Boston University

OpenBU **http://open.bu.edu**

Theses & Dissertations Boston University Theses & Dissertations

2019

Orchestration and management of application functions over virtualized cloud infrastructures

https://hdl.handle.net/2144/39521

Boston University

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

ORCHESTRATION AND MANAGEMENT OF APPLICATION FUNCTIONS OVER VIRTUALIZED CLOUD INFRASTRUCTURES

by

NABEEL AKHTAR

B.Sc., Lahore University of Management Sciences (LUMS), 2011
M.S., Koç University, 2013

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2019

© Copyright by
NABEEL AKHTAR
2019

Approved by

First Reader ______________________________

Abraham Matta, PhD
Professor of Computer Science

Second Reader ______________________________

Vatche Ishakian, PhD
Assistant Professor of Computer Information Systems
Bentley University

Acknowledgments

First and foremost, I would like to thank my advisor Prof. Ibrahim Matta, who has given me tremendous support during my PhD journey. Ibrahim is not only a superb advisor, but also a great collaborator and a considerate friend, and I feel very lucky to have him.

I also thank my thesis committee members, Azer Bestavros, Rich West, Jonathan Appavoo, and Vatche Ishakian, for reading my thesis and providing helpful comments.

I thank my fellow lab-mates and friends, who made the PhD journey fun and kept me going during my lowest moments.

I also thank my family for their generous support. Last but not least, I must express my deepest appreciation and greatest admiration to my lovely wife, Afifa Mehmood, for her endless support during my PhD journey.

ORCHESTRATION AND MANAGEMENT OF APPLICATION FUNCTIONS OVER VIRTUALIZED CLOUD INFRASTRUCTURES

NABEEL AKHTAR

Boston University, Graduate School of Arts and Sciences, 2019

Major Professor: Abraham Matta, Professor of Computer Science

ABSTRACT

Next-generation networks are expected to provide higher data rates and ultra-low latency in support of demanding applications, such as virtual and augmented reality, robots and drones, *etc*. To meet these stringent requirements of applications, edge computing constitutes a central piece of the solution architecture wherein functional components of an application can be deployed over the edge network to reduce bandwidth demand over the core network while providing ultra-low latency communication to users. In this thesis, we provide solutions to resource orchestration and management for applications over a virtualized *client-edge-server* infrastructure.

We first investigate the problem of optimal placement of pipelines of application functions (virtual service chains) and the steering of traffic through them, over a multi-technology edge network model consisting of both wired and wireless millimeter-wave (mmWave) links. This problem is $\mathcal{NP}$-hard. We provide a comprehensive "microscopic" binary integer program to model the system, along with a heuristic that is one order of magnitude faster than optimally solving the problem. Extensive evaluations demonstrate the benefits of orchestrating virtual service chains (by distributing them over the edge network) compared to a baseline "middlebox" approach in terms of overall admissible virtual capacity.

Next, we look at the problem of finding the optimal configuration parameters, such as memory and CPU, for application functions running as serverless functions, *i.e.* they run in stateless compute containers that are event-driven, ephemeral, and fully managed by the cloud provider. While serverless computing is a relatively simpler computing model, configuring such parameters correctly while minimizing cost and meeting delay constraints is not trivial. To solve this problem, we present a framework that uses Bayesian Optimization to find the optimal configuration for serverless functions. The framework uses statistical learning techniques to intelligently collect samples with the goal of predicting the cost and execution time of a serverless function across unseen configuration values. Our framework uses the predicted cost and execution time to select the "best" configuration parameters for running a single or a chain of serverless functions (service chains). Evaluations on a commercial cloud provider and a wide range of simulated distributed cloud environments confirm the efficacy of our approach.

Contents

1 Introduction **1**

1.1 Background and Motivation 3

1.2 Challenges 8

1.2.1 Virtual Function Placement and Traffic Steering 8

1.2.2 Configuring the Parameters of Virtual Functions 9

1.3 Thesis Contributions 11

1.4 Roadmap of thesis 12

2 Related Work **13**

2.1 Virtual Function Placement and Traffic Steering 13

2.2 Configuring Virtual Functions 15

3 Virtual Function Placement and Traffic Steering **18**

3.1 Introduction 18

3.2 Background and Related Work 21

3.2.1 Placement 23

3.2.2 Traffic Steering 24

3.3 System Model 25

3.4 Mathematical Model 28

3.4.1 Variables 29

3.4.2 BIP Formulation 30

3.5 Evaluation Model, Parameters and Proposed Heuristic 35
3.5.1 Edge Network Graphs 35
3.5.2 Input Flow Parameters 42
3.5.3 Proposed Heuristic 43
3.6 Evaluation Results 46
3.7 Summary 56

4 Configuring Serverless Functions using Statistical Learning 57

4.1 Introduction 57
4.2 System Description 61
4.3 COSE: The *Performance Modeler* component 63
4.3.1 Our Approach: Leveraging Bayesian Optimization 65
4.3.2 Adapting BO for Serverless Functions 70
4.4 COSE: The *Config Finder* Component 72
4.5 Experimental Results: Running COSE on Amazon Lambda 75
4.5.1 Representative Functions 75
4.5.2 Evaluation Results 76
4.6 Evaluation in a Distributed Cloud Environment 78
4.6.1 Modeling Cloud Provider 78
4.6.2 Modeling cost and execution time 79
4.6.3 Simulation Results 81
4.7 Related Work 87
4.8 Summary 88

5 Conclusion and Research Directions 89

5.1 Summary of Contributions 89

5.2 Open Research Directions 90

Bibliography **91**

Curriculum Vitae of Nabeel Akhtar **98**

List of Tables

1.1 Comparison of Virtualization Services 7

3.1 Network Parameters. 29

3.2 Traffic Parameters. 30

3.3 Variables. 31

3.4 Graph Parameters and Characteristics 39

3.5 Evaluation Parameters 40

3.6 Flow Parameters 44

4.1 Serverless platforms 58

List of Figures

1.1 Virtual Reality under the traditional client-server model 4

1.2 Virtual Reality application hosted on edge-cloud 4

3.1 Function virtualization for 5G mobile network 19

3.2 Virtual Reality use case 27

3.3 Augmented Reality use case 27

3.4 Santa Monica network topology 36

3.5 Palo Alto network topology 37

3.6 Multi-technology edge network consisting of processing and routing nodes. 38

3.7 Probability of successful bit delivery over a *mmWave* link 40

3.8 Cost vs running time comparison of BIP vs Heuristic 47

3.9 Flow Acceptance Ratio for *Wire*, *Single* and *Dual* networks with BIP and Heuristic 48

3.10 Virtual Capacity Allocated for *Wire*, *Single* and *Dual* networks with BIP and Heuristic 49

3.11 Average Link Utilization for all links as a function of incoming flows for *Wire*, *Single* and *Dual* networks 50

3.12 Average Link Utilization for *Wire* links as a function of incoming flows for *Wire*, *Single* and *Dual* networks 51

3.13 Average Link Utilization for *mmWave* links as a function of incoming flows for *Single* and *Dual* networks 52

3.14 CDF of *mmWave* link utilization for *Single* networks 53
3.15 Flow Acceptance Ratio under middlebox schenario for *Wire*, *Single* and *Dual* networks 54
3.16 Virtual Capacity Allocated under middlebox schenario for *Wire*, *Single* and *Dual* networks 55

4.1 Serverless function's performance with different memory sizes and co-location 59
4.2 System overview 62
4.3 Example of AIAD getting stuck at a local minimum 64
4.4 Bayesian Optimization example 67
4.5 COSE performance on Amazon Lambda for *I/O Intensive* serverless function 76
4.6 BO's Convergence and EI 82
4.7 Configuration with COSE 83
4.8 COSE for serverless function with changing execution model 83
4.9 Delay bounded chaining of serverless functions 85

List of Abbreviations

5G	. . . .	Fifth Generation
AIAD	. . . .	Additive Increase Additive Decrease
AR	. . . .	Augmented Reality
AWS	. . . .	Amazon Web Services
BIP	. . . .	Binary Integer Program
BO	. . . .	Bayesian Optimization
CDF	. . . .	Cumulative Distribution Function
COdc	. . . .	Central Offices Data Centers
COs	. . . .	Central Offices
COSE	. . . .	Configuring Serverless Framework
CP	. . . .	Cloud Provider
CPU	. . . .	Central Processing Unit
EC	. . . .	Elliptical Curve
EC2	. . . .	Amazon Elastic Compute Cloud
EI	. . . .	Expected Improvement
eNodeB	. . . .	Evolved Node B
EPC	. . . .	Evolved Packet Core
FaaS	. . . .	Function as a Service
GP	. . . .	Gaussian Process

GPU	....	Graphics Processing Unit
HDM	....	Head Mounted Device
HDR	....	High Dynamic Range
HTTP	....	Hyper Text Transfer Protocol
ILP	....	Integer Linear Program
ITU-R	....	International Telecommunication Union's Radio-communication Sector
L2/L3	....	Layer 2/Layer 3
LOS	....	Line of Sight
mmWave	....	Millimeter Wave
MPI	....	Maximum Probability of Improvement
NFV	....	Network Function Virtualization
$\mathcal{NP}$	....	Non-deterministic Polynomial
OPEX	....	Operating Expense
P-GW	....	Packet Data-network Gateway
PN	....	Processing Nodes
QoS	....	Quality of Service
RAN	....	Radio Access Network
RN	....	Routing Nodes
S-GW	....	Serving Gateway
SDN	....	Software Defined Networking
SFC	....	Service Function Chaining
UCB	....	Upper Confidence Bound
UE	....	User Equipment
VM	....	Virtual Machine

VNE	....	Virtual Network Embedding
VR	....	Virtual Reality
VF	....	Virtual Function

Chapter 1

Introduction

Next-generation networks are expected to go beyond the delivery of static content or streaming content, such as telephony, web browsing, and low-resolution video. They are envisioned to support billions of devices, adapt to sudden and frequent changes in demand, and seamlessly process user information at much higher data rates (exceeding 500 Mbps) and ultra-low latencies (less than 5 milliseconds) [25, 47, 67].

Potential next-generation applications, such as augmented/virtual reality, robots and drone control, healthcare, *etc.*, have stringent network delay and bandwidth requirements. Currently, application servers are supported on public cloud and users experience network delay in the range of 20-100 milliseconds [67]. With such high network delays, the current client-server model cannot support next-generation applications given their stringent requirements. To decrease network latencies and increase data rates, we need to change both the network design, and the application design, and we need to use available virtual resources efficiently. We envision that the following changes need to be made to decrease latencies and increase data rates.

- Use a *client-edge-core model*, instead of the traditional *client-server model*. In the *client-edge-core model*, the clients experience higher speed and very low network delays over the edge network where applications can run. The edge network, closer to end-users, has 1-5ms of network delays and can provide speeds of 500 Mbps or higher.

The edge network can be a multi-technology network (*e.g.*, using mmWave links along with wired links to support low latency and high bandwidth requirements [23]). Furthermore, instead of using bare-metal machines, virtualization technologies can be introduced at the edge of the network. Virtualization means that the applications are not tied to a specific piece of physical hardware and multiple application workloads can run simultaneously on the same piece of hardware. The edge network should be able to support different virtualization services, such as Virtual Machines (VMs) [28], Containers [34], and Serverless [27, 38]. The flexibility in choosing different virtualization services enables edge resources to be effectively used depending on the burstiness of the application's computations, *i.e.*, the ratio of computation time to the total running time of the application. In particular, VMs are appropriate for supporting long-term computations where a user is responsible for installing all systems and application programs given a certain amount of resources (e.g., memory and CPU) allocated to the VM. On the other hand, a Serverless computing model is typically more appropriate for short-term computations where a user configures resource parameters to run application programs, and it's up to the cloud provider to provide the necessary underlying execution infrastructure that may take the form of either VMs or Containers.

- Change the way the applications are designed to support the *client-edge-core model* of the network, where low-latency components are hosted on the edge of the network and other components are hosted on the network core. Applications should have a modular design, where different application modules can run as Virtual Functions (VFs) either at the edge or the core of the network. This modular application design inherently avoids vendor lock-in. Depending on the functionality of a VF and application requirements, a VF can either run on a Virtual Machine, Container, or as a Serverless Function.

- Intelligently configure multiple system parameters, such as memory, CPU, location,

etc., for running VFs on cloud providers. Configuring such parameters correctly can significantly decrease the cost while meeting the performance criteria. However, choosing the “best” parameters for minimizing cost and meeting delay constraints is not trivial.

These aforementioned changes come with their own set of unique challenges. The first challenge includes finding efficient ways to decompose applications into VFs, which is inherently dependent not only on finding the appropriate virtualization service to use for each application VF, but also on the way these VFs are chained together to perform a certain task. The second challenge involves the placement of the decomposed VFs along the *client-edge-core model*, together with resource management to strictly adhere to the stringent application quality-of-service (QoS) requirements. While typical static placement algorithms are $\mathcal{NP}$-hard, any suggested placement or resource management technique needs to quickly scale resources and adapt to frequent and sudden changes in demand. Finally, all the aforementioned challenges need to be resolved while taking into consideration the cost incurred for using cloud and edge resources. Choosing the “best” configuration (CPU, RAM, location, etc.) for running virtual resources on a cloud provider should not only attempt to minimize cost but also meet the performance needs of the application.

1.1 Background and Motivation

Redesigning The Applications:

Next-generation applications, such as Augmented and Virtual reality (AR and VR), are envisioned to have strict network requirements. Qualcomm and AT&T [25, 67], two of the leading network operators, have termed AR/VR as the “killer applications” that will change the way the network is currently designed. First, AR and VR videos include new formats, such as stereoscopic, high dynamic range (HDR) [29], and 360°, at increased resolutions (more than 8K) and frame rates (higher than 90 fps). This drastically increases the

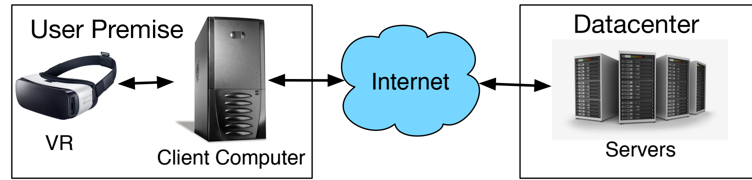


Figure 1.1: Virtual Reality under the traditional client-server model

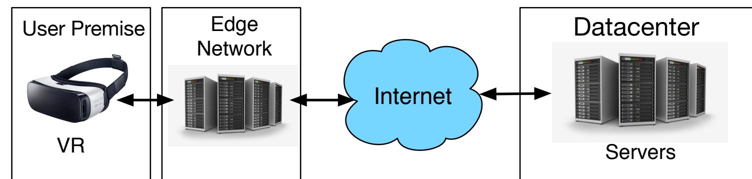


Figure 1.2: Virtual Reality application hosted on edge-cloud

bandwidth requirement of the network to more than 500 Mbps. Second, AR/VR content is highly interactive. For traditional non-interactive content that can use buffering, network latency is generally not an issue. However, with interactive content, low latency is required to ensure responsiveness. Consider, for example, an online gaming application using VR and AR where the user has a head mounted device (HMD), the latency between an action (head movement) and reaction (the display getting updated) is crucial. For such scenarios, network latency of less than 5ms is expected.

To support such data-intensive, latency-sensitive applications, there is a need to redesign both the application architecture and the supporting network and computing infrastructure. Under the current client-server model, content is hosted in servers running mostly in datacenters. The network delays experienced by a typical networked application range from 20-100 milliseconds. This is largely due to the propagation and transmission delays over the wide-area Internet. By hosting the application at the network edge, network delays could be decreased.

The value proposition is promising since it enables us to host next generation, edge-

driven applications, and thus meet their stringent delay and bandwidth requirements. Hosting an application at the edge of the network comes with a set of challenges [79]. Figure 1.1 shows a VR application as currently used under the client-server model. The VR set needs to be connected to a local computer that runs most of the computations for the VR set and transfers data from/to the VR set. The computer also needs to be connected to the central servers running in a datacenter with a good Internet connection. Such client-server model may only support simple interactive VR applications. More complex VR applications would typically experience computation and/or network bottlenecks. Campus, home, and enterprise networks today simply cannot support the high bandwidth and low latency required by the envisioned highly interactive VR applications.

Figure 1.2 shows a recent trend in support of edge-driven applications. Edge resources are connected with application servers with good Internet connections, and intensive compute tasks are run at the network edge. One recent example is Parsec Gaming [64], where users rent machines on the network edge, *e.g.*, to play online multi-player games. Since compute intensive tasks run at the network edge, the user premise computer can be a simple machine, *e.g.*, a Raspberry Pi. By moving the computation to the network edge, Parsec enables people to play compute intensive games on simple machines while providing lower network delays. Similarly, Google has recently launched Google's Stadia [9], which uses edge computing nodes that can handle part of the workload much closer to the end-user than Google's data centers.

While edge resources help lower network latency, they are scarce, and utilizing them is costly. To efficiently use edge resources, and to fulfill application requirements, we believe that applications can be divided into modules that can run as virtual functions, either in parallel or chained in a sequence, to provide a required service. The advantage of dividing the application into smaller modules leads to a gain in flexibility on where we could run

them; some options include running a module locally, on a server located at the edge, or even across multiple edge-cloud providers. Although dividing the applications into smaller modules has software engineering challenges, such flexibility enables the infrastructure to run optimizations specifically for a single module (set) – for example, using GPU to perform encoding/transcoding of video, for in-network anomaly detection processing, or for Elliptical Curve (EC) cryptography operations. The majority of the tasks, *e.g.*, 3D construction of gaming scenes, currently done on the *servers* in the core datacenter, could be performed at the edge of the network, thus significantly reducing user perceived and network latencies. The state of the game still needs to be shared with a central server running in the core datacenter. However, we believe that this information is more delay tolerant as it does not directly impact the user experience. These modules are independent of each other and they can run on a single or multiple machines at the edge. Note that the same application module (or VF) is capable of supporting multiple application flows, *i.e.*, a single VF instance can serve multiple clients at the same time.

Redesigning The Edge Infrastructure:

To support next-generation applications that run on the network edge and core, and to support modular application design, there is a need for change in the network infrastructure. Currently, datacenters are typically geographically far from the end-users, which leads to lower bandwidth and high network latencies. Edge computing aims to lower the delays. However, it is important that the edge network is designed taking into account different types of user applications running as virtual functions. Application flows originating from a user equipment (UE) are served by the application virtual functions that are running at the network edge, and by the application server running at the core (datacenter). To provide better connectivity, the network edge can consist of heterogeneous technologies, *e.g.*, using millimeter-Wave (mmWave) links along with wired links, to increase the overall

capacity of the edge network and to lower network delays. Recently, Facebook's Connectivity Lab tested up to 20 Gbps links over 13 km with mmWave technology in Southern California [54], and Facebook's Terragraph project [35] is developing a multi-node gigabit wireless edge network for dense urban areas.

The edge network should also be able to support different virtualization services. Currently, most edge networks are assumed to support Virtual Machines only. Given recent advancements in edge infrastructure technologies and devices (*e.g.*, AWS Greengrass which includes support for AWS Lambda [41]), and continual evolution of applications, it is becoming apparent that the future edge network will support heterogeneous virtualization services, *e.g.*, Virtual Machines (VMs), Containers, and Serverless computing platforms. By having the flexibility of choosing among different virtualization services, the placement of application virtual functions can better adapt to the needs of the application. Table 1.1 shows a comparison of different virtualization services offered by current cloud providers. Provisioning time refers to the time to start a VF, utilization measures the usage of resources, charging granularity refers to the minimum amount of execution time charged, and cost per unit time is the price that the user pays for the VF per unit time.

Table 1.1: Comparison of Virtualization Services

Technology	Provisioning time	Utilization	Charging Granularity	Cost per unit time
Virtual Machine	seconds to minutes	low	minutes to hours	low
Container	seconds	high	minutes	high
Serverless	milliseconds	highest	milliseconds	highest

VMs are cheapest in terms of cost per unit of time and allow for full control of the virtual resources. Their utilization is typically low, they have a high provisioning time (a few minutes), and the charging granularity is minutes to hours. Furthermore, VMs are less flexible, and harder to manage and scale. Serverless has the highest cost per

unit of time and the least control of virtual resources. They inherently support stateless execution of functions, but provide out-of-the-box scalability. Their provisioning time and charging granularity is milliseconds, so users only pay when the serverless function runs, which leads to higher utilization of cloud resources. Serverless is more suitable for bursty computations [27], where the user runs small computations intermittently. In this case, the total cost is low. If the VF is constantly being used, serverless can be significantly more expensive as compared to VMs and Containers. Serverless has limitations [56] on the amount of memory allocated to the function (*e.g.*, maximum of 3GB on AWS Lambda) and its execution time (*e.g.*, maximum of 300 seconds on AWS Lambda). Furthermore, users have no control over the environment, *i.e.*, one cannot custom install packages and software on the running environment.

1.2 Challenges

1.2.1 Virtual Function Placement and Traffic Steering

Placement of application service graphs, consisting of virtual functions (VFs), across multiple nodes, and possibly across multiple cloud providers, over the edge-cloud, and steering traffic through the virtual functions, is a hard problem to solve. A set of VFs may need to be chained together to provide a certain service and this chain needs to be placed on the virtualized infrastructure. Note that multiple instances of the same VF may be deployed in the network, and each VF instance can support multiple users.

VF Placement: The task of creating and deploying virtual service chains, annotated with their resource requirements, is similar to the Virtual Network Embedding (VNE) and Network Function Virtualization (NFV) placement problem [37, 68, 42, 61]. However, application function virtualization brings a unique set of challenges. The application functions can be placed on different types of virtualization technologies, *e.g.*, placing a VF doing

video encoding on GPUs. The characteristics of the functions can also be very diverse, so a function with "bursty" computation may be best placed on a serverless platform, while a function running for longer times may be best placed on a VM. Depending on the location of customers, multiple sets of the same service chain might be placed on the network. For reliability, replicating VF chains and placing them strategically over the network is also needed. Similar to VNE, the placement of application (virtual) service graphs is $\mathcal{NP}$-hard. Formulated as an optimization problem, VF placement and chaining reduces to an integer program, which is $\mathcal{NP}$-hard and intractable for larger inputs.

VF Traffic Steering: Traffic steering through VFs residing at different locations is challenging. Software-Defined Networking (SDN) offers a flexible control approach to install traffic forwarding rules. However, SDN capabilities have been limited to L2/L3 forwarding functions and cannot support application VFs. SDN based solutions have been proposed [36, 66, 85] that extend the current L2/L3 functions of SDN to provide a policy enforcement layer for VF traffic steering. Although extended SDN mechanisms have enabled VF traffic steering, finding the best path through a set of VFs under multiple constraints, such as delay and throughput, is $\mathcal{NP}$-complete [65].

1.2.2 Configuring the Parameters of Virtual Functions

Configuring appropriate resources to either VMs/Containers or Serverless functions is essential to the cost and performance of running application programs. As noted earlier, VMs/Containers support computations over a longer time scale. Once the resources are reserved for VMs/Containers, the users typically install their systems and application programs, and the resources are usually kept anywhere from a few hours to multiple days. Because of the longer timescales for managing VMs/Containers' resources, the configuration parameters for resources are either tuned offline [24], or they are re-configured

using monitoring services [12]. Although the serverless platform is built on top of VMs or Containers, serverless application functions require little/no setup time, and they typically finish execution in a few milliseconds. Typically, a serverless function is periodically triggered when certain events happen. Serverless functions are an ideal candidate for adaptive online resource configuration since savings resulting from a proper cloud configuration are most significant for recurring jobs [22].

