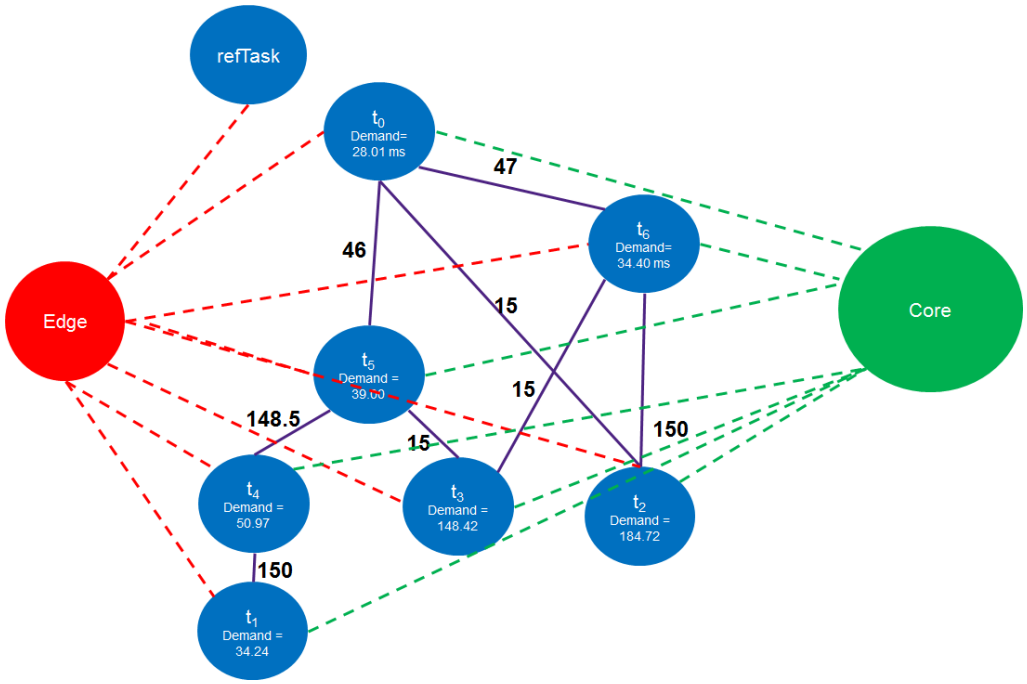

Figure 4.2: Graph from LQN model

edge cloud and core cloud introduce associated network latencies. In order to reduce this network latency the concept of an extended cloud has been introduced in [50]. In the

extended cloud scheme, data is cached on one or more edge clouds to improve scalability and efficiency of applications deployed on the core cloud to take advantage of the fact that edge cloud is closer to the end-user than core cloud and moving data and few portions of deployed application closer to the end-user of the system will improve system performance. Figure 4.3 shows the extended cloud architecture with one core cloud and one or more edge clouds.

An end user can use mobile devices or work stations to access the edge cloud (or simply edge) located possibly in the same area or vicinity. It is connected to the core cloud (or simply core) via a dedicated high-speed network to facilitate information exchange. However, due to its distant location, any request made by an end user takes more time to access the core when compared to the edge. In the past a lot of research has concentrated on the edge cloud to develop algorithms to maintain data consistency across different clouds [51] [52] [53], to reduce data access time [54], to partition data between edge and core clouds [55], etc.

Given the nature of the future applications listed in Chapter 1, low latency will be important. Therefore, it is vital to carefully assign tasks so that latency constraints are respected. Even though the edge has low latency because of proximity to the end user, it lacks computing power and storage capacity relative to core that may reduce the system performance in some cases when it must also access core. This edge to core communication adds extra latency to the overall user experience, which may be significant depending on the traffic patterns and deployment of the given application.

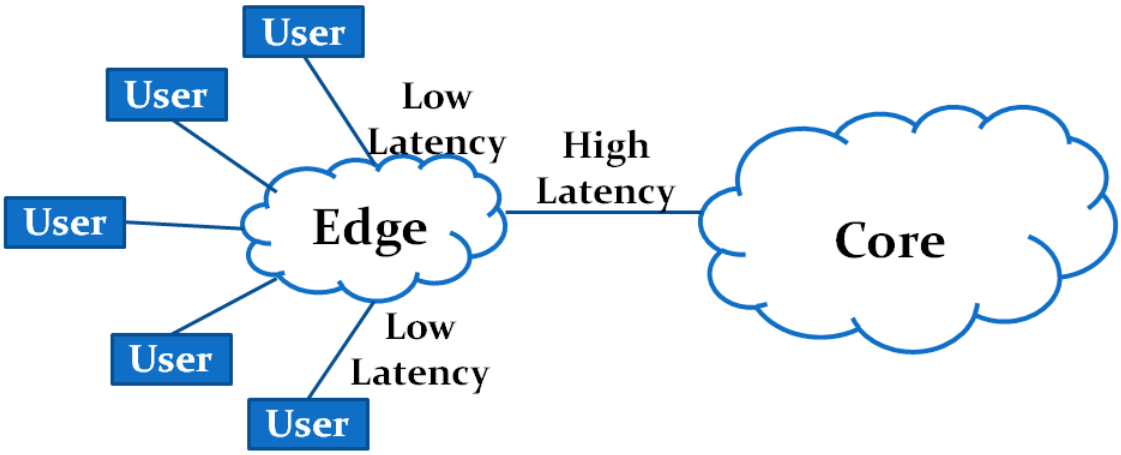

Figure 4.3: Extended cloud architecture

The major recurring cost factor for edge and core clouds is power consumption. All the hosts running on edge and core clouds constantly consume electric power and produce lots of heat that requires high quality air conditioning equipment which consumes even more power to prevent overheating of hosts. Since power is expensive and can easily overwhelm the economics of managing a sustained edge/core cloud network, it is imperative to minimize the amount of power consumed by only turning on minimum number of hosts required to support a particular application on the edge or core clouds. Next we formulate the problem for edge-core allocation to address this issue.

4.3 Edge-Core Allocation Problem Formulation

Edge-Core allocation can be modeled as a graph partitioning, if an application is modeled as a graph of a certain kind, where we divide a graph *G* into two subgraphs. Each subgraph is a collection of tasks (vertices) that are assigned to the edge/core cloud. Tasks (vertices) assigned to each cloud are limited by the capacity of each cloud. Since

capacity of edge cloud is very small relative to core cloud, we end up with a non-uniform graph partitioning scheme.

For graph *G* defined in Section 4.1, each node n_i is assigned two attributes:

1. PR^i: Processing requirement (service demand),
2. MR^i: Memory requirement.

An arc has arc weight c_{ij} associated with it. For task-to-task arcs, $c_{ij} = network\ delay \times number\ of\ calls\ between\ n_i\ and\ n_j$. For task-to-cloud arcs (red arcs in Figure 4.2); $c_{ij} = PT_i$ (calculated in Equation (2)). Network delay is introduced from edge to core or vice- versa. For communicating nodes in the same cloud (both in edge, or both in core), network delay is assumed to be negligible, but edge-core network delay is not.

The objective is to assign each node n_i to one of the hosts in the core or edge cloud such that the total power consumed by all the hosts used to deploy an application on the core and edge clouds is minimized provided the following conditions are satisfied:

1. Response time is less than a given threshold limit.
2. Processing capacity of each host in the edge and core cloud is greater than processing requirements of all the tasks allocated to that host.
3. Memory capacity of each host in the edge and core cloud is greater than the memory requirements of all the tasks allocated to that host.

We can express the power minimization objective mathematically as follows. Power cost of a host (P_h) on the edge or core can be expressed as the sum of execution cost and fixed cost. Execution cost can be approximated as linear function of CPU utilization (U_h) [56]

which in turn is proportional to host capacity used *(Ω_h)* to run all tasks allocated to that host. Hence, the power cost of a host can be written as:

$$P_h = C_{fh} + C_h * \Omega_h \tag{3}$$

where C_{fh} and C_h are machine specific constants and can be derived from linear regression. C_h is the power cost due to a unit of utilization (full utilization of a single core) on this host. C_{fh} denotes the fixed cost of host h, associated with the host being included in the deployment, with zero utilization. Therefore, the objective function to minimize becomes:

$$P_{total} = \sum_{j=1}^{m} \sum_{i=1}^{n_j} P_{\mathrm{h}}^{i} \tag{4}$$

where m is the total number of edge and core clouds and n_j is the total number of hosts used for an application in cloud j. The constraints on the objective function mentioned in (4) are as follows:

$$\sum_{i,j} h_{ij} c_{ij} \leq \Psi \tag{5}$$

$$\forall j \in (1, \dots m): \sum_{i} MR^{i} \times X_{ij} \leq MC_{\mathrm{h}}^{j} \tag{6}$$

$$\forall j \in (1, \dots m): \sum_{i} PR^{i} \times X_{ij} \leq PC_{\mathrm{h}}^{j} \tag{7}$$

$$\forall j \in (1, \dots m) \; and \; \forall k \in \left(1, \dots n_j\right): \sum_{i} MR^{i} \times Y_{ikj} \leq MC_{\mathrm{h}}^{k} \tag{8}$$

$$\forall j \in (1, \dots m) \; and \; \forall k \in (1, \dots n_j): \sum_{i} PR^{i} \times Y_{ikj} \leq PC_{\mathrm{h}}^{k} \tag{9}$$

where Ψ is the response time threshold, h_{ij} is equal to 1 only if node (or task) i and node (or task) j are not allocated to the same edge or core cloud and is zero otherwise, MR^i is memory required by i^{th} task to run on a given host, MC_{h}^j is the memory capacity of a host on j^{th} cloud, MC_{h}^k is the memory capacity of k^{th} host on a given cloud, PR^i is the processing power required by the i^{th} task to run on a given host, PC_{h}^j is the processing power capacity of a host on j^{th} cloud, and PC_{h}^k is the processing power capacity of k^{th} host on a given cloud. X_{ij} is equal to 1 when task i is allocated to cloud j otherwise it is zero. Y_{ikj} is equal to 1 when task i is allocated to host k on cloud j otherwise it is zero. Equation 5 represents the response time constraint where the response time represented by arc cut weight is checked against a threshold value. Equations 6 and 7 represent overall memory and processing requirement constraints for all cloud networks under consideration. Equations 8 and 9 represent memory and processing requirement constraints for individual host(s) on different cloud networks. Next, the ability of state of the art algorithms to solve this problem are evaluated.

4.4 Evaluation of State of the Art Algorithms applied to Edge-Core Deployment

Existing algorithms in literature have tried to solve the stated problem by multiple methods/techniques as briefly explained in Chapter 3. In this section, a brief evaluation of the state of the art methods is given in light of problems defined by Equations (4), (5), (6), (7) and the requirement for a quick and robust algorithm.

Bin packing [57] [24] [25] cannot consider a large number of constraints as it becomes a multi-dimensional bin packing problem with is very slow to solve and solution

is not always guaranteed [13]. Bin Packing cannot handle latency constraints by itself. Hill climbing is not robust and fails to find a feasible solution very often [58], VNS and Tabu search need a rich pool of potential solution neighborhoods to search which often leads to failure in finding an acceptable solution [30]. Move based algorithms also suffer from the fact that they frequently return poor quality solutions and cannot handle larger problem size [59]. Genetic algorithms and simulated annealing algorithms are too slow to converge to an acceptable solution to be used in practice. Furthermore, these algorithms are highly sensitive to the initial starting solution [59]. Verbelen [8] presents three algorithms based on KL and SA taking into account processing capacity constraints while minimizing response time. However they are missing memory capacity constraints and do not address power minimization which is an important aspect of the allocation problem in cloud computing.

A summary of the state of the art is given in Table 4.1.

Table 4.1: Comparison of state of the art methods. ** If incorporated into objective function.*

Method	Decision Quality	Scalable	Extensibility	Can handle latency constraint	Run time
Bin packing	Heuristic	Yes	Weak	No	Fast
VNS	Local Optimal	Yes	Weak	*	Fast
GA	Heuristic	Yes	Good	Yes	Slow
Tabu search	Local Optimal	Yes	Weak	*	Medium
Hill Climbing	Heuristic	Yes	Good	*	Fast
SA	Heuristic	No	Good	*	Slow
Flow networks	Local Optimal	Yes	Good	Yes	Medium
KL/FM	Local Optimal	Yes	Good	Yes	Fast

5. New Edge-Core Partitioning Algorithm

In this chapter, a graph representation of the problem derived from LQN model is discussed first and then the details of the proposed algorithm to solve the power minimization problem are presented. The proposed algorithm is named Heuristic Algorithm for Service-centre Resource UTilization (HASRUT).

The algorithm tries to minimize power consumption in a given application by allocating these tasks to edge and core clouds such that all given constraints are met. The HASRUT algorithm uses a multi-level graph partitioning algorithm and has these main elements:

1. PHASE 1: assignment of tasks to edge and core clouds:
    i) Graph coarsening (explained in 5.2.1),
    ii) Random initial allocation,
    iii) Graph partitioning using a modified form of the Kernighan-Lin (KL) algorithm, and
    iv) Uncoarsening and refinement.
2. PHASE 2: assignment of tasks to hosts for power calculation:
    i) For each cloud, bin-pack the tasks assigned to it on the hosts in the cloud,
    ii) Calculate the total power consumption using Equation (4).

Local optima are avoided by using many random initial starting solutions (or allocations). For each initial solution the HASRUT algorithm calculates power consumption associated with the final deployment and chooses the solution having minimum power.

5.1 HASRUT Algorithm

The HASRUT algorithm takes a graph described in Section 4.1 as an input. Each task on the graph has one or several replicas deployed in the service center or cloud infrastructure, which is assumed to consist of *m* identical machines. The solution given by the HASRUT optimization algorithm is a deployment:

- dividing service demand between multiple task replicas, where applicable,
- allocating host reservations to tasks,
- minimizing power consumption, and
- meeting memory and processing capacity constraints.

To meet the service demand of a task, more than one replica of a task may be required (scaling out). The total processor utilization of task *t* is given by:

$$UT_t = f \times DT_t \tag{10}$$

where *f* is throughput in user requests/sec. and if it is assumed that nodes are single-core with maximum utilization of U_{nom}, then the replicas must fit into a single core each. If there are m_t replicas of task *t*, of equal total demand per user request of $DR_{t,r} \leq U_{nom}$, then the number of replicas of task *t* must be at least this value:

$$m_t = ceiling(f * \frac{DT_t}{U_{nom}}) \tag{11}$$

After the replication is done, the graph is updated with the new task replicas. The arc weights of replicated tasks interacting with other tasks are updated.

5.2 Multilevel Graph Partitioning Approach

Multilevel partitioning has three phases: coarsening, partitioning and refinement, uncoarsening and refinement.

5.2.1 Graph Coarsening

Graph coarsening is the first phase of the proposed multi-level graph partitioning algorithm. It groups vertices together and builds a condensed, smaller graph whose nodes are these groups. As the number of vertices increases, the number of possible partitions grows exponentially. The main advantage of coarsening is that it reduces the number of possible partitions, but at the cost of reduced accuracy. Therefore a good partition of the coarsened graph is much easier to identify than a good partition of the original graph. The price paid for this reduction in complexity is that only a small number of the possible fine graph partitions are represented and are therefore examinable on the coarse graph. However, by working on different levels, the local refinement scheme improves the partition on multiple scales. A partition obtained from the coarsened graph may not directly induce a very good partition of the original graph; the repeated application of the local refinement largely resolves this problem [60].

5.2.1.1 Construction of a Coarse Graph

During coarsening, a larger graph G_i is reduced to another graph G_{i+1} that has fewer nodes compared to G_i by removing arcs and merging nodes connected by removed arcs. The weight of the combined node is the sum of weights of the combined nodes (as shown in Figure 5.1). When two combined nodes have an arc to a third node, these two

arcs are collapsed into one arc with a summed arc weight (as shown in Figure 5.2). Coarsening can be done multiple times such that a set of arcs with common nodes are merged at each coarsening step. Collapsing heavily weighted arcs is beneficial in finding a small arc cut as these are unlikely to be in the best minimum cut.

The Heavy-Edge Matching (HEM) algorithm described in [61] is employed to perform coarsening. A given node in the graph is chosen for merging with its neighboring node such that the arc joining these two nodes has the maximum weight of all incident arcs for the given node. There is no particular order in which nodes are coarsened. A node cannot be merged twice in a given coarsening step to simplify node tracking during uncoarsening phase.

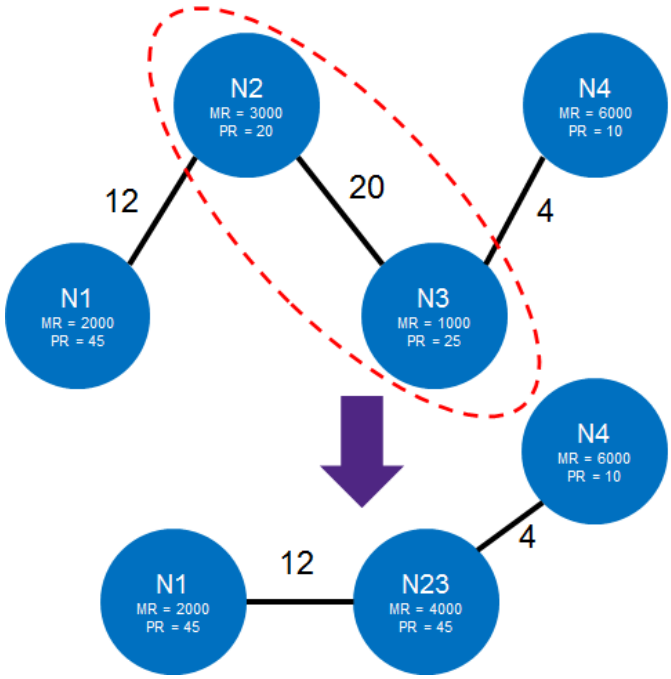

Figure 5.1: Coarsening example 1

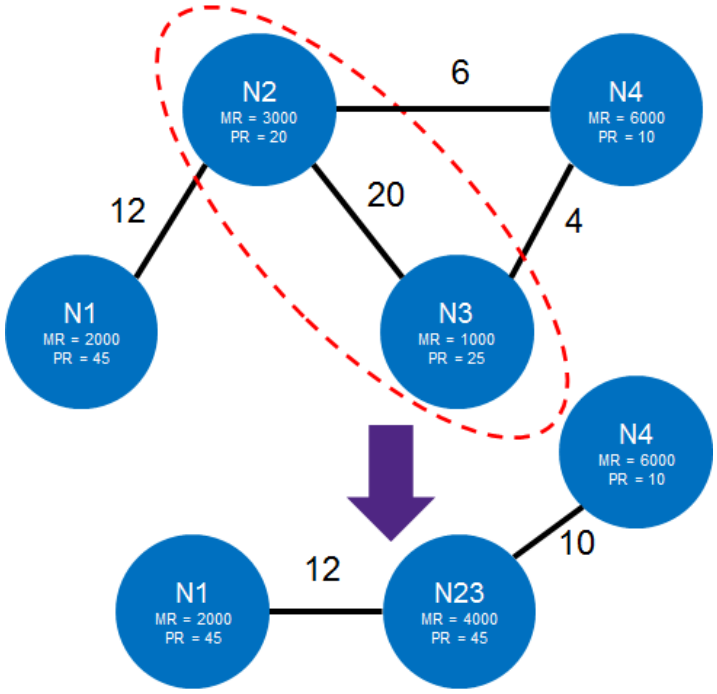

Figure 5.2: Coarsening example 2

5.2.2 Initial Allocation, Partitioning and Refinement

After coarsening, the graph is next partitioned and partitions are refined using the modified KL-based algorithm that takes into account cloud memory and processing capacity while keeping response time below a certain threshold value. The primary objective is to choose a feasible deployment of tasks on the given infrastructure of cloud networks. This step assumes that there are enough machines and resources available to find such a deployment.

5.2.2.1 Initial Allocation

For creating an initial allocation the modified KL-like algorithm is used. However, it requires a starting allocation, which is created by randomly allocating tasks

to the edge and core clouds. This has two advantages: first it gives an initial solution to start modified KL iterations; second it provides a wide sampling of the search space in an effort to find global optimum. The starting allocation may meet memory and processing constraints for the edge and core cloud along with a response time threshold. In such a case, the HASRUT algorithm does not refine this allocation and takes the starting allocation as the final deployment for next step. For cases where the starting deployment does not satisfy constraints or the response time threshold, the HASRUT algorithm refines this allocation as explained in next section. It is to be noted that a bin packing scheme could also be used [8] to create an initial allocation.