In this thesis, we focus on configuring the parameters of serverless functions. These application Virtual Functions run as independent modules on cloud providers. These VFs have a wide variety of use cases, with different types of evolving techniques used for data processing, such as Deep Learning, Encoding/Transcoding, and 3D reconstruction. Hence, these functions have diverse behaviors and resource requirements (memory, CPU, GPU, disk). To run these VFs on a cloud provider, a user needs to select different types of resources such as VMs, Containers, or Serverless Functions. Moreover, the users need to select the resource configurations – the number of CPU cores, amount of memory, disk space – for each type of resource, which is not trivial. To meet the service quality and to minimize the cost paid to a cloud provider, choosing the right configuration for each VF is essential. However, selecting the best cloud configuration parameters is difficult due to different types of execution models, co-location of VFs on resources by the cloud provider, various input workloads, and diverse choice of configuration parameters. Existing solutions either do not fully address all the aforementioned challenges, or they assume complete knowledge of the underlying cloud infrastructure. Since this information is not available to the user, these techniques cannot be applied by the user for configuring resources on the cloud provider.

1.3 Thesis Contributions

The contributions of this thesis center around the resource allocation and resource configuration for application Virtual Functions. The contributions are as follows.

- Edge computing constitutes a central piece of the solution architecture wherein functional components of an application can be deployed over the edge network so as to reduce bandwidth demand over the core network while providing ultra-low latency communication to users. We investigate the joint optimal placement of virtual service chains consisting of virtual application functions (components) and the steering of traffic through them, over a multi-technology edge network model consisting of both wired and wireless mmWave links. This problem is $\mathcal{NP}$-hard. We provide a comprehensive "microscopic" binary integer program to model the system, along with a heuristic that is one order of magnitude faster than solving the corresponding binary integer program. Extensive evaluations demonstrate the benefits of managing virtual service chains (by distributing them over the edge network) compared to a baseline "middlebox" approach in terms of overall admissible virtual capacity. We observe significant gains when deploying mmWave links that complement the wired physical infrastructure.

- We present COSE, a framework that uses Bayesian Optimization to find the optimal configuration for serverless functions. Running application Virtual Functions (VFs) requires users to configure multiple parameters, such as memory, CPU, cloud provider, *etc*. While serverless is a relatively simpler computing model, configuring such parameters correctly while minimizing cost and meeting delay constraints is not trivial. To solve this problem, COSE uses statistical learning techniques to intelligently collect samples with the goal of predicting the cost and execution time of a serverless function across unseen configuration values. Our framework uses the predicted cost and execu-

tion time, to select the “best” configuration parameters for running a single or a chain of serverless functions (service chains), while satisfying customer objectives such as minimizing cost or satisfying delay constraints.

1.4 Roadmap of thesis

The rest of the thesis is organized as follows. In Chapter 2, we discuss work related to this thesis. In Chapter 3, we give the details of the application virtual function placement and routing problem, and present simulation results for both synthetic and real edge network topologies. In Chapter 4, we present COSE, an adaptive learning framework for finding the “best” configuration parameters for serverless functions. Chapter 5 concludes this thesis with future work.

Chapter 2

Related Work

2.1 Virtual Function Placement and Traffic Steering

Placement of functions deals with the efficient instantiation of VF instances on processing nodes to satisfy the demands of the system while minimizing the overall cost. Since different application flows can have different service chain requirements, a virtual service graph, with resource requirements, is created for each flow. The task of allocating resources for virtual service chains is similar to the Virtual Network Embedding (VNE) problem [37, 68].

In the VNE problem, a virtual network is embedded in a substrate network. The optimal resource allocation leads to customized end-to-end guaranteed service to the end-users. The VNE problem can be divided into two sub-problems. In *Virtual Node Mapping*, the virtual nodes are mapped on physical nodes, and in *Virtual Link Mapping*, the virtual links are mapped on physical links and connect virtual nodes. Solving the VNE problem is $\mathcal{NP}$-hard. Even with a given virtual node mapping, the problem of optimally mapping the virtual links onto the physical network links reduces to the unsplittable flow problem [53], which is also $\mathcal{NP}$-hard. Proposed solutions for solving the VNE problem can be divided into two types. The *One Stage VNE* problem solves both the Virtual Node Mapping and Virtual Link Mapping jointly [48, 57, 75]. In the *Two Stage VNE* problem, the Virtual Node Mapping is done at the initial stage, and then the Virtual Link Mapping is

performed at the second stage [31]. There is strong coordination between both stages, and the process is repeated until the desired mapping is achieved [84]. Since the VNE problem is $\mathcal{NP}$-hard, the time required to find an optimal solution for large problem sizes becomes intractable. Taking the complexity of VNE into account, there are two approaches to solve VNE. *Exact solutions* propose optimal techniques to solve small instances of the problem and to create a baseline case for testing *Heuristic solutions*. Exact solutions can be achieved by formulating the problem as an Integer Linear Program (ILP) and using algorithms, such as branch and bound, branch and cut, and branch and price [83], to optimally solve small instances of the problem in a reasonable time. *Heuristic solutions* attempt to find a reasonable solution, compromising optimality for faster execution time for the mapping.

In our work, we look at the virtual function placement and traffic steering for application virtual functions. Unlike the VNE problem, where a static network is embedded onto a substrate network, we embed Virtual Function service chains (graphs) onto the edge network. The embedding is done in an *online* fashion, where for each flow, we embed a VF service chain onto the substrate network. Based on the demand of the flow, there can be multiple instances of a virtual function on the edge network. Moreover, unlike the VNE problem, VF graphs have a directional flow, and the flow must pass through VFs in a predefined order. Based on the computation performed by the VFs, the bandwidth demand for flows changes as they traverse through the VF chain. In this thesis, we provide a *One Stage* solution to the optimal placement of virtual service chains and traffic steering through them, over a multi-technology edge network model consisting of both wired and wireless links. The *Exact solution* is provided by formulating the problem as an ILP. To solve the problem faster, we propose a *Heuristic solution* that is an order of magnitude faster than the *Exact solution* and gives results closer to the optimal solution.

2.2 Configuring Virtual Functions

Our work focuses on resource configuration for Serverless Functions. Although Serverless Functions have a simpler execution model, they are an ideal candidate for resource configuration since savings resulting from a proper cloud configuration are most significant for recurring jobs [22].

Serverless Measurement & Monitoring: As serverless computing is gaining popularity, a significant amount of research is focusing on measuring and monitoring different aspects of the serverless paradigms [13, 14]. Previous works [80, 59, 58, 49] perform a detailed study of different commercial serverless platforms and characterize their architecture, performance, and resource management strategies. [44] describes the important aspects of serverless platforms and addresses the challenges in designing these systems. [39] and [27] suggest extending the serverless paradigm to the edge of the network for IoT and Edge devices. Moreover, there have been several commercial products like dashbird [16], SignalFx [17], and Thundra [18] that help developers monitor their serverless applications in real-time. Unlike previous works which focus on monitoring and understanding the behavior of serverless functions and cloud provider serverless platforms, our work focuses on finding the "best" configuration parameters for a serverless function or a chain of serverless functions, regardless of the underlying details and complexities of the serverless platform used for running these functions.

Cloud Provider Resource Configuration: Commercial cloud providers have developed systems that suggest suitable configuration parameters for running tasks on the cloud provider's infrastructure. Google provides machine type recommendation system [15] that helps to maximize the resource utilization of user VM instances. The recommendations are based on the system metrics gathered by Google's monitoring system for the previous eight days. AWS provides auto-scaling service [12] to the users for EC2 instances. It

automatically adds/removes EC2 instances based on user-specified conditions for the load on the instances.

The cluster managing systems are provided by cloud providers where the user specifies the needs for the application, and cloud providers suggest the cluster configuration. Google's Borg [78] is a large-scale cluster manager, and schedules applications on large Google clusters. Brog offers the users a declarative job specification language and real-time job monitoring tools. Mesos [46] offers cluster resources to computing frameworks, such as Hadoop and MPI. It provides near-optimal data locality when sharing a cluster among diverse frameworks. Paragon [32] and Quasar [33] are online schedulers that use historical data of previously deployed applications to decide the cluster share for a new application.

Currently, cloud providers do not provide a resource configuration facility for serverless computing. Moreover, the aforementioned technique assumes that the user has complete knowledge of the underlying cloud infrastructure. Since the cloud providers don't share this information with the users, a user cannot apply these techniques for serverless computing.

Service Provider Resource Configuration: In these systems, a user infers the performance/cost of using given resources and picks the cloud provider configurations in the hope of minimizing the cost. CherryPick [24] uses Bayesian Optimization to predict suitable VM configuration for an application running on a cloud provider. CloudCmp [55] recommends a suitable cloud provider for running user application. It looks at the elastic computing, persistent storage, and networking services offered by cloud providers in choosing a suitable cloud provider. Both CherryPick and CloudCmp are offline tools that help users configure resources *before* the users deploy their applications. Ernest [76] builds the performance model of machine learning applications by running it on small data-sets.

It uses a statistical sampling technique to collect a small number of training data to build the model. Using this performance model it predicts the application behavior for large data-sets across different resources. WebPerf [50] estimates the latency model of an application running on a cloud provider using the causal dependencies of the application and the latency models of cloud APIs under hypothetical configurations. Elastisizer [45] suggests a suitable cluster size for big-data applications using job profiling, simulations, and estimation using black-box/white-box techniques. ARIA [77] builds the job profile and performance model for MapReduce and Hadoop applications. It uses the performance model to suggest resource configurations that satisfy delay constraints on the execution time for MapReduce jobs.

Unlike the vast body of prior work, which focus on VMs, Containers, or cluster configurations, our COSE framework targets serverless workloads. COSE works in scenarios where the users have limited to no visibility over the factors that affect the performance of the serverless function. Moreover, COSE runs online and adapts to the changes in the execution model of an application running as serverless.

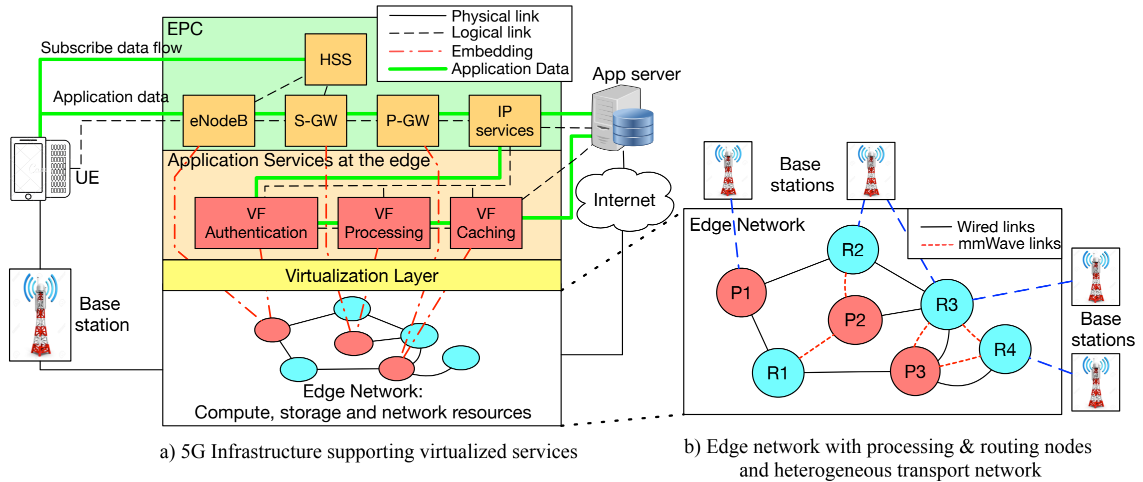
Chapter 3

Virtual Function Placement and Traffic Steering

3.1 Introduction

Next-generation mobile networks are expected to go beyond the delivery of static or streaming content, such as telephony, web browsing, and low-resolution video. They should be capable of serving many billions of users and smart devices at much higher data rates (over 500 Mbps) and ultra-low latencies (less than five milliseconds) [25, 67, 47]. Potential 5G applications include robots and drones, virtual and augmented reality, health-care, etc. Traditional network and application architectures can not support these stringent application requirements. Advances in the physical network infrastructure, *e.g.*, the integration of Gigabit Ethernet and millimeter wave (*mmWave*) technologies, and the virtualization of network and application functions are key to achieving these 5G goals [25, 67, 47].

The virtualization of network functions, termed Network Function Virtualization (NFV), aims to decouple network software from proprietary, dedicated hardware appliances, termed "middleboxes" (*e.g.*, traffic shapers, Network Address Translation boxes). Similarly, application virtualization allows an application to work in an isolated virtualized environment. Moreover, in cloud-based or service-oriented application architectures, an application can be composed of many application components, where each component can run as a Virtual Function (VF). Thus, under application service virtualization, multiple VFs can run



a) 5G Infrastructure supporting virtualized services

b) Edge network with processing & routing nodes and heterogeneous transport network

Figure 3.1: Function virtualization for 5G mobile network

on any general-purpose computer within a virtual machine, in an operating system container, or as a serverless "Function as a Service" (FaaS). The flexibility with which VFs can be deployed and managed — *i.e.*, chained, allocated resources, migrated — allows their hosting "close" to the users, in an edge cloud/datacenter, thus meeting the 5G application requirements of ultra-low latency and high throughput.

Figure 3.1a illustrates the evolution of cellular networks to 5G, where network services are moved from radio base stations and gateways into the edge cloud. In a traditional LTE architecture, user traffic traverses a series of devices on its way to the application server: the base station (eNodeB), a serving gateway (S-GW), and finally a packet data network gateway (P-GW) that connects to the outside world. On the other hand, in a virtualized environment, these network functions are envisioned to run virtualized, anywhere on the edge resources. They are chained together in a particular order based on processing requirements — in Figure 3.1a example, (eNode, S-GW, P-GW). To steer traffic across these VFs, Software Defined Networking (SDN) mechanisms are leveraged so that routes are established programmatically between components of the service chain.

Applications running on the edge network can also have different service chain requirements (*e.g.*, Authentication, Processing, Caching in Figure 3.1), and multiple application flows may need to use the same VF. Thus, understanding where to place VFs, or instances of the same VF, that are necessary to satisfy service chain requirements of different application flows, subject to physical resource (host and network) constraints, is a challenging problem. Furthermore, a 5G edge network may consist of multiple link technologies, *e.g.*, Ethernet and *mmWave*, that may have different characteristics suitable for possibly different types of application flows.

Our Contribution: In this chapter, leveraging optimization theory, we investigate the joint placement of virtual service chains consisting of virtual application functions (components) and the steering of traffic through them, over a multi-technology edge network model consisting of both wired and *mmWave* links. Our contributions are:

- We propose a detailed "microscopic" binary integer program (BIP) to find the optimal placement of virtual functions.
- BIP is $\mathcal{NP}$-hard (*i.e.*, computationally expensive), so we provide a heuristic that is one order of magnitude faster than BIP.
- Our workload model captures virtual service chains that correspond to the needs of future applications described as "killer applications" (*i.e.*, virtual and augmented reality) over the edge network.
- Extensive evaluation results demonstrate the benefits of managing virtual service chains (by distributing them over the edge network) compared to a baseline "middlebox" approach (where all functions are run on one host).
- We observe significant gains when deploying *mmWave* links that complement the wired physical infrastructure. Moreover, most of the gains are attributed to only 30% of these

mmWave links, which indicates that judicial placement of *mmWave* links is key for maximum gains.

- We show that the gains are highest in the high node density networks, where *mmWave* links can be easily established between the nodes.

- To the best of our knowledge, this is the first work to study a multi-technology based edge infrastructure envisioned for 5G networks. The developed model can be used by a 5G "service" provider to allocate resources to service chains optimally, and by a 5G "infrastructure" provider to understand the benefits of deploying *mmWave* links.

Chapter Organization: The chapter is organized as follows: Section 3.2 provides a background and reviews related work. Section 3.3 describes our system model. Section 3.4 explains our mathematical formulation. Section 3.5 presents our evaluation model, parameters and proposed heuristic. Results are shown in Section 3.6. Section 3.7 concludes the chapter.

3.2 Background and Related Work

This section provides a review of some of the most prominent research work on function placement and traffic steering, and the industry's direction to support high data rate and ultra-low latency applications on next-generation mobile networks (5G and beyond). According to "IMT-2020", a program developed by the International Telecommunication Union's Radiocommunication Sector (ITU-R) for 5G, the peak data rates are expected to be around 10 Gbits/s, while end-to-end latency is expected to be less than 5 ms [11]. To meet these strict requirements, there is a need for changes in the infrastructure (*e.g.*, using millimeter wave) and for having elasticity in hosting VFs at the edge of the network. Users accessing application servers hosted in the public network experience average de-

lays of 50-100ms, while such applications hosted in the operator's cloud experience delays ranging from 20-50ms. However, these delays are still significantly higher than those expected from a network that supports future applications. To meet the strict requirements of 5G network applications for delays of 1-5 ms, the edge computing paradigm that places computation closer to end-users is necessary [25, 67]. As an example, *Telefonica*, one of the world's largest telecom operator, is using their central offices (COs) as datacenters (COdc). These COdc are closer to the end-users (at the network edge) and are capable of hosting user VFs [62].

Figure 3.1a shows the case where service-chain components are running as virtualized functions at the edge of the network. Here, all the traffic from users passes through Authentication, Processing, and Caching, which are running at the edge of the network, before arriving at the Application Server. Note that the operator's network services (*e.g.*, S-GW and P-GW), which are part of the Evolved Packet Core (EPC), can also be virtualized and hosted in the edge datacenter, as shown in Figure 3.1a. However, in this work, we are specifically studying virtual functions for applications running on the 5G network. *The internal functional split of the 5G RAN, and virtual EPCs is beyond the scope of this work.*

Figure 3.1b shows an example of an edge network, consisting of processing nodes (P1-P3) and routing/switching nodes (R1-R4). This edge network covers a small geographical area, *e.g.*, a medium-size city. As the name suggests, processing nodes have the processing power and can host VFs, while routing nodes are responsible for routing traffic through the network. Note that a processing node can also act as a routing node. All the nodes are SDN enabled and can be programmed for traffic routing. The nodes are connected with two different link technologies, namely *Wire* and millimeter wave (*mmWave*) links. The *mmWave* technology is considered an important aspect of 5G networks. The enormous

amount of spectrum available in the *mmWave* band, and the ease and flexibility of deploying *mmWave* infrastructure, will greatly increase the network capacity, as well as decrease latency when *mmWave* links are used to create shortcuts between nodes [63].

Application service components are hosted at processing nodes. These components run as VFs and are dynamically instantiated, migrated, or removed from the network based on the system requirements. Applications have strict requirements for their traffic to traverse virtualized services in a certain order, *e.g.*, authentication followed by caching. This is known as "Service Function Chaining" (SFC). SFC is an important capability of virtualized networks as it provides both modularity and elasticity. A single function in a service chain is dynamically changed/updated without having any impact on other functions. The efficient placement of virtualized functions and traffic steering through service chains are challenging problems.

3.2.1 Placement

Placement of functions deals with the efficient instantiation of virtual function (VF) instances on processing nodes to satisfy the demands of the system while minimizing the overall cost. Since different application flows can have different service chain requirements, a virtual service graph, with resource requirements, is created for each flow. This graph is embedded on a virtualized physical infrastructure, as shown in Figure 3.1a. The task of creating and deploying virtual service chains is similar to the Virtual Network Embedding (VNE) problem [37, 68]. Similar to VNE, this task is $\mathcal{NP}$-hard. Different VF placement schemes have been proposed [42, 61]. Formulated as an optimization problem, VF placement and chaining reduces to an integer program, which is $\mathcal{NP}$-hard and intractable for larger inputs. Hence, most solutions focus on designing heuristic or meta-heuristic algorithms for solving the VF placement with service chaining [42, 72]. These

solutions aim to find a sub-optimal placement quickly and are based on simple cost functions and constraints. In this work, *we aim to find an optimal placement based on a detailed system model that captures many complexities that arise with virtualized services for a 5G network, including multi-technology links and detailed service demands. Moreover, we provide a heuristic solution to quickly solve the problem while sacrificing little on the quality of the results.*

3.2.2 Traffic Steering

Traffic steering through VF instances residing at different locations brings a different set of challenges. Traditionally, traffic is directed through a desired sequence of network functions (middleboxes) using manual configurations, which cannot be imported to the VF paradigm. Since resources are dynamically allocated, there is a need for autonomic traffic steering. SDN offers a flexible control approach and enables traffic forwarding. However, SDN capabilities have been limited to L2/L3 forwarding functions and cannot support VFs. SDN based solutions have been proposed [36, 66, 85], which extend the current L2/L3 functions of SDN to provide a policy enforcement layer for VF traffic steering. Although extended SDN mechanisms have enabled VF traffic steering, finding the best path through a set of VFs under multiple constraints is $\mathcal{NP}$-complete [65]. Previous work focuses on finding paths given the cost function of a single link technology [42, 21]. In this work, *we consider multiple link technologies, each has its own cost definition. Furthermore, the link cost function takes into account multiple cost metrics to accurately model the link technologies.*

3.3 System Model

This section describes our envisioned 5G system model for edge computing. We also describe our use cases (augmented and virtual reality applications) which have stringent processing and communication requirements that "thin" clients/mobile devices and traditional networks fail to support.

Our model of the 5G infrastructure consists of a multi-technology edge network, where nodes are connected with wireless *mmWave* and wired links, as shown in Figure 3.1b. Nodes that are closer than a threshold distance are connected with *mmWave* links. There are two types of nodes in the network. Routing Nodes (RN) are OpenFlow enabled routers that forward packets to the next hop toward their destination. Processing Nodes (PN) are RNs with processing power, so a PN can also host Virtual Functions (VFs). A PN has multiple processing cores. For simplicity, we assume that a single core can only host a single VF instance.

There are costs associated with using the network. There is a fixed cost of running a VF instance on a PN. There are two different types of cost associated with using a communication link, namely, fixed cost and usage cost. A fixed cost is incurred if the link is being used, regardless of the amount of traffic flowing through the link. A usage cost is based on the cost per unit of traffic flowing through the link.

Each flow in the network has a source node, destination node, capacity demand, delay demand, and service chain. We model the long-term average rate requirement for an application. The capacity demand is the bit rate that a flow needs on each link as it goes from its source to destination. The delay demand is the maximum delay that packets of the flow can experience as they move from the source to destination. A service chain, as we discussed earlier, is an ordered list of VFs that the flow should pass through before reaching the destination node. This is shown in Figure 3.1a where an application flow

passes through VFs running *Authentication*, *Processing*, and *Caching* before reaching the destination application server.

Online vs Offline

The resource allocation problem consists of placement of VFs and traffic steering, and it can be done either *online* or *offline*. In the *online* case, the resources are dynamically allocated for each flow as the flow arrives to the system. In the *offline* case, all the flow demands are known in advance and the resources are simultaneously allocated for all flows. Both the *online* and *offline* cases are $\mathcal{NP}$-hard [71]. The *offline* resource provisioning case is not always possible, especially when users' behavior cannot be accurately predicted. In this work, we only consider the *online* case. In the next section, we provide a detailed Binary Integer Programming (BIP) formulation for this problem, which can be used for both *online* and *offline* cases. Note that the *online* case is merely the *offline* case with a single flow.

To evaluate our system, we model the workload of service chains inspired by applications such as augmented and virtual reality applications. VR/AR applications requires high throughput and ultra-low latency. It is believed that VR applications, where users interact with other users, would need bandwidth up to 500Mbps and latency less than 5ms [25]. The challenge in advancing and deploying such applications is that traditional architectures (using remote clouds/datacenters) fail to satisfy such stringent requirements. To overcome this challenge, the VR application should be refactored as a chain of VFs that get deployed at the edge cloud. For instance, the 3D distributed game described in [52] may be decomposed into a chain of VFs as illustrated in Figure 3.2. The aim is to move most computation from Application Servers to the 5G edge network, to reduce latency and increase throughput. As shown in Figure 3.2, when a user's request arrives, it first

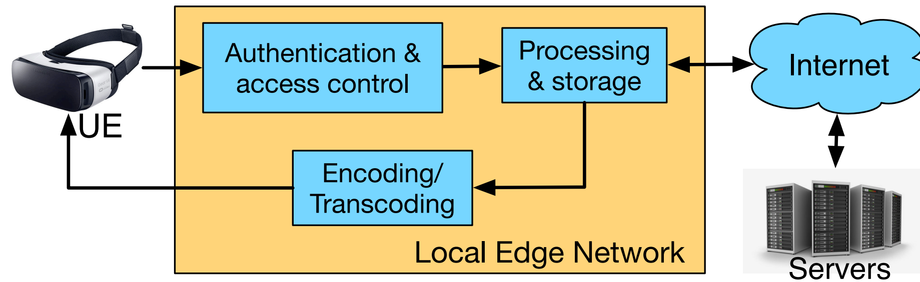


Figure 3.2: Virtual Reality use case

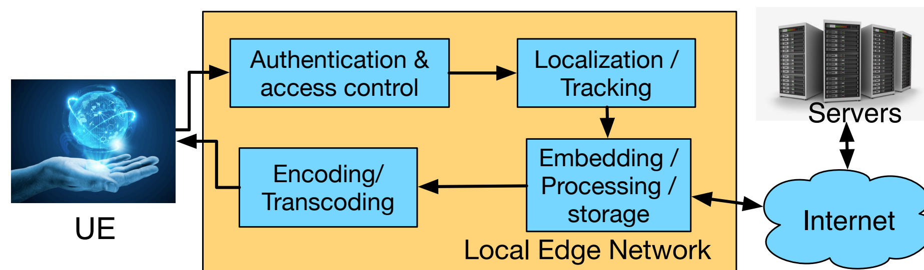


Figure 3.3: Augmented Reality use case

goes through the *Authentication and Access Control* VF to identify the user and check if the user is allowed to make the request. The request then moves to the *Processing and Storage* VF where the request is processed and actions are taken. These actions are also propagated to application servers over the Internet to update the global state of the game. This VF also has storage capability so it can provide caching and deliver data directly to the user. The delivered data finally moves through the *Encoding/Transcoding* VF, where data is encoded/transcoded before being sent to the user. A similar case for AR is shown in Figure 3.3.

3.4 Mathematical Model

In this section, we present the Binary Integer Programming (BIP) formulation for the joint placement of virtualized services (VFs) and traffic steering across the service chains. Although our formulation targets our envisioned 5G system model described in Section 3.3, it can be applied to other scenarios by making appropriate changes to cost functions or constraints. Our model can be used by a 5G "service" provider to optimally allocate resources to service chains, as we describe in this section. Specifically, we aim to minimize the operational (OPEX) cost by maximizing the resource utilization of the physical infrastructure while satisfying the network delay and bandwidth requirements of the application virtual functions. Note that since we are interested in modeling the network requirements of an application, our constraints are on network parameters. For machine-specific application requirements, such as the amount of memory, or the number of GPUs, additional constraints can be added to the model. All network parameters are described in Table 3.1. (Later in Section 3.5, we use this model, in conjunction with a network graph generation model, to also understand the benefits of deploying *mmWave* links from the point of view of a 5G "infrastructure" provider.)

In our model, a physical (or logical) network $G(V,E)$ is made up of nodes V, and links E between the nodes. Each link has capacity $c(u,v)$ and latency $l(u,v)$. The fixed cost of using a link is given by $k^c_{(u,v)}$, and it captures the cost incurred if the link is used by any flow. The usage cost of a link is given by $k^d_{(u,v)}$, which represents the cost per unit of flow that passes through the link. The cost of starting a new virtualized function instance on a Processing Node (PN) is given by h^n_i. Each PN has a set of cores available O_v, and each virtual function runs on a single core. Each VF has a load limit U_s (in Mbps) on the total bit rate that is served by the VF instance. Each PN can host certain types of VF n, as indicated by M^n_i. The volume of the incoming flow and outgoing flow through a VF changes based

Notation	Description
$G(V, E)$	Network graph, V is the set of nodes: Routing Nodes (RNs) and Processing Nodes (PNs), and E is the set of all links (u, v).
$w_{(u,v)}$	binary {0,1}: 1 if there exists a physical link between nodes u and v, 0 otherwise.
$c(u, v)$	Capacity of link (u, v).
$l(u, v)$	Latency of link (u, v).
$k^c_{(u,v)}$	Fixed cost of using link (u, v). If any amount of traffic, greater than zero, passes through link (u, v), we incur this cost.
$k^d_{(u,v)}$	Usage cost of using link (u, v). It is the cost of unit flow that passes through link (u, v).
h^n_i	Fixed cost of instantiating a VF instance of type n on node $i \in V$.
O_v	Set of cores available at node $v \in V$. Each core can support one VF.
U_s	Load (in Mbps) that can be served by a single VF $s \in S$.
M^n_i	binary {0,1}: 1 if VF $n \in S$ can be supported at node i, 0 otherwise.
ϕ_s	Ratio of outgoing to incoming flow rate through VF $s \in S$.

Table 3.1: Network Parameters.

on the computation performed by the VF, *e.g.*, an encryption VF encrypts incoming traffic, so the amount of outgoing traffic leaving the VF is more than the incoming traffic. The ratio of the outgoing bit rate (in Mbps) over the incoming bit rate (in Mbps) for a VF is given by ϕ_s.

Table 3.2 shows the traffic parameters. Each flow f in the network has a start node s^f, destination node t^f, initial capacity demand (in Mbps) d^f, latency demand (in milliseconds) l^f, and a service chain C^f. A flow is unsatisfied if any of its constraints are not met. As the flow traverses through the VFs in its service chain, its capacity demand changes based on the VF's ϕ_i. The capacity demand of a flow between two VFs is given by $d^{f(m \to n)}$.

3.4.1 Variables

Table 3.3 describes our model variables in detail. This includes decision variables, and derived variables (*i.e.*, variables dependent on decision variables).

Notation	Description
F	Set of all flows in the network.
s^f	Start node of flow $f \in F$.
t^f	Destination node of flow $f \in F$.
d^f	Initial capacity demand of flow $f \in F$.
l^f	Latency demand of flow $f \in F$. Maximum delay that a flow $f \in F$ can tolerate on the path from source to destination.
K	Set of all different VFs that can be placed on nodes.
C^f	Service chain of flow $f \in F$. Set of VFs that flow $f \in F$ needs to traverse in a specific order, *i.e.* $n_1 \rightarrow n_2 \rightarrow ... \rightarrow n_l$, where $n_i \in K$.
C^f_{st}	$C^f_{st} = [n_{s^f} \rightarrow C^f \rightarrow n_{t^f}]$. The service chain of flow $f \in F$ which includes s^f and t^f nodes. To ensure that the flow starts at node s^f and ends at node t^f, two imaginary VFs n_{s^f} and n_{t^f} are introduced at s^f and t^f nodes, respectively. Since VFs n_{s^f} and n_{t^f} are only present at s^f and t^f nodes, these nodes are selected as the start and end nodes on the flow's path.
$d^{f(m \rightarrow n)}$	Capacity demand of flow $f \in F$ from VF m to n. $$d^{f(m \rightarrow n)} = d^f \prod_{i=s^f}^{m} \phi_i, \quad (\text{note}: \ \phi_{s^f} = 1)$$

Table 3.2: Traffic Parameters.

3.4.2 BIP Formulation

3.4.2.1 Objective Function

Our objective is to find the optimal placement of VFs that minimizes the resource fragmentation in the system, *i.e.*, maximizes the utilization of resources, while satisfying the delay and bandwidth requirements of the applications. Since physical resources in the network are usually leased or rented from third parties, we aim to maximize the utilization of resources that are already in use as long as we can satisfy the flow demands. Following are the costs that we consider, and we aim to minimize.

<table>
<tr><th>Variables</th><th>Description</th></tr>
<tr><td>$x^{f(m \to n)}_{(u,v)}$</td><td>binary {0,1}: 1 if link (u, v) is used to reach from VF m to n in the service chain C^f of flow $f \in F$, and 0 otherwise.</td></tr>
<tr><td>$x_{(u,v)}$</td><td>binary {0,1}: 1 if any flow uses link (u, v), and 0 otherwise. Note that it is not a decision variable, as it can be derived from $x^{f(m \to n)}_{(u,v)}$.
$x_{(u,v)} = 1$ if
$$\sum_{f \in F} \sum_{(m \to n) \in C^f_{st}} x^{f(m \to n)}_{(u,v)} + \sum_{f \in F} \sum_{(m \to n) \in C^f_{st}} x^{f(m \to n)}_{(v,u)} > 0 \;\; \forall (u, v) \in E \tag{3.1}$$
and 0 otherwise.
Equation (3.1) above can also be written as a set of linear constraints as shown below.
$$x_{(u,v)} \leq \sum_{f \in F} \sum_{(m \to n) \in C^f_{st}} x^{f(m \to n)}_{(u,v)} + \sum_{f \in F} \sum_{(m \to n) \in C^f_{st}} x^{f(m \to n)}_{(v,u)} \;\; \forall (u, v) \in E$$
$$x_{(u,v)} \geq x^{f(m \to n)}_{(u,v)} \;\; \forall f \in F, \; \forall (m \to n) \in C^f_{st}, \; \forall (u, v) \in E$$
$$x_{(u,v)} \geq x^{f(m \to n)}_{(v,u)} \;\; \forall f \in F, \; \forall (m \to n) \in C^f_{st}, \; \forall (u, v) \in E$$
Since $x_{(u,v)}$ is symmetrical, we also want to enforce that $x_{(u,v)} = x_{(v,u)}$</td></tr>
<tr><td>S^{fn}_{ia}</td><td>binary {0,1}: 1 if VF $n \in C^f_{st}$ is placed at core a of node i for flow $f \in F$, and 0 otherwise.</td></tr>
<tr><td>X^n_{ia}</td><td>binary {0,1}: 1 if any VF $n \in K$ is placed on core a of node i, 0 otherwise. Note that it is not a decision variable as it can be derived from S^{fn}_{ia}.
$X^n_{ia} = 1$ if
$$\sum_{f \in F} S^{fn}_{ia} \geq 1 \quad \forall n \in C^f, \forall i \in V, \forall a \in O_i \tag{3.2}$$
and 0 otherwise.
Equation (3.2) above can also be written as a set of linear constraints as shown below.
$$X^n_{ia} \leq \sum_{f \in F} S^{fn}_{ia} \;\; \forall n \in C^f, \forall i \in V, \forall a \in O_i$$
$$X^n_{ia} \geq S^{fn}_{ia} \;\; \forall n \in C^f, \forall i \in V, \forall a \in O_i, \forall f \in F$$</td></tr>
</table>

Table 3.3: Variables.

VF Deployment Cost

To run a VF on a node, we assume a pricing/cost model that is similar to Amazon EC2 "dedicated host", in which a fixed cost is paid for leasing/renting the node on which the VF instance is run.

$$\mathbb{V}_c = \sum_{i \in V} \sum_{n \in K} \sum_{a \in O_i} h_i^n X_{ia}^n \tag{3.3}$$

Link Fixed Cost

If a link is used (in any direction) by any of the flows, regardless of the flow demand, we pay a fixed cost. Different link technologies (namely, *Wire* and *mmWave* links) can have different fixed costs, which we explain in detail later in Section 3.5.

$$\mathbb{E}_c = \sum_{(u,v) \in E} k_{(u,v)}^c x_{(u,v)} \quad \forall (u,v) \in E, \quad u > v \tag{3.4}$$

Link Usage Cost

This link usage cost is based on the amount of link resources used by flows. It represents the cost per unit of flow going through a link.

$$\mathbb{E}_d = \sum_{(u,v) \in E} k_{(u,v)}^d \sum_{f \in F} \sum_{(m \to n) \in C_{st}^f} x_{(u,v)}^{f(m \to n)} d^{f(m \to n)} \tag{3.5}$$

Our objective is to minimize the cost of the system and fragmentation of the resources in the system, while satisfying the flow demands. The objective function is given by:

$$minimize(\ \mathbb{V}_c + \mathbb{E}_c + \mathbb{E}_d \ )$$

This cost minimization is subject to the following constraints:

3.4.2.2 Link Capacity Constraint

$$\sum_{f \in F} \sum_{(m \to n) \in C_{st}^{f}} d^{f(m \to n)} x_{(u,v)}^{f(m \to n)} \leq c(u,v) \qquad \forall (u,v) \in E \tag{3.6}$$

Each link has a capacity limit. Flows passing through a link should not exceed the capacity of the link.

3.4.2.3 Flow Latency Constraint

$$\sum_{(m \to n) \in C_{st}^{f}} \sum_{(u,v) \in E} l(u,v) x_{(u,v)}^{f(m \to n)} \leq l^{f} \qquad \forall f \in F \tag{3.7}$$

Each flow has a latency constraint. A flow, moving from source to destination, should not experience latency greater than its (end-to-end) latency requirement. Here we are only considering network delays, *i.e.,* propagation and transmission delays.

3.4.2.4 Physical Link Constraint

$$x_{(u,v)}^{f(m \to n)} \leq w_{(u,v)} \qquad (m \to n) \in C_{st}^{f} \tag{3.8}$$

A virtual link along the path of a flow should be using one of the existing physical links given by $w(u,v)$.

3.4.2.5 Flow Constraint

$$\sum_{j \in V} x_{(i,j)}^{f(m \to n)} - \sum_{k \in V} x_{(k,i)}^{f(m \to n)} = \sum_{a \in O_i} S_{ia}^{fm} - \sum_{a \in O_i} S_{ia}^{fn} \tag{3.9}$$

$\forall i \in V$, $(m \to n) \in C_{st}^{f}$, where VF n is after VF m in the service chain C_{st}^{f}.

This constraint ensures that there is a single continuous path between pair of nodes on which VFs m and n are placed.

3.4.2.6 VF Placement Constraint

$$S_{ia}^{fn} \leq M_i^n \quad \forall f \in F, \forall n \in C_{st}^f, \forall i \in V, \forall a \in O_i \tag{3.10}$$

VF $n \in C_{st}^f$ can only be hosted on nodes that can host VF n.

3.4.2.7 Single VF Node Selection Constraint

$$\sum_{i \in V} \sum_{a \in O_i} S_{ia}^{fn} = 1 \qquad \forall n \in C_{st}^f \tag{3.11}$$

Only a single node is selected to host a VF in the service chain C_{st}^f of flow $f \in F$.

3.4.2.8 Node Capacity Constraint

$$\sum_{n \in K} \sum_{a \in O_i} X_{ia}^n \leq |O_i| \quad \forall i \in V \tag{3.12}$$

Each free core at a node can host a single VF. The number of VFs hosted at a node is limited by the number of cores available at that node.

3.4.2.9 VF Capacity Constraint

$$\sum_{f \in F} d^{f(m \to n)} S_{ia}^{fn} \leq U_n \quad \forall i \in V,\ \forall a \in O_i,\ \forall n \in C^f \tag{3.13}$$

Each VF at a node has a capacity limit and can only serve flow demands (in Mbps) within that limit.

3.4.2.10 Single VF per core

$$\sum_{n \in C^f} S_{ia}^{fn} <= 1 \qquad \forall f \in F \ \ \forall i \in V \ \ \forall a \in O_i \tag{3.14}$$

Each core at a node can host at most one VF.

3.5 Evaluation Model, Parameters and Proposed Heuristic

In this section, we present our evaluation model and parameters for both the edge network and the workload of VR and AR service chains. We then provide a description of our proposed heuristic.

3.5.1 Edge Network Graphs

We used different types of edge network topologies for the simulation. We generated synthetic graphs using BRITE [60], a widely used network graph generator. Moreover, we used real edge network topologies for two cities, Santa Monica (CA, USA) and Palo Alto (CA, USA). Next, we explain the topologies generated using each technique.

3.5.1.1 Synthetic Edge Network Topology using BRITE

BRITE is a widely used network graph generator [60]. BRITE supports multiple graph generation models, including models for flat and hierarchical graphs. BRITE separates the placement of the nodes from the process of growing the graph and interconnecting the nodes. We use BRITE's *random node placement* model for placing nodes in a plane, and BRITE's *Waxman* model for interconnecting the nodes probabilistically [81]. BRITE network has 25 nodes and node density of 6.25 $nodes/km^2$.

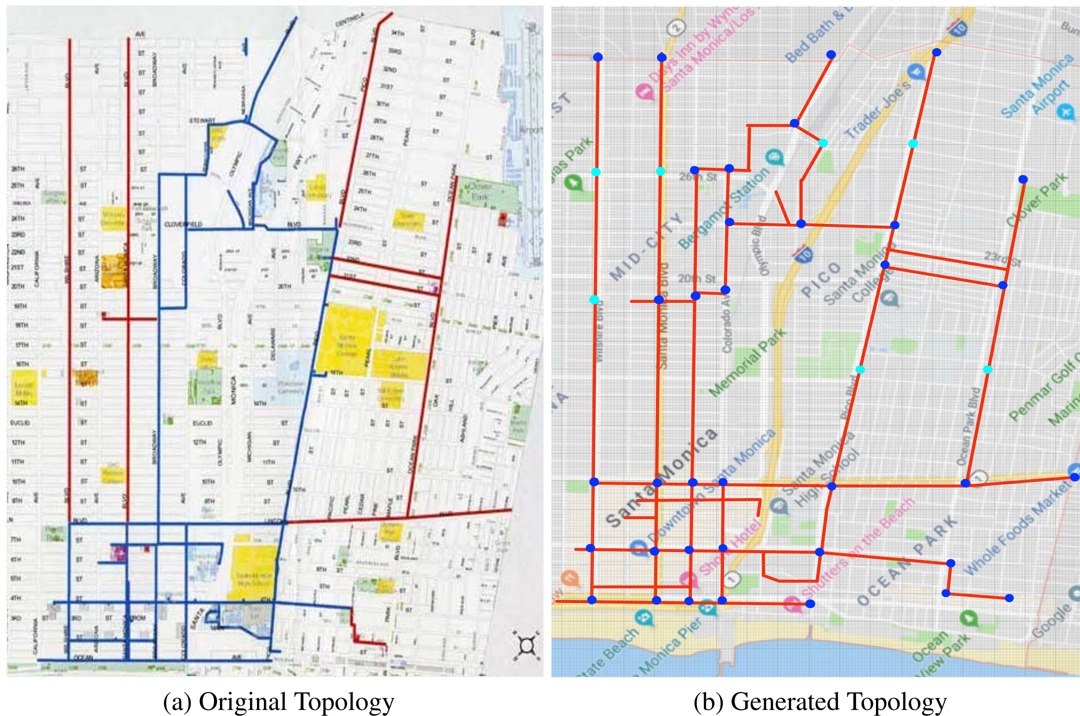
3.5.1.2 Real Edge Network Topology

We used real city network topologies to generate edge networks. We used the optical fiber network topology of two cities in the USA. The original wired network topologies for Santa Monica and Palo Alto are shown in Figures 3.4a and 3.5a, respectively. We mapped the original topologies onto the Google Maps [40], as shown in Figures 3.4b and 3.5b. Mapping a topology on Google Maps helped us find coordinates of different points on the

maps. The original maps do not show the router nodes. We assumed there exists a router node at the intersection of the wires and the end of each wire. Moreover, we introduced some additional router nodes within long-distance links for the Santa Monica topology, marked by cyan color in Figure 3.4b.

Santa Monica:

We used the wired network topology for Santa Monica, CA, USA. As shown in Figure 3.4, Santa Monica's wired network topology is a distributed bus topology [43], with few connections between backbone bus lines. Distributed bus topology is a widely used edge network topology for city networks. Santa Monica wired network has 43 nodes and a node density of 3.61 $nodes/km^2$.



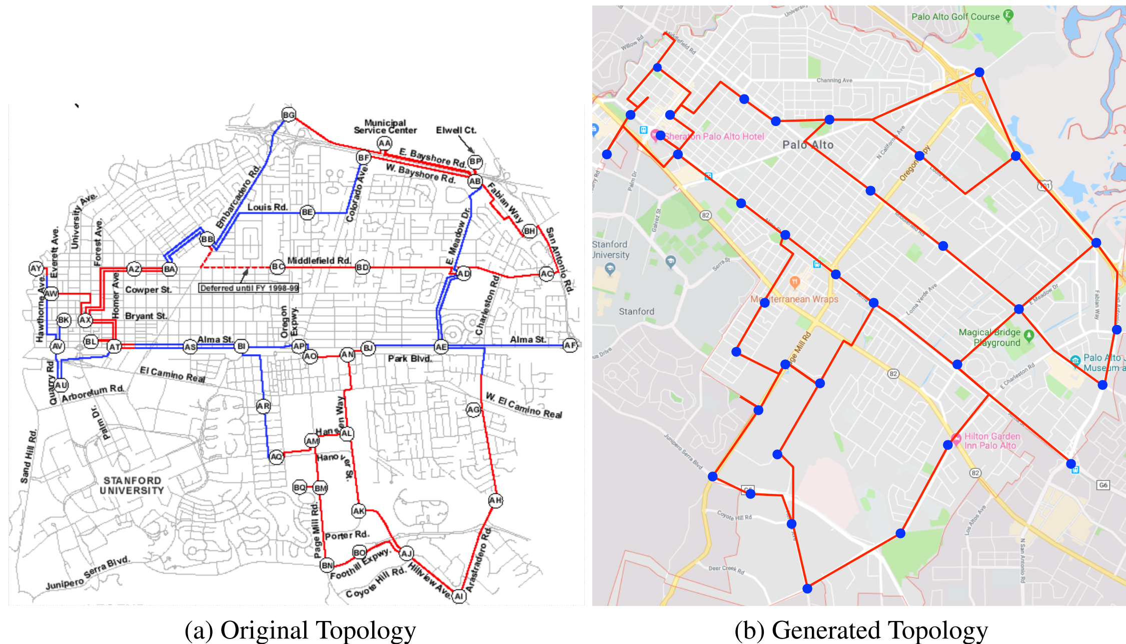
(a) Original Topology (b) Generated Topology

Figure 3.4: Santa Monica network topology

Palo Alto:

We used the wired network topology for Palo Alto, CA, USA. As shown in Figure 3.5,

Palo Alto's wired network topology is a ring topology with few connections across the diameter of the ring network. Ring topology is a widely used edge network topology for city networks. Palo Alto wired network has 36 nodes and a node density of 0.91 $nodes/km^2$.