5.2.2.2 Refinement of the Starting Allocation

After starting with the random starting allocation, modified KL iterations create a better deployment for the coarsened graph by moving tasks from the edge to core and vice-versa.

The pseudocode of the main refinement algorithm is shown in Figure 5.5. It has two major loops called the outer and inner loops. The outer loop tries to find a better cut weight by moving vertices (tasks) from one cloud to another in each iteration. For instance it can try to move tasks from one edge allocation to another edge/core cloud or vice versa to find a better cut weight allocation. The outer loop is terminated after 3 tries if it cannot improve the best partition so far, indicating that the algorithm has reached a local optimum. A vertex can only be moved once during an iteration of the outer loop to avoid outer loop cycling i.e., going around and around a loop of identical solutions. The inner loop iteratively selects a vertex to move. Normally the vertex with the largest gain

is selected to move, if it satisfies the memory and processing capacity constraints. Note that this also could move a vertex with a negative gain which gives the algorithm the possibility to escape to some extent from local optima.

Once a vertex is moved, the gains of all of its neighbors are updated to reflect the move and this vertex is added to *MovedList* that keeps track of all vertices that have been moved once. This process completes when no more suitable moves are found. In order to descend as deep as possible to find local optima, the inner loop only accepts moves with positive gain and then starts a new iteration when a better allocation is found. In this new iteration, *MovedList* is cleared so that moves that are already in *MovedList* from the previous iteration can again be performed to come closer to the optimum.

Instead of creating balanced allocations, it is ensured that no allocation gets overloaded in terms of violating total memory and processing capacity of that cloud. Forbidding vertex moves that would violate any capacity constraint of the cloud is a trivial but ineffective option as it causes the algorithm to get stuck in local optima, from which there is no escape (see Figure 5.3). To avoid local optima, moves are allowed that reduce the amount of overload on the given cloud(s). If a cloud is heavily overloaded in terms of memory or processing capacity (i.e. tasks allocated to that cloud have larger total memory and processing requirements than the total capacity of the cloud), a vertex is selected with highest gain that will reduce the amount of overload on this allocation. This way situations as in Figure 5.3 will also converge to a better solution such as is given by allocation shown in .

When two possible moves have equal gain, the tie is broken by:

- Making a move that does not exceed the capacity of target cloud,

- If still a tie, make the move to non-empty cloud,
- If still a tie, make a move randomly.

A threshold value for the arc cut weight is introduced that can be compared to the cut weight of given deployment (combination of allocations). Arc cut weight is the weight on arcs joining nodes allocated to different clouds. If the cut weight of current deployment is less than or equal to this threshold value, the deployment is feasible for response time and the algorithm is terminated. This provides a mechanism to save execution time to find an acceptable deployment, instead of continuing until all vertices are moved.

5.2.3 Uncoarsening and Refinement

Next the graph is uncoarsened, which creates new nodes/tasks in the current deployment, and modified-KL iterations are applied to refine the deployment. Once all tasks have been uncoarsened, a final deployment is created that contains tasks from the original graph. At this point the total memory and processing capacity constraints on edge and on core are satisfied. But in order to calculate the total power consumption, tasks have to be assigned to individual hosts without exceeding their capacities. This is explained in the next section.

5.3 Power Calculation

The power calculation is done with a detailed host-by-host allocation using offline two-dimensional bin packing. Hosts on the edge and core are assumed to be homogeneous and the number of hosts on the edge and core allocation is also known.

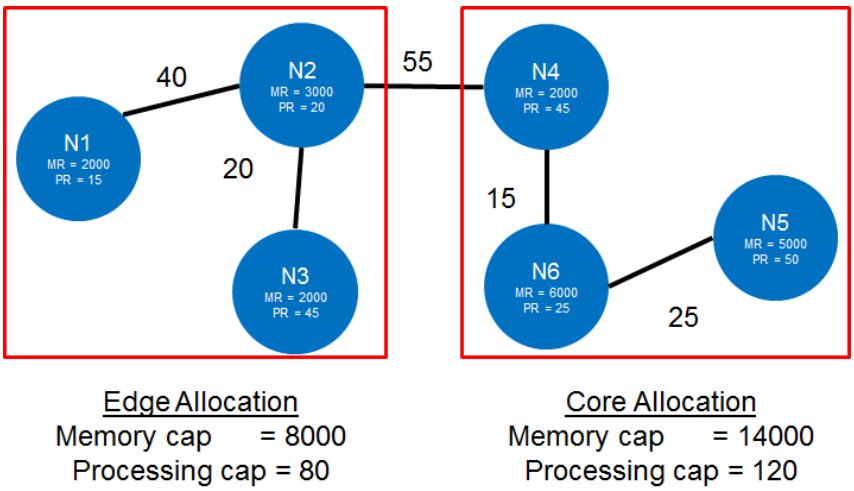

Figure 5.3: Local optima example

This determines the bin size for allocating tasks to their constituent hosts to facilitate utilization calculation that is required to compute power for given deployment. If the two-dimensional bin-packing algorithm is unable to find a feasible packing, then a failure is reported. This can happen if constraints are too hard to meet or the algorithm is unable to converge to a proper solution.

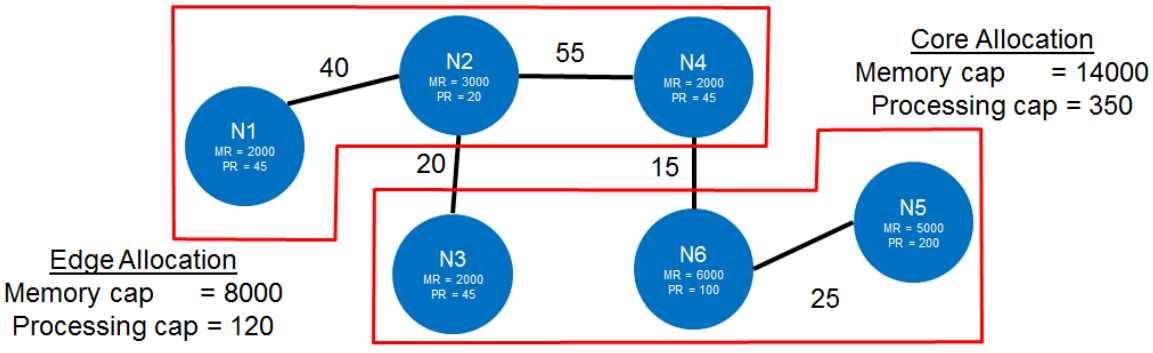

Figure 5.4: Local optima issue fixed

5.3.1.1 Details of HASRUT Algorithm

The complete HASRUT algorithm can be described as follows and is shown in Figure 5.6. The task graph is read in and converted to a node-replicated graph. Next, depending on the number of nodes in the input graph ($Graph_{input}$) it may be coarsened to create coarsened graph ($Graph_{coarsened}$). If number of nodes in $Graph_{input}$ is more than 50, heavy edge matching (HEM) is used to create $Graph_{coarsened}$. Coarsening can be done in multiple steps if needed to reduce the number of nodes in $Graph_{coarsened}$.

Next a check for infeasibility is performed which determines if there is not enough capacity on the edge or core side to host the tasks. This is done by comparing maximum of memory and processing requirements of all individual tasks to the maximum of memory and processing capacity of all individual hosts in edge and core. If the maximum task processing or memory requirement is greater than the maximum host processing or memory capacity then it is impossible to deploy task(s) on any of the hosts which makes any deployment infeasible. Another infeasibility check is done by comparing total memory and processing requirements of all tasks to total memory and processing capacity of all hosts on the edge and core cloud. If total memory and processing requirement of all tasks is more than total memory and processing capacity of all hosts then this will again result in infeasible deployment as all tasks cannot be assigned to hosts without violating their capacities.

If no infeasibility is detected, a random allocation is created that randomly assigns tasks from given graph to edge and core allocations. These random allocations are refined by using the modified KL-based algorithm shown in Figure 5.5. Once the refined allocation is found on the coarsened graph, the graph is uncoarsened and allocations are

refined again using the modified KL-based algorithm. This process of uncoarsening and refinement is repeated until the nodes/tasks of original graph ($Graph_{input}$) are obtained from successive coarsened graph(s). This final refined deployment is then used to calculate power based on Equation (3). The final deployment and its corresponding power are stored in a list.

Once all generated random allocations have been refined, the refined allocation with the minimum power is output as the solution. The algorithm ensures that all initial random allocations are unique. It is to be noted that not all possible combinations of allocations are tried in interest of saving run time. Instead a certain number of random allocations are generated, sufficient to provide a good covering of the solution space.

5.4 Competing Algorithms

In evaluating HASRUT in Chapter 7, two other competing algorithms were used for comparison. The details of these algorithms are included here.

5.4.1 Extended modified MLKL Algorithm

The modified KL algorithm shown in Figure 2.7, was extended for comparison with the HASRUT algorithm by adding the multi-level scheme and power consumption calculation using the same 2D bin packing algorithm as used for HASRUT algorithm. Pseudocode of this extended algorithm is shown in Figure 5.7.

INPUTS: *current_deployment,* graph, numbers of hosts on edge and core clouds, *threshold*

- *best_deployment* ← *current_deployment*
- Compute initial gains for all moves of tasks from one cloud to the other
- Compute the total free processing capacity and the total free memory capacity on the edge and core clouds
- Find a better allocation. **Repeat:**
  - Initialize *moved_list* to empty list
  - Try all possible moves with positive gain. **Repeat** :
    - *move_found* ←*false*
    - To find task to move. **Repeat**:
      - Select task to move, not in *moved_list*, with highest gain. Assume that the move would shift task *t* from cloud *p* to cloud *p′*.
      - **IF** cloud *p* is overloaded (processing or memory capacity):
        - **IF** moving task *t* to cloud *p′* would result in reduced processing and memory capacities in cloud *p′* that are still greater than the processing and memory capacities in cloud *p* prior to the move, then *move_found* ← *true*. Break out of loop.
      - **Else** *move_found* ← *true* and break out of loop
    - **IF** *move_found* is *true*:
      - Update *current_deployment* by performing the move and add task to *moved_list*
      - Update gains of neighbors of moved tasks
      - Update free processing and memory capacities on edge and core
      - **IF** *cut_weight(current_deployment)* ≤ *threshold*:
        - *best_deployment* ← *current_deployment*
        - Output *best_deployment* and exit algorithm.
      - **Else_IF** *current_deployment* has improved memory and processing capacity load than *best_deployment*:
        - *best_deployment* ← *current_deployment*
      - **IF** there are no possible moves with positive gain:
        - Break out of loop.
    - **ELSE** exit loop

OUTPUT: *best_deployment*

Figure 5.5: Pseudocode of the modified KL-based refinement algorithm

INPUTS: Input task graph ($Graph_{input}$), numbers of hosts, memory and processing capacity of hosts on edge and core clouds

- **IF** total memory and processing requirement of all tasks is more than total memory and processing capacity of all hosts on edge and core cloud then quit as no feasible deployment possible.
- Use HEM coarsening scheme to create coarsened graph $Graph_{coarsened}$ from input task graph $Graph_{input}$
- **IF** maximum task memory and processing requirement is less than or equal to maximum host memory and processing capacity*:*
    - Initialize *list_of_all_deployments* to *empty_list*
    - **For** all random initial deployments generated from tasks in coarsened graph:
        - **IF** this is a new unrepeated *current_initial_deployment:*
            - Perform modified KL with *current_initial_deployment* to create *current_deployment*
            - Uncoarsen and refine *current_deployment* via modified KL at each uncoarsening step to create *current_final_deployment*
            - Find *current_power* using 2D bin packing for *current_final_deployment*
            - Add *current_final_deployment* and *current_power* to *list_of_all_deployments*

OUTPUT: deployment having minimum power in *list_of_all_deployments*

Figure 5.6: Pseudocode for the entire HASRUT algorithm

INPUTS: Input task graph ($Graph_{input}$), numbers of hosts, memory and processing capacity of hosts on edge and core clouds

- Coarsen the task graph $Graph_{input}$ to create $Graph_{coarsened}$ by using HEM scheme
- Create an *initial_deployment* by randomly allocating tasks from $Graph_{coarsened}$ to edge and core clouds
- Perform modified KL as explained in Fig. 2.7 with *initial_deployment* to create *current_deployment*
- Uncoarsen and refine *current_deployment* via modified KL as explained in Fig. 2.7 at each uncoarsening step to create *current_final_deployment*

OUTPUT: Power for *current_final_deployment* using 2D bin packing algorithm

Figure 5.7: Pseudocode of the extended modified MLKL algorithm

5.4.2 Extended SA algorithm

SA has also been extended for comparison against the HASRUT algorithm. Pseudocode of the extended SA algorithm is shown in Figure 5.8. The memory constraint has been added and power consumption calculation is done using same 2D bin packing algorithm as used for the HASRUT algorithm. Epoch length L is calculated as $s \times N \times (K - 1)$ where N is number of tasks, K is number of machines used and s is equal to 50. The value of the cooling coefficient α is taken to be 0.908. The values of these control parameters were taken from [8]. The starting temperature $\boldsymbol{T_0}$ value is between 200-350 and varies depending on the example. In the case where there is not enough memory or processing capacity on target cloud, a task is moved based on the probability calculated from the capacity that is violated (i.e. capacity is negative). For instance, if moving a task from cloud p to cloud p' meets memory capacity but overloads cloud p' for processing capacity then the move probability is calculated based on the overloaded processing capacity as it will be a negative number. If both capacities are violated, the move is performed based on the capacity that gives maximum probability.

INPUTS: Input task graph ($Graph_{input}$), *current_deployment*, initial temperature (T_0), final temperature T_{final}, epoch length L, cooling coefficient α

- *best_deployment* ← *current_deployment*
- $T \leftarrow T_0$
- **Repeat** until stopping conditions are met:
    - *counter* ← 0
    - **Repeat** until *counter* ≥ *L:*
        - *counter* ← *counter* + 1
        - Calculate gain g_i to move task i. Assume that the move would shift task t from cloud p to cloud p'
        - **IF** $g_i \geq 0$:
            - **IF** there is enough processing and memory capacity on cloud p' by moving task i from cloud p to cloud p' perform move
            - **ELSE:**
                - Select the overloaded processing or memory capacity on cloud p' if task i is moved from cloud p to cloud p'
                - perform move with probability $e^{\frac{selected\ capacity\ overload\ on\ cloud\ p'}{T}}$
        - **Else_IF** ($g_i < 0$):
            - **IF** there is enough processing and memory capacity on cloud p' *by* moving task i from cloud p to cloud p' perform move with probability $e^{\frac{g_i}{T}}$
        - **IF** *cut_weight(current_deployment)* < *cut_weight(best_deployment)*:
            - *best_deployment* ← *current_deployment*
        - $T \leftarrow \alpha \times T$

OUTPUT: Power for *best_deployment* using 2D bin packing algorithm

Figure 5.8: Pseudocode for the extended SA algorithm

6. Experimental Setup

This chapter describes the experimental setup, including the test models, comparison algorithms, and metrics for comparison, and software and hardware details. The goals of these experiments are (1) to validate the effectiveness of the new algorithm, and (2) to assess the characteristics of the new algorithm. The algorithm was tested and validated by comparing it with modified versions of two competing and well-established algorithms (KL and SA) over a set of 49 test cases that are representative of future-oriented applications.

6.1 Edge and Core Characteristics

In all our experiments, the number of hosts on the edge cloud is 10 while there is an infinite number of hosts on the core cloud. Identical hosts are used in the edge or core cloud with each host having memory capacity equal to 8 GB and processing capacity equal to 1 GHz. Communication latency between edge and core cloud is taken to be 50 milliseconds. It is assumed that an individual host can have a maximum utilization of 80%.

6.2 Test Models

The characteristics of future applications are likely to differ significantly from the characteristics of typical current applications. Some applications may require higher processing capacity while others may have higher memory demand. Future applications are predicted to require more data exchange between tasks or with the user while simultaneously having much tighter latency constraints. To model such applications,

validate and verify the optimization algorithms a set of 49 test cases was created. The architecture of each application was defined by the structure of an LQN model. Figure 6.1 shows one of the test models.

The first task *refEntry* in Figure 6.1 is an example of a task that does not receive any requests. Demands are the total average amounts of host processing and average number of calls for service operations required to complete an entry. For example, task *e581* has a processing demand of 64.57 ms. and *t6* is host processor assigned to it. Every task (entity) has a host processor, which is the physical entity that carries out the operations. This separates the logic of an operation from its physical execution. Calls are indicated on the arrows from one task to other; they are requests for service from one entry to an entry of another task. The structure of LQN shows fan-in and fan out of calls. This shows the variability in interaction between different layers of an application. We use think times to represent network latency between every pair of communicating tasks that are not located in the same cloud. We constructed 49 test models which show various combinations of system attributes as shown in Table 6.1.

The test cases are randomly generated application architectures, represented by LQN model structures, in which workload parameters (CPU demands of entries, and numbers of calls) were generated randomly. The test cases are constructed as tree like structures with service requests distributed across different servers. A memory demand parameter (also chosen randomly) is added for more complete view of applications. Each entry has a mean service time chosen randomly from a uniform distribution U(1, 100) milliseconds, number of calls U(1, 20), memory requirement for each task U(512, 64000)

bytes. Details of the application test cases are shown in Appendix A and summarized in Table 6.2 below.

Table 6.1: Parameters used to create test models

Test Case type	No. of Test cases of each type	Service Demand	No. of calls (inter-task communication)	Total no. of tasks
Type 1	8	Low	Low	Low
Type 2	7	Low	Low	High
Type3	7	Low	High	Low
Type 4	4	Low	High	High
Type 5	6	High	Low	Low
Type 6	7	High	Low	High
Type 7	7	High	High	Low
Type 8	3	High	High	High

Table 6.2: Summary of test cases in Appendix A

Minimum Theoretical Response Time (msec)	1481.75
Maximum Theoretical Response Time (msec)	10100.1
Average Minimum Response Time (msec)	4046.2
Average Maximum Response Time (msec)	5112.89
Average Number of tasks before replication	17.12
Average Number of tasks after replication	28.69
Average Minimum Power Consumption (W)	32.22

The response time threshold is set at a value between the minimum theoretical response time (the total processing time) and the maximum theoretical response time

(processing + communication time). It is expressed as percentage of processing time. The minimum theoretical number of hosts is given by the minimum of the sum of hosts used in edge and core clouds calculated based on memory (MR) and processing requirement (PR) of all tasks and corresponding memory and processing capacity of each host in the edge and core clouds. It is calculated as in Equation (12). The minimum number of hosts is then the maximum of hosts calculated from MR and PR. The minimum power consumed by a cloud is calculated from the minimum number of hosts e.g. if the minimum number of hosts is equal to 14.39, then the total minimum power consumed is equal to 14 fully utilized hosts and one host at 0.39 utilization, resulting in total minimum power consumption of 28.617 W. The power consumed by a single host is given by Equation (3).

The attributes of an application model, derived from the above are shown in Table A.1 of the Appendix A in terms of total number of tasks, minimum and maximum response time, minimum number of hosts from demand/memory capacity, and minimum power consumption. Table 6.2 summarizes test case details by providing bounds and average values of theoretical response time, number of tasks and power consumption.

$$Minimum\ number\ of\ hosts = max\left\{\left\lceil\frac{\sum_{j=1}^{n} MR^{j}}{memory\ capacity\ of\ a\ host}\right\rceil, \left\lceil\frac{\sum_{j=1}^{n} PR^{j}}{processing\ capacity\ of\ a\ host}\right\rceil\right\} \quad (12)$$

where n is the total number of tasks of an application.

6.3 Competing Algorithms

Results are compared to those obtained via modified versions of Simulated Annealing (SA) and multi-level KL (MLKL) algorithm presented in [8]. These algorithms were implemented in Jython 2.5.4rc1 [62] environment using JGraphT library [63]. Details of these algorithms are presented in Section 5.4.

6.4 Metrics

HASRUT and the competing algorithms are compared on the basis of the following metrics:

1. *Power consumption of the final deployment*: The main objective of our research is minimization of power consumption. Power consumption has been compared for HASRUT and competing algorithms relative to minimum theoretical power consumption for all test models as shown in Appendix A.
2. *Response time of final deployment*: Response time is an important constraint as it affects QoS. The response time requirement or threshold should be met, but it is desirable to go even farther below that value. Response time is also compared against minimum theoretical response time as shown for test models defined in Appendix A.
3. *Run time:* Total algorithm run time is also one the main evaluation metrics. This is important as the user wants to have the deployment calculated as fast as possible, meeting all the constraints and minimizing power, especially in cases where system requirements change dynamically.

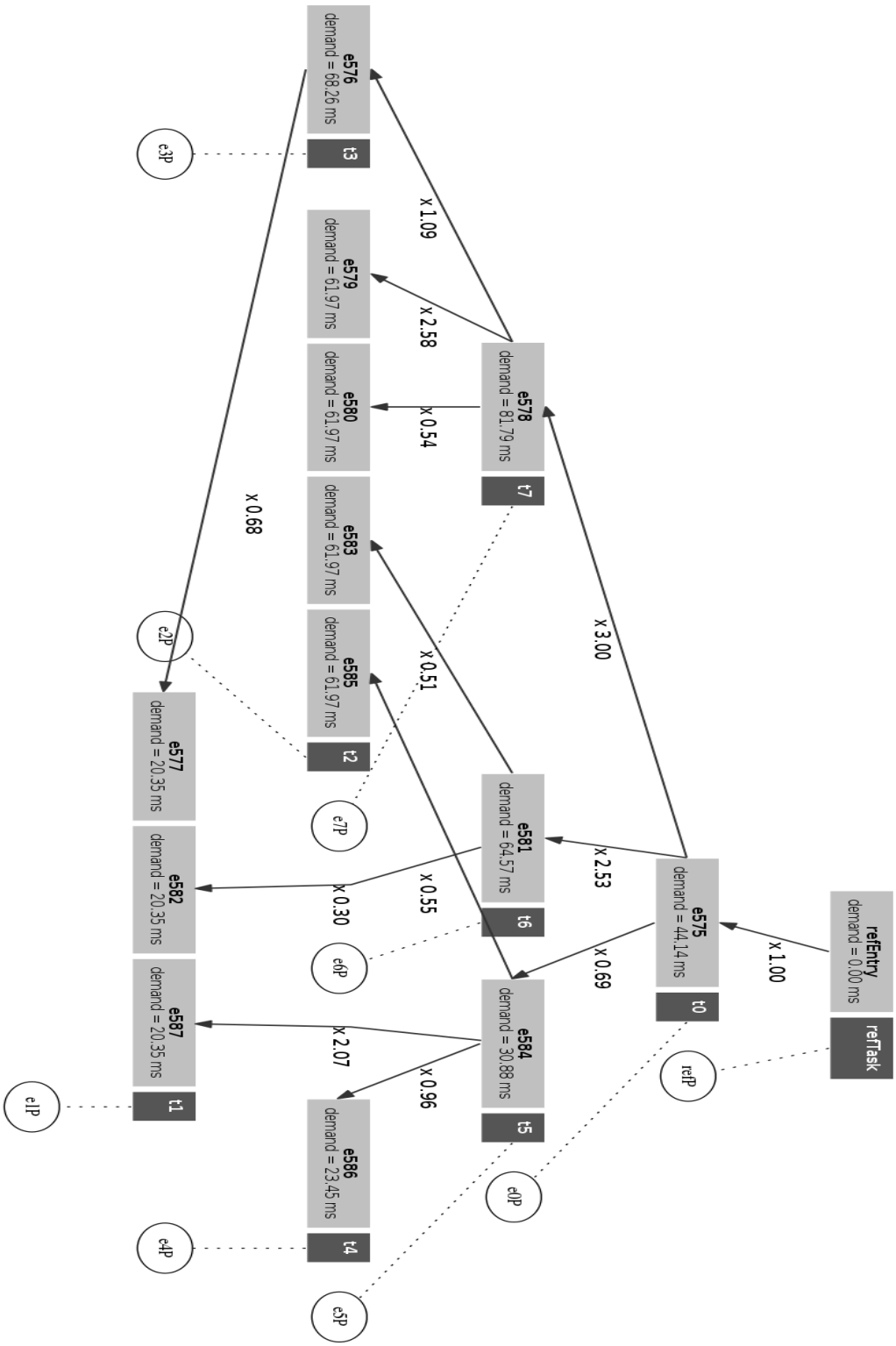

Figure 6.1: Example of generated test model

4. *Number of failures*: Failure is the inability of an algorithm to satisfy one or more of the constraints within some time limit. It can also occur when the tasks cannot be bin packed on the hosts available on their assigned cloud. This is a measure of the robustness of the algorithm. A smaller number of failures indicates a more robust algorithm.
5. *Time required to compute the first feasible result*: The first feasible solution can be very useful for dynamic systems where fast calculation of deployment is required because of changing system requirements and load. However, it does not optimize the power consumption.

All algorithms are run 5 times for each test case and the metrics reported are an average value over these 5 runs. HASRUT algorithm uses 500 initial random allocations. This number was chosen based on experimentation and generally produced the best deployments within a reasonable run time for the test cases under consideration.

6.4.1 Performance Profiles

Performance Profiles [64] are a useful way to summarize the comparison of different algorithms on a given metric. Performance profiles are distribution functions for a performance metric. In this thesis, a performance profile plots the ratio of a metric to the smallest value of that metric among the competing algorithms against the percentage of total testcases having that ratio or better. The ratio approach provides information on the percentage improvement and eliminates negative effects of allowing a small portion of problems to dominate the conclusions. Performance

profiles are shown for evaluating HASRUT, SA and MLKL for power consumption, robustness, response time constraint and run time metrics.

Interpretation of Performance Profile: For performance profile shown in Figure 7.1, say a vertical line is drawn perpendicular to the x-axis at value of 1.05. This line crosses the MLKL line at around 0.55 which means that for about 55% of the test cases MLKL has results within 5% larger than the lowest power consumption. Next this vertical line crosses the SA line at about 0.7, which means that for about 70% of the test cases SA has results within 5% of the lowest power. Further, this vertical line doesn't cross the HASRUT algorithm line at all because HASRUT has the lowest power for 100% of the models shown by the point (0,1). So, for a given performance profile, a curve near the top leftmost corner of the plot is considered superior to the other curves.

6.5 Hardware and Software

Experiments were carried out on a PC with Microsoft Windows 7 Enterprise OS and following characteristics:

- Processor: Intel Core i7-3770
- CPU speed : 3.4 GHz
- Physical Memory: 16 GB
- Free Disk Space: 144 GB

While executing the experiments, no other applications were running on the machine. This ensures that system resources were used mostly by the running experiment and there is no resource contention that might interfere with the results.

HASRUT algorithm has been implemented in Jython version 2.5.4rc1 [62] using Java graph library JGraphT [63]. HEM coarsening and node replication algorithm have been implemented in Python 2.7.6 [65].

7. Experimental Results

In this chapter, experimental results are presented that compare the HASRUT algorithm and competing algorithms on the test models described in Chapter 6. Various evaluation metrics have been used to compare all algorithms. Discussions and observations derived from results are also presented. Experiments are performed under two major categories. The first category caters to validation of results obtained from proposed HASRUT algorithm. The solution characteristics of the HASRUT algorithm are explored in the second experiment category.

7.1 Experiment 1: Validation of Effectiveness of HASRUT Algorithm

7.1.1 Evaluation of Power Consumption, Run Time and Response Time

HASRUT and competing algorithms are compared based on response time, power consumption and run time. Power is computed by assuming it is a piecewise linear function of utilization as shown in Equation (3), and the arc cut weight represents the response time for a given deployment. Run time is measured from the start of the algorithm (including graph input time) to when the algorithm stops, having produced a final deployment that it considers is best i.e. meeting all constraints and having small power consumption.

A network delay of 50 ms between edge and core clouds is assumed. The edge cloud is assumed to have 10 hosts and the core cloud is assumed to have an unlimited number of hosts and hence unlimited memory and processing capacity.

Detailed results are provided in Table A.2 in Appendix A. Figures 7.1-7.3 summarize the results in the form of performance profiles. The ideal place on a

performance profile is exactly where HASRUT algorithm is on the profile in Figure 7.1, at (0,1), i.e. the algorithm gets the best result among the competitors in 100% of the cases.

Table 7.1 shows statistics of the three metrics. MLKL seems to be a better algorithm across all metrics when looking at these results. This is because HASRUT and SA include results from test cases for which MLKL failed to produce any feasible result. Table 7.2 shows results for comparable subset of models. Comparable subsets of models are the set of models for which all of the algorithms return at least one solution. From this table, HASRUT algorithm is superior in terms of power consumption, response time and number of failures as compared to MLKL and SA. This matches the conclusion from performance profiles. The best average values for each metric is shown in boldface in all

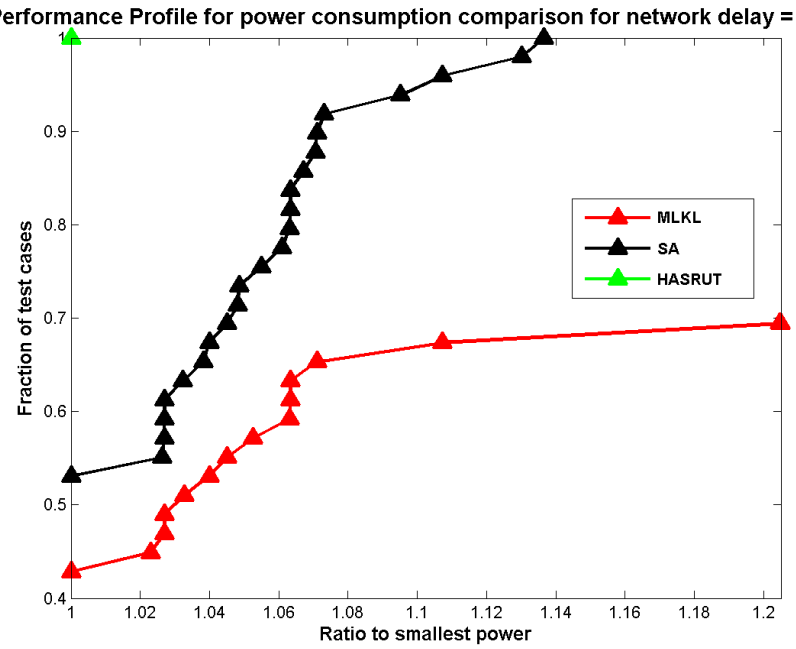

Figure 7.1: Power consumption performance profile for HASRUT, SA and MLKL for network delay = 50 ms

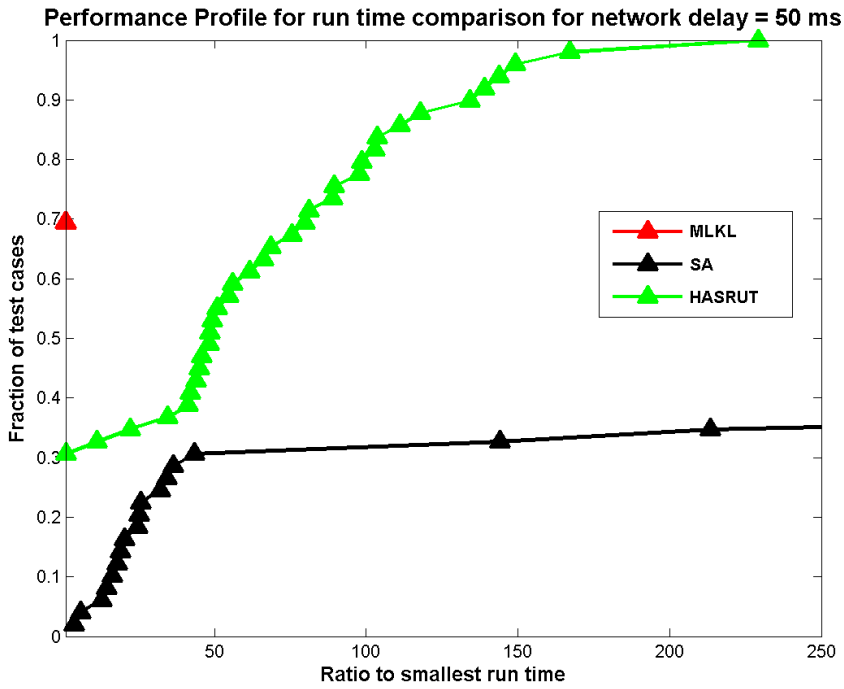

Figure 7.2: Run-time performance profile for HASRUT, SA and MLKL for network delay = 50 ms

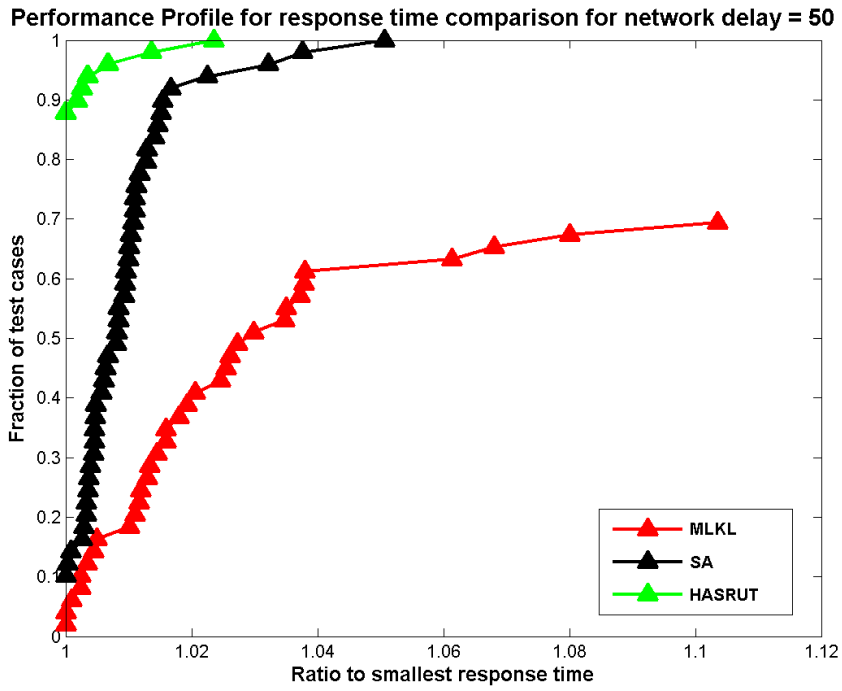

Figure 7.3: Response time performance profile for HASRUT, SA and MLKL for network delay = 50 ms

tables. It can be seen from Table 7.1 and Table 7.2 that power consumption, response time and number of hosts does not change much but there is considerable difference in the values for run time and number of failures.

This evaluation shows that:

- For all test cases under consideration, the HASRUT algorithm always finds the lowest or equal power compared to SA and MLKL as shown in performance profile for power consumption in Figure 7.1. It is to be noted that power is highly dependent on number of hosts used by an algorithm. Effect of utilization on power is very negligible. This is because C_f=0.3 is much less than C_{fh}=1.62 (see Equation (3)).
- The HASRUT algorithm performs better for larger problem sizes with high inter-task communication like test cases of type 7 and type 8 mentioned in Chapter 6 in terms of finding minimum power (see Table 7.3).
- The HASRUT algorithm also finds a lower response time relative to MLKL and SA as shown in Figure 7.3, Table 7.2 and Table 7.3. For some test cases SA and

Table 7.1: Comparison between HASRUT, SA and MLKL for network delay = 50ms

		HASRUT	SA	MLKL
Response Time	**avg**	4066.59	4092.26	**4065.29**
	std dev	1713.91	1700.26	1546.97
Power	**avg**	38.53	39.41	**37.46**
	std dev	17.85	17.70	15.61
No. Hosts	**avg**	21.49	22.03	**20.82**
	std dev	10.09	9.98	8.66
Run Time	**avg**	19.19	405.33	**0.23**
	std dev	18.20	410.08	0.19
No. Failures	**avg**	**0.00**	**0.00**	3.92
	std dev	0.00	0.00	0.98

Table 7.2: Comparison between HASRUT, SA and MLKL for network delay = 50ms for comparable subset of models

		HASRUT	SA	MLKL
Response Time	**avg**	**3991.67**	4012.92	4065.29
	std dev	1564.37	1554.78	1546.97
Power	**avg**	**36.65**	37.73	37.46
	std dev	15.41	15.18	15.61
No. Hosts	**avg**	**20.29**	20.96	20.82
	std dev	8.54	8.39	8.66
Run Time	**avg**	17.83	367.42	**0.23**
	std dev	18.84	368.74	0.19
No. Failures	**avg**	**0.00**	**0.00**	3.92
	std dev	0.00	0.00	0.98

Table 7.3: Comparison between HASRUT, SA and MLKL for network delay = 50ms for comparable subset of models for type 7 and 8 test cases

		HASRUT	SA	MLKL
Response Time	**avg**	**6243.7**	6246.7	6258.9
	std dev	1316.9	1311.8	1312.6
Power	**avg**	**57.7**	58.8	59.6
	std dev	16.1	15.2	15.3
No. Hosts	**avg**	**32.3**	33	33.5
	std dev	9.0	8.4	8.5
Run Time	**avg**	44.1	995.3	**0.5**
	std dev	29.8	427.5	0.2
No. Failures	**avg**	**0.00**	**0.00**	4.3
	std dev	0.00	0.00	0.7

MLKL find a deployment with a smaller response time compared to HASRUT; HASRUT finds a deployment with relatively lower power consumption instead at the cost of increased response time.

7.1.2 Evaluation of First Feasible Result

The run time of MLKL is always better than HASRUT algorithm, though it fails very frequently. However if power optimization is not important, the first feasible result

returned by HASRUT algorithm can be used instead of the final result. As shown in Figure 7.4, the first feasible HASRUT result is obtained in much less time than the final HASRUT result, and also generally in much less time than MLKL. Details for this experiment are shown in Table A.5 in the Appendix A. SA is not included in the first feasible solution results because its implementation is not structured to assess allocation feasibility at intermediate solutions. Power consumption and response time reported by the first feasible result from the HASRUT algorithm is not always the best but the results are acceptable as can be seen in the performance profiles in Figure 7.5 and Figure 7.6 . Statistics of the values are shown in Table 7.4 and Table 7.5, which shows that the average solution time for the first feasible solution is two orders of magnitude faster than for the full HASRUT algorithm. However, there is ~1.5% degradation in the average power consumption value reported by the first feasible solution relative to full HASRUT algorithm for all test cases.