(a) Original Topology

(b) Generated Topology

Figure 3.5: Palo Alto network topology

3.5.1.3 Adding *mmWave* links

The initial topologies that we generate represents a base edge network that consists of only wired links. We then augment this base graph with *mmWave* links to obtain three different types of graph, which are described next.

Wire: This is the initial graph generated with only wired links. An example of such graph is shown in Figure 3.6a.

Single: *mmWave* links are added to the *Wire* graph if the distance between any two nodes in the graph is less than a given distance/*mmWave*-range (elaborated on below). However,

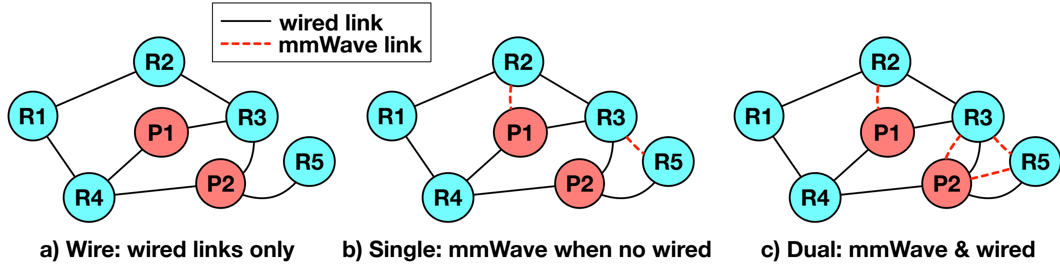


Figure 3.6: Multi-technology edge network consisting of processing and routing nodes.

if there is already a wired link between the two nodes, a *mmWave* link is not added. So we have only a *single* type of link technology (*mmWave* or *Wire*) between any two nodes, as shown in Figure 3.6b.

Dual: *mmWave* links are added to the *Wire* graph if the distance between any two nodes in the graph is less than a given distance/*mmWave*-range (elaborated on below). In this scenario, two nodes may have *dual* technology links, *i.e.*, both *mmWave* and *Wire* links, as shown in Figure 3.6c. *Dual* has the maximum number of possible *mmWave* links between nodes in the network.

Characteristics of different graphs for our evaluation, in terms of nodes, area and links, are summarized in Table 3.4. Graphs generated using BRITE covers a small area of 4.0 km^2 and has the highest node density of 6.25 $nodes/km^2$. Santa Monica has area of 11.90 km^2 and a node density of 3.61 $nodes/km^2$ and Palo Alto has area of 39.65 km^2 and the lowest node density of 0.91 $nodes/km^2$. Because of the differences in the node density, BRITE has the highest percentage of *mmWave* links. Santa Monica has the second highest percentage of *mmWave* links and Palo Alto has the lowest percentage of *mmWave* links.

Table 3.5 shows the various parameters used in our evaluation campaign. The range of a *mmWave* link is defined by variable $range_{mm}$. Two nodes in the network cannot have a *mmWave* link if their distance is beyond $range_{mm}$. $range_{mm}$ is chosen to be 500m, which can be achieved in urban environments with LOS [26]. The capacity of *mmWave* links can

BRITE					
Type	**# Nodes**	**Area**	**Technology**	**avg. # of links**	**%age of links**
Dual	25	4.0 km^2	*mmWave*	47.4	65.5
			Wire	25	34.5
Single	25	4.0 km^2	*mmWave*	35.6	58.8
			Wire	25	41.2
Wire	25	4.0 km^2	*mmWave*	0	0.0
			Wire	25	100
Santa Monica					
Type	**# Nodes**	**Area**	**Technology**	**avg. # of links**	**%age of links**
Dual	43	11.90 km^2	*mmWave*	90	62.50
			Wire	54	37.50
Single	43	11.90 km^2	*mmWave*	65	54.62
			Wire	54	45.38
Wire	43	11.90 km^2	*mmWave*	0	0.0
			Wire	54	100
Palo Alto					
Type	**# Nodes**	**Area**	**Technology**	**avg. # of links**	**%age of links**
Dual	36	39.65 km^2	*mmWave*	44	50.57
			Wire	43	49.43
Single	36	39.65 km^2	*mmWave*	18	29.51
			Wire	43	70.49
Wire	36	39.65 km^2	*mmWave*	0	0.0
			Wire	43	100

Table 3.4: Graph Parameters and Characteristics

vary in the 1 Gbps–10 Gbps range, based on channel conditions [82]. We have taken the link capacity $c(u,v)_{mm}$ to be 2 Gbps for *mmWave* links [82], and $c(u,v)_w$ to be 10 Gbps for *Wire* links.

The fixed cost for using a *mmWave* link, $k_{(u,v)}^{c_{mm}}$, is kept low by setting it to 1, since it is less costly to establish *mmWave* links between two sites if they are within the range $range_{mm}$. On the other hand, the fixed cost for *Wire* links is higher, and so we set it to 50, since *Wire* links are usually leased / rented from an infrastructure provider.

The usage cost for *mmWave* links, $k_{(u,v)}^{d_{mm}}$, is dependent on link performance and is set

Parameter	Description	Value
$range_{mm}$	*mmWave* range	500 m
$c(u,v)_{mm}$	Capacity of *mmWave* links	2 Gbps
$c(u,v)_{w}$	Capacity of *Wire* links	10 Gbps
$k^{c_{mm}}_{(u,v)}$	Fixed cost for using *mmWave* link	1
$k^{c_{w}}_{(u,v)}$	Fixed cost for using *Wire* link	50
$k^{d_{mm}}_{(u,v)}$	Cost per unit flow for using *mmWave* link	$1/PS$
$k^{d_{w}}_{(u,v)}$	Cost per unit flow for using *Wire* link	1
$l_{(u,v)}$	Latency of link (u, v) is the sum of propagation and transmission delays	-
h^n_i	Fixed cost of instantiating a VF instance of type n on node i	200
$\lvert O_v \rvert$	Number of cores available at processing node v	10
U_s	Capacity of the VF s	15 Gbps
$ratio_{PN}$	Ratio of processing nodes to routing nodes	0.3

Table 3.5: Evaluation Parameters

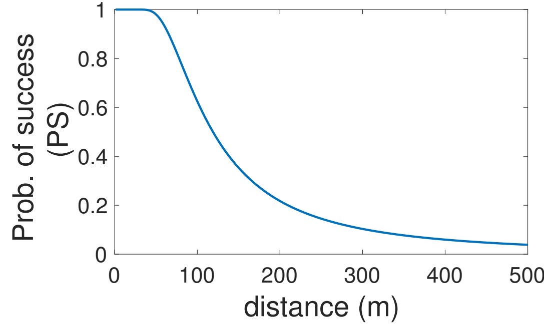


Figure 3.7: Probability of successful bit delivery over a *mmWave* link

to $1/PS$, where PS is the probability that a bit sent over the link successfully reaches the other side. PS is obtained using the empirical studies on *mmWave* technology described in [69, 73]. Figure 3.7 shows PS as a function of distance. Note that the usage cost $k^{d_{mm}}_{(u,v)}$ becomes significantly higher as the distance between the two nodes connected via a *mmWave* link increases. Hence, shorter *mmWave* links are favored over longer *mmWave* links. For *Wire* links, the usage cost $k^{d_w}_{(u,v)} = 1$, since the cost (delivery performance penalty) associated with using *Wire* is relatively much lower. The latency of a link is given by $l(u, v)$,

and is equal to the sum of propagation and transmission delays. Note that there will be zero or negligible queuing delays when demands match allocated capacities.

We select a fraction of the nodes in the network graph to be processing nodes (PNs). This ratio, denoted by $ratio_{PN}$, is set to 0.3, *i.e.* only 30% of the nodes are PNs. Each PN node has $|O_v|$ cores available, and we set $|O_v| = 4$. This means that each PN can host at most 4 VFs. The capacity of a single VF U_s is set to 15 Gbps. The cost associated with instantiating a VF h_i^n is set to 200. It represents the cost of leasing a virtual machine or container from the edge datacenter. A high value has the effect of packing as many flows as possible on a VF as long as the flow demands can still be fulfilled. Next, we explain the process of selecting processing nodes.

3.5.1.4 Processing Node Selection

We assume that any node in the network can be chosen as a processing node. $ratio_{PN}$ fraction of nodes are selected as processing nodes. Processing nodes are selected such that the sum of the distances from each node to the closest processing node is minimized.

We formulate this problem as Integer Linear Program (ILP). The distance from the node i to processing node (PN) p is denoted by c_{pi}. X is the total number of processing nodes. Note that the nodes and the processing nodes share the same set of points. We define the following variables:

$z_{pi} = 1$ if node i is satisfied by PN p, 0 otherwise

$x_p = 1$ if PN p is being used, 0 otherwise

We formulate the problem as integer-optimization model.

$$minimize \sum_{p \in N} \sum_{i \in N} c_{pi} z_{pi}$$

Subject to:

1. The total number of processing nodes are equal to X

$$\sum_{p \in N} x_p = X$$

2. A single processing node p is selected for each node i

$$\sum_{p \in N} z_{pi} = 1 \quad \forall i \in N$$

3. $x_p = 1$ if node p is selected as PN

$$z_{pi} \leq x_p \quad \forall p \in N \quad \forall i \in N$$

$$\sum_{p \in N} z_{pi} \geq x_p \quad \forall x \in N$$

We used CPLEX solver[1] to solve the ILP above for the selection of processing nodes for Santa Monica and Palo Alto network graphs. Since the graphs generated using BRITE are synthetic, we generated multiple graph topologies and randomly selected processing nodes.

3.5.2 Input Flow Parameters

There are two different types of flow in the network, each type has different service chain requirements representing either Virtual Reality (VR) or Augmented Reality (AR). For each of the generated network graphs, we generate five sets of flows, where each incoming flow is either VR or AR flow with probability 0.5. Each flow starts and ends at the same node (representing the user/client), which is randomly selected. We only consider the allocation of the service chains on the edge network. Flow parameters for VR and AR

[1] IBM ILOG CPLEX Optimizer,
http://www-01.ibm.com/software/integration/optimization/cplex-optimizer

flows are described in Table 3.6.

For VR flows, the initial VR flow demand d^f_{VR} is normally distributed with mean 10 Mbps and standard deviation 2 Mbps. The flow passes through three different VFs, as shown in Figure 3.2, with each VF s having a different value for ϕ_s (ratio of outgoing to incoming flow rate). The values that we considered for ϕ_s for different VR VFs are shown in Table 3.6. As an example, a VR flow generates requests at 10 Mbps. These requests pass through the *Authentication & Access control* VF and yield 9 Mbps of admissible requests. These admissible requests pass through the *Processing & Storage* VF yielding data at the rate of 180 Mbps, which after the *Encoding / Transcoding* VF becomes 144 Mbps. Similarly, AR flows passes through four different VFs, as shown in Figure 3.3, and AR flow parameters are described in Table 3.6.

3.5.3 Proposed Heuristic

We used the CPLEX solver to solve the BIP that we described in Section 3.4.2. The running time for obtaining the optimal solution for each of our evaluation experiments is significantly high because of the $\mathcal{NP}$-hardness of the problem. To reduce the running time, Algorithm 1 shows a fast heuristic whose solution we compare against the CPLEX solution in terms of performance and running time.

This heuristic takes a flow and returns a least cost path, while fulfilling the flow requirements. It takes the current state of the network graph (G with nodes, edges, residual link capacities, fixed and dynamic costs, processing nodes) as input, along with the input flow requirements, *i.e.*, source and destination nodes, service chain, flow latency and ϕ_i (bandwidth ratio after the use of each VF along the chain).

Initially (line 1), we use the function *getFeasibleGraph* to get a subgraph (G') from the original graph G that includes only those links that have enough capacity to satisfy the

VR Flow		
Parameter	**Description**	**Value**
d^f_{VR}	Initial flow demand	μ = 10 Mbps σ = 2 Mbps
l^f_{VR}	Latency demand	μ = 5 ms σ = 1 ms
$\phi_{A\&AC}$	Ratio of outgoing to incoming flow rate through the Authentication & access control VF	0.9
$\phi_{P\&S}$	Ratio of outgoing to incoming flow rate through the Processing & storage VF	20
$\phi_{E\&T}$	Ratio of outgoing to incoming flow rate through the Encoding / Transcoding VF	0.8
AR Flow		
Parameter	**Description**	**Value**
d^f_{AR}	Initial flow demand	μ = 150 Mbps σ = 20 Mbps
l^f_{AR}	Latency demand	μ = 4 ms σ = 1 ms
$\phi_{A\&AC}$	Ratio of outgoing to incoming flow rate through the Authentication & access control VF	0.9
$\phi_{L\&Tk}$	Ratio of outgoing to incoming flow rate through the Localization / Tracking VF	0.9
$\phi_{E/P/S}$	Ratio of outgoing to incoming flow rate through the Embedding / Processing / Storage VF	1
$\phi_{E\&T}$	Ratio of outgoing to incoming flow rate through the Encoding / Transcoding VF	0.8

Table 3.6: Flow Parameters

flow end-to-end rate demand. *getNearbyPN* (line 2) finds the q nearest (in number of hops) processing nodes ($PN^{s^f}_q$) from the source (s^f) on subgraph G' using Dijkstra's shortest path algorithm. After getting $PN^{s^f}_q$ processing nodes, *getShortestPaths* is invoked (line 3) which calculates all possible least-cost paths through every permutation of the processing nodes in $PN^{s^f}_q$. While calculating paths, *getShortestPaths* makes sure that for each path, the segment from the source to the first processing node has available capacity that is at least equal to the rate outgoing from the source (d^f). Also, from the first processing node

Algorithm 1 Service Chain Placement Heuristic

Input:
f: incoming flow
$G(V, E)$: Network graph, V is set of nodes and E is set of links
PN: set of processing nodes, where $PN \subseteq V$
q: number of nearest processing nodes used for virtual function placement
Output: *minPath*

1: $G' = getFeasibleGraph(G, f)$; // subgraph G'(V,E'), E' can carry flow demand
2: $PN_q^{s^f} = getNearbyPN(G', s^f, PN, q)$;// get set of q nearby processing nodes
3: $P^f = getShortestPaths(PN_q^{s^f}, G, f)$;//all possible paths through processing nodes
4: minPath = *null*
5: minPathCost = ∞
6: **for** path p in P^f **do**
7:     **if** $pathFeasible(p, l^f, d^f)$ **then**
8:         $c_p^f = getCost(p, f)$ // cost = fixed cost + usage cost + VF placement cost
9:         **if** $c_p^f <$ minPathCost **then**
10:             minPathCost = c_p^f
11:             minPath = p
12:         **end if**
13:     **end if**
14: **end for**

to the last processing node, it has the maximum possible capacity required by the flow, and from the last node to the destination, it has at least a capacity of $d^f \prod_{i=s^f}^{n_l} \phi_i$.

Next, we evaluate each path individually. We perform additional feasibility checks using *pathFeasible* in line 7. *pathFeasible* checks if the path's latency is less than the flow's end-to-end latency requirement and the path can provide/deploy the function chain. If the path is feasible, we calculate the cost of allocating the flow f on the path p using the function *getCost* (line 8); the cost includes link usage and VF deployment cost along the path p. Here, we take a greedy approach where we try to use VFs that are already deployed along the path, otherwise collocate other missing VFs on the same processing node(s) if feasible. After evaluating all paths in P^f, we pick the path with the lowest cost for the flow.

3.6 Evaluation Results

In this section, we discuss the results of our study where we evaluate the performance and cost of allocating service chains as flows arrive to the edge network. We consider the following performance metrics: (1) *Flow Acceptance Ratio*: is the ratio of flows accepted (*i.e.*, resources are available to allocate to these flows) to the total number of flow arrivals, (2) *Virtual Capacity Allocated*: is the total virtual capacity of all links along the service chains of accepted flows, and (3) *Average Link Utilization*: is the ratio of link usage over link capacity averaged over all links, or over each of the two types of link (*Wire* and *mmWave*). Results are shown for BRITE, Santa Monica and Palo Alto topologies. For each type of typology, results with 90% confidence intervals are shown for *Wire*, *Single*, and *Dual* networks for both BIP and Heuristic.

Observations: Before presenting the details of our results, we summarize our main observations as follows: (1) Augmenting the physical (*Wire*) infrastructure with *mmWave* links yields significantly higher flow acceptance ratio and virtual capacity allocated (up to 20% higher); (2) Most significant gains using *mmWave* links are in high node density network, where nodes are closer to each other, and a reliable *mmWave* links can be established between the nodes. (3) These *mmWave* links should complement the connectivity provided by *Wire* links and only a small number of *mmWave* links needs to be deployed to achieve most performance gains; (4) The flexibility in resource allocation afforded by decomposing applications into service chains that can be deployed *anywhere* on the edge infrastructure yields significant gains (up to three times higher accepted virtual capacity) over a traditional "middlebox" static deployment; and (5) The proposed heuristic decreases the running time by up to one order of magnitude when compared with BIP while giving performance results close to BIP.

The cost versus running time for BIP and our heuristic is shown in Figure 3.8 for

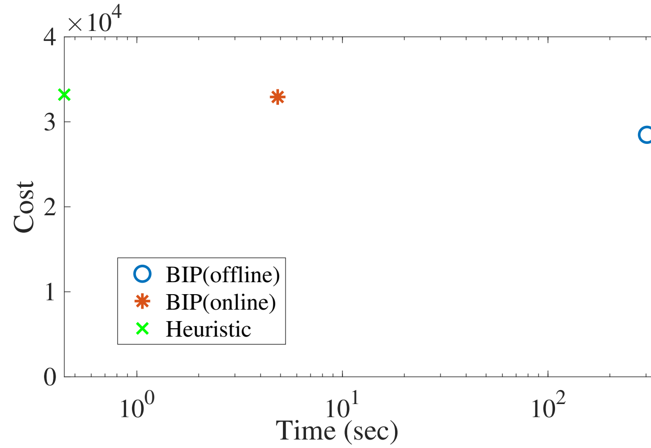
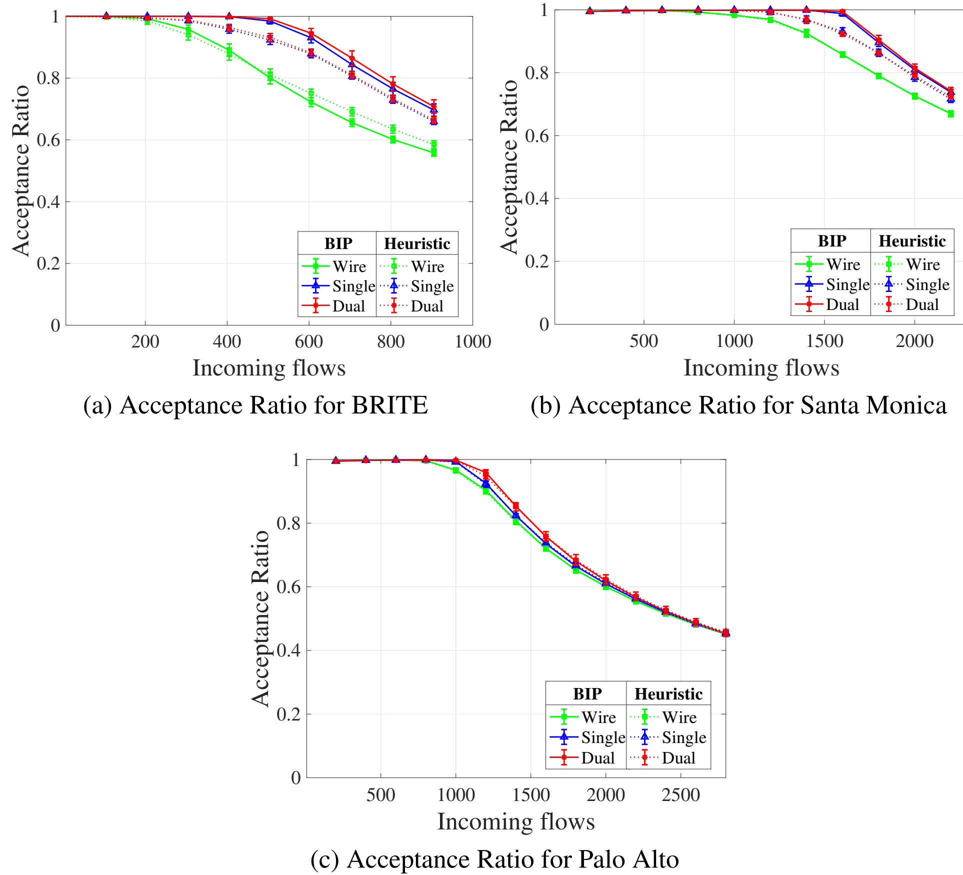


Figure 3.8: Cost vs running time comparison of BIP vs Heuristic

the *Dual* scenario. As explained in Section 3.3, in the BIP *online* case, the resources are dynamically allocated for each flow as it arrives, while in the *offline* case, all flow demands are known in advance and resources are simultaneously allocated for all flows. Since *offline* has advance knowledge of all flow demands, it can efficiently allocate the flows on the network and the cost is lowest. However, the running time for *offline* is orders of magnitude larger than the online case. The proposed heuristic yields a cost comparable to the BIP *online* case, with running time that is one order of magnitude lower. The *offline* resource provisioning is not always possible since we cannot accurately predict incoming flows. For this reason, in the remainder of the chapter, results are shown for the BIP online case.

Figure 3.9 shows the flow acceptance ratio of BRITE, Santa Monica, and Palo Alto as a function of incoming flows for different types of network. The high node-density networks of BRITE and Santa Monica with *mmWave* links (*Single* and *Dual*) accept more flows than the same network with only wired links (*Wire*). The low-density network of Palo Alto yields little/no gain with *mmWave* links. As the density of nodes decreases, the probability of having a *mmWave* link between two nodes (within range) is small. This



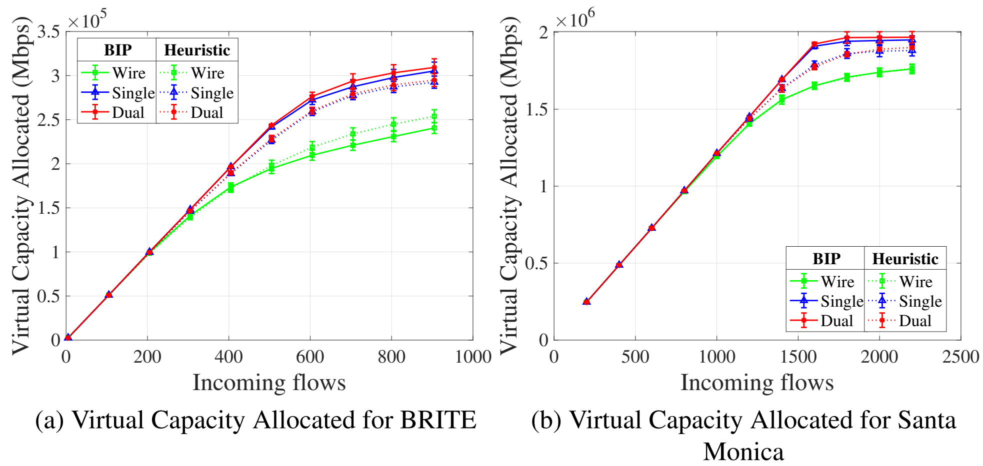
(a) Acceptance Ratio for BRITE (b) Acceptance Ratio for Santa Monica

(c) Acceptance Ratio for Palo Alto

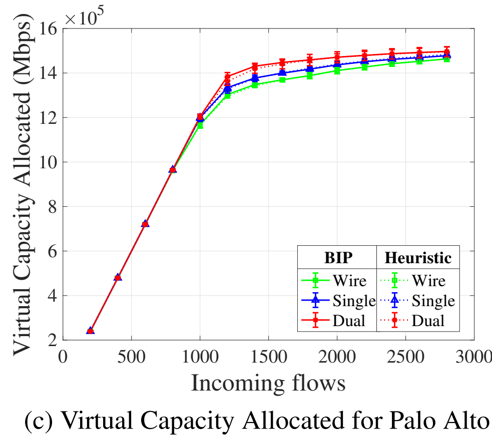
Figure 3.9: Flow Acceptance Ratio for *Wire*, *Single* and *Dual* networks with BIP and Heuristic

yields a network with sparse *mmWave* links, thus a small increase in the connectivity and capacity of the network, and a little/no increase in the accepted flows. This difference is more significant in high-density networks. The percentage of *mmWave* links are higher in high-density networks as nodes are closer to each other, and the probability of having *mmWave* links between two nodes (within range) is higher. This yields a network that is better connected and with increased capacity. Thus, we observe a higher number of accepted flows.

Since each flow can have different capacity requirements along its virtual service chain, the number of flows accepted does not necessarily mean that the network capacity is ef-



(a) Virtual Capacity Allocated for BRITE (b) Virtual Capacity Allocated for Santa Monica

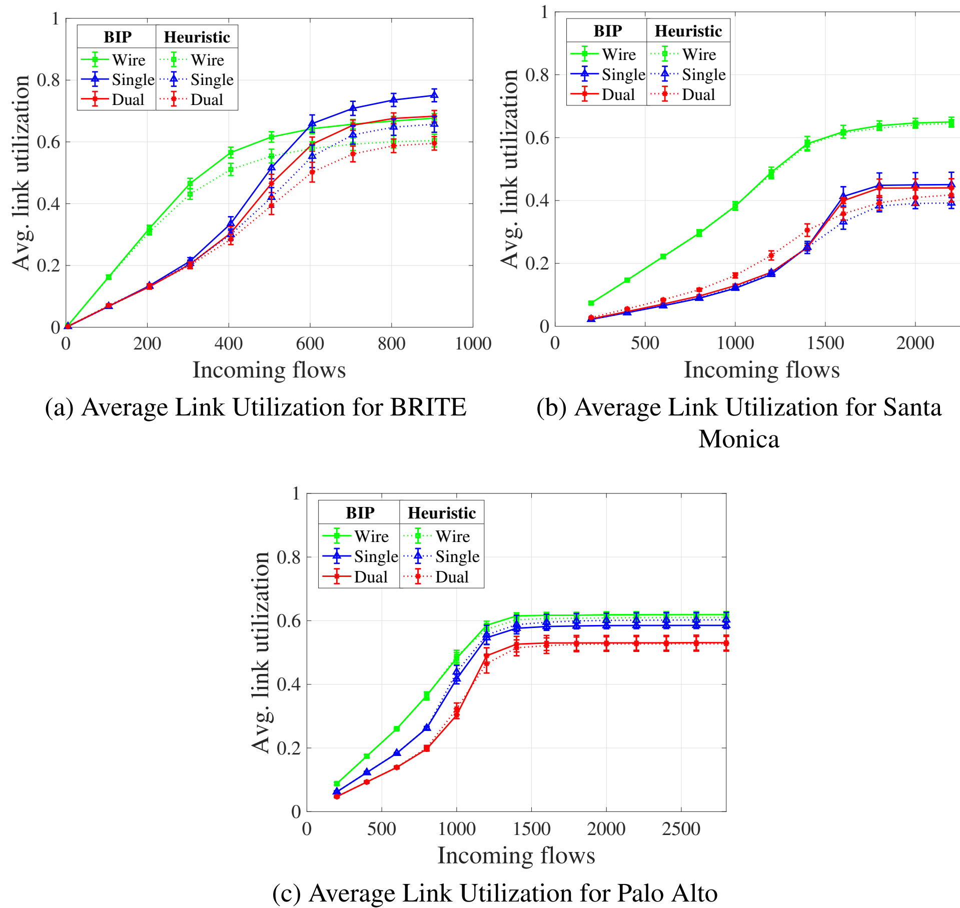


(c) Virtual Capacity Allocated for Palo Alto

Figure 3.10: Virtual Capacity Allocated for *Wire*, *Single* and *Dual* networks with BIP and Heuristic

ficiently allocated. Figure 3.10 shows the virtual capacity allocated for BRITE, Santa Monica and Palo Alto. Again, we see that *Single* and *Dual* have higher virtual capacity allocated than *Wire* for high node-density networks. There is little/no gain for Palo Alto, which has low node density. For both Figure 3.9 and Figure 3.10, results obtained by the proposed heuristic are very close to BIP.

Figures 3.11 to 3.13 show the average link utilization for both *mmWave* links and *Wire* links. We observe that the *Wire* network has higher link utilization because the network has lower capacity and links get congested quickly. Figures 3.12 and 3.13 show the link



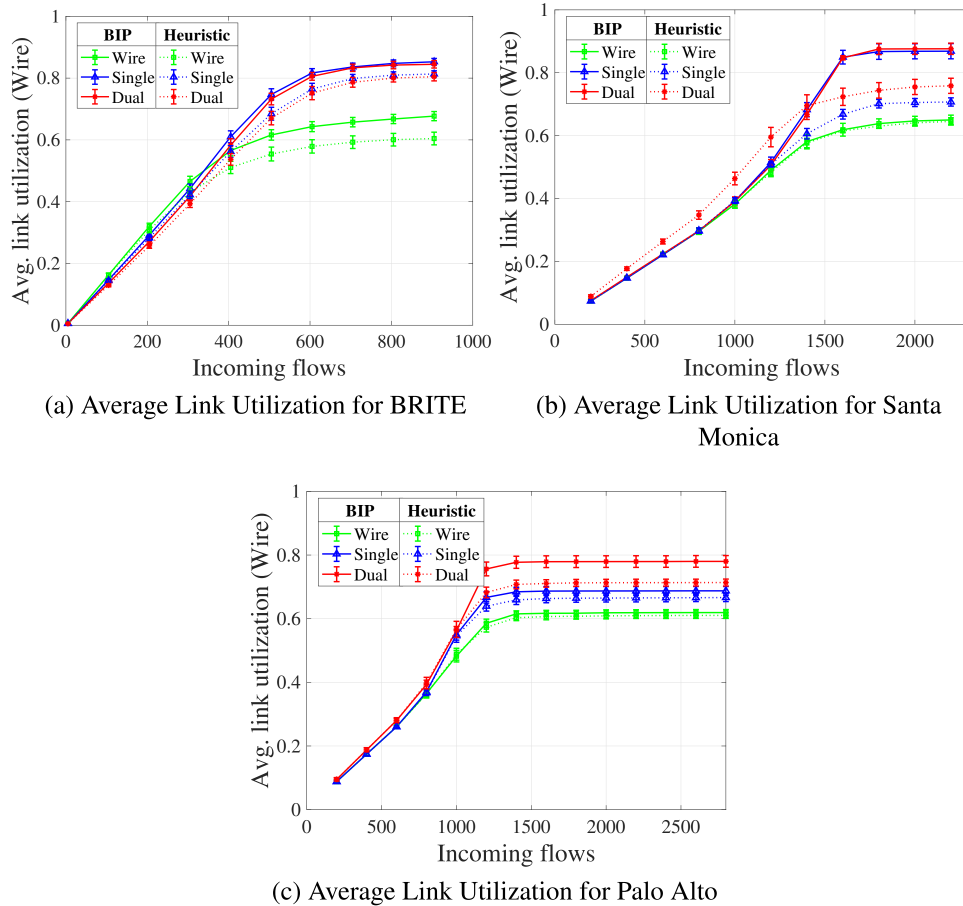
(a) Average Link Utilization for BRITE

(b) Average Link Utilization for Santa Monica

(c) Average Link Utilization for Palo Alto

Figure 3.11: Average Link Utilization for all links as a function of incoming flows for *Wire*, *Single* and *Dual* networks

utilization for *Wire* links, and for *mmWave* links, respectively. We see in Figure 3.12 that *Wire* links are better utilized (up to 20%) when there are *mmWave* links in the high node-density networks. The existence of *mmWave* links makes the network better connected, which leads to better utilization of the resources and higher number of flows accepted. Figure 3.13 shows that *mmWave* links are better utilized (up to 10% in BRITE) in *Single* networks compared to *Dual* networks, although the acceptance ratio and virtual capacity allocated for both networks are the same. However, *mmWave* links have higher usage cost. Thus, initially, when the network is not yet congested, only a few *mmWave* links are used.



(a) Average Link Utilization for BRITE

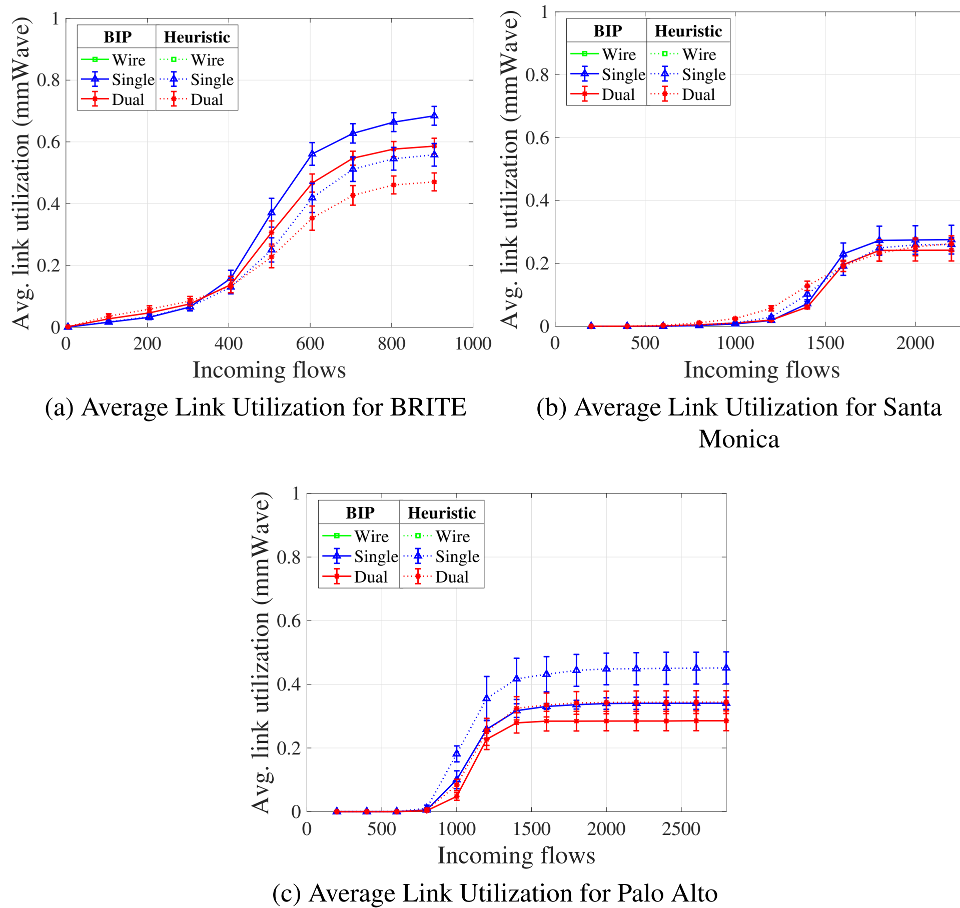
(b) Average Link Utilization for Santa Monica

(c) Average Link Utilization for Palo Alto

Figure 3.12: Average Link Utilization for *Wire* links as a function of incoming flows for *Wire*, *Single* and *Dual* networks

So initially, the average utilization for *mmWave* links is low, as shown in Figure 3.13. On the other hand, as more flows enter the system and the network becomes congested, more and more *mmWave* links are used to satisfy the flow demands. This leads to higher utilization of *mmWave* links, but at a higher cost. In all the graphs, the gain is significant in the high node-density networks. We also provide a comparison with the proposed heuristic. We observe that the heuristic performance is close to the performance given by BIP.

Figure 3.14 shows the CDF of utilization of the *mmWave* links for *Single* scenarios for BRITE, Santa Monica and Palo Alto. We observe that in the *Single* scenario, upto 60%



(a) Average Link Utilization for BRITE

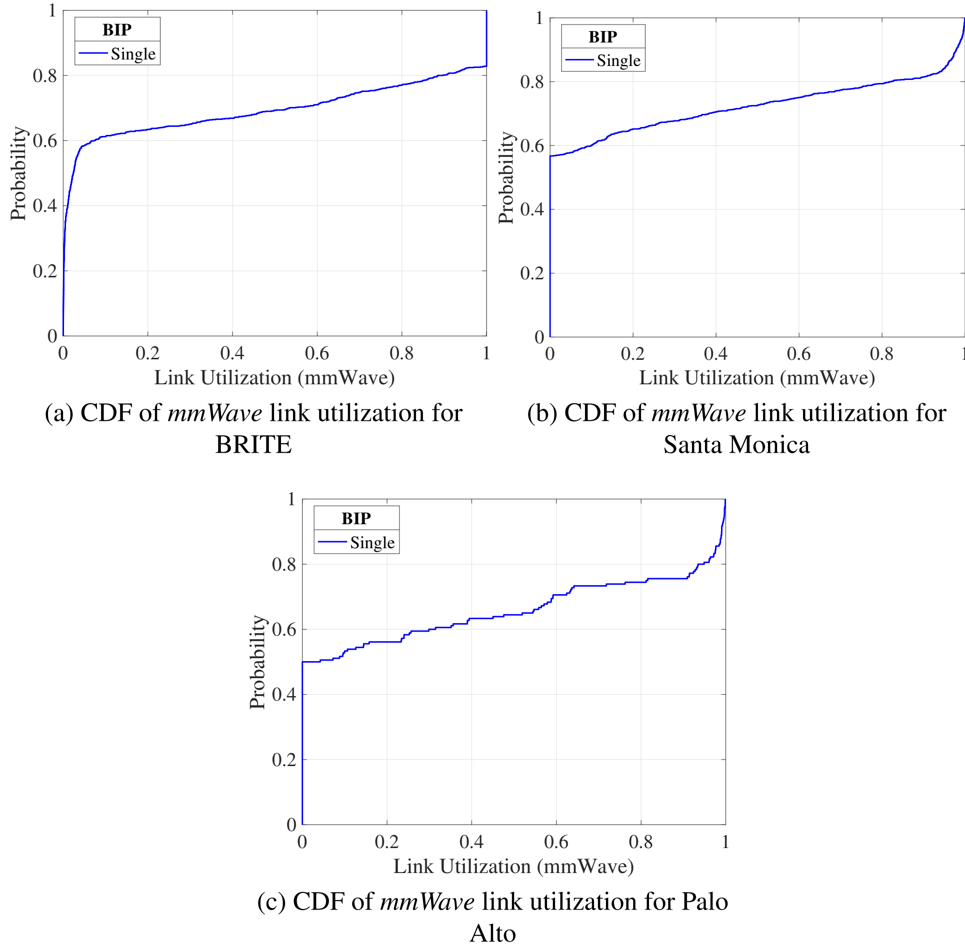
(b) Average Link Utilization for Santa Monica

(c) Average Link Utilization for Palo Alto

Figure 3.13: Average Link Utilization for *mmWave* links as a function of incoming flows for *Single* and *Dual* networks

of the links have utilization of less than 6%, and around 20% of the links are completely saturated with utilization close to 100%. This shows that significant performance gains can be achieved by judiciously deploying a small number of *mmWave* links.

Middlebox Scenario: To highlight the benefit of using (optimal) distributed virtual NF placement, we compare it with a traditional middlebox scenario. In the middlebox scenario, a powerful hardware appliance, with all the required services, is placed at the edge of the network. For each network (*i.e.*, *Wire*, *Single* and *Dual*), we chose a single Processing Node (PN) to host the middlebox, *i.e.*. We set this middlebox to be 10 times more



(a) CDF of *mmWave* link utilization for BRITE

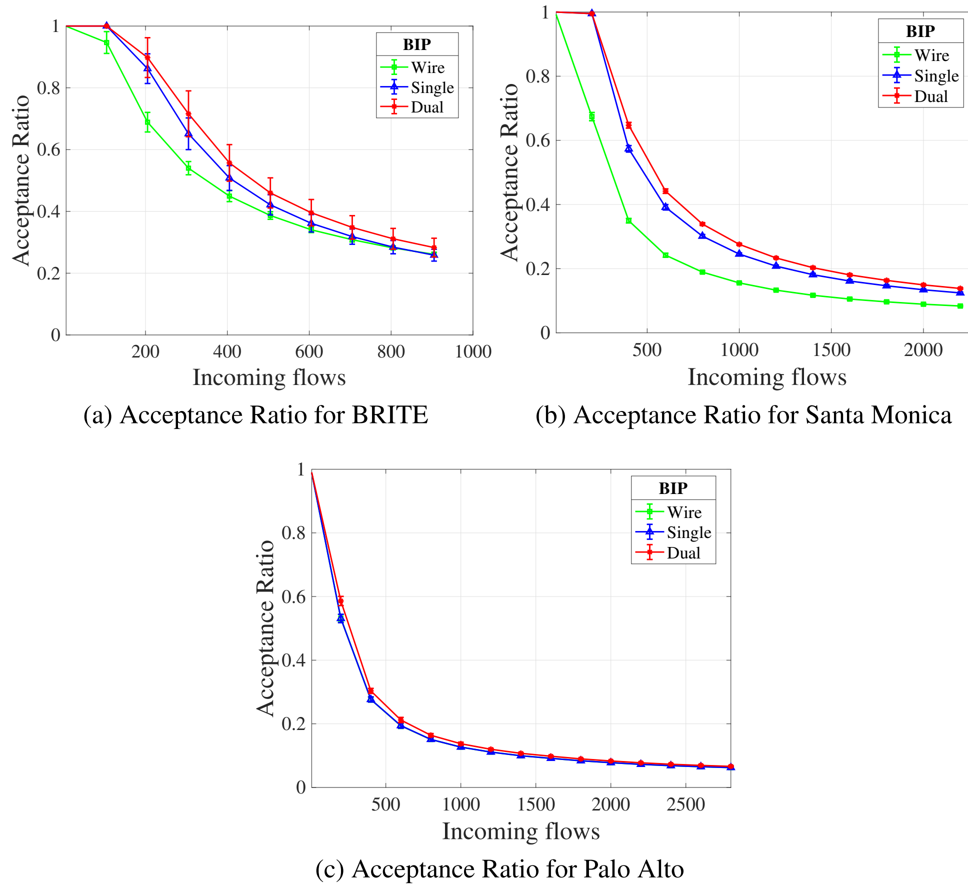
(b) CDF of *mmWave* link utilization for Santa Monica

(c) CDF of *mmWave* link utilization for Palo Alto

Figure 3.14: CDF of *mmWave* link utilization for *Single* networks

powerful (*i.e.*, it can serve 10 times more flows) than a virtualized service placed on a PN, and it runs all the needed services. Figures 3.15 and 3.16 show the flow acceptance ratio and virtual capacity allocated, respectively, for the middlebox scenario. The number of flows accepted in the middlebox case (Figure 3.15) are far lower than that accepted in the distributed VF placement scenario (Figure 3.9). As shown in Figures 3.10 and 3.16, the virtual capacity allocated for the distributed VF placement scenario is three times higher than the traditional middlebox scenario for higher density networks.

Discussion: The results clearly show the benefits of introducing *mmWave* links in the net-



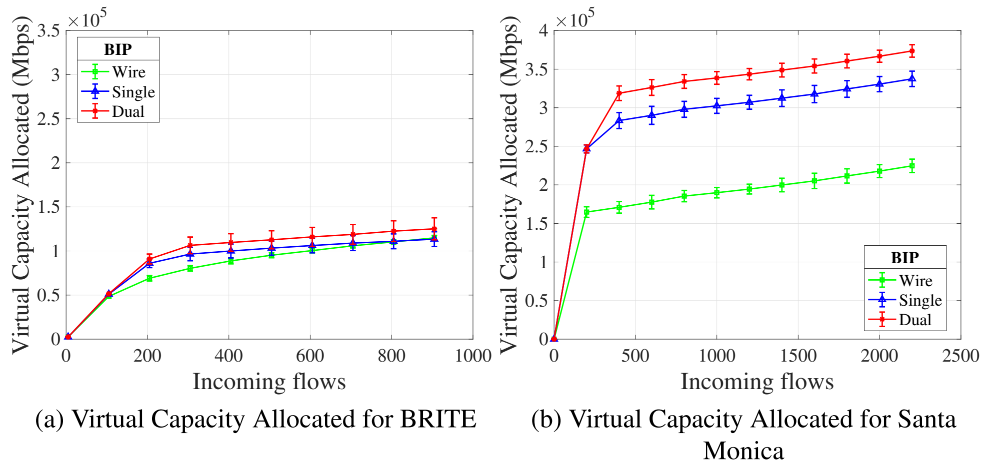
(a) Acceptance Ratio for BRITE

(b) Acceptance Ratio for Santa Monica

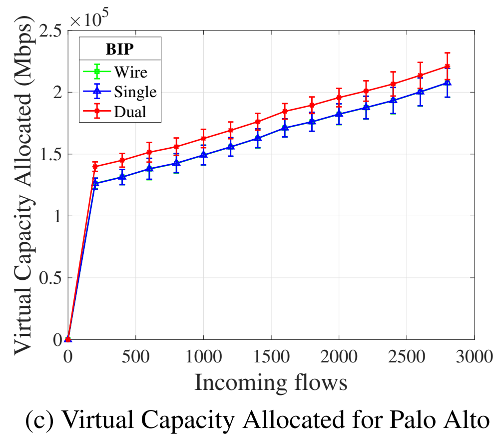
(c) Acceptance Ratio for Palo Alto

Figure 3.15: Flow Acceptance Ratio under middlebox schenario for *Wire*, *Single* and *Dual* networks

work with higher node density. However, it is important to wisely deploy these *mmWave* links. As shown in Table 3.4, the *Dual* network has a larger number of *mmWave* links compared to the *Single* network. However, if we look at the marginal utility of using *Dual* over *Single*, the gains are negligible. The flow acceptance ratio and the virtual capacity allocated (Figures 3.9 and 3.10) for both cases are within the 90% confidence interval. Furthermore, the average utilization of links is higher in *Single* compared to *Dual* (Figure 3.11) for high density network, which means links are better utilized in the former. Figure 3.14 also shows that only a small number of *mmWave* links are needed to achieve most performance gains. This leads us to conclude that a small number of *mmWave* links should



(a) Virtual Capacity Allocated for BRITE (b) Virtual Capacity Allocated for Santa Monica



(c) Virtual Capacity Allocated for Palo Alto

Figure 3.16: Virtual Capacity Allocated under middlebox schenario for *Wire*, *Single* and *Dual* networks

be introduced such that the overall connectivity between the nodes is increased, rather than to just increase the capacity of the network.

We also note that the middlebox scenario fails to take advantage of introducing *mmWave* links, as the number of flows accepted for the *Wire* network is similar to that for networks with additional *mmWave* links (Figures 3.10 and 3.16).