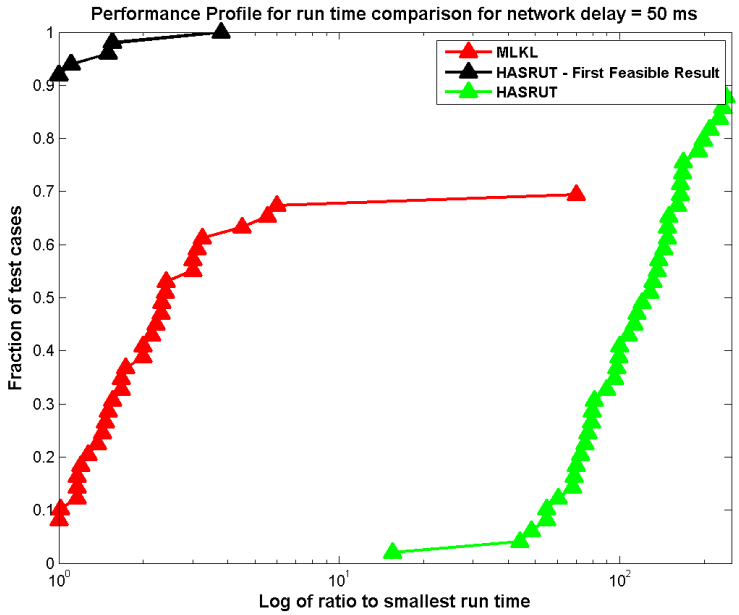

Figure 7.4: Performance profile for run-time comparison between first feasible result of HASRUT, final result of HASRUT and final result of MLKL

7.1.3 Evaluation of Robustness

Robustness is a critical attribute of any algorithm, i.e. the algorithm should not fail to provide a feasible deployment if one exists. The HASRUT algorithm and SA did not fail on any of the 49 test models we evaluated, compared to MLKL that has a very high rate of failures (30% of test cases). MLKL fails to find a feasible deployment because it gets stuck at local infeasible minima for the move calculations and cannot

Table 7.4: Comparison between HASRUT, HASRUT first feasible result and MLKL for network delay = 50ms

		HASRUT	HASRUT- First Feasible Result	MLKL
Response Time	**avg**	4066.24	4076.66	**4065.29**
	std dev	1713.44	1706.24	1546.97
Power	**avg**	38.53	39.09	**37.46**
	std dev	17.85	17.77	15.61
Run Time	**avg**	19.19	**0.17**	0.23
	std dev	18.20	0.17	0.19
No. Failures	**avg**	**0.00**	**0.00**	3.92
	std dev	0.00	0.00	0.98

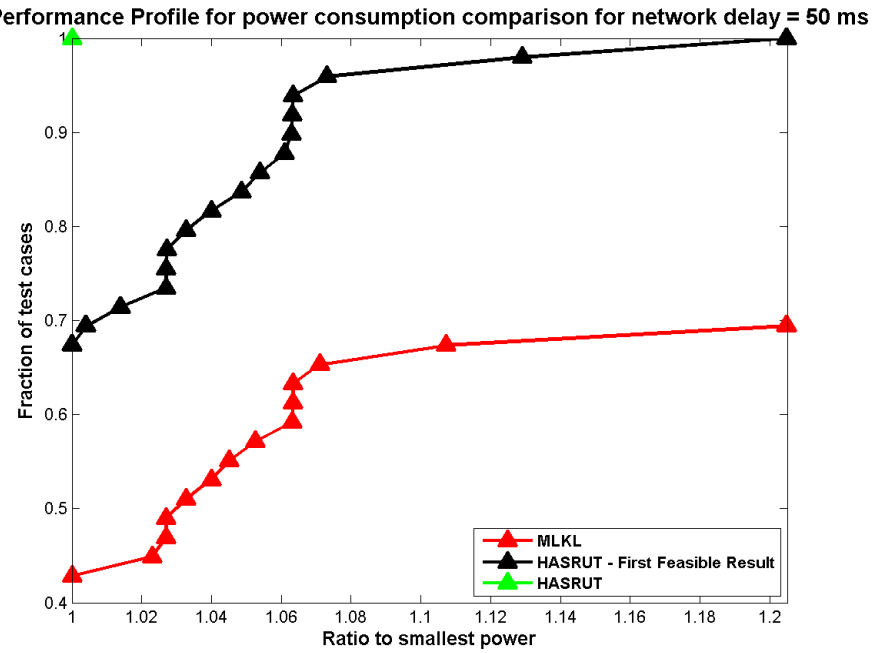

Figure 7.5: Performance profile for power consumption comparison between first feasible result of HASRUT, final result of HASRUT and final result of MLKL

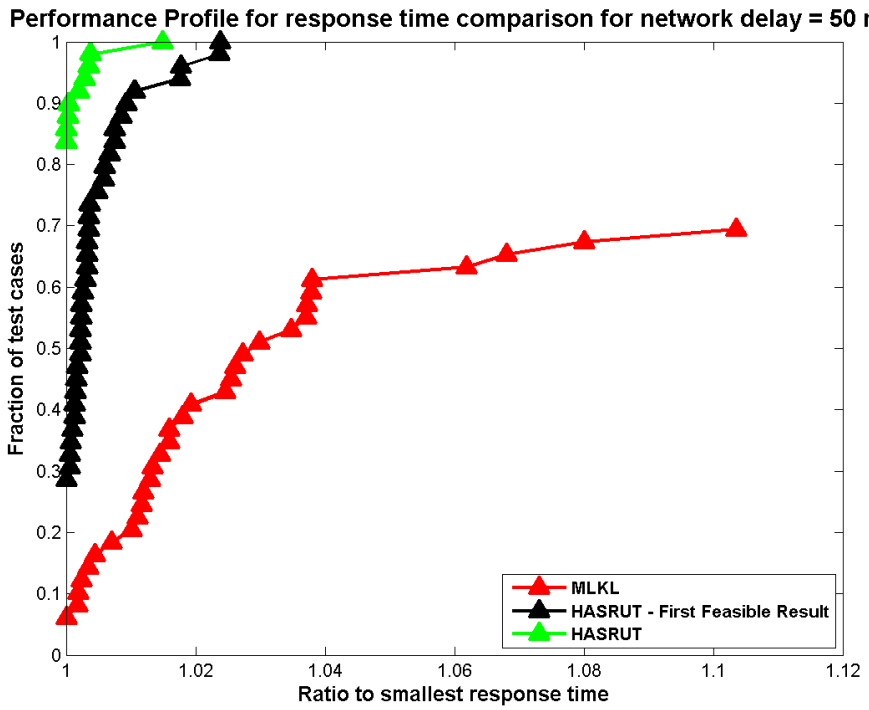

Figure 7.6: Performance profile for response time comparison between first feasible result of HASRUT, final result of HASRUT and final result of MLKL

Table 7.5: Comparison between HASRUT, HASRUT first feasible result and MLKL for network delay = 50ms for comparable subset of models

		HASRUT	HASRUT- First Feasible Result	MLKL
Response Time	**avg**	**3991.17**	4007.11	4065.29
	std dev	1563.59	1561.65	1546.97
Power	**avg**	**36.65**	37.28	37.46
	std dev	15.41	15.24	15.61
Run Time	**avg**	17.83	**0.13**	0.23
	std dev	18.84	0.10	0.19
No. Failures	**avg**	**0.00**	**0.00**	3.92
	std dev	0.00	0.00	0.98

satisfy the response time constraint. So it is relatively less robust than the HASRUT algorithm and cannot be used in critical applications where robustness is a primary requirement. The failure rate of MLKL algorithm for the test cases under consideration is shown in Figure 7.7.

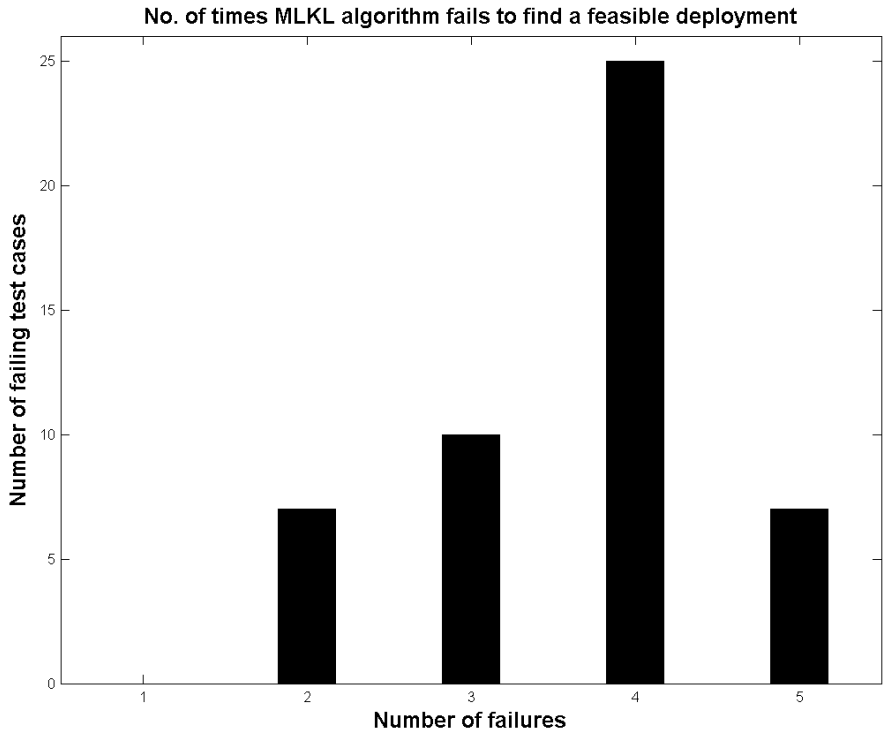

Figure 7.7: Number of times MLKL algorithm fails to find a feasible deployment out of 5 attempts

7.1.4 Impact of Variation of Network Delay

Network latency has an obvious impact on response time. However it also affects power consumption and algorithm run time because of trade-offs that must be made to meet the response time constraint. The limited capacity of the edge cloud is also a factor: it is unlikely that all of the communicating tasks can be accommodated on the edge cloud to eliminate the cloud-to-cloud network delay.

The test cases may become infeasible if overly long network communication delays are assumed. The same experiments were repeated by changing the network delay value from 50 ms to 100 ms and 200 ms to perform this evaluation. Results are summarized in Figure 7.8 - Figure 7.13. The actual scale of values is shown in Table 7.6 - Table 7.9 to augment the performance profiles.

A few observations based on this evaluation are as follows:

1. The HASRUT algorithm is still able to find a deployment with the least power consumption when compared to SA and MLKL. Network delay variation did not have any impact on the relative performance of the compared algorithms in terms of power consumption results as shown in Figure 7.8 and Figure 7.9. This is also reflected in Table 7.7 and Table 7.9 where average power consumption of the HASRUT algorithm is better than MLKL and SA.
2. As network delay increases the response time differences between all algorithms also increase. Note that the range of response time ratios is not that large anyway. The HASRUT algorithm generally comes up with better response time and with better or equal power compared to MLKL and SA as can be seen in Figure 7.10 and Figure 7.11. The scale of values in Table 7.7 and Table 7.9 shows that average response time of HASRUT algorithm is better than MLKL and SA.

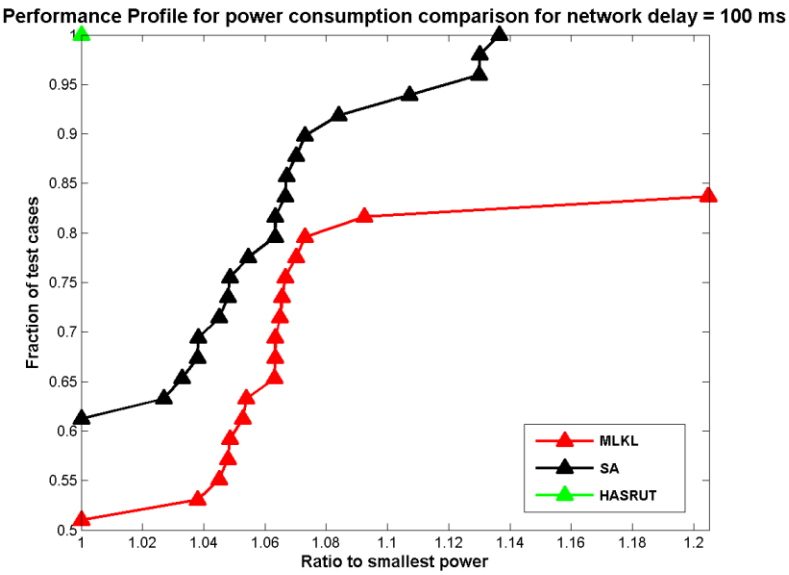

Figure 7.8: Power consumption performance profile for HASRUT, SA and MLKL for network delay = 100 ms

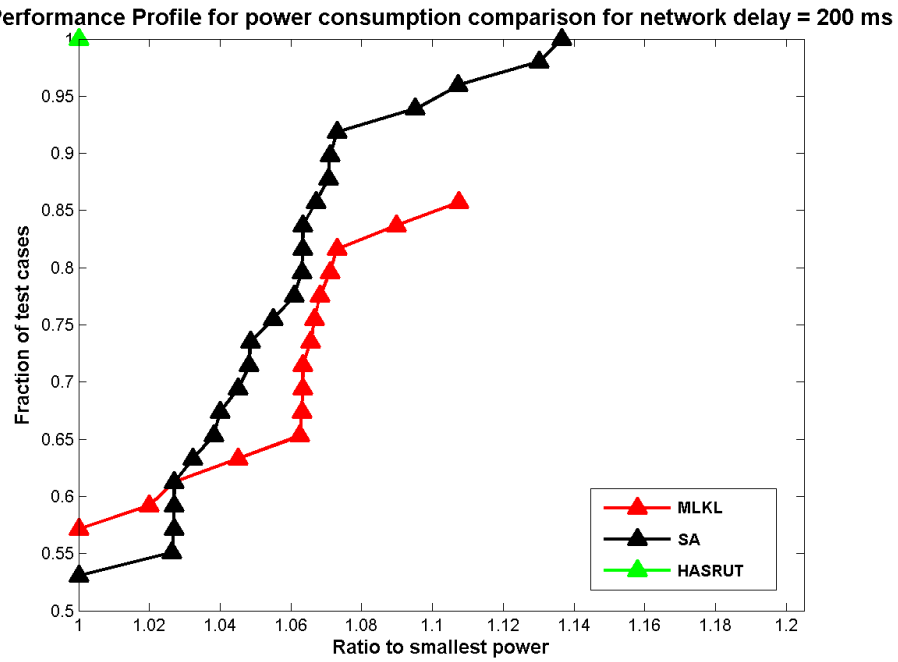

Figure 7.9: Power consumption performance profile for HASRUT, SA and MLKL for network delay = 200 ms

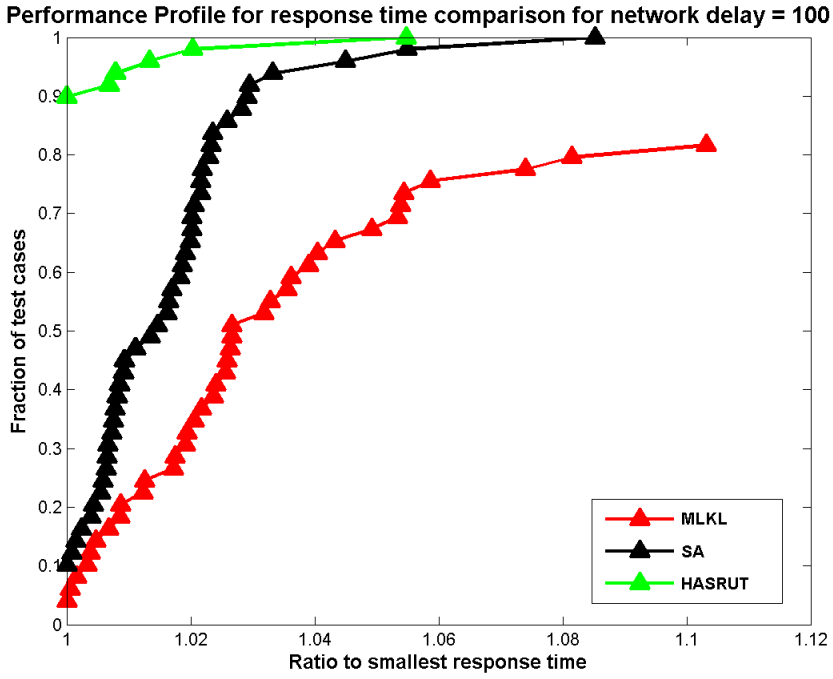

Figure 7.10: Response time performance profile for HASRUT, SA and MLKL for network delay = 100 ms

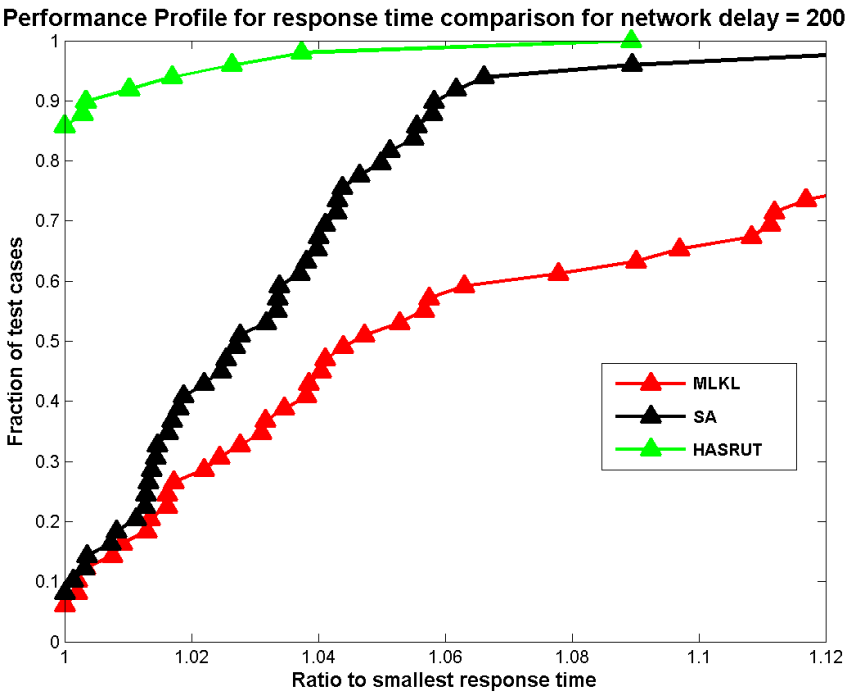

Figure 7.11: Response time performance profile for HASRUT, SA and MLKL for network delay = 200 ms

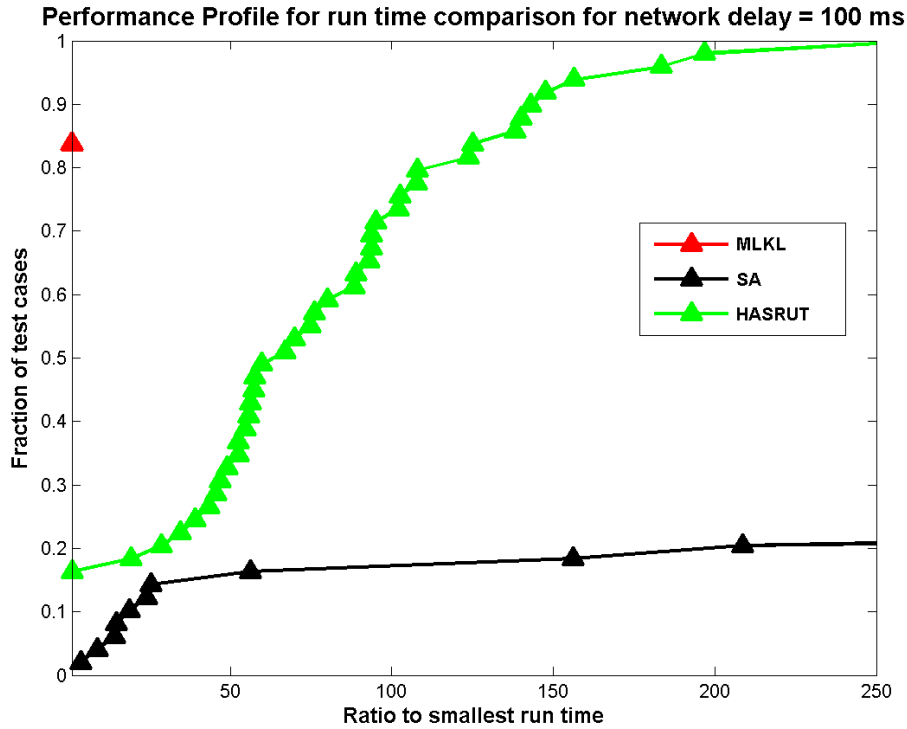

Figure 7.12: Run time performance profile for HASRUT, SA and MLKL for network delay = 100 ms

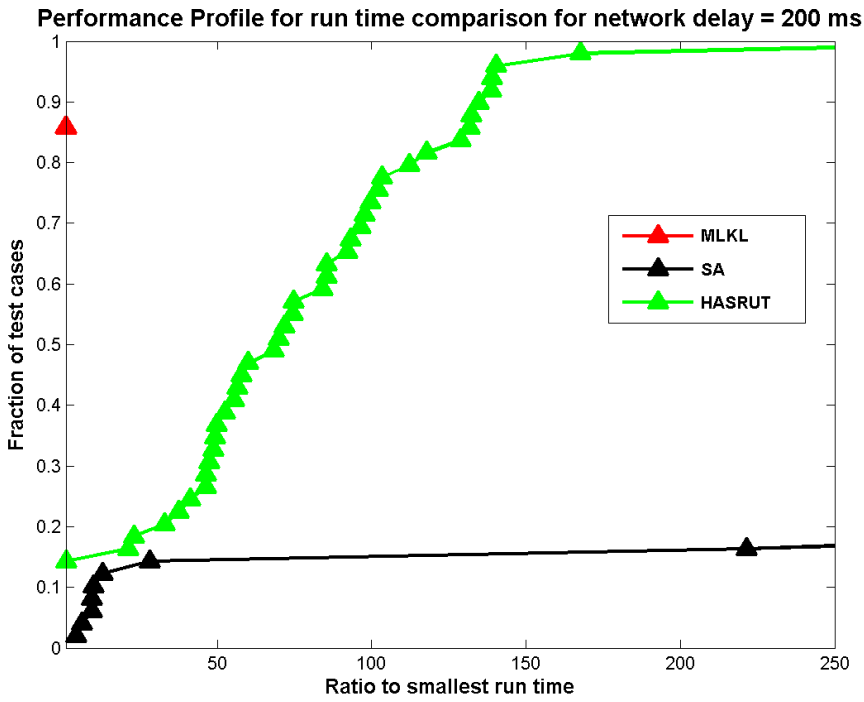

Figure 7.13: Run time performance profile for HASRUT, SA and MLKL for network delay = 200 ms