3.7 Summary

In this chapter, we studied the problem of allocating resources at the edge in support of envisioned 5G applications, *e.g.*, virtual and augmented reality. We presented a model of an edge network with multiple link technologies, namely, *Wire* and *mmWave*. We also developed a workload model that consists of the service chains with varying capacity requirements as the traffic flow traverses its chain. We formulated a binary integer optimization problem whose objective is to minimize the cost of deploying these service chains over the edge network, while satisfying their high throughput and ultra-low latency requirements. We also introduced a fast heuristic to solve the problem. Our extensive evaluations demonstrate the benefits of managing virtual service chains (by distributing them over the edge network) compared to a baseline "middlebox" approach (where all services are run on one host) in terms of overall admissible virtual capacity.

Moreover, we observe significant gains when deploying a small number of *mmWave* links that complement the *Wire* physical infrastructure. We show that a network topology with a high density of nodes has the highest performance gains since a large number of reliable *mmWave* links are formed between the nodes, thus increasing both the capacity and connectivity of the network.

Chapter 4

Configuring Serverless Functions using Statistical Learning

4.1 Introduction

Serverless computing has emerged as a new and compelling paradigm for the deployment of cloud applications and services. It promises new capabilities that make writing scalable microservices easier and cost effective. Most of the prominent cloud computing providers have released serverless computing platforms, and there are also several open-source efforts including OpenLambda [44] and OpenWhisk [2].

The serverless paradigm [30] at its core provides developers with a simplified programming model for creating cloud applications that abstracts away most, if not all, operational concerns. They no longer have to worry about provisioning and managing servers, and other infrastructure issues. Instead, they can focus on the business aspects of their applications. The paradigm also lowers the cost of deploying cloud code by charging for execution time – following a "pay as you go" pricing model [1, 8] – rather than for allocated resources.

In serverless application-development [38], a developer implements the business functionality as a stateless or composition of stateless functions using one or a combination of the programming languages supported by major cloud providers. Currently, Python and

	AWS Lambda	**Google Function**	**IBM Cloud Function**
Memory (MB)	$64 \times$ i i = {2,3, ... 47}	$128 \times$ i i = {2,4,8,16,24}	{256 ... 2048}
Language	Python, Nodejs & others	Nodejs	Python, Nodejs, & others
Billing	Execution time based on memory	Execution time based on memory & CPU-power	Execution time based on memory
Configurable	memory	memory & CPU-power	memory

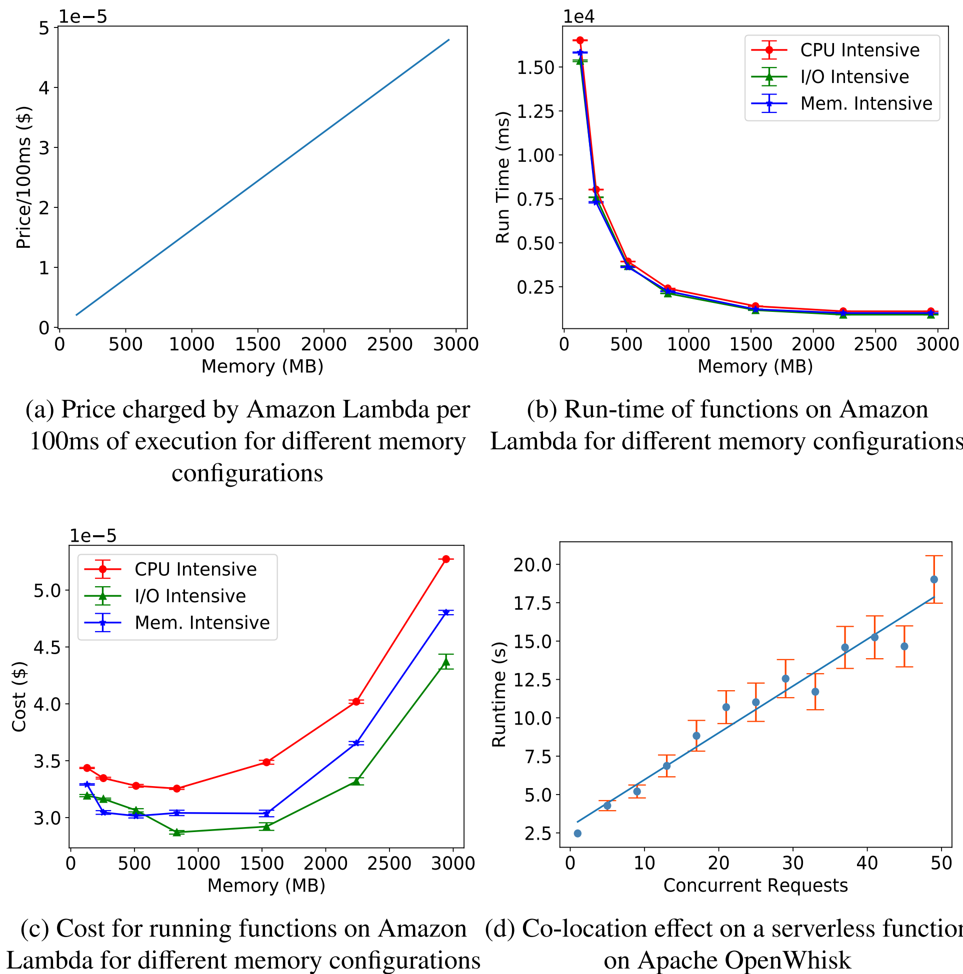
Table 4.1: Serverless platforms

Nodejs are the most common scripting languages supported by major serverless platforms (*c.f.* Table 4.1). The developer then submits the code to the cloud provider along with dependencies (*e.g.* libraries), and specifies configuration parameters such as memory size or CPU power. The cloud provider stores this code, and on invocation – which can be triggered through events or HTTP requests – executes this code either in containerized environments [2] or virtual machines over varying underlying physical infrastructures, with the specified configurations. Table 4.1 highlights serverless platforms from three major cloud providers[1]. The table shows programming languages supported, billing methodology, and memory size or CPU-power options that a user can select.

Serverless computing has given a much-needed agility to developers, abstracted away the management and maintenance of physical resources, and provided them with a relatively small set of configuration parameters: memory and CPU. While relatively simpler, configuring the "best" values for these parameters while minimizing cost and meeting performance and delay constraints poses a new set of challenges. This is due to several factors that can significantly affect the running time of serverless functions.

To highlight the effects of parameter configuration on the performance and cost of

[1]Note that Microsoft Azure Functions does not provide users with the ability to configure functions and the cost is based on per-second resource consumption and execution time. In this work, we focus on configurable functions where users can configure serverless functions to meet service requirements.



(a) Price charged by Amazon Lambda per 100ms of execution for different memory configurations

(b) Run-time of functions on Amazon Lambda for different memory configurations

(c) Cost for running functions on Amazon Lambda for different memory configurations

(d) Co-location effect on a serverless function on Apache OpenWhisk

Figure 4.1: Serverless function's performance with different memory sizes and co-location

serverless functions, we deployed serverless functions written in Python on AWS Lambda. In AWS Lambda, a customer is allowed to configure the amount of memory allocated to a serverless function. We ran this function for different memory sizes and studied the effect of varying memory sizes on the performance of the function, *i.e.* run-time. Figure 4.1b shows how the run-time of these serverless functions decreases with the increase of memory size allocated to the function. However, the marginal improvement in the run-time decreases as the memory increases. Figure 4.1c shows the cost of corresponding runs, which

is the product of price (Figure 4.1a) and run-time (Figure 4.1b)[2]. As shown in Figure 4.1c, choosing too small a value or too large a value for memory can result in higher costs for running the function[3]. This behavior is because the pricing model as exposed by the cloud providers is tightly coupled with the amount of resources specified to execute the serverless function (*c.f.* Figure 4.1a), and the dependency between memory and CPU resource allocation – AWS Lambda allocates CPU power linearly in proportion to the amount of memory configured [3]. We observed similar behavior when setting configurable parameters on Google Function. Running similar experiments on Apache OpenWhisk showed that co-location has a significant impact on the run-time. Figure 4.1d shows the effect of co-locating serverless functions on Apache OpenWhisk running on a machine with a single CPU core.

The examples above highlights some factors that can affect the performance of running serverless functions. However, they are not the only factors. A recent study [80] showed that the underlying infrastructure and resource provisioning can vary significantly depending on multiple factors including function placement, cold starts, I/O and network conditions, type of VMs/containers, and co-location with other functions. The user is oblivious to all these other factors, and has only limited control over a few parameters affecting performance, *i.e.* memory and processing power.

Given the issues raised above and the limited control a user has over the underlying system parameters, finding the "best" configuration to run a function while minimizing cost and meeting performance and delay constraints poses a new set of challenges. The problem is even more challenging when a user is running a chain of interdependent functions – where a user can still meet the performance requirement of the chain by trading off the performance of some of the functions in the chain for lower cost – and when a user is

[2]Note that a small difference in run-time translates to a larger difference in cost.
[3]Our results are consistent with recent studies [49] on the cost of executing serverless functions.

presented with the option to pick between multiple locations, *i.e.* edge and core [19] [20] [39].

In this chapter we present COSE, a framework that uses Bayesian Optimization to statistically learn the relationship between cost/run-time and unseen configurations of a serverless function. Using this learned relationship, henceforth referred to as *performance model* of the serverless function, our framework is able to pick the best configuration for a serverless function which not only minimizes the cost but also meets user-specified performance criteria such as response time/delay of running a serverless function or chain of these functions. Our framework is lightweight and has the ability to dynamically adapt to changes in the execution time of a serverless function. It can be incorporated into an offering by cloud providers; it could be implemented as a value-added proposition by service providers; or it could be directly leveraged by customers. We evaluate our framework not only on a commercial cloud provider, where we successfully found optimal/near-optimal configurations in as few as five samples[4], but also over a wide range of simulated distributed cloud environments that confirm the efficacy of our approach.

4.2 System Description

Figure 4.2 provides an overview of our COSE framework. It consists of two main components: a *Performance Modeler* component, which is responsible for learning the application's *performance model*, *i.e.* the relation between cost/run-time and configurations for the serverless function, and the *Config Finder* component whose goal is to find the "best" configuration that minimizes cost while satisfying the delay bound on the running time of the serverless function. As indicated earlier, COSE can be incorporated into an offering by cloud providers; it could be implemented as a value-added proposition by service

[4] A configuration "sample" refers to a serverless function invocation at certain parameter values.

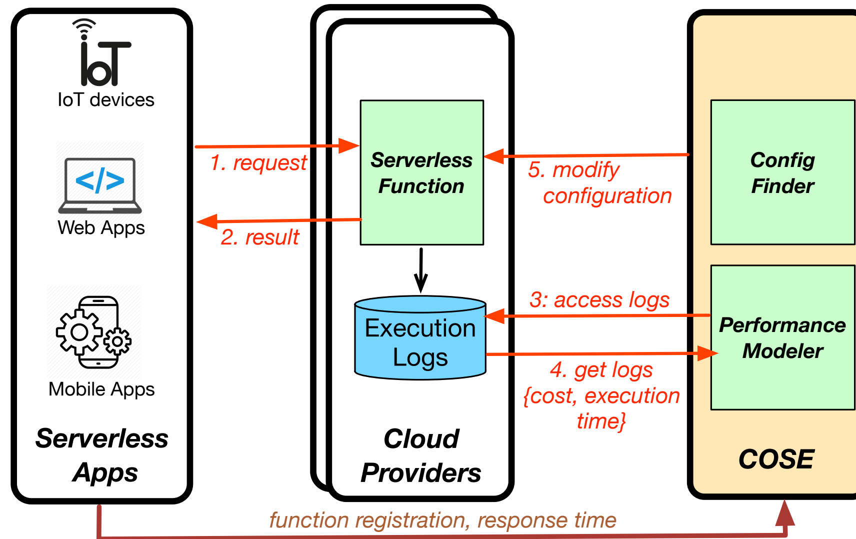


Figure 4.2: System overview

providers; or it could be directly leveraged by customers. For the rest of this work, we will assume that our COSE framework has been adopted by a Service Provider, and through standard APIs, a client registers her serverless function with the COSE service.

Figure 4.2 highlights the interactions between our COSE framework and its environment. Application clients, *e.g.* mobile and IoT devices, issue requests to the cloud provider to invoke a serverless function. Once the function is invoked, a trace log, containing the cost and execution time, is generated and stored. Our framework acts as a monitoring service and utilizes the information from the trace to learn the performance model of the function. After the learning phase converges, COSE uses *Config Finder* to find the "best" configuration that minimizes cost while satisfying the delay bound on the running time of the serverless function or a chain of functions. To account for delays associated with different locations/services supported by a designated cloud provider (e.g. Amazon Lambda's "edge" vs. "core"), a client reports the response times of its serverless function invocations to COSE[5]. If a change to the previous configuration is needed, COSE connects to the

[5]While this requires changes to the client, in practice, techniques to estimate the response time across

designated cloud provider using APIs to modify the configuration. Next, we discuss the approaches and choices for the components of our COSE framework.

4.3 COSE: The *Performance Modeler* component

Our COSE framework has been entrusted to execute a single or a set of serverless functions on a designated cloud provider. It has the ability to configure the parameters of the serverless function, *e.g.* amount of memory, or the location of running the function by requesting it from the Cloud Provider (CP). The goal is to learn the application's performance model. There are several ways to achieve this:

1. *Exhaustive Search for the best cloud configuration:* This method runs the serverless function under all or a subset of possible configurations to find the configuration that minimizes the cost [4]. This methodology has very high overhead. Amazon Lambda alone has over 45 different memory types with a choice of location between "edge" and "core". To learn configurations across multiple CPs needs hundreds, if not thousands, of function executions. Moreover, the performance of the function can vary depending on the type of physical resources the function is deployed to execute on, and whether the function is co-located with other functions. This can lead to repeating the exhaustive search again to find the best configuration.

2. *Algorithms for parameter descent:* As an alternative to doing exhaustive search, this method performs the search using parameter descent algorithms. The algorithms choose parameter values in the direction of decreasing cost. For example, if the memory value 512MB gives lower cost than 448MB,[6] the algorithm chooses a value greater than

geographically distributed clients can be incorporated without requiring any changes to the client.

[6]Recall that AWS memory options available from AWS Lambda increases or decreases in 64MB increments.

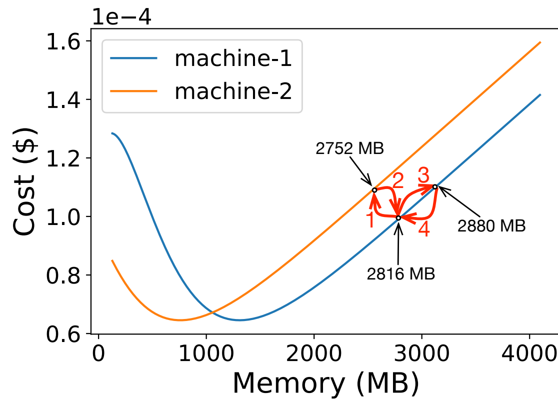


Figure 4.3: Example of AIAD getting stuck at a local minimum

512MB in anticipation of decreasing the cost further. Algorithms such as Additive Increase Additive Decrease (AIAD) and Gradient Descent, can be used. Such algorithms have tendency to get stuck in local minima, which leads to sub-optimal configuration for a serverless function.

Figure 4.3 illustrates a simple example where AIAD gets stuck at a local minimum. AIAD, as its name implies, uses a small fixed amount – 64MB in our example – to either increase or decrease requests for resources. Imagine a CP with two machines with different hardware configurations: machine-1 and machine-2. The performance of the serverless function will be different on machine-1 and machine-2 since these are shared resources and the performance depends on the utilization of each machine. Initially, AIAD requests a large memory configuration, *e.g.* 2816MB. Our serverless function is placed on machine-1. Using AIAD, it descends in the direction of decreasing cost. When the memory requested decreases from 2816 MB to 2752 MB (shown by arrow 1), the CP decides – potentially due to cost savings from colocating it with other functions – to place the serverless function on machine-2. Since the cost of execution is high at memory 2752 MB, AIAD changes direction and asks for a higher memory value

of 2816 MB, which makes the CP place the function back on machine-1 (shown by arrow 2). In the next step, AIAD will further go in this same cost-reducing direction and ask for a higher memory value of 2880 MB (shown by arrow 3). However, the cost becomes higher at 2880 MB when compared with 2816 MB, so AIAD will change direction and in the next iteration, ask for 2816 MB (shown by arrow 4). The process will keep repeating and AIAD will be stuck at a local minimum. We implemented AIAD and Gradient Descent and tested them on commercial cloud providers.

Another drawback of parameter descent algorithms is that they do not continually learn the underlying relation between cost/execution time and configuration, and so if the underlying conditions or requirements change, the whole process needs to be repeated.

3. *Statistical Learning for finding the best configuration:* This approach uses a statistical learning model to predict the performance of a serverless function under different configurations. It involves sampling different configuration values to successfully model the performance of the function and predict the configuration that will minimize the cost. In this chapter, we use **Bayesian Optimization** as the statistical learning approach to find the "best" configuration for a serverless function.

4.3.1 Our Approach: Leveraging Bayesian Optimization

The objective of Bayesian Optimization (BO) is to optimize over a black-box function. In our case, the function that we want to learn is the relationship between performance/cost and all possible configurations, not to be confused with the serverless function/code itself that we want to execute. Knowing this relationship, one can readily locate the configuration that minimizes cost, or that meets a certain performance/delay requirement.

BO constructs a probabilistic model for a black-box function in a predefined parameter space and exploits this model to make decisions about where to next sample/evaluate the

function. It uses the information from all previous observations of the black-box function to find the next sample. The goal is to learn the black-box function in a few number of samples. Compared to deterministic searching/learning, BO *dynamically adapts* its search based on its current characterization and confidence interval of the prediction model. BO dynamically picks the next sample that gives more information and avoids unnecessary samples. BO stops searching when it has high confidence in the predicted model and the expected improvement for the predicted model is small for new samples.

4.3.1.1 Problem definition

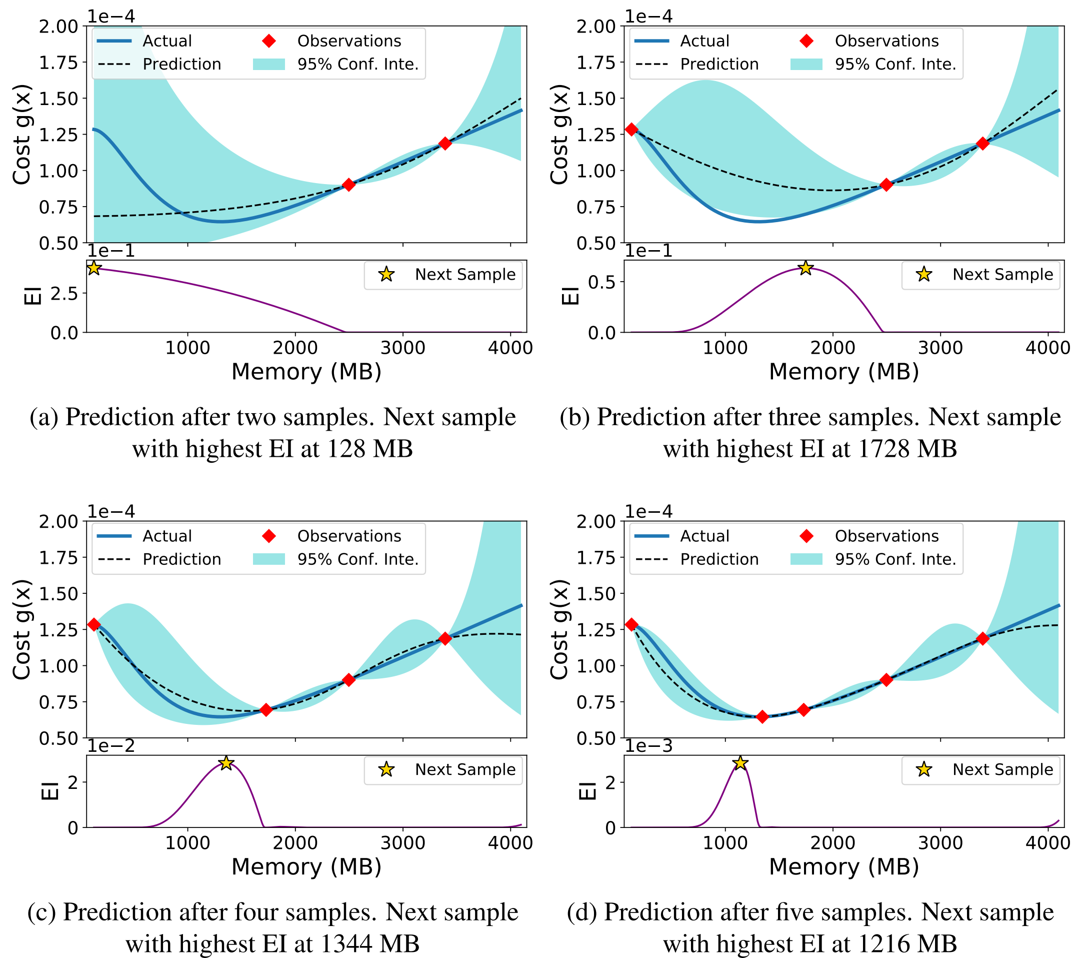
We formally define the cost minimization problem as follows: Given a cost $g(x)$ to execute a function on a CP premise, the objective is to find a configuration x that will minimize the cost.

$$\underset{x \in C}{\text{minimize}}\, g(x) \tag{4.1}$$

where x is a particular configuration in the set of all possible configurations C. C is a hyper-rectangle where each parameter in a configuration x is bounded $\{x \in \mathbb{R}^d : a_i \leq x_i \leq b_i\}$. It is expensive to solve $g(x)$ for all possible values of $x \in C$, so we use BO to intelligently search for an appropriate solution with a small number of samples.

4.3.1.2 BO at work

BO observes the objective function $g(x)$ at different sampled values. It models $g(x)$ as a stochastic process and computes a confidence interval for $g(x)$ based on the samples collected. Figure 4.4 shows a simple example where configuration x consists of a single dimension, *i.e.,* memory for a serverless function. The *actual* underlying function is given by the solid blue line. The confidence interval is an area around the predicted function where the actual function is passing through with 95% probability. In Figure 4.4a, there



(a) Prediction after two samples. Next sample with highest EI at 128 MB

(b) Prediction after three samples. Next sample with highest EI at 1728 MB

(c) Prediction after four samples. Next sample with highest EI at 1344 MB

(d) Prediction after five samples. Next sample with highest EI at 1216 MB

Figure 4.4: Bayesian Optimization example

are only two samples collected and the confidence interval is higher in the region further away from the observed values. The black dashed line is the predicted objective function of $g(x)$. As BO collects more samples (in Figure 4.4b and 4.4c), the confidence interval gets smaller and the prediction is closer to the actual values. BO intelligently samples the next point to evaluate/observe $g(x)$, based on the so-called *acquisition function* – Expected Improvement (EI) in this case. EI is calculated for each possible configuration in the search space. A configuration with the highest EI value is selected as the next sample. As shown in the lower part of Figure 4.4a, the highest EI value is at 128 MB and this value is used as

a configuration value for the next run of the serverless function as shown in Figure 4.4b.