Table 7.6: Comparison between HASRUT, SA and MLKL for network delay = 200ms

		HASRUT	SA	MLKL
Response Time	**avg**	**4118.99**	4206.79	4129.19
	std dev	1726.95	1681.69	1528.39
Power	**avg**	38.54	39.23	**37.91**
	std dev	17.99	17.76	16.65
No. Hosts	**avg**	21.51	21.98	**21.07**
	std dev	10.16	10.01	9.18
Run Time	**avg**	21.45	238.58	**0.21**
	std dev	23.23	252.68	0.18
No. Failures	**avg**	**0.00**	**0.00**	3.65
	std dev	0.00	0.00	0.90

Table 7.7: Comparison between HASRUT, SA and MLKL for network delay = 200ms for comparable subset of models

		HASRUT	SA	MLKL
Response Time	**avg**	**3955.27**	4055.86	4129.19
	std dev	1562.32	1531.13	1528.39
Power	**avg**	**37.29**	37.99	37.91
	std dev	16.74	16.59	16.65
No. Hosts	**avg**	**20.69**	21.17	21.07
	std dev	9.22	9.13	9.18
Run Time	**avg**	19.59	220.46	**0.21**
	std dev	21.72	243.94	0.18
No. Failures	**avg**	**0.00**	**0.00**	3.65
	std dev	0.00	0.00	0.90

Table 7.8: Comparison between HASRUT, SA and MLKL for network delay = 100ms

		HASRUT	SA	MLKL
Response Time	**avg**	**4083.22**	4124.00	4094.63
	std dev	1713.02	1693.23	1645.51
Power	**avg**	38.54	39.27	**37.45**
	std dev	17.98	17.66	14.96
No. Hosts	**avg**	21.51	21.96	**20.85**
	std dev	10.16	9.96	8.26
Run Time	**avg**	23.11	322.86	**0.21**
	std dev	26.46	338.18	0.17
No. Failures	**avg**	**0.00**	**0.00**	3.67
	std dev	0.00	0.00	0.92

Table 7.9: Comparison between HASRUT, SA and MLKL for network delay = 100ms for comparable subset of models

		HASRUT	SA	MLKL
Response Time	**avg**	**4001.55**	4047.59	4094.63
	std dev	1645.59	1627.88	1645.51
Power	**avg**	**36.66**	37.50	37.45
	std dev	15.12	14.80	14.96
No. Hosts	**avg**	**20.37**	20.88	20.85
	std dev	8.36	8.17	8.26
Run Time	**avg**	21.15	269.03	**0.21**
	std dev	24.86	269.58	0.17
No. Failures	**avg**	**0.00**	**0.00**	3.67
	std dev	0.00	0.00	0.92

3. The HASRUT algorithm and SA never failed to produce a feasible result while MLKL algorithm has very frequent failures. These properties do not change with variation of network delay.
4. The run-time comparisons were not affected at all. MLKL was the fastest algorithm as expected, followed by HASRUT and then SA as shown in Figure 7.12 and Figure 7.13.

7.2 Experiment 2: Characteristics of HASRUT Algorithm Solutions

7.2.1 HASRUT and Edge-Core Communication

In this experiment, the adaptability of the HASRUT algorithm is tested by forcing tasks onto the edge by reducing the size of the core.The effect on power consumption and response time is examined. This also models the effect of a limited capacity available on the core cloud because it is busy with multiple applications thereby creating tighter memory and processing constraints for a given deployment. For this experiment, the T1 test model (see Figure 7.14) was used and network latency was assumed as 50 ms. A minimum of 18 combined hosts on edge and core cloud is required. Out of these 18 hosts, there are 4 hosts on the edge so this leaves a minimum of 14 processors required on the core. Results of experimentation are shown in Table 7.10

The conclusions from this experiment follow:

1. If the core has 14 or more hosts, all hosts on core are used.
2. Since there is at least 1 task pinned on the edge, a minimum of 1 host is always used on the edge. The remaining tasks are deployed on the core, which saturates

at a maximum of 17 hosts because tasks for this test case require no more than total of 18 hosts.

Table 7.10: Impact of limiting size of core cloud

Number of core hosts available	HASRUT		
	Response time	**Power**	**No. of hosts used in core/edge**
12	No Feasible Result	No Feasible Result	No Feasible Result
14	4103.75	31.52	14/4
15	4005.75	31.52	15/3
16	3978.42	31.52	16/2
17	3955.08	31.52	17/1
20	3955.08	31.52	17/1
50	3955.08	31.52	17/1
100	3955.08	31.52	17/1

3. The HASRUT algorithm tries to keep the heavily communicating tasks together as much as it can. So when the core is large, it keeps them all together in the core. But as the core shrinks, more tasks are forced to communicate across the edge-core divide leading to increase in response time.
4. The response time improves as number of hosts on core cloud increases. This is because more tasks are moved from edge to core with increase in core size as the core is able to accommodate more tasks and hence more communication is between tasks on the same cloud, thereby reducing the response time.
5. It is interesting to note that the power consumption does not change. This is because the HASRUT algorithm uses same number of total hosts in all solutions and has similar loadings on the hosts.

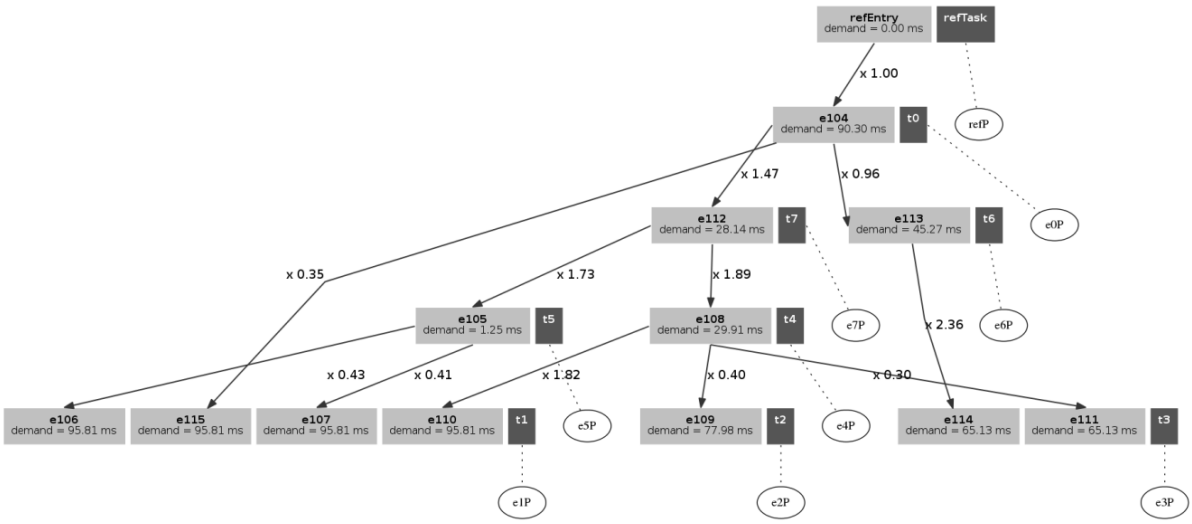

Figure 7.14: LQN model for test case T1

7.2.2 Impact of Problem Size

Problem size i.e., the total number of tasks after replication, also has an impact on the results reported by HASRUT, SA and MLKL. Based on Figure 7.15, Figure 7.16 and Figure 7.17, it can be concluded that the HASRUT algorithm finds solutions relatively quickly. More importantly, the solution time scales linearly with the problem size. The HASRUT algorithm has a far superior run time compared to SA however MLKL is very fast when compared to the HASRUT algorithm (see Figure 7.15 and Figure 7.16). This is not surprising because MLKL is used to find a good deployment for every randomly created initial partition in the HASRUT algorithm. To counter this run time disadvantage against MLKL, using the first feasible result from HASRUT provides even faster results (see Figure 7.17) while maintaining equal or better power consumption and better robustness as compared to MLKL.

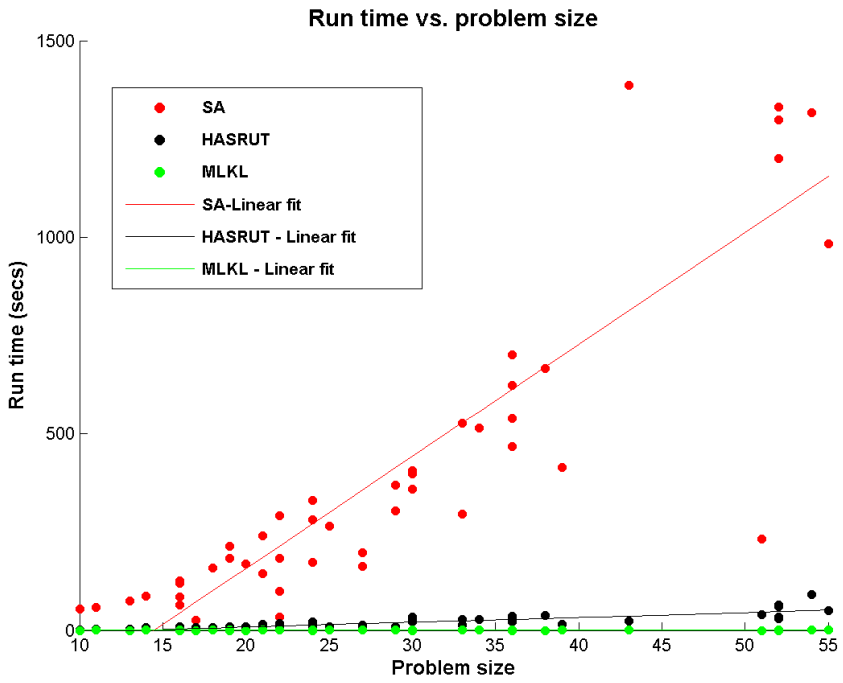

Figure 7.15: Run time vs. problem size comparison of SA, HASRUT algorithm and MLKL

7.2.3 Evaluation of Quality of Results with Time for HASRUT Algorithm

In this evaluation, variation of the quality of the solution of the HASRUT algorithm compared with solution time is presented. As the solution time progresses, the HASRUT algorithm finds deployments with lower power consumption values while meeting the response time constraint. Initially, the HASRUT algorithm finds a deployment that has a high power consumption and response time. As the solution time progresses, new deployments are found that have better power and response times.

Figure 7.18 shows a plot of how power consumption and response time varies with solution time for test case T48 (see Table A.6 in Appendix A for details). At 5 seconds, the algorithm finds a deployment that has the lowest power value so far, but a higher response time than previous deployments. For the remaining time, the HASRUT

algorithm keeps the power consumption value at this lower value while improving response time. The response time threshold value for this test case is 4841.97 msec.

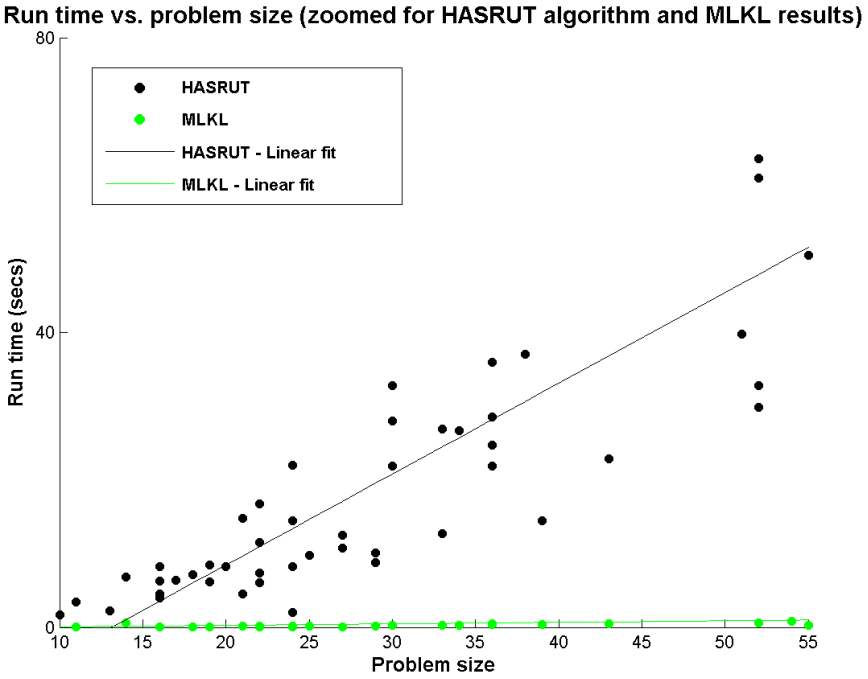

Figure 7.16: Run time vs. problem size comparison of HASRUT algorithm and MLKL

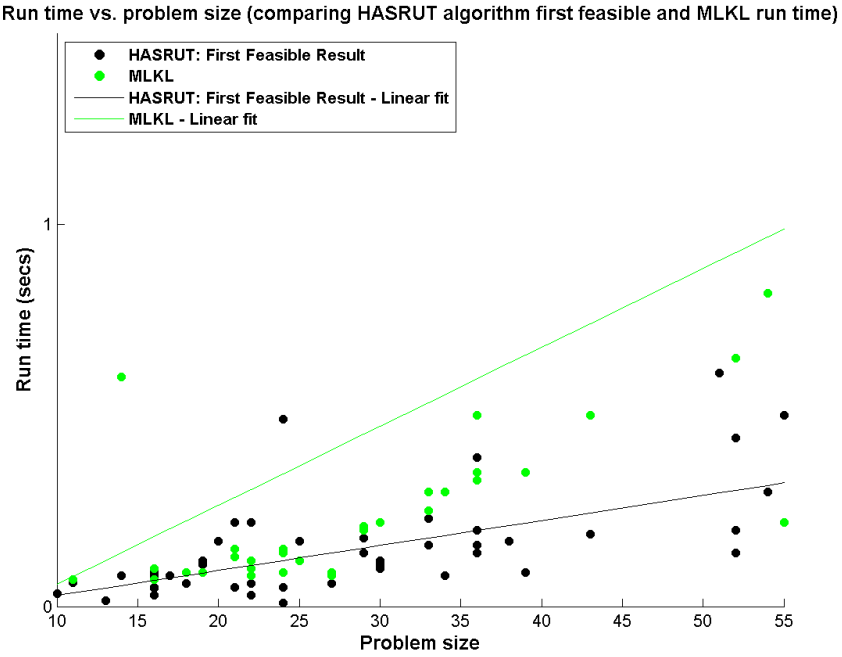

Figure 7.17: Run time vs. problem size comparison for HASRUT algorithm first feasible result and MLKL

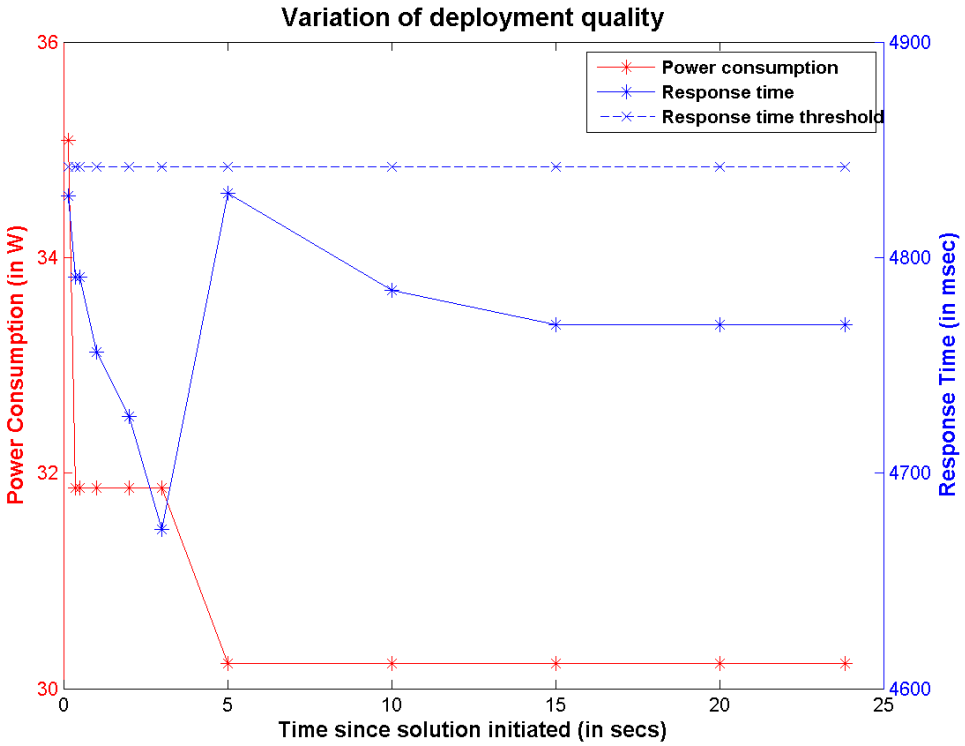

Figure 7.18: Variation of power consumption and response time for HASRUT algorithm as solution progresses

8. Summary

The main objective of this work is to develop an algorithm that is fast and robust and provides good quality allocations between an edge and a core cloud, and that heuristically minimizes power consumption subject to latency constraints, and to constraints on memory and processing capacity of hosts in the clouds.

A new approach called HASRUT has been described in this thesis for deployment of applications between an edge and core cloud considering network latency and communication delays. New test models have been devised to simulate future oriented latency sensitive applications to validate proposed algorithm results. HASRUT runs quite fast when compared to SA and never fails to find a solution, as compared to the high failure rate of the MLKL algorithm.

8.1 Conclusions

1. The HASRUT algorithm is robust, fast and returns low power consumption solutions for edge-core task allocation. It is fast enough for use in real-time control of clouds. Solution time scales linearly with application size, and useful solutions are available well before the algorithm completes.
2. Network latency has little impact on the effectiveness of the HASRUT algorithm.
3. The extended Simulated Annealing algorithm is much slower than the other two algorithms and hence unsuitable for use in real-time task allocation.
4. The extended multi-level Kernighan-Lin algorithm is not sufficiently robust for use in real-time task allocation.

5. Power consumption depends almost entirely on the number of hosts that are turned on because the running costs are much smaller.
6. The HASRUT algorithm does not force edge-core communication until it has to do so. It keeps the highly communicating tasks together as much as it can on either the edge or core cloud.
7. The solution found by the HASRUT algorithm evolves over time by finding deployments with lower power consumption values that meet the response time constraint. Response time is sacrificed at the cost of finding minimum power over solution time span.

8.2 Summary of Contributions

The main contribution of this thesis is a novel algorithm that allocates tasks to edge or core clouds in a way that consumes little power, while meeting constraints on response time, host memory and host processing capacity. The HASRUT algorithm is robust and fast compared to Simulated Annealing (SA) and MLKL. The first feasible result from the HASRUT algorithm provides an even faster solution that also provides a low power allocation. The algorithm is also easily extendable to handle additional capacity constraints (e.g. number of software licenses, etc.).

8.3 Future Research

In this thesis, various assumptions were made to simplify the problem at hand. These assumptions present various future research opportunities:

1. Constraints considered in this research are response time, memory and processing capacity. The HASRUT algorithm can be extended to take more constraints into account such as task grouping where two or more tasks always need to be grouped together no matter what host they are allocated to and task host affinity where a particular task can only be allocated on a specific host or group of hosts.
2. To simplify the problem, only one core and one edge cloud are assumed in the network. The algorithm can also be extended to use multiple core and edge clouds. This will introduce multiple competing moves from one cloud (core/edge) to another which will likely require some techniques like first fit or best fit to decide where to move a given task.
3. If the application is to be deployed over one core and many identical edges, with symmetrical usage patterns over all edges (so, the deployment is the same on each edge), then only one edge needs to be represented in the graph model, and HASRUT can be applied with suitable modifications for the core capacity requirement. Then any number of edges can be solved, even thousands of edge clouds.
4. Another research area to explore is to allow heterogeneous hosts on edge and core cloud. This will introduce some challenges during power calculation using multi-dimensional bin packing.
5. Multiple applications running on same edge and core cloud is another problem that was not considered in this research. To make things more realistic these applications can also be interacting with one another. This introduces extra dependencies between interacting tasks and will impact their deployment.

6. Future oriented applications can have an arbitrary call-graph topology which poses a more complex structure to deploy applications.
7. Execution speed is assumed same for hosts on edge and core cloud. The problem can be extended if core host execution speed be different from edge hosts.

References

[1] Y. Sun, J. White, J. Gray, A. Gokhale and D. Schmidt, "Model-Driven Automated Error Recovery in Cloud Computing," in *Model-driven Analysis and Software Development: Architectures and Functions*, J. Osis and E. Asnina, Eds., Hershey, PA, USA, IGI Global, 2011.

[2] "Cloud Computing Definition," [Online]. Available: http://www.service-architecture.com/articles/cloud-computing/cloud_computing_definition.html. [Accessed 8 August 2015].

[3] S. Chien-Chung, S. Chavalit and J. Chaiporn, "Sensor Information Networking Architecture and Applications," *IEEE Personal Communications,* pp. 52-59, 2001.

[4] S. Elaine and P. Adrian, "Designing Secure Sensor Networks," *IEEE Wireless Communications,* vol. 11, no. 6, pp. 38-43, December 2004.

[5] S. S. Dumit and D. P. Aggrawal, "Self-Organizing and Energy-Efficient Network of Sensors," in *MILCOM 2002. Proceedings*, Oct. 7-10,2002.

[6] J. Chinneck, M. Litoiu and C. Woodside, "Real-Time Multi-Cloud Management Needs Application Awareness," in *5th ACM/SPEC International Conference on Performance Engineering ICPE 2014*, Dublin, Ireland, March 22 -26,2014.

[7] A. Faisal, D. Petriu and M. Woodside, "Network Latency Impact on Performance of Software Deployed Across Multiple Clouds," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '13*, Ontario, Canada, 2013.

[8] T. Verbelen, T. Stevens, F. D. Turck and B. Dhoedt, "Graph Partitioning Algorithms for Optimizing Software Deployment in Mobile Cloud Computing," *Future Gener Comput Syst,* vol. 29, no. 2, pp. 451-9, 2013.

[9] "Private Cloud Explained," [Online]. Available: http://www.eci.com/cloudforum/private-cloud-explained.html. [Accessed 8 August 2015].

[10] "Hybrid Cloud Storage," [Online]. Available: https://ussignal.com/blog/5-use-cases-for-hybrid-cloud-storage. [Accessed 8 August 2015].

[11] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders and C. Schulz, "Recent Advances in Graph Partitioning," in *Algorithm Engineering – Selected Topics, to app., ArXiv:1311.3144*, 2014.

[12] M. Woodside, "Tutorial Introduction to Layered Modeling of Software Performance," May 2002. [Online]. Available: http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialf.pdf. [Accessed 8 August 2015].

[13] D. Johnson, A. Demers, D. Ullman, M. Garey and R. Graham, "Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM Journal of Computing,* vol. 3, pp. 299-325, 1974.

[14] E. Coffman, M.R.Garey, D.S.Johnson and R.E.Tarjan, "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms," *SIAM Journal on Computing,* vol. 9, pp. 808-826, 1980.

[15] V. Cerny., "Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm," *Journal of Optimization Theory and Applications,* vol. 45, no. 1, pp. 41-51, January 1985.

[16] S. Kirkpatrick, C. D. Gelatt and M. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, no. 4598, pp. 671-680, May 1983.

[17] **H. H. Hoos and T. Stützle, Stochastic Local Search Foundations and Applications, Morgan Kaufmann / Elsevier, 2005.**

[18] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning," *Bell System Technical Journal,* vol. 49, no. 2, p. 291–307, 1970.

[19] C. Mattheyses and R. Fiduccia, "A Linear-Time Heuristic for Improving Network Partitions," *In Papers on Twenty-five years of electronic design automation,* p. 247, 1998.

[20] M.Stoer and F.Wagner, "A simple min-cut algorithm," *Journal of the ACM (JACM),* vol. 44, no. 4, pp. 585-591, July 1997.

[21] L. Ford and D. Fulkerson, "Maximal Flow Through a Network," *Canadian Journal of Mathematics,* vol. 8, pp. 399-404, 1956.

[22] J. W. Chinneck, "Practical Optimization: A Gentle Introduction," [Online]. Available: http://www.sce.carleton.ca/faculty/chinneck/po.html.. [Accessed 8 August 2015].

[23] S. Bokhari, "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in Distributed Processor Systems," *IEEE Trans. Software Engineering,* vol. 7,

no. 6, pp. 335-341, 1981.

[24] A. Paul, S. Addya, B. Sahoo and A. Turuk, "Application of Greedy Algorithms to Virtual Machine Distribution Across Data Centers," in *India Conference (INDICON), 2014 Annual IEEE*, Pune, India, 11-13 Dec. 2014.

[25] E. Coffman, E. Garry and D. S. Johnson, "An Application of Bin-Packing to Multiprocessor Scheduling," *SIAM Journal on Computing,* vol. 7, no. 1, pp. 1-17, 1978.

[26] C. M. Woodside and G. G. Monforton, "Fast Allocation of Processes in Distributed and Parallel Systems," *IEEE Transaction. on Parallel and Distributed Systems,* vol. 4, no. 2, pp. 164-174, 1993.

[27] S. Chao, J. Chinneck and R. Goubran, "Assigning Service Requests in Voice-Over-Internet Gateway Multiprocessors," *Computers and Operations Research,* vol. 31, no. 14, pp. 2419-2437, 2004.

[28] C. Tang, M. Steinder, M. Spreitzer and G. and Pacifici, "A Scalable Application Placement Controller For Enterprise Data Centers," in *Proceedings of 16th International Conference on the World Wide Web*, Banff, Alberta, Canada, 2007.

[29] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko and A. and Tantawi, "Dynamic Placement for Clustered Web Applications," in *Proceedings of the 15th international Conference on World Wide Web*, Edingburgh, Scotland, 2006.

[30] T. Davidović, P. Hansen and N. Mladenović, "Permutation Based Genetic, Tabu and

Variable Neighborhood Search Heuristics for Multiprocessor Scheduling with Communication Delays," *Asia-Pacific Journal of Operational Research,* vol. 22, no. 3, pp. 297-326, Sept. 2005.

[31] I. Ahmad and M. K. Dhodhi, "Multiprocessor Scheduling in a Genetic Paradigm," *Parallel Computing,* vol. 22, p. 395–406, 1996.

[32] E. S. H. Hou, N. Ansari and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Trans. Parallel and Distributed Systems,* pp. 113-120, Feb 1994.

[33] [Online]. Available: http://www.ro.feri.uni-mb.si/predmeti/int_reg/Predavanja/Eng/3.Genetic%20algorithm/_18.html. [Accessed 8 August 2015].

[34] A. Thesen, "Design and Evaluation of a Tabu Search Algorithm for Multiprocessor Scheduling," *J.Heuristics,* p. 1998, 141-160.

[35] D. S. Johnson, C. R. Aragon, L. McGeoch and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning," *Operations Research,* vol. 37, no. 6, pp. 856-892, November 1989.

[36] M.-W. Park and Y.-D. Kim, "A Systematic Procedure for Setting Parameters in Simulated Annealing Algorithms," *Computers & Operations Research,* vol. 25, no. 3, pp. 207-217, March 1998.

[37] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering,* pp. 85-93, 1977.

[38] S. Bokhari, "Dual Processor Scheduling with Dynamic Reassignment," *IEEE Trans.*

Softw. Eng, vol. 5, no. 4, pp. 326-334, July,1979.

[39] V. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Transactions on Computers,* vol. 37, no. 11, pp. 1384-1397, 1988.

[40] M. Leng and S. Yu, "An Effective Multi-Level Algorithm Based on Ant Colony Optimization for Bisecting Graph," *Lecture Notes in Computer Science,* p. 4426:138, 2007.

[41] L. Sun and M. Leng, "An Effective Multi-level Algorithm Based on Simulated Annealing for Bisecting Graph," *Lecture Notes in Computer Science,* p. 4679:1, 2007.

[42] L. Sun, M. Leng and S. Yu, "A New Multi-level Algorithm Based on Particle Swarm Optimization for Bisecting Graph," *Lecture Notes in Computer,* p. 4632:69, 2007.

[43] Y. Saab, "An Effective Multilevel Algorithm for Bisecting Graphs and Hypergraphs," *IEEE Transactions on Computers,* vol. 53, no. 6, p. 641–652, 2004.

[44] C. Aykanat, B. Cambazoglu and B. Ucar, "Multi-level Direct K-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices," *Journal of Parallel and Distributed Computing,* vol. 68, no. 5, p. 609–625, Oct 2007.

[45] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing,* vol. 20, no. 1, p. 359–392, 1999.

[46] F. Pellegrini, "Scotch and LibScotch 5.1 User's Guide,LaBRI, Université

Bordeaux," August 2008. [Online]. Available: http://www.labri.fr/~pelegrin/scotch/.

[47] C. Coello, G. B. Lamont and D. Veldhuizen, Evolutionary Algorithms for Solving Multi-Objective Problems, vol. 2, Springer US, 2007.

[48] S. Porto and C. Ribeiro, "A Tabu Search Approach to Task Scheduling on Heterogeneous Processors Under Precedence Constraints," *International Journal of High-Speed Computing,* vol. 7, pp. 47-71, 1995.

[49] K. A. Dowsland, "Simulated Annealing," in *Modern Heuristic Techniques for Combinatorial Problems*, C. R. Reeves, Ed., John Wiley & Sons, 1993, pp. 20-69.

[50] M. Litoiu, J. Chinneck, M. Woodside and K. Salem, "Extended Cloud Computing," in *21st Centre for Advanced Studies Conference (CASCON)*, Toronto, 2011.

[51] L. Ramaswamy and J. Chen, "Efficient Delivery of Dynamic Content: The Cooperative EC Grid Project," in *2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, San Jose, CA, 2005.

[52] L. Ramaswamy, L. Liu and A. Iyengar, "Cache Clouds: Cooperative Caching of Dynamic Documents in Edge Networks," in *In Proc. Of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Columbus, OH, 2005.

[53] S. Narravula, H. Jin, K. Vaidyanathan and D. Panda, "Designing Efficient Cooperative Caching Schemes for Multi-Tier Data-Centers over RDMA-enabled Networks," in *Sixth IEEE International Symposium on Cluster Computing and the Grid*, 16-19 May 2006.

[54] M. Smit, M. Shtern, B. Simmons and M. Litoiu, "Partitioning Applications for Hybrid and Federated Clouds," in *22nd Centre for Advanced Studies Conference (CASCON)*, Toronto, ON, 2012.

[55] M. Björkqvist, L. Y. Chen, M. Vukolic and a. X. Zhang, "Minimizing Retrieval Latency For Content Cloud," in *INFOCOM*, Shanghai, April 2011.

[56] A. Kansal, F. Zhao, J. Liu, N. Kothari and A. Bhattacharya, "Virtual Machine Power Metering and Provisioning," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*, New York, USA, 2010.

[57] D. Niz and R. Rajkumar, "Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems," *Intl. Jnl. of Embedded Systems,* 2006.

[58] "Hill Climbing," [Online]. Available: http://wwwic.ndsu.edu/juell/vp/cs724s00/hill_climbing/index.html. [Accessed 8 August 2015].

[59] K. Schloegel, G. Karypis and V. Kumar, "Graph Partitioning for High Performance Scientific Simulations," in *CRPC Parallel Computing Handbook*, Morgan Kaufman, 2000.

[60] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning," in *In Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, New York, USA, 1995.

[61] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning," *SIAM Journal on Scientific Computing,* vol. 20, no. 1, pp. 359-392,

1999.

[62] "Jython," [Online]. Available: http://www.jython.org/. [Accessed 8 August 2015].

[63] "JGraphT," [Online]. Available: http://www.jgrapht.org. [Accessed 8 August 2015].

[64] E. D. Dolan and J. J. More, "Benchmarking Optimization Software with Performance Profiles," *Math. Program.,* vol. 91, pp. 201-213, 2002.

[65] "Python," [Online]. Available: https://www.python.org/. [Accessed 8 August 2015].

[66] **K. Schloegel, G. Karypis and V. Kumar, "Graph partitioning for high performance scientific simulation," in *CRPC Parallel Computing Handbook*, Morgan Kaufmann, 2000.**

[67] I. Ahmad and Y.-K. Kwok, "Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using A Parallel Genetic Algorithm," *J. Parallel and Distrib. Computing,* vol. 47, pp. 58-77, 1997.

A. Appendix

A.1 Test Cases

Test cases used for evaluating HASRUT and the competing algorithms are shown in Table A.1.

Table A.1: Summary of Test Cases

Test case name	Test case type	No. of tasks	No. of tasks replicas	No. of arcs after replication	Minimum theoretical response time (ms)	Maximum theoretical response time (ms)	Minimum theoretical number of hosts used (edge+core)	Minimum theoretical power consumption (C_f = 0.3; C_{fh} = 1.62; in W)
T1	1	12	24	25	3931.75	4537.75	from PR 7.86 from MR 14.39	28.617(MR)
T2	4	13	22	37	3536.7	4399.2	from PR 7.07 from MR 9.48	19.044(MR)
T3	5	15	27	50	3753.1	4767.6	from PR 15.01 from MR 10.48	30.42(PR)
T4	5	17	22	28	2569.55	3670.5	from PR 10.28 from MR 7.26	20.904 (PR)
T5	5	15	27	54	4263.7	5354.7	from PR 8.53 from MR 14.39	28.617(MR)
T6	8	41	55	101	6907.45	9404.7	from PR 13.82 from MR 22.99	44.16(MR)
T7	8	39	51	59	4794.4	7289.35	from PR 6.39 from MR 26.12	43.78 (MR)
T8	7	29	52	81	7181.3	9375.65	from PR 14.36 from MR 33.19	55.14(MR)
T9	1	9	11	14	1481.75	2010.7	from PR 3.7 from MR 5.6	11.4(MR)
T10	5	16	29	50	3925.0	4966.7	from PR	38.28(PR)

							19.6 from MR 14.7	
T11	3	12	21	28	2955.15	3714.2	from PR 14.78 from MR 8.13	28.734(PR)
T12	1	6	10	13	1731.25	2080.55	from PR 3.46 from MR 4.42	9.43(MR)
T13	7	23	39	52	5285.85	6701.5	from PR 26.43 from MR 21.99	51.67(PR)
T14	2	11	16	19	2022.7	2719.5	from PR 4.04 from MR 5.33	11.319(MR)
T15	2	10	22	24	3564.9	4243.25	from PR 17.82 from MR 10.19	34.506(PR)
T16	1	8	16	27	2204.25	2736.55	from PR 5.88 from MR 9.92	19.176(MR)
T17	1	6	13	21	2313.5	2785.7	from PR 5.78 from MR 8.16	17.028(MR)
T18	2	9	16	16	2598.8	3275.05	from PR 12.99 from MR 10.11	24.957(PR)
T19	7	26	43	70	5610.2	7131.85	from PR 28.05 from MR 24.75	55.395(PR)
T20	3	22	36	58	5936.0	6787.15	from PR 18.26 from MR 15.35	36.258(PR)
T21	4	20	34	51	5424.5	6388.7	from PR 18.08 from MR 10.84	36.204(PR)
T22	6	17	30	49	4199.8	5518.8	from PR 20.99 from MR 12.65	40.317(PR)
T23	6	19	33	50	5159.35	6345.1	from PR 10.86 from MR	26.823(MR)

							13.81	
T24	6	23	30	57	3428.0	4769.0	from PR 17.14 from MR 12.50	34.302(PR)
T25	3	15	22	29	3095.7	3897.95	from PR 9.52 from MR 8.38	19.056(PR)
T26	4	18	33	54	5528.4	6513.35	from PR 27.64 from MR 17.75	53.65(PR)
T27	6	16	24	34	3086.95	4291.4	from PR 15.44 from MR 14.68	30.55(PR)
T28	8	31	52	75	5718.4	8265.3	from PR 28.59 from MR 29.54	57.46(MR)
T29	3	15	24	33	2961.35	3761.6	from PR 11.84 from MR 16.5	32.49(MR)
T30	1	9	21	40	3231.1	3699.85	from PR 9.94 from MR 10.87	21.081(MR)
T31	1	11	14	21	2221.4	2803.65	from PR 4.44 from MR 8.98	17.27(MR)
T32	7	24	52	84	8301.49	9656.75	from PR 16.60 from MR 28.35	55.485(MR)
T33	2	11	16	28	2238.15	2844.45	from PR 7.46 from MR 10.7	21.03(MR)
T34	6	18	30	47	4362.85	5646.35	from PR 9.7 from MR 11.84	22.992(MR)
T35	5	17	36	54	5397.45	6552.3	from PR 19.63 from MR 15.55	38.289(PR)
T36	2	11	17	23	2424.35	3093.15	from PR 12.12 from MR 9.54	24.696(PR)
T37	2	10	16	23	2303.95	2916.2	from PR	22.896(PR)

							11.52 from MR 8.14	
T38	4	17	38	67	6152.35	7060.75	from PR 15.38 from MR 30.21	59.283(MR)
T39	1	10	19	34	2478.4	2907.0	from PR 4.96 from MR 11.61	22.923(MR)
T40	6	14	25	39	3753.05	4963.15	from PR 7.51 from MR 11.99	23.037(MR)
T41	3	13	18	29	2101.2	2950.35	from PR 4.20 from MR 12.13	24.699(MR)
T42	7	33	52	65	6888.3	8464.6	from PR 13.78 from MR 35.63	69.009(MR)
T43	5	14	29	51	4440.8	5458.7	from PR 8.88 from MR 9.20	18.96(MR)
T44	2	12	19	29	2699.05	3401.05	from PR 5.4 from MR 11.64	22.932(MR)
T45	6	21	36	67	5021.0	6370.1	from PR 10.04 from MR 18.6	36.36(MR)
T46	3	15	24	39	3255.65	4016.25	from PR 6.51 from MR 14.78	28.734(MR)
T47	3	12	20	23	3231.35	3976.2	from PR 6.46 from MR 13.86	26.838(MR)
T48	7	20	36	48	4492.9	5947.35	from PR 8.99 from MR 14.77	28.731(MR)
T49	7	34	54	104	8099.2	10100.1	from PR 16.2 from MR 27.78	53.694(MR)

A.2 Experimental Result on Test cases

Experimental results on test cases defined in Table A.1 are shown in Table A.2 for network delay of 50 msec. Bold numbers indicate best results for a given metric across all algorithms for a given test case. E.g. if power of HASRUT is 31.52 is in bold that means that it has lowest reported power relative to SA and MLKL. Table A.3 shows results for network delay of 200 msec and Table A.4 shows results for network delay of 100 msec. Finally, first feasible result from HASRUT is compared against SA and MLKL for network delay of 50 msec in Table A.5.