Bayesian Optimization has two key parts. The first part is a probabilistic surrogate model, which consists of a prior over functions that express assumptions about the function being optimized. The second part is choosing an *acquisition function*, which is used to construct a utility function from the model posterior, allowing us to determine the next configuration point to evaluate.

We use the Gaussian Process (GP) as prior for BO. The GP is widely accepted as a good surrogate model for BO [74]. Moreover, the GP is the only choice that is computationally tractable.

4.3.1.3 Choosing a Kernel and Mean function

We want to calculate the conditional distribution of function $g(x)$ at multiple unobserved points in the configuration space. The resulting distribution is a multivariate normal, with a mean μ vector and covariance kernel that depends on the location of the unevaluated points with respect to evaluated points and their measured values. The kernel should have the property that, points closer in the input space are more strongly correlated. If $||x - x'|| < ||x - x''||$ for some norm $||.||$, then the kernel value $\sum_o(x, x') > \sum_o(x, x'')$. We used the Matern kernel, which is a commonly used kernel for BO. It does not require strong smoothness and is a preferred model for practical functions [74]. We use the mean function as $\mu(x) = \mathbb{E}[g(x)]$.

4.3.1.4 Acquisition function

The acquisition function calculates the utility for sampling at each point in configuration space and selects the point with highest utility as the next value to sample for objective

function $g(x)$. The objective function $g(x)$ will be sampled at

$$x_t = argmax_x\, A(x \mid S_{1:t-1})$$

where A is the acquisition function, $S_{1:t-1} = \{(x_1, y_1), ..., (x_{t-1}, y_{t-1})\}$ are the $t-1$ samples drawn from $g(x)$. The sample obtained can be a noisy sample $y_t = g(x_t) + \epsilon$. This sample is added to the previous sample space $S_{1:t} = \{S_{1:t-1}, (x_t, y_t)\}$.

Possible choices of acquisition functions are maximum probability of improvement (MPI), expected improvement (EI), and upper confidence bound (UCB). We chose EI as it is the most widely used acquisition function. EI has been shown to outperform MPI, and unlike UCB, it does not require parameter tuning [74].

Expected Improvement (EI): Expected improvement picks a point from the parameter space that maximizes the expected improvement over the current best point. It is defined as

$$EI(x) = \mathbb{E}\, max(g(x^+) - g(x), 0)$$

where $g(x^+)$ is the value of the "best" configuration sample so far with the minimum cost. Intuitively, we sample at a point x where we are most likely to see an improvement when compared to the best configuration value we have seen so far.

EI can be evaluated in closed form using integration by parts, as described by Jones *et al.* [51]. The resulting expression is

$$EI(x) = \begin{cases} (g(x^+) - \mu(x) - \omega)\Phi(Z) + \sigma(x)\phi(Z) & if\ \sigma(x) > 0 \\ 0 & if\ \sigma(x) = 0 \end{cases} \tag{4.2}$$

where $Z = \frac{g(x^+) - \mu(x) - \omega}{\sigma(x)}$. $\mu(x)$ and $\sigma(x)$ are the mean and the standard deviation of the GP posterior prediction at x. Φ and ϕ are the standard normal cumulative distribution function

and the standard normal probability density function, respectively. Intuitively, the first term in Equation (2) is the exploitation term, and the second term is the exploration term. The parameter ω is used to define the amount of exploration, where higher values lead to more exploration, and lower values lead to exploitation. Specifically, increasing the value of ω decreases the importance of improvement predicted by the GP posterior mean $\mu(x)$ relative to the importance of potential improvement in regions of high prediction uncertainty given by the large standard deviation $\sigma(x)$ value.

4.3.2 Adapting BO for Serverless Functions

To make BO run efficiently and accurately for serverless functions, we made changes to the classical BO. These changes are highlighted next.

(i) *Initial Points:* The choice of the initial points can guide the search for the optimal solution in Bayesian Optimization. A random choice of initial points can lead to longer convergence time. Since a serverless function tends to have a convex relation between the cost of execution and the chosen configuration (cf. Figure 4.1c), we chose the initial points as uniformly distributed in the search space. For this work, we chose four initial points.

(ii) *Reduce the Search Space:* We reduce the search space by discretizing the possible configuration parameters. BO has a computational complexity of $O(C^4)$, where C is the number of data samples. We used the *memory values* and the *cloud providers* as the set of parameters to choose from. We follow the choices of memory values available to customers on Amazon Lambda as shown in Table 4.1. Possible memory values that can be selected on Amazon Lambda are between 128MB and 3008MB, with the offset of 64 MB, *i.e.,* $m \in \{128MB, 192MB, 256MB, ..., 3008MB\}$. We calculated EI only for these discrete values of possible input memory size, which

decreased the running time significantly. For this work, we have only two cloud providers or two locations/services supported by a cloud provider, *Edge-Cloud* and *Core-Cloud*, which are readily discrete.

(iii) *Handling Noise:* A cloud environment is shared and there are uncertainties introduced because of the sharing of resources. Co-location of functions, cold-start, hardware failures, resource-overuse, *etc*., can impact the execution time of a serverless function running under the same configuration. Gaussian Process regression can be extended naturally to observations with independent normally distributed noise of known variance [70]. Since we have little or no knowledge of the underlying cost-configuration relation, this variance is not known. We assume that the noise is of common variance and it is included as a hyperparameter α of the Gaussian Process. Finding the best value for α is outside the scope of this work. Our experimental results show that $\alpha = 0.01$ captures most uncertainties in the serverless cloud platforms and we chose this value for our system.

(iv) *Accurately Predicting Changing Performance Model:* BO assumes that the target performance model is not changing while the samples are being collected. However, in practice, the target cost-configuration relation can change because of multiple reasons, *e.g.* migration of the serverless function to a different machine by the cloud provider, change in execution time based on the change in the input data to the function, *etc*. To predict the dynamics in the target cost-configuration relation, we keep a history of the configuration points sampled, and we discard the "old" sampled points as we collect new samples using a sliding window approach. This helps BO sample points again in the search space where it had sampled in the past, thus capturing the changes in the target cost-configuration relation.

(v) *Convergence Criteria:* As mentioned earlier, we use Expected Improvement (EI) as the convergence criterion for BO. When the EI for the next collected sample is below 5%, we consider that BO has converged and we henceforth use our BO predicted cost/run-time vs. configuration model to find the least-cost configuration that satisfies the delay constraint on the serverless function.

Our COSE-based service provider runs BO to predict the performance model of the serverless function, *i.e.* the black-box cost/run-time vs. configuration relation. It will keep on sampling until the BO has converged on the performance model. Once converged, COSE can use the predicted model to find the "best" configuration that satisfies the delay constraint for a serverless function. In Section 4.4, we show how delay constraints for serverless functions are met.

4.4 COSE: The *Config Finder* Component

Given the performance model of a serverless function, predicted by the *Performance Modeler*, it is easy for the second component of our COSE framework, *Config Finder*, to locate the configuration that minimizes the cost. However, real-world applications typically have delay constraints on the running time of a serverless function or chain of functions. In this section, we discuss how *Config Finder* picks the least-cost configurations that satisfy delay constraints.

The cost of running a serverless function f is given by:

$$g^f(x) = t^f(x) \times p^f(x)$$

where $t^f(x)$ is the execution time of the function using configuration x, and $p^f(x)$ is the price (cost per unit time) for running the function using configuration x. $p^f(x)$ is pro-

vided by the cloud provider and the predicted $g^f(x)$ is provided by the BO, so we can use these values to calculate the predicted execution time $t^f(x)$ for a serverless function under configuration x. The total time to run a function (end-to-end delay) is given by:

$$T^f(x) = t^f(x) + d(x)$$

where $d(x)$ is the delay other than the execution time of a function (such as network, queuing delay, *etc*.) Note that $d(x)$ is specific to a cloud provider or location and we estimate it by taking the difference $T^f(x) - t^f(x)$ of multiple samples collected by BO for the cloud providers or locations. This will help us predict the total time to run a serverless function, *i.e.* response time, on any cloud provider for a given configuration value.

With all the information above, it is easy for a service provider to find a configuration that satisfies the (end-to-end) delay constraint for a single serverless function. However, the problem gets complicated when we have a chain of functions (service chain) that need to execute one after another. All current serverless cloud providers support the chaining of functions. The service provider needs to select a configuration for each serverless function such that the cost to execute the service chain is minimized, while satisfying the delay constraint on the running time of the whole chain. The *Config Finder* module in COSE solves this problem. Using Integer Linear Programming (ILP), we formulate the problem as an optimization problem. *Config Finder* solves the ILP to find the best configuration for a chain of functions. Note that to solve for a single function, we consider the degenerate case of a chain of size one. Next we formulate the optimization problem.

We assume that for each serverless function f in chain F, we choose the cloud provider $v \in V$ and the memory $m \in M$ such that the total cost for placing the chain is minimized and the delay constraint D_F on service chain F is satisfied.

Define $Y_x^f \in \{0,1\} = 1$ if function $f \in F$ is deployed using configuration $x \in C$, 0

otherwise.

The objective of the *Config Finder* is to minimize the total price paid for the chain of functions. This is given by:

$$minimize\left(\sum_{f\in F}\sum_{x\in C} g^f(x)Y_x^f\right) \tag{4.3}$$

subject to:

1) The delay requirement for the service chain is satisfied:

$$\sum_{f\in F}\sum_{x\in C} T^f(x)Y_x^f \leq D_F \tag{4.4}$$

where $T^f(x)$ is the predicted end-to-end delay for running serverless function $f \in F$ using configuration $x \in C$.

2) A single configuration $x \in C$ is selected for each serverless function in the chain.

$$\sum_{x\in C} Y_x^f = 1 \quad \forall f \in F \tag{4.5}$$

The solution to this problem yields a least-cost feasible solution, *i.e.* the resulting Y_x^f, that gives the configuration x of each serverless function f in the chain. Note that one can argue that ILP takes long to solve since it is $\mathcal{NP}$-hard. However, since a chain typically consists of a few functions (in our case, less than five), the total time to execute this ILP on a CPLEX solver[7] is only a few milliseconds. For chains consisting of a large number of functions, a heuristic can be used to solve it. ILP can also be approximated using LP relaxation for large chain sizes [10].

[7] IBM ILOG CPLEX Optimizer,
http://www-01.ibm.com/software/integration/optimization/cplex-optimizer

4.5 Experimental Results: Running COSE on Amazon Lambda

We test the proposed COSE system on AWS Lambda, a very popular serverless cloud provider. We start by describing the class of functions that we tested on AWS Lambda.

4.5.1 Representative Functions

we test our COSE framework across four different representative functions for serverless computing. These functions represent the different types of computation (combination of *I/O-*, *CPU-*, *network-* and *memory*-intensive tasks) that a serverless application performs.

- *CPU-intensive:* This is a function that calculates the trigonometric function *atan* of multiple numbers, hence making it more CPU-heavy function.

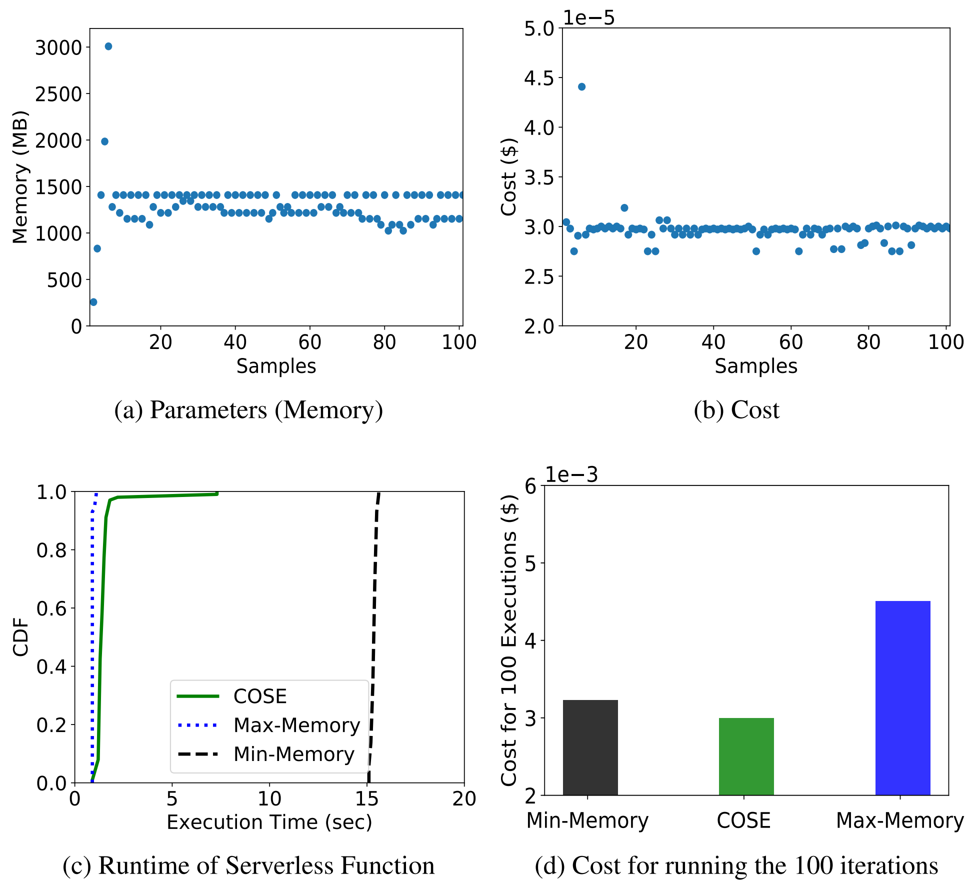
- *Memory-intensive:* This function applies a filter on a large image. This requires extensive use of memory.

- *I/O-intensive:* This function performs multiple I/O related operations on a file, *i.e.,* opening, reading and closing a file.

- *Network-intensive:* This function downloads a large file from a server.

These functions were implemented in Python3.6/3.7 and deployed on AWS Lambda. Each function was deployed as a separate AWS Lambda function. Figure 4.1b shows the runtime for the *CPU-intensive, memory-intensive* and *I/O-intensive* functions under different memory configurations. We do not show the results for the *network-intensive* function since change in memory had little/no impact on the running time of the function. This is because the network resources allocated to a function do not change as we change the memory requested. Figure 4.1c shows the price-memory relation for CPU-, memory- and I/O-intensive functions.

Even though CPU- and I/O-intensive functions do not use more than a certain amount of memory, their performance is affected by the memory requested for the function. The reason for that is, AWS Lambda assigns CPU share to each function in proportion to the memory configured for the function. Hence more memory will assign more CPU cycles to a function.



(a) Parameters (Memory)

(b) Cost

(c) Runtime of Serverless Function

(d) Cost for running the 100 iterations

Figure 4.5: COSE performance on Amazon Lambda for *I/O Intensive* serverless function

4.5.2 Evaluation Results

We ran the *CPU-intensive*, *I/O-intensive* and *memory-intensive* functions shown in Figure 4.1 on Amazon Lambda using COSE. Since the behavior of these functions is very

similar to each other, we show results for only the *I/O-intensive* function here. To get an estimate of the optimal memory value, *i.e.* the memory value that minimizes the price, we ran the serverless functions multiple times across different memory values. As seen in Figure 4.1c, the *I/O-intensive* function has the lowest price in the memory range 900MB-1400MB.

We use COSE to find the optimal configuration for this function (with the goal of minimizing the cost, with no delay requirement on the execution time of the function). For the first few requests, as shown in Figure 4.5a, COSE explores different memory values and once it learns the underlying cost-memory relation, it starts suggesting optimal/near-optimal memory values in the range 900-1400MB. The corresponding cost for individual requests (function invocations) is given in Figure 4.5b. Again, COSE finds the optimal/near-optimal cost for the function ($\$2.9 \times 10^{-5}$ as seen in Figure 4.1c for the *I/O-intensive* function).

To compare COSE with static configurations, we invoked the *I/O-intensive* function 100 times with the maximum and minimum memory values, possible on AWS Lambda, to get the best/worst running times for the function, and also the corresponding cost. Figure 4.5c shows the running time of the serverless function when invoked with configurations picked by COSE, *maximum-memory*=3008MB, and *minimum-memory*=128MB. The *minimum-memory* configuration takes, on average, around 15 seconds to complete a request, while the *maximum-memory* configuration takes, on average, 1 second to complete the request. COSE performance is very close to *maximum-memory*. However, the cost incurred when using COSE is even less than the cost for *minimum-memory*, as shown in Figure 4.5d, due to lower execution time under COSE.

4.6 Evaluation in a Distributed Cloud Environment

While the results of running our framework on AWS Lambda highlight the utility of our COSE framework, evaluating additional aspects of our framework presents a new set of challenges because we have little or no knowledge about the underlying infrastructure, the decisions made by the cloud provider regarding the allocation of resources, which functions are co-located, if the function had cold-start or warm-start, and queuing, propagation and other delays in the system. While it was possible in the previous section to compare the performance of an *I/O-intensive* serverless function in a simple scenario by exhaustively searching the memory space, and finding where the optimal memory value for this function lies, this approach may not be practical for scenarios where we have multiple functions (and possibly chains of functions). To establish the efficacy of our COSE framework, we model a distributed cloud provider and evaluate the framework across a set of multiple functions using extensive simulations. Since in the simulated cloud environment we know the target function that COSE is trying to optimize, we can compare the performance of COSE against the "ground truth".

4.6.1 Modeling Cloud Provider

We model our cloud provider by adopting the following aspects of commercial cloud providers.

i *Co-location*: We modeled the effect of co-location of functions by deploying the open-source serverless platform, Apache OpenWhisk [2], on Chameleon Cloud [5]. We deployed multiple functions on the same machine. The effect of co-location is given in Figure 4.1d. We modeled this in the cloud provider.

ii *Life-time and Cold-start*: If a function is not executed for a certain period of time (i.e.

function *life-time*), the function is evicted by the cloud provider, and the subsequent request for running the function will experience extra delay (i.e. function *cold-start*). We use Amazon Lambda function's *life-time* of 26 minutes and *cold-start* delay of 0.25 seconds, as shown by previous studies [80, 6].

iii *Edge-cloud and core-cloud*: To compare across different cloud providers or different locations provided by one cloud provider, we model two types of clouds, *edge-cloud* and *core-cloud*. We assume that *edge-cloud* is closer to the user and thus has a smaller round trip delay. However, *edge-cloud* is more expensive when compared with the *core-cloud*.

iv *Dynamic serverless function*: To test the adaptive performance of COSE, we run COSE for a dynamic function whose execution time changes over time.

v *Modeling price and execution time of a serverless function*: We develop an analytical model for the cost and execution time based on the experimental results of running these functions on Amazon Lambda. The analytical model is explained in detail in the next section.

4.6.2 Modeling cost and execution time

Our analytical model for cost and execution time of serverless functions is based on AWS Lambda's pricing and execution model.

Cloud provider's pricing model: We use Amazon Lambda's pricing model. Amazon uses a linear pricing model, as shown in Figure 4.1a. We use this pricing model for the *core-cloud*. Since the price for *edge-cloud* is higher than the *core-cloud*, we set the edge resource price to be 1.5 times the price of resources at the *core-cloud*. This linear pricing

model is captured by the following equation for serverless function f:

$$p^f(v,m) = K(v) \times m \tag{4.6}$$

where m is the memory used by function f, $v \in V$ is the cloud provider, and $K(v)$ is a constant and its value depends on the cloud provider's pricing.

Execution time model: The execution time for representative functions is shown in Figure 4.1b for Amazon Lambda. The execution time for these functions follows an exponential decay. In other words, the running time of a function decreases as we increase the memory requested for it. However, after a certain memory size, the change in running time becomes too small or negligible. The execution time for function f is given by:

$$t^f(v,m) = t^f(v,m_{min}) + t^f(v,m_{max}) \times e^{-\lambda(m-m_{min})} + h(v) \tag{4.7}$$

where $t^f(v,m_{min})$ is the running time for function f at the lowest possible memory ($m_{min} = 128MB$ for Amazon Lambda), $t^f(v,m_{max})$ is the running time at the highest possible memory ($m_{max} = 3008MB$ for Amazon Lambda), v is the cloud provider, and λ is the decay constant. By changing $t^f(v,m_{min})$, $t^f(v,m_{max})$ and λ, we can fit the execution model for any serverless function. The constant $h(v)$ captures the delay due to cold-start and co-location, and its value depends on the current state of the cloud provider.

Cost for running a serverless function: The total cost $g^f(v,m)$ for running function f on cloud provider $v \in V$ is given by the product of the price per second ($p^f(v,m)$) and the total execution time ($t^f(v,m)$).

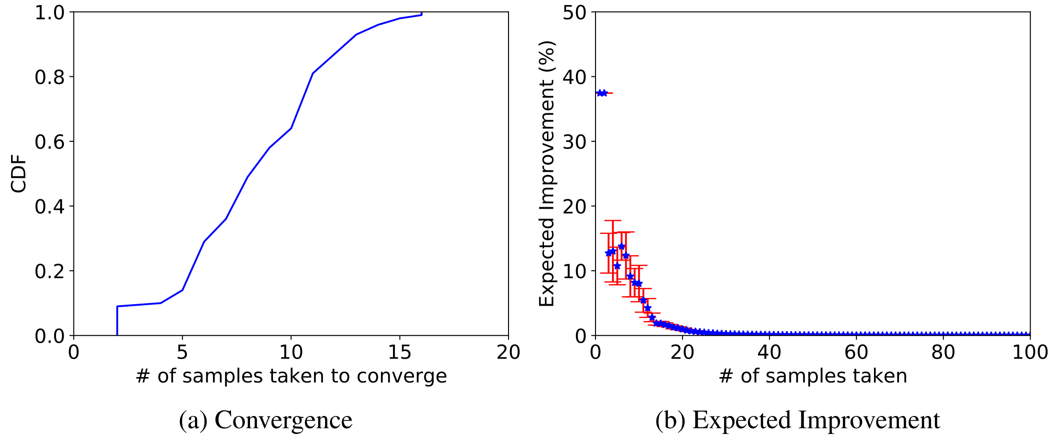
$$g^f(v,m) = p^f(v,m) \times t^f(v,m)$$

4.6.3 Simulation Results

Using the above models allows us to simulate a cloud provider that has two parameters to be optimized, *i.e.* selection of location (edge vs. core) and memory value for deploying a serverless function. Since we have more control over the execution model and resource management, we evaluated the convergence and accuracy-related aspects of COSE. In addition, we evaluated some unique scenarios, for example, dynamically changing the underlying execution model to evaluate how well COSE adapts to changes. Our evaluation showed that COSE can learn the optimal or near-optimal configurations for a serverless function with as few as 13-15 samples and can adapt to changes well[8]. Also, COSE showed significant cost savings without compromising on performance. We provide the simulation parameters at [7].

Convergence: COSE uses Bayesian Optimization (BO) to predict the price function. A small convergence time for BO means that COSE can quickly find the "best" configuration that minimizes the price paid while satisfying the delay constraint. In this experiment, we looked at how long it takes for BO to converge and find the underlying cost-configuration relation. Since we are using a *cloud provider model*, we know the underlying performance function. As explained in Section 4.3.1, we use expected improvement (EI) as the convergence criterion. Figure 4.6a shows the CDF (taken over 100 runs) of the number of configuration samples taken for BO to converge. We observed that BO can converge, 95% of the time, with as few as 15 samples. In Figure 4.6b we show how EI decreases as the number of configuration samples increases. The first few samples have the highest EI value. However, as COSE takes more samples, the EI value rapidly decreases. With each new sample, BO improves its understanding of the underlying performance function and subsequent configuration samples contribute little to improving the prediction.

[8]We note that for a commercial cloud provider with one parameter, COSE was able to find a near-optimal configuration in 5 samples



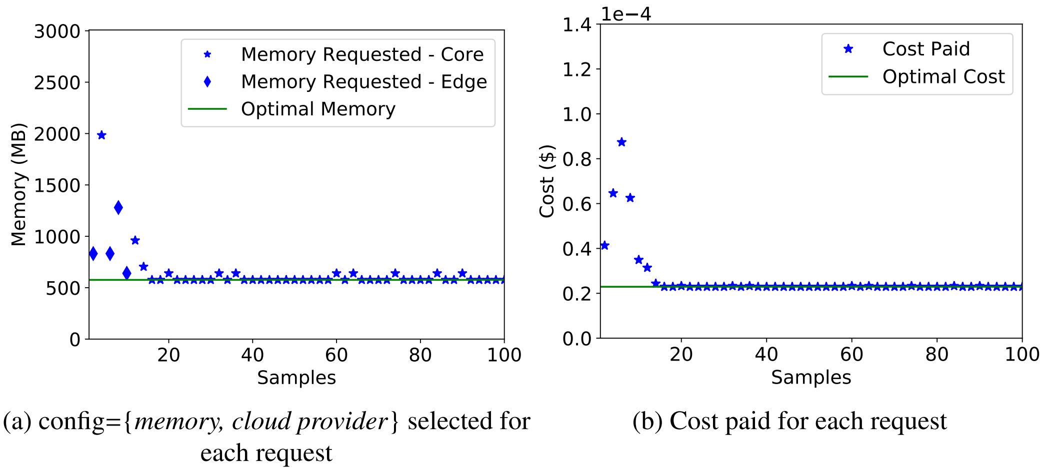
(a) Convergence

(b) Expected Improvement

Figure 4.6: BO's Convergence and EI

Configuration Selection: After BO converges, COSE starts picking the "best" possible configuration for serverless functions using *Config Finder*. We used a function whose execution model had an optimal configuration of {*memory* = *576MB, location* = *core-cloud*}. In Figures 4.7a and 4.7b, we show the configurations that COSE picked for each request and their corresponding cost. For the first few requests (up to 15 requests), COSE is exploring different configurations. After the BO in COSE converges, *Config Finder* starts picking optimal/near-optimal configurations for the function, *i.e.* the corresponding price paid of each serverless request is lowest.

Dynamicity: The performance of a serverless function can be affected by factors like co-location, hardware, resource provisioning policy, *etc*. In case any of these factors changes, the configurations that were optimal before the change may no longer be optimal. We designed COSE so it is resilient in the face of such changes and is able to find new optimal configurations. We tested COSE's ability to adapt to a change in the underlying execution model. We created 500 requests for a serverless function. The first 250 requests follow a certain execution model and have certain optimal configurations. For the next 250 requests, we change the execution model hence the optimal configurations. We observed that



(a) config={*memory, cloud provider*} selected for each request

(b) Cost paid for each request

Figure 4.7: Configuration with COSE

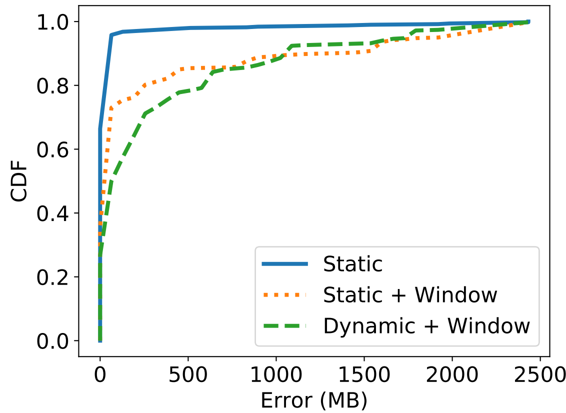


Figure 4.8: COSE for serverless function with changing execution model

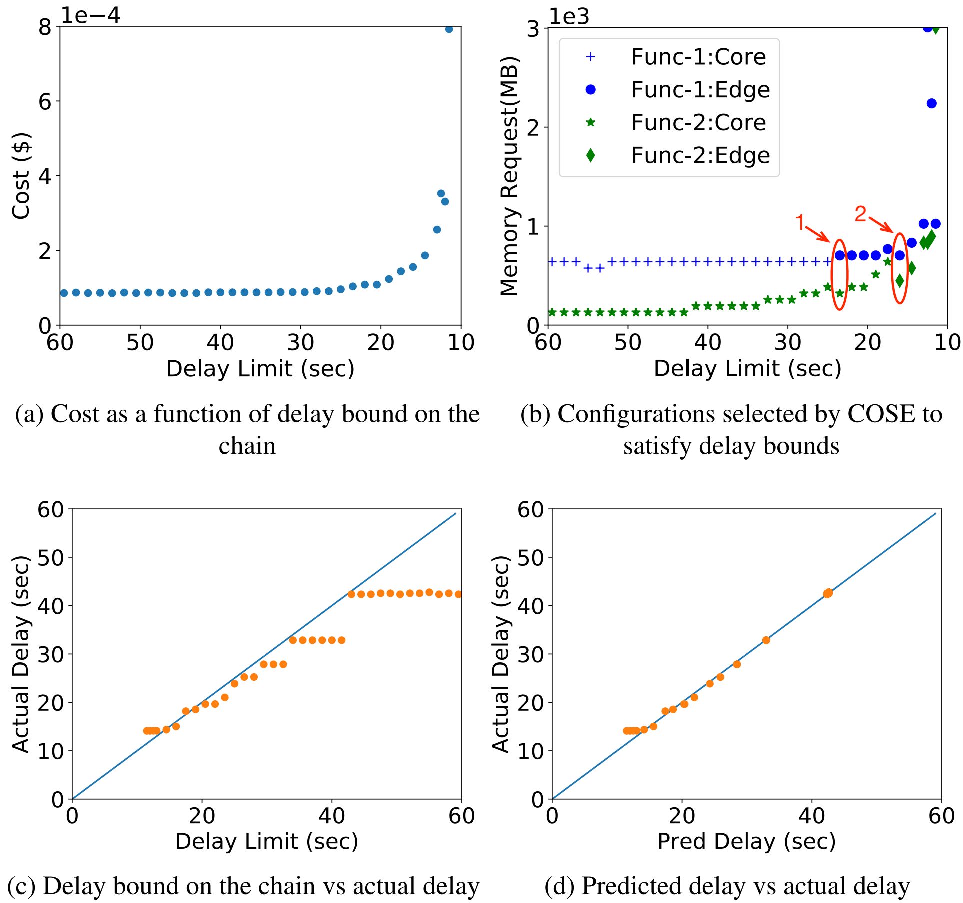
depending on the history, COSE can successfully unlearn the previous execution model, learn the new model, and start predicting the new optimal configurations.

In Figure 4.8, we show the performance of COSE in terms of sample error in memory for three scenarios. The sample error E^s is defined as the difference between the sampled memory and the optimal memory: $E^s = |m^s - m^o|$, where m^s is the memory being sampled by BO and m^o is the optimal memory. In the *static* case, the underlying function's execution model does not change, and COSE remembers the full history, hence it takes a

few samples for COSE to converge and the error is small. In the *static+window* case, the underlying function's execution model does not change, and COSE remembers a limited history (last 40 samples). As BO loses historical data, the acquisition function periodically samples configuration points for exploration, and hence, there is high variability in memory selection leading to the higher error value. In the third (dynamic+window) case, COSE not only remembers a limited history but the serverless function's execution model also changes. We see the highest error in this scenario since COSE is constantly collecting samples to adapt to the changing execution model, and it takes time proportional to the history, to unlearn the previous execution model and learn the new one. That is a trade-off that COSE can make: more history would yield less sampling, but it would be slow to adapt to changes. A shorter history would result in more sampling, but COSE would adapt to changes more quickly. In all three scenarios, COSE converges to optimal/near-optimal values.

Delay Bounded Chaining: Serverless applications can have a chain of functions that triggers one another, where the output of one function serves as input to the next. This is called *service chaining*. Service chains usually have a delay constraint. As an example, think of the login functionality of an application. The login process can comprise of multiple serverless functions, *i.e.* getting user input, retrieving user information from a database, evaluating credentials, and on success, showing appropriate options to the user. To meet certain quality of experience, an application administrator may wish to complete the whole login process within a certain amount of time. As each individual action is a serverless function, picking the right configurations is critical to the performance of the application and the cost of cloud usage. Here we show how COSE successfully finds the optimal configurations for serverless functions comprising a delay-bounded chain.

In this experiment, each request is a service chain consisting of two or more functions.



(a) Cost as a function of delay bound on the chain

(b) Configurations selected by COSE to satisfy delay bounds

(c) Delay bound on the chain vs actual delay

(d) Predicted delay vs actual delay

Figure 4.9: Delay bounded chaining of serverless functions

Each function has a different execution model, hence different optimal configuration. As explained in Section 4.4, COSE uses BO's prediction of price and execution time to formulate the problem as an Integer Linear Program (ILP). COSE solves this ILP to find a suitable configuration for each function in the chain. Initially, we let the BO collect configuration samples and wait for it to converge. Once BO has converged, we observe the "best" configuration selected by COSE for each function in the chain such that the total (end-to-end) delay of the chain satisfies the delay bound. Although we tested COSE for different chain sizes, we show results for the service chain of size two for simplicity.

In Figure 4.9a, we look at how the delay bound affects the cost of cloud-usage. For

loose delay requirement, COSE finds the "best" location and memory for both functions, hence lower cost. As the delay requirement becomes more stringent, COSE has to make a decision of either increasing the memory available to a function or placing it on the *edge-cloud* to reduce the delays. Both of these choices will raise the cost and that is why we see an increase in the cost as the delay bound becomes tighter. In Figure 4.9b, we show the corresponding configurations selected for varying delay bound. Initially, because of higher delay bound, COSE runs both functions on the *core-cloud* to lower the usage cost and selects the memory that lowers the cost. However, as the delay bound becomes smaller, COSE has to increase the memory available to either function or change the location where they are deployed. As the delay bound reduces to 42 seconds, COSE starts increasing the memory available to the second function. At around 24 seconds of delay bound, COSE cannot keep both functions on the *core-cloud* to meet the delay requirement. As shown by arrow 1, COSE moves the first function to the *edge-cloud* and decreases the memory needed for the second function on the *core-cloud*. As the delay bound becomes even smaller, COSE moves both functions to the *edge-cloud* as shown by arrow 2 at around 15 seconds. Since the *edge-cloud* has lower delays, COSE selects smaller memory values, compared to previous values, to minimize the cost while fulfilling the delay requirement.

In Figure 4.9c, we look at the actual delay experienced by the chain. COSE meets the delay requirement of the chain under most delay bounds. When the delay bound is higher than 42 seconds, we do not see an increase in the actual delay experienced by the chain because at the optimal configurations, the chain's total delay is 42 seconds. As explained in Section 4.4, COSE uses its estimation of delays in selecting the configurations of the serverless functions in the chain. It is critical that the predicted delay is close to the actual delay. Figure 4.9d shows that the actual delay experienced by the chain is very close to the delay predicted by COSE.

4.7 Related Work

As serverless computing is gaining popularity, there has been a significant amount of research that measures different aspects of the serverless paradigms [13, 14]. Detailed studies on different commercial serverless platforms aim to characterize and understand the architecture and resource management by the cloud provider [80, 59, 58]. By having a better insight into the cloud provider's serverless platform, users can tailor their applications to efficiently use the cloud provider. In COSE, our focus is on modeling the application behavior at different configurations, regardless of the underlying architecture and resource management scheme used by cloud providers.

Commercial cloud providers have developed systems that suggest suitable configuration parameters to the user for running her tasks. Google provides a machine type recommendation system [15] that helps to maximize the resource utilization of user VM instances. AWS provides auto-scaling service [12] to the users for EC2 instances. Cloud provider's cluster managing systems, such as Google's Borg [78], Mesos [46], Paragon [32] and Quasar [33], allow the user to specify the need for the application and the system finds the best configuration. Currently, cloud providers do not provide a resource configuration facility for serverless computing. Moreover, to port any of these techniques for serverless computing, the user needs the complete knowledge of the underlying cloud infrastructure. Since this information is not available to the user, these techniques cannot be applied by the user for serverless computing.

Systems have been developed for users to infer the cost-performance relationship across different cloud configurations. CherryPick [24] uses Bayesian Optimization to predict suitable VM configuration for an application in a cloud provider. CloudCmp [55] recommends a suitable cloud provider for running user application. Both CherryPick and CloudCmp are offline tools that are helpful to users *before* they deploy their applications.

Ernest [76] builds the performance model of machine learning applications. WebPerf [50] estimates the latency model of a web application. ARIA [77] builds the job profile and performance model for MapReduce and Hadoop applications. Unlike the large body of prior work, which focus on a particular application, the COSE framework can be used for any application running as a serverless function. Moreover, the aforementioned approaches perform resource configuration at the beginning of deployment/execution of an application, while COSE monitors the application continually and adapts its configuration dynamically.

4.8 Summary

In this chapter, we present COSE, a statistical learning based configuration finder for serverless functions. COSE uses Bayesian Optimization to learn the cost and execution time model for serverless functions across unseen configuration values. COSE supports function chaining, and has the ability to dynamically adapt to changes in the execution time of serverless functions. Our results on commercial cloud and simulated distributed cloud environments show that COSE provides optimal/near-optimal configurations for serverless functions in a few configuration samples.

Chapter 5

Conclusion and Research Directions

5.1 Summary of Contributions

In this thesis, we studied the problem of resource orchestration and management of application functions over virtualized cloud infrastructures. Initially, we proposed decomposing applications into application functions that can be placed on the edge network. We also developed a workload model that consists of service chains with varying capacity requirements as the traffic flow traverses its chain. We formulated a binary integer optimization problem whose objective is to minimize the cost of deploying these service chains over the edge network, while satisfying their high throughput and ultra-low latency requirements. We also introduced a fast heuristic to solve the problem. Our extensive evaluations demonstrate the benefits of managing virtual service chains (by distributing them over the edge network) compared to a baseline "middlebox" approach (where all services are run on one host) in terms of overall admissible virtual capacity. Moreover, we observe significant gains when deploying a small number of *mmWave* links that complement the *Wire* physical infrastructure.

Next, we present COSE, a statistical learning based configuration finder for application virtual functions. COSE uses Bayesian Optimization to learn the cost and execution time model for serverless functions across unseen configuration values. COSE supports function chaining, and has the ability to dynamically adapt to changes in the execution time of

serverless functions. Our results on commercial cloud and simulated distributed cloud environments show that COSE provides optimal/near-optimal configurations for serverless functions in a few configuration samples.

5.2 Open Research Directions

Application decomposition and virtual function placement can be used for other envisioned future applications, *e.g.*, healthcare, and the control of robots and drones. We believe this work is a first step toward further analysis and implementation of edge cloud-based applications.

In this work, we showed the benefits of using *mmWave* links. However, finding the optimal placement of *mmWave* links such that the gains are maximized is an exciting problem. Long-range *mmWave* links can be formed by introducing repeater nodes. This is especially helpful for networks where node density is low.

Configuration finder COSE can be used as a service over larger scale multi-cloud providers. This will enable studying a wide range of workloads, application requirements, and cloud resource provisioning and pricing policies. We intend to extend our COSE simulator to accommodate more complex scenarios, such as service graphs. Note that although we did not test COSE for functions with varying input workload, we believe COSE can be used for such scenarios if the input workload can be classified (*e.g.*, based on size) and COSE is trained for each class separately.

Bibliography

[1] Amazon Lambda Pricing Model. https://aws.amazon.com/lambda/pricing/.

[2] Apache OpenWhisk. https://openwhisk.apache.org/.

[3] AWS Lambda Function Configuration. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html.

[4] AWS Lambda Power Tuning. https://github.com/alexcasalboni/aws-lambda-power-tuning.

[5] Chameleon Cloud. https://www.chameleoncloud.org.

[6] Cold Starts in AWS Lambda. https://mikhail.io/serverless/coldstarts/aws/.

[7] COSE Simulation Parameters. https://bit.ly/2HPwY3n.

[8] Google Function Pricing Model. https://cloud.google.com/functions/pricing.

[9] Google's Stadia. http://stadia.com.

[10] LP Relaxation and Rounding. http://pages.cs.wisc.edu/~shuchi/courses/787-F09/scribe-notes/lec10.pdf.

[11] Report ITU-R M.[IMT-2020.TECH PERF REQ] - Minimum requirements related to technical performance for IMT-2020 radio interface(s).

[12] Amazon Auto Scaling. https://aws.amazon.com/ec2/autoscaling/, 2018.

[13] AWS Lambda CPU allocation . https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-with-aws-lambda-41cd44da3cac, 2018.

[14] Chaos of AWS Lambda. https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e, 2018.

[15] Google Cloud Recommendations. https://cloud.google.com/compute/docs/instances/apply-sizing-recommendations-for-instances, 2018.

[16] Serverless Monitoring - Dashbird. `https://dashbird.io/`, 2018.

[17] Serverless Monitoring - SignalFx. `https://www.signalfx.com/solutions/serverless-monitoring/`, 2018.

[18] Serverless Monitoring - Thundra. `https://docs.thundra.io/`, 2018.

[19] AWS Lambda at Edge. `https://aws.amazon.com/lambda/edge/`, 2019.

[20] OpenWhisk at Edge. `https://github.com/kpavel/openwhisk-light`, 2019.

[21] Addis, B., Belabed, D., Bouet, M., and Secci, S. Virtual network functions placement and routing optimization. In *IEEE CloudNet* (2015).

[22] Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I., and Zhou, J. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 21–21.

[23] Akhtar, N., Matta, I., Raza, A., Goratti, L., Braun, T., and Esposito, F. Virtual Function Placement and Traffic Steering over 5G Multi-Technology Networks. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)* (June 2018), pp. 114–122.

[24] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 469–482.

[25] AT&T. Enabling Mobile Augmented and Virtual Reality with 5G Networks.

[26] Azar, Y., Wong, G. N., Wang, K., Mayzus, R., Schulz, J. K., Zhao, H., Gutierrez, F., Hwang, D., and Rappaport, T. S. 28 GHz propagation measurements for outdoor cellular communications using steerable beam antennas in New York city. In *IEEE ICC)* (June 2013).

[27] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[28] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 164–177.

[29] CAI, H., AND CHUNG, T. Improving the Quality of High Dynamic Range Images. *Lighting Research & Technology 43*, 1 (2011), 87–102.

[30] CASTRO, P., ISHAKIAN, V., MUTHUSAMY, V., AND SLOMINSKI, A. Serverless programming (function as a service). In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 2658–2659.

[31] CHOWDHURY, N. M. M. K., RAHMAN, M. R., AND BOUTABA, R. Virtual network embedding with coordinated node and link mapping. In *IEEE INFOCOM 2009* (April 2009), pp. 783–791.

[32] DELIMITROU, C., AND KOZYRAKIS, C. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst. 31*, 4 (Dec. 2013), 12:1–12:34.

[33] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 127–144.

[34] DOCKER. https://www.docker.com/, 2018.

[35] FACEBOOK TERRAGRAPH: SOLVING THE URBAN BANDWIDTH CHALLENGE. https://terragraph.com/.

[36] FAYAZBAKHSH, S. K., SEKAR, V., YU, M., AND MOGUL, J. C. FlowTags: Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions. In *ACM SIGCOMM - HotSDN* (2013).

[37] FISCHER, A., BOTERO, J. F., BECK, M. T., DE MEER, H., AND HESSELBACH, X. Virtual Network Embedding: A Survey. *IEEE Communications Surveys Tutorials 15*, 4 (Fourth 2013), 1888–1906.

[38] FOX, G. C., ISHAKIAN, V., MUTHUSAMY, V., AND SLOMINSKI, A. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028* (2017).

[39] GLIKSON, A., NASTIC, S., AND DUSTDAR, S. Deviceless edge computing: Extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference* (New York, NY, USA, 2017), SYSTOR '17, ACM, pp. 28:1–28:1.

[40] GOOGLE. Google Maps.

[41] GREENGRASS, A. https://aws.amazon.com/greengrass/, 2018.

[42] Guo, L., Pang, J., and Walid, A. Dynamic Service Function Chaining in SDN-enabled networks with middleboxes. In *IEEE ICNP* (2016).

[43] Hahne, E. L., Choudhury, A. K., and Maxemchuk, N. F. Improving the fairness of distributed-queue-dual-bus networks. In *Proceedings. IEEE INFOCOM 90: Ninth Annual Joint Conference of the IEEE Computer and Communications Societies The Multiple Facets of Integration* (June 1990), pp. 175–184 vol.1.

[44] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (Denver, CO, 2016), USENIX Association.

[45] Herodotou, H., Dong, F., and Babu, S. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 18:1–18:14.

[46] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 295–308.

[47] Hu, Y., Patel, M., Sabella, D., Sprecher, N., and Young, V. Mobile edge computing:A key technology towards 5G. *ETSI W. Paper 11* (2015).

[48] Inführ, J., and Raidl, G. R. Introducing the virtual network mapping problem with delay, routing and location constraints. In *Proceedings of the 5th International Conference on Network Optimization* (Berlin, Heidelberg, 2011), INOC'11, Springer-Verlag, pp. 105–117.

[49] Ishakian, V., Muthusamy, V., and Slominski, A. Serving deep learning models in a serverless platform. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on* (2018), IEEE, pp. 257–262.

[50] Jiang, Y., Sivalingam, L. R., Nath, S., and Govindan, R. Webperf: Evaluating what-if scenarios for cloud-hosted web applications. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 258–271.

[51] Jones, D. R., Schonlau, M., and Welch, W. J. Efficient global optimization of expensive black-box functions. *J. of Global Optimization 13*, 4 (Dec. 1998), 455–492.

[52] Jurgelionis, A., Fechteler, P., Eisert, P., Bellotti, F., David, H., Laulajainen, J. P., Carmichael, R., Poulopoulos, V., Laikari, A., Perälä, P., De Gloria, A., and

Bouras, C. Platform for distributed 3d gaming. *Int. J. Comput. Games Technol.* (2009).

[53] Kolliopoulos, S. G., and Stein, C. Improved approximation algorithms for unsplittable flow problems. In *Proceedings 38th Annual Symposium on Foundations of Computer Science* (Oct 1997), pp. 426–436.

[54] Lab, F. C. Facebook demonstrates record-breaking data rate using millimeter-wave technology. https://code.fb.com/connectivity/, Nov 2016.

[55] Li, A., Yang, X., Kandula, S., and Zhang, M. Cloudcmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2010), IMC '10, ACM, pp. 1–14.

[56] Limits, A. L. https://docs.aws.amazon.com/lambda/latest/dg/limits.html, 2018.

[57] Liu, W., Xiang, Y., Ma, S., and Tang, X. Completing virtual network embedding all in one mathematical programming. In *2011 International Conference on Electronics, Communications and Control (ICECC)* (Sep. 2011), pp. 183–185.

[58] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., and Pallickara, S. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)* (April 2018), pp. 159–169.

[59] McGrath, G., and Brenner, P. R. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (June 2017), pp. 405–410.

[60] Medina, A., Lakhina, A., Matta, I., and Byers, J