Table A.2: Experimental results for network delay = 50 ms (negative value represents infeasible deployment)

Test model #	Simulated Annealing (SA)					HASRUT					MLKL				
	Response Time	Power	$\#_{total}$ of hosts	Run time	# of failures	Response Time	Power	$\#_{total}$ of hosts	Run time	# of failures	Response Time	Power	$\#_{total}$ of hosts	Run time	# of fail ures
T1	3970.42	34.52	19.2	19.22	***0/5***	***3956.08***	***31.52***	18	2	***0/5***	4053.25	***31.52***	18	***0.09***	4/5
T2	3597.2	***21.56***	12	183.33	***0/5***	***3551.7***	***21.56***	12	11.5	***0/5***	3599.2	***21.56***	12	***0.08***	3/5
T3	3798.6	41.24	22.5	197.39	***0/5***	***3768.1***	***38.52***	21	12.5	***0/5***	3900.1	***38.52***	21	***0.09***	3/5
T4	2599.5	25.76	14	32.79	***0/5***	***2592.05***	***24.14***	13	6.04	***0/5***	2768.55	***24.14***	13	***0.08***	4/5
T5	4319.7	34.96	20	162.78	***0/5***	***4276.7***	***33.34***	19	10.73	***0/5***	4338.7	***33.34***	19	***0.08***	4/5
T6	6944.25	***49.5***	28	982.77	***0/5***	6937.45	***49.5***	28	50.46	***0/5***	***6912.85***	51.12	29	***0.22***	4/5
T7	5045.3	***68.34***	41	232.41	***0/5***	***4801.9***	***68.34***	41	39.81	***0/5***	-1	-1	-1	-1	5/5
T8	7221.6	***70.73***	41	1201.1	***0/5***	7270.97	***70.73***	41	32.83	***0/5***	-1	-1	-1	-1	5/5
T9	1553.05	14.07	8	59.08	***0/5***	***1496.75***	***12.45***	***7***	3.4	***0/5***	1651.65	***12.45***	7	***0.07***	3/5
T10	3977.85	***51.25***	28	304.35	***0/5***	***3938.05***	***51.25***	***28***	10.14	***0/5***	4000.7	***51.25***	28	***0.21***	4/5
T11	2985.15	35.21	19	145	***0/5***	***2960.15***	***33.59***	18	4.5	***0/5***	2970.15	***33.59***	18	***0.13***	2/5
T12	1754.3	***10.76***	6	54.5	***0/5***	***1746.25***	***10.76***	6	1.7	***0/5***	-1	-1	-1	-1	5/5
T13	5322.5	63.01	34	414.86	***0/5***	***5292***	***61.39***	33	14.5	***0/5***	5315.85	64.62	35	***0.35***	4/5
T14	2055.9	***12.55***	7	119.79	***0/5***	***2039.3***	***12.55***	7	6.29	***0/5***	-1	-1	-1	-1	5/5
T15	3603.13	***40.98***	22	98.68	***0/5***	***3569.9***	***40.98***	22	7.4	***0/5***	3676.5	***40.98***	22	***0.12***	4/5
T16	2226.75	24.44	14	126.27	***0/5***	***2211.75***	***22.82***	13	4.2	***0/5***	2234.25	24.44	14	***0.1***	4/5
T17	2354.85	***17.94***	10	75.7	***0/5***	***2327.3***	***17.94***	10	2.2	***0/5***	-1	-1	-1	-1	5/5
T18	2628.8	***29.82***	16	63.79	***0/5***	***2603.8***	***29.82***	16	4.5	***0/5***	2702.4	***29.82***	16	***0.1***	3/5
T19	5620.2	***74.84***	41	1386.2	***0/5***	***5615.2***	***74.84***	41	22.9	***0/5***	5682	***74.84***	41	***0.5***	4/5
T20	5977.75	52.14	29	700.52	***0/5***	***5941.42***	***50.51***	28	28.5	***0/5***	-1	-1	-1	-1	5/5
T21	5469.95	44.3	24	515	***0/5***	***5435.1***	***42.6***	23	26.7	***0/5***	5623.8	44.3	24	***0.3***	4/5
T22	4224.35	***53.28***	29	358.82	***0/5***	***4209.35***	***53.28***	29	21.88	***0/5***	-1	-1	-1	-1	5/5
T23	5201.9	***34.04***	19	527.92	***0/5***	***5176.83***	***34.04***	19	12.75	***0/5***	5237.55	***34.04***	19	***0.25***	3/5
T24	3554	44.02	24	405.86	***0/5***	***3443***	***42.4***	23	28	***0/5***	-1	-1	-1	-1	5/5

T25	3150.65	23.92	13	290.6	***0/5***	***3110.22***	***22.29***	12	16.7	***0/5***	3301.2	22.29	12	***0.1***	2/5
T26	5582.45	61.75	33	295.54	***0/5***	***5535.9***	***60.13***	32	26.86	***0/5***	5680.85	61.75	33	***0.3***	3/5
T27	3161.48	***37.03***	20	172.5	***0/5***	***3091.95***	***37.03***	20	8.21	***0/5***	3339.48	***37.03***	20	***0.15***	3/5
T28	5801.55	***81.48***	45	1568.1	***0/5***	***5737.03***	***81.48***	45	61	***0/5***	-1	-1	-1	-1	5/5
T29	3004.65	37.57	21	331.27	***0/5***	***2971.78***	***35.95***	20	14.5	***0/5***	3004.65	37.57	21	***0.14***	4/5
T30	3266.66	27.28	15	239.84	***0/5***	***3236.1***	***25.66***	14	14.8	***0/5***	3278.05	27.28	15	***0.15***	4/5
T31	2265.45	***20.77***	12	86.5	***0/5***	***2240.8***	***20.77***	12	6.8	***0/5***	2246.05	***20.77***	12	0.6	4/5
T32	***8321.5***	61.68	35	1298	***0/5***	8435.2	***60.06***	34	29.8	***0/5***	-1	-1	-1	-1	5/5
T33	2283.15	***28.16***	16	119.75	***0/5***	***2245.65***	***28.16***	16	8.25	***0/5***	2329.23	***28.16***	16	***0.07***	4/5
T34	4401.95	27.21	15	397.2	***0/5***	***4382.55***	***25.59***	14	32.85	***0/5***	4452.5	27.21	15	***0.22***	4/5
T35	5427.45	***57.73***	32	466.68	***0/5***	***5409.95***	***57.73***	32	24.7	***0/5***	5514.35	***57.73***	32	***0.5***	3/5
T36	2469.35	***27.94***	15	24.69	***0/5***	***2431.85***	***27.94***	15	6.4	***0/5***	-1	-1	-1	-1	5/5
T37	2333.95	***29.37***	16	84.91	***0/5***	***2308.95***	***29.37***	15	3.93	***0/5***	2371.9	***29.37***	16	***0.07***	4/5
T38	6191.42	***59.69***	34	666.63	***0/5***	***6171.5***	***59.69***	34	37	***0/5***	-1	-1	-1	-1	5/5
T39	2526.7	***25.78***	15	213.3	***0/5***	***2489.12***	***25.78***	15	8.45	***0/5***	-1	-1	-1	-1	5/5
T40	3775.55	28.17	16	264.2	***0/5***	***3760.55***	***26.55***	15	9.73	***0/5***	3828.05	***26.55***	15	***0.12***	3/5
T41	2146.2	27.18	16	158.8	***0/5***	***2116.2***	***25.56***	15	7.19	***0/5***	2196.7	27.18	16	***0.09***	4/5
T42	6913.3	***70.55***	41	1331.2	***0/5***	***6910.8***	***70.55***	41	63.59	***0/5***	6910.9	72.17	42	***0.65***	4/5
T43	***4455.8***	26.96	15	370.27	***0/5***	4560.77	***23.72***	13	8.8	***0/5***	4547.35	28.58	16	***0.20***	2/5
T44	2722.65	***25.91***	15	183.39	***0/5***	***2709.93***	***25.91***	15	6.17	***0/5***	2778.9	***25.91***	15	***0.09***	1/5
T45	5049	***38.65***	22	623.7	***0/5***	***5036***	***38.65***	22	36	***0/5***	5040.65	***38.65***	22	***0.35***	4/5
T46	3296.9	***31.11***	18	281.15	***0/5***	***3263.15***	***31.11***	18	21.99	***0/5***	-1	-1	-1	-1	5/5
T47	3286.15	31.09	18	168.21	***0/5***	***3238.85***	***29.47***	17	8.24	***0/5***	-1	-1	-1	-1	5/5
T48	***4549.85***	33.47	19	539.5	***0/5***	4561.9	***30.23***	17	21.87	***0/5***	4560.6	33.47	19	***0.33***	4/5
T49	***8130.1***	61.55	35	1317.1	***0/5***	8145.1	***59.93***	34	91.09	***0/5***	8171	61.55	35	***0.82***	3/5

Table A.3: Experimental results for network delay = 200 ms (negative value represents infeasible deployment)

Test Models	Simulated Annealing (SA)					HASRUT					MLKL				
	Response Time	Power	$\#_{total}$ of hosts	Run time	# of failures	Response Time	Power	$\#_{total}$ of hosts	Run time	# of failures	Response Time	Power	$\#_{total}$ of hosts	Run time	# of failures
T1	4029.08	***31.52***	18	235.44	***0/5***	***3955.08***	***31.52***	18	1.9	***0/5***	4417.75	***31.52***	18	***0.09***	3/5
T2	3718.7	***21.56***	12	82.58	***0/5***	***3596.7***	***21.56***	12	15.08	***0/5***	3786.7	***21.56***	12	***0.09***	4/5
T3	3991.1	***38.52***	21	121.93	***0/5***	***3783.1***	***38.52***	21	12.13	***0/5***	4341.1	***38.52***	21	***0.09***	3/5
T4	2689.55	25.76	14	26.16	***0/5***	***2659.55***	***24.14***	13	***6.2***	***0/5***	-1	-1	-1	-1	5/5
T5	4487.7	34.96	20	72.53	***0/5***	***4315.7***	***33.34***	19	14.56	***0/5***	4563.7	***33.34***	19	***0.11***	4/5
T6	6967.45	51.13	29	806.27	***0/5***	7027.45	***49.5***	28	54	***0/5***	***6910.62***	52.75	30	***0.16***	4/5
T7	5154.2	***68.34***	41	558.79	***0/5***	***4854.4***	***68.34***	41	***90***	***0/5***	-1	-1	-1	-1	5/5
T8	7254.6	***70.73***	41	455.16	***0/5***	***7244.7***	***70.73***	41	***48.75***	***0/5***	-1	-1	-1	-1	5/5
T9	1766.95	14.07	8	27.8	***0/5***	***1541.75***	***12.45***	7	3.5	***0/5***	1738.5	***12.45***	7	***0.05***	4/5
T10	4136.4	***51.25***	28	146.37	***0/5***	***3977.2***	***51.25***	28	10.72	***0/5***	3985.0	***51.25***	28	***0.22***	2/5
T11	3075.15	35.21	19	79.7	***0/5***	***2975.15***	***33.59***	18	4.5	***0/5***	3095.68	***33.59***	18	***0.12***	4/5
T12	1823.58	***10.76***	6	30.7	***0/5***	***1791.25***	***10.76***	6	1.65	***0/5***	2017.62	***10.76***	6	***0.05***	4/5
T13	5397.05	***61.39***	33	246.2	***0/5***	***5310.45***	***61.39***	33	23.94	***0/5***	5543.45	***61.39***	33	***0.35***	4/5
T14	2155.5	***12.55***	7	60.95	***0/5***	***2089.1***	***12.55***	7	***6.4***	***0/5***	-1	-1	-1	-1	5/5
T15	3717.83	***40.98***	22	81	***0/5***	***3584.9***	***40.98***	22	7.36	***0/5***	3788.1	***40.98***	22	***0.13***	2/5
T16	2294.25	24.44	14	62.64	***0/5***	***2234.25***	***22.82***	13	4.14	***0/5***	2264.25	24.44	14	***0.10***	4/5
T17	2478.9	***17.94***	10	33.75	***0/5***	***2368.63***	***17.94***	10	2.3	***0/5***	2450.7	19.55	11	***0.10***	4/5
T18	2718.8	***29.82***	16	42.96	***0/5***	***2618.8***	***29.82***	16	4.44	***0/5***	3193.8	***29.82***	16	***0.08***	3/5
T19	5650.2	***74.84***	41	292.75	***0/5***	***5630.2***	***74.84***	41	21.33	***0/5***	5753.8	***74.84***	41	***0.46***	3/5
T20	6043.0	***50.52***	28	415.58	***0/5***	***5957.67***	***50.52***	28	27.78	***0/5***	6060.6	***50.52***	28	***0.33***	2/5
T21	5484.5	44.3	24	289.12	***0/5***	***5466.9***	***42.68***	23	27.00	***0/5***	5636.3	***42.68***	23	***0.28***	4/5
T22	4449.4	***53.28***	29	186.54	***0/5***	***4238.0***	***53.28***	29	21.49	***0/5***	4567.8	***53.28***	29	***0.22***	3/5
T23	5359.48	***34.04***	19	255.94	***0/5***	***5270.28***	***34.04***	19	12.37	***0/5***	5355.75	***34.04***	19	***0.26***	4/5
T24	3584.4	***42.40***	23	174.67	***0/5***	***3488.0***	***42.40***	23	28.31	***0/5***	3584.4	***42.40***	23	***0.24***	4/5
T25	3315.5	23.92	13	147.39	***0/5***	***3153.77***	***22.29***	12	17.15	***0/5***	3506.7	23.92	13	***0.13***	3/5
T26	5744.6	61.75	33	62.04	***0/5***	***5558.4***	***60.13***	32	25.78	***0/5***	5648.4	61.75	33	***0.28***	4/5

T27	3385.08	***37.03***	20	100.55	***0/5***	***3106.95***	***37.03***	20	8.42	***0/5***	3408.15	***37.03***	20	***0.14***	2/5
T28	5978.0	***81.48***	45	785.56	***0/5***	***5829.2***	***81.48***	45	62.4	***0/5***	6105.0	83.10	46	***0.73***	4/5
T29	3134.55	37.57	21	177.56	***0/5***	***3003.08***	***35.95***	20	14.47	***0/5***	3337.35	37.57	21	***0.14***	3/5
T30	3373.37	27.28	15	137.58	***0/5***	***3292.23***	***25.66***	14	13.27	***0/5***	3418.9	27.28	15	***0.13***	3/5
T31	2397.6	***20.77***	12	40.33	***0/5***	***2299.0***	***20.77***	12	6.74	***0/5***	2320.0	***20.77***	12	***0.06***	4/5
T32	***8381.5***	***61.68***	35	830.55	***0/5***	8694.67	***61.68***	35	***29.46***	***0/5***	-1	-1	-1	-1	5/5
T33	2418.15	***28.16***	16	53.67	***0/5***	***2268.15***	***28.16***	16	7.97	***0/5***	2622.48	***28.16***	16	***0.08***	4/5
T34	4500.45	27.21	15	211.93	***0/5***	***4441.65***	***25.59***	14	33.29	***0/5***	4721.45	27.21	15	***0.24***	4/5
T35	5517.45	***57.73***	32	1000	***0/5***	***5447.45***	***57.73***	32	23.08	***0/5***	5488.72	***57.73***	32	***0.27***	4/5
T36	2764.75	***27.94***	15	46.52	***0/5***	***2454.35***	***27.94***	15	6.3	***0/5***	2514.35	***27.94***	15	***0.12***	4/5
T37	2423.95	***29.38***	16	42.77	***0/5***	***2323.95***	***29.38***	16	3.99	***0/5***	2575.75	***29.38***	16	***0.08***	4/5
T38	6308.15	***59.69***	34	380.77	***0/5***	***6222.35***	***59.69***	34	***38.4***	***0/5***	-1	-1	-1	-1	5/5
T39	2683.5	25.79	15	118.54	***0/5***	***2535.8***	***24.17***	14	5.8	***0/5***	2544.1	25.78	15	***0.1***	3/5
T40	***3843.05***	28.17	16	118.94	***0/5***	3944.65	***24.93***	14	***9.24***	***0/5***	-1	-1	-1	-1	5/5
T41	2281.2	***25.56***	15	83.13	***0/5***	***2161.2***	***25.56***	15	6.73	***0/5***	2581.2	27.18	16	***0.09***	4/5
T42	6978.7	72.17	42	836.31	***0/5***	***6928.3***	***70.55***	41	62.46	***0/5***	6942.11	***70.55***	41	***0.67***	4/5
T43	***4500.8***	26.96	15	169.12	***0/5***	4902.69	***23.72***	13	9.3	***0/5***	4672.4	25.34	14	***0.2***	3/5
T44	2793.35	***25.92***	15	96.71	***0/5***	2742.58	***25.92***	15	6.42	***0/5***	***2733.35***	27.54	16	***0.13***	2/5
T45	***5081.0***	***38.65***	22	300.1	***0/5***	5095.93	***38.65***	22	38.67	***0/5***	5147.1	***38.65***	22	***0.3***	4/5
T46	3420.65	***31.11***	18	143.13	***0/5***	***3285.65***	***31.11***	18	22.25	***0/5***	3420.65	***31.11***	18	***0.16***	4/5
T47	3450.55	***29.48***	18	85	***0/5***	***3261.35***	***29.48***	17	7.89	***0/5***	3555.15	***29.48***	17	***0.11***	2/5
T48	4780.7	35.1	21	243.97	***0/5***	4769.0	***30.23***	17	23.84	***0/5***	***4720.7***	33.48	19	***0.32***	4/5
T49	8248.4	***61.56***	35	692.78	***0/5***	***8129.2***	***61.56***	35	118	***0/5***	8386.4	***61.56***	35	***0.84***	2/5

Table A.4: Experimental results for network delay = 100 ms (negative value represents infeasible deployment)

Test Model	Simulated Annealing (SA)					HASRUT					MLKL				
	Response Time	Power	$\#_{total}$ of hosts	Run time	# of failures	Response Time	Power	$\#_{total}$ of hosts	Run time	# of failures	Response Time	Power	$\#_{total}$ of hosts	Run time	# of failures
T1	4008.42	31.52	18	310.43	***0/5***	***3943.42***	***31.52***	18	1.75	***0/5***	4174.75	***31.52***	18	***0.09***	3/5
T2	3627.11	21.56	12	130.43	***0/5***	***3566.7***	***21.56***	12	14.68	***0/5***	3661.7	***21.56***	12	***0.08***	3/5
T3	3844.1	38.52	21	176.52	***0/5***	***3768.1***	***38.52***	21	12.96	***0/5***	4047.1	***38.52***	21	***0.12***	4/5
T4	2629.55	25.76	14	24.51	***0/5***	***2614.05***	***24.14***	13	6.25	***0/5***	-1	-1	-1	-1	5/5
T5	4375.2	34.96	20	115.25	***0/5***	***4289.7***	***33.34***	19	15	***0/5***	4500.7	34.96	20	***0.12***	4/5
T6	6972.15	51.13	29	1194.8	***0/5***	6967.45	***49.5***	28	52.9	***0/5***	***6912.5***	52.75	30	***0.2***	4/5
T7	4854.4	***68.34***	41	879.75	***0/5***	***4819.4***	***68.34***	41	100.3	***0/5***	-1	-1	-1	-1	5/5
T8	***7265.7***	***70.73***	41	751.96	***0/5***	7413.17	***70.73***	41	29.6	***0/5***	-1	-1	-1	-1	5/5
T9	1624.35	14.07	8	42.22	***0/5***	***1496.75***	***12.45***	7	3.8	***0/5***	1545.95	***12.45***	7	***0.05***	2/5
T10	4030.7	***51.25***	28	231.5	***0/5***	***3951.1***	***51.25***	28	10	***0/5***	4000.7	***51.25***	28	***0.22***	2/5
T11	3015.15	35.2	19	109.9	***0/5***	***2961.15***	***33.59***	18	4.5	***0/5***	3025.42	35.2	19	***0.13***	3/5
T12	1777.42	***10.76***	6	43.3	***0/5***	***1761.25***	***10.76***	6	1.8	***0/5***	-1	-1	-1	-1	5/5
T13	5341.45	***61.39***	33	342.4	***0/5***	***5298.15***	***61.39***	33	23.8	***0/5***	5315.85	64.63	35	***0.34***	1/5
T14	2089.1	***12.55***	7	90.6	***0/5***	***2055.9***	***12.55***	7	6.1	***0/5***	-1	-1	-1	-1	5/5
T15	3641.37	***40.99***	22	90.14	***0/5***	***3574.9***	***40.99***	22	7.1	***0/5***	3660.9	***40.99***	22	***0.13***	4/5
T16	2249.25	24.4	14	96.3	***0/5***	***2219.25***	***22.8***	13	4.2	***0/5***	2234.25	24.4	14	***0.08***	3/5
T17	2396.2	***17.94***	10	55.6	***0/5***	***2341.07***	***17.94***	10	2.3	***0/5***	2382.1	19.6	11	***0.08***	4/5
T18	2658.8	***29.82***	16	45.71	***0/5***	***2608.8***	***29.82***	16	4.5	***0/5***	2618.8	***29.82***	16	***0.08***	4/5
T19	5630.2	***74.84***	41	418.27	***0/5***	***5625.2***	***74.84***	41	22.1	***0/5***	5773.3	***74.84***	41	***0.45***	4/5
T20	6013.3	***50.52***	28	569.24	***0/5***	***5946.83***	***50.52***	28	27.54	***0/5***	5998.3	***50.52***	28	***0.31***	2/5
T21	5469.5	44.3	24	427.8	***0/5***	***5445.7***	***42.68***	23	28.6	***0/5***	5585.2	44.3	24	***0.28***	2/5
T22	4315.4	***53.28***	29	277.87	***0/5***	***4218.9***	***53.28***	29	21.55	***0/5***	4383.8	***53.28***	29	***0.23***	3/5
T23	5237.55	***34.04***	19	372.95	***0/5***	***5209.28***	***34.04***	19	12.7	***0/5***	5397.65	***34.04***	19	***0.27***	4/5
T24	3538.7	44.02	24	266.04	***0/5***	***3458.0***	***42.4***	23	28	***0/5***	3540.0	***42.4***	23	***0.2***	3/5
T25	3205.6	23.92	13	229.3	***0/5***	***3124.7***	***22.29***	12	16.1	***0/5***	3205.6	23.92	13	***0.13***	3/5
T26	5663.3	61.75	33	62.63	***0/5***	***5543.4***	***60.13***	32	26.5	***0/5***	5649.5	***60.13***	32	***0.3***	3/5

T27	3236.02	***37.03***	20	129.9	***0/5***	***3096.95***	***37.03***	20	8.7	***0/5***	3349.48	***37.03***	20	***0.13***	3/5
T28	5820.9	***81.48***	45	1174.5	***0/5***	***5737.03***	***81.48***	45	62.15	***0/5***	-1	-1	-1	-1	5/5
T29	3004.65	37.57	21	319.48	***0/5***	***2982.22***	***35.95***	20	15.4	***0/5***	3111.15	37.57	21	***0.15***	4/5
T30	3266.67	27.82	15	229.69	***0/5***	***3261.67***	***25.66***	14	14	***0/5***	3325.0	27.28	15	***0.13***	4/5
T31	2309.5	***20.77***	12	61.8	***0/5***	***2260.2***	***20.77***	12	6.5	***0/5***	2270.7	***20.77***	12	***0.07***	4/5
T32	8341.5	***61.68***	35	1047.3	***0/5***	***8321.5***	***61.68***	35	94	***0/5***	***8321.5***	***61.68***	35	***0.68***	4/5
T33	2328.15	***28.16***	16	85	***0/5***	***2253.15***	***28.16***	16	7.5	***0/5***	2313.15	***28.16***	16	***0.08***	4/5
T34	4431.65	27.21	15	279.5	***0/5***	***4402.25***	***25.59***	14	32.46	***0/5***	4542.15	27.21	15	***0.22***	3/5
T35	5457.45	***57.73***	32	1317.2	***0/5***	***5422.45***	***57.73***	32	23.4	***0/5***	-1	-1	-1	-1	5/5
T36	2573.15	***27.94***	15	47.72	***0/5***	***2439.35***	***27.94***	15	6.3	***0/5***	2469.35	***27.94***	15	***0.11***	4/5
T37	2363.95	***29.4***	16	73.3	***0/5***	***2313.95***	***29.4***	16	3.9	***0/5***	2439.85	***29.4***	16	***0.09***	4/5
T38	6230.25	***59.69***	34	508.2	***0/5***	***6190.65***	***59.69***	34	35.9	***0/5***	-1	-1	-1	-1	5/5
T39	2580.95	25.78	15	164.12	***0/5***	***2507.1***	***24.17***	14	5.79	***0/5***	2511.25	25.78	15	***0.11***	4/5
T40	***3798.05***	28.17	16	177	***0/5***	3848.85	***24.93***	14	9.58	***0/5***	3933.05	26.55	15	***0.16***	4/5
T41	2191.2	27.18	16	114.6	***0/5***	***2131.2***	***25.56***	15	6.73	***0/5***	2351.0	27.18	16	***0.09***	3/5
T42	6938.3	***70.55***	41	106.13	***0/5***	***6938.3***	***70.55***	41	64.76	***0/5***	7057.5	***70.55***	41	***0.68***	3/5
T43	***4470.8***	26.96	15	276.6	***0/5***	4715.8	***23.72***	13	9	***0/5***	4711.75	28.58	16	***0.23***	4/5
T44	2746.2	***25.9***	15	135.65	***0/5***	***2720.82***	***25.9***	15	6.13	***0/5***	2866.02	***25.9***	15	***0.11***	4/5
T45	5080.73	***38.65***	22	454.55	***0/5***	***5041.6***	***38.65***	22	65	***0/5***	5044.73	***38.65***	22	***0.33***	4/5
T46	3338.15	***31.11***	18	205	***0/5***	***3270.65***	***31.11***	18	21.9	***0/5***	3338.15	***31.11***	18	***0.14***	3/5
T47	3340.95	31.09	18	123.9	***0/5***	***3246.35***	***29.48***	17	7.5	***0/5***	3274.35	***29.48***	17	***0.13***	4/5
T48	***4606.8***	33.47	19	383.8	***0/5***	4638.8	***30.23***	17	22.45	***0/5***	4792.9	31.86	18	***0.28***	4/5
T49	8146.0	***61.56***	35	1049.7	***0/5***	***8114.42***	***61.56***	35	118.8	***0/5***	8242,8	***61.56***	35	***0.83***	4/5

Table A.5: First feasible result from HASRUT, HASRUT algorithm final result and MLKL comparison for network delay= 50ms (negative values represent infeasible deployment)

Test model #	HASRUT First Feasible Result				HASRUT Final Result				MLKL			
	Response Time	Power	Run time	# of failures	Response Time	Power	Run time	# of failures	Response Time	Power	Run time	# of failures
T1	3968.92	31.52	0.02	***0/5***	***3956.08***	***31.52***	2	***0/5***	4053.25	***31.52***	***0.09***	4/5
T2	3568.7	***21.56***	0.048	***0/5***	***3551.7***	***21.56***	11.5	***0/5***	3599.2	***21.56***	***0.08***	3/5
T3	3760.6	38.52	0.063	***0/5***	***3768.1***	***38.52***	12.5	***0/5***	3900.1	***38.52***	***0.09***	3/5
T4	2638.05	24.14	0.063	***0/5***	***2592.05***	***24.14***	6.04	***0/5***	2768.55	***24.14***	***0.08***	4/5
T5	4316.95	34.96	0.079	***0/5***	***4276.7***	***33.34***	10.73	***0/5***	4338.7	***33.34***	***0.08***	4/5
T6	6986.2	*51.12*	0.16	***0/5***	6937.45	***49.5***	50.46	***0/5***	***6912.85***	51.12	***0.22***	4/5
T7	4814.4	*69.29*	0.66	***0/5***	***4801.9***	***68.34***	39.81	***0/5***	-1	-1	-1	5/5
T8	7270.9	***70.73***	0.6	***0/5***	7270.97	***70.73***	32.83	***0/5***	-1	-1	-1	5/5
T9	1532.3	**12.45**	0.001	***0/5***	***1496.75***	***12.45***	3.4	***0/5***	1651.65	***12.45***	***0.07***	3/5
T10	3950.12	***51.25***	0.14	***0/5***	***3938.05***	***51.25***	10.14	***0/5***	4000.7	***51.25***	***0.21***	4/5
T11	3030.35	**33.59**	0.04	***0/5***	***2960.15***	***33.59***	4.5	***0/5***	2970.15	***33.59***	***0.13***	2/5
T12	1746.25	***10.76***	0.11	***0/5***	***1746.25***	***10.76***	1.7	***0/5***	-1	-1	-1	5/5
T13	5295.2	61.39	0.3	***0/5***	***5292***	***61.39***	14.5	***0/5***	5315.85	64.62	***0.35***	4/5
T14	2046.8	***14.17***	0.03	***0/5***	***2039.3***	***12.55***	6.29	***0/5***	-1	-1	-1	5/5
T15	3574.9	***40.98***	0.05	***0/5***	***3569.9***	***40.98***	7.4	***0/5***	3676.5	***40.98***	***0.12***	4/5
T16	2225.07	22.82	0.06	***0/5***	***2211.75***	***22.82***	4.2	***0/5***	2234.25	24.44	***0.1***	4/5
T17	2327.3	***17.94***	0.05	***0/5***	***2327.3***	***17.94***	2.2	***0/5***	-1	-1	-1	5/5
T18	2621.3	***29.82***	0.05	***0/5***	***2603.8***	***29.82***	4.5	***0/5***	2702.4	***29.82***	***0.1***	3/5
T19	5657.25	***74.84***	0.16	***0/5***	***5615.2***	***74.84***	22.9	***0/5***	5682	***74.84***	***0.5***	4/5
T20	5951.42	50.51	0.19	***0/5***	***5941.42***	***50.51***	28.5	***0/5***	-1	-1	-1	5/5
T21	5442.6	44.30	0.14	***0/5***	***5435.1***	***42.6***	26.7	***0/5***	5623.8	44.3	***0.3***	4/5
T22	4209.35	***53.28***	0.13	***0/5***	***4209.35***	***53.28***	21.88	***0/5***	-1	-1	-1	5/5
T23	5267.77	***34.04***	0.17	***0/5***	***5176.83***	***34.04***	12.75	***0/5***	5237.55	***34.04***	***0.25***	3/5
T24	3443.0	42.4	0.12	***0/5***	***3443***	***42.4***	28	***0/5***	-1	-1	-1	5/5

T25	3108.8	23.92	0.38	***0/5***	***3110.22***	***22.29***	16.7	***0/5***	3301.2	22.29	***0.1***	2/5
T26	5535.9	60.13	0.25	***0/5***	***5535.9***	***60.13***	26.86	***0/5***	5680.85	61.75	***0.3***	3/5
T27	3096.95	***37.03***	0.05	***0/5***	***3091.95***	***37.03***	8.21	***0/5***	3339.48	***37.03***	***0.15***	3/5
T28	5780.37	***81.48***	0.77	***0/5***	***5737.03***	***81.48***	61	***0/5***	-1	-1	-1	5/5
T29	2997.1	36.10	0.09	***0/5***	***2971.78***	***35.95***	14.5	***0/5***	3004.65	37.57	***0.14***	4/5
T30	3241.1	27.28	0.05	***0/5***	***3236.1***	***25.66***	14.8	***0/5***	3278.05	27.28	***0.15***	4/5
T31	2240.8	***20.77***	0.1	***0/5***	***2240.8***	***20.77***	6.8	***0/5***	2246.05	***20.77***	0.6	4/5
T32	***8311.49***	61.68	0.3	***0/5***	8435.2	***60.06***	29.8	***0/5***	-1	-1	-1	5/5
T33	2253.15	***28.16***	0.06	***0/5***	***2245.65***	***28.16***	8.25	***0/5***	2329.23	***28.16***	***0.07***	4/5
T34	4392.25	27.21	0.11	***0/5***	***4382.55***	***25.59***	32.85	***0/5***	4452.5	27.21	***0.22***	4/5
T35	5422.77	***57.73***	0.09	***0/5***	***5409.95***	***57.73***	24.7	***0/5***	5514.35	***57.73***	***0.5***	3/5
T36	2431.85	***27.94***	0.05	***0/5***	***2431.85***	***27.94***	6.4	***0/5***	-1	-1	-1	5/5
T37	2316.45	***29.37***	0.03	***0/5***	***2308.95***	***29.37***	3.93	***0/5***	2371.9	***29.37***	***0.07***	4/5
T38	6175.25	***61.31***	0.25	***0/5***	***6171.5***	***59.69***	37	***0/5***	-1	-1	-1	5/5
T39	2489.12	***25.78***	0.11	***0/5***	***2489.12***	***25.78***	8.45	***0/5***	-1	-1	-1	5/5
T40	3782.75	28.17	0.18	***0/5***	***3760.55***	***26.55***	9.73	***0/5***	3828.05	***26.55***	***0.12***	3/5
T41	2123.7	25.56	0.1	***0/5***	***2116.2***	***25.56***	7.19	***0/5***	2196.7	27.18	***0.09***	4/5
T42	6903.3	***70.55***	0.28	***0/5***	***6898.8***	***70.55***	63.59	***0/5***	6910.9	72.17	***0.65***	4/5
T43	***4547.35***	28.58	0.09	***0/5***	4560.77	***23.72***	8.8	***0/5***	4547.35	28.58	***0.20***	2/5
T44	2716.2	***27.54***	0.14	***0/5***	***2709.93***	***25.91***	6.17	***0/5***	2778.9	***25.91***	***0.09***	1/5
T45	5036	***38.65***	0.3	***0/5***	***5031***	***38.65***	36	***0/5***	5040.65	***38.65***	***0.35***	4/5
T46	3270.65	***31.11***	0.4	***0/5***	***3263.15***	***31.11***	21.99	***0/5***	-1	-1	-1	5/5
T47	3246.45	29.47	0.05	***0/5***	***3238.85***	***29.47***	8.24	***0/5***	-1	-1	-1	5/5
T48	***4576.8***	31.86	0.19	***0/5***	4561.9	***30.23***	21.87	***0/5***	4560.6	33.47	***0.33***	4/5
T49	***8114.2***	61.55	0.34	***0/5***	8145.1	***59.93***	91.09	***0/5***	8171	61.55	***0.82***	3/5

Table A.6: First feasible result from HASRUT, HASRUT algorithm final result and MLKL comparison for network delay= 200ms (negative values represent infeasible deployment)

Test Model	HASRUT First Feasible Result				HASRUT Final Result				MLKL			
	Response Time	Power	Run time	# of failures	Response Time	Power	Run time	# of failures	Response Time	Power	Run time	# of failures
T1	4120.42	33.14	***0.009***	0/5	***3955.08***	***31.52***	1.9	0/5	4417.75	***31.52***	0.09	3/5
T2	3694.7	***21.56***	***0.08***	0/5	***3596.7***	***21.56***	15.08	0/5	3786.7	***21.56***	0.09	4/5
T3	***3783.1***	***38.52***	***0.06***	0/5	***3783.1***	***38.52***	12.13	0/5	4341.1	***38.52***	0.09	3/5
T4	2826.55	***24.14***	***0.03***	0/5	***2659.55***	***24.14***	6.2	0/5	-1	-1	-1	5/5
T5	4468.7	34.96	***0.06***	0/5	***4315.7***	***33.34***	14.56	0/5	4563.7	***33.34***	0.11	4/5
T6	7103.45	51.12	0.5	0/5	7027.45	***49.5***	54	0/5	***6910.62***	52.75	0.16	4/5
T7	4874.4	69.29	***0.61***	0/5	***4854.4***	***68.34***	90	0/5	-1	-1	-1	5/5
T8	7619.6	***70.73***	***0.44***	0/5	***7244.7***	***70.73***	48.75	0/5	-1	-1	-1	5/5
T9	1564.75	14.07	0.063	0/5	***1541.75***	***12.45***	3.5	0/5	1738.5	***12.45***	0.05	4/5
T10	3998.1	***51.25***	***0.18***	0/5	***3977.2***	***51.25***	10.72	0/5	3985.0	***51.25***	0.22	2/5
T11	3243.75	***33.59***	***0.05***	0/5	***2975.15***	***33.59***	4.5	0/5	3095.68	***33.59***	0.12	4/5
T12	***1791.25***	***10.76***	***0.033***	0/5	***1791.25***	***10.76***	1.65	0/5	2017.62	***10.76***	0.05	4/5
T13	5408.95	***61.39***	***0.09***	0/5	***5310.45***	***61.39***	23.94	0/5	5543.45	***61.39***	0.35	4/5
T14	2112.7	14.17	***0.09***	0/5	***2089.1***	***12.55***	6.4	0/5	-1	-1	-1	5/5
T15	3604.9	***40.98***	***0.06***	0/5	***3584.9***	***40.98***	7.36	0/5	3788.1	***40.98***	0.13	2/5
T16	2317.55	***22.82***	***.08***	0/5	***2234.25***	***22.82***	4.14	0/5	2264.25	24.44	0.10	4/5
T17	2423.77	***17.94***	***0.016***	0/5	***2368.63***	***17.94***	2.3	0/5	2450.7	19.55	0.10	4/5
T18	2698.8	***29.82***	***0.048***	0/5	***2618.8***	***29.82***	4.44	0/5	3193.8	***29.82***	0.08	3/5
T19	5813.8	***74.84***	***0.19***	0/5	***5630.2***	***74.84***	21.33	0/5	5753.8	***74.84***	0.46	3/5
T20	***5957.67***	***50.52***	***0.2***	0/5	***5957.67***	***50.52***	27.78	0/5	6060.6	***50.52***	0.33	2/5
T21	***5496.9***	44.3	***0.08***	0/5	***5466.9***	***42.68***	27.00	0/5	5636.3	***42.68***	0.28	4/5
T22	***4238***	***53.28***	***0.11***	0/5	***4238.0***	***53.28***	21.49	0/5	4567.8	***53.28***	0.22	3/5
T23	***5249.28***	35.66	***0.23***	0/5	5270.28	***34.04***	12.37	0/5	5355.75	***34.04***	0.26	4/5
T24	***3488***	***42.4***	0.1	0/5	3488.0	***42.40***	28.31	0/5	3584.4	***42.40***	0.24	4/5
T25	***3148.2***	23.92	0.22	0/5	3153.77	***22.29***	17.15	0/5	3506.7	23.92	***0.13***	3/5
T26	***5558.4***	***60.13***	0.16	0/5	5558.4	***60.13***	25.78	0/5	5648.4	61.75	0.28	4/5

T27	3146.95	***37.03***	0.05	0/5	3106.95	***37.03***	8.42	0/5	3408.15	***37.03***	0.14	2/5
T28	5890.8	***81.48***	0.14	0/5	5829.2	***81.48***	62.4	0/5	6105.0	83.10	0.73	4/5
T29	***3003.08***	***35.95***	0.09	0/5	3003.08	***35.95***	14.47	0/5	3337.35	37.57	0.14	3/5
T30	***3292.23***	***25.66***	0.22	0/5	3292.23	***25.66***	13.27	0/5	3418.9	27.28	***0.13***	3/5
T31	***2299***	***20.77***	0.08	0/5	2299.0	***20.77***	6.74	0/5	2320.0	***20.77***	***0.06***	4/5
T32	8726.23	63.3	0.14	0/5	8694.67	***61.68***	29.46	0/5	-1	-1	-1	5/5
T33	2298.15	***28.16***	0.05	0/5	2268.15	***28.16***	7.97	0/5	2622.48	***28.16***	0.08	4/5
T34	4499.25	27.21	0.12	0/5	4441.65	***25.59***	33.29	0/5	4721.45	27.21	0.24	4/5
T35	5488.72	***57.73***	0.14	0/5	5447.45	***57.73***	23.08	0/5	5488.72	***57.73***	0.27	4/5
T36	***2454.35***	***27.94***	0.08	0/5	2454.35	***27.94***	6.3	0/5	2514.35	***27.94***	0.12	4/5
T37	2343.95	***29.38***	0.03	0/5	2323.95	***29.38***	3.99	0/5	2575.75	***29.38***	0.08	4/5
T38	***6255.85***	***59.69***	0.17	0/5	6222.35	***59.69***	38.4	0/5	-1	-1	-1	5/5
T39	***2521.3***	25.78	0.11	0/5	2535.8	***24.17***	5.8	0/5	2544.1	25.78	0.14	3/5
T40	***3900.65***	26.55	0.17	0/5	3944.65	***24.93***	9.24	0/5	-1	-1	-1	5/5
T41	***2161.2***	***25.56***	0.06	0/5	2161.2	***25.56***	6.73	0/5	2581.2	27.18	0.09	4/5
T42	6948.3	***70.55***	0.2	0/5	6928.3	***70.55***	62.46	0/5	6942.11	***70.55***	0.67	4/5
T43	4935.8	25.34	0.14	0/5	4902.69	***23.72***	9.3	0/5	***4672.4***	25.34	0.2	3/5
T44	2762.58	***25.92***	0.12	0/5	2742.58	***25.92***	6.42	0/5	***2733.35***	27.54	0.13	2/5
T45	5147.93	***38.65***	0.39	0/5	5095.93	***38.65***	38.67	0/5	5147.1	***38.65***	***0.3***	4/5
T46	3302.05	***31.11***	0.49	0/5	3285.65	***31.11***	22.25	0/5	3420.65	***31.11***	***0.16***	4/5
T47	3291.35	***29.48***	0.17	0/5	3261.35	***29.48***	7.89	0/5	3555.15	***29.48***	***0.11***	2/5
T48	4828.5	35.09	0.16	0/5	4769.0	***30.23***	23.84	0/5	***4720.7***	33.48	0.32	4/5
T49	8159.2	***61.56***	0.3	0/5	8129.2	***61.56***	118	0/5	8386.4	***61.56***	0.84	2/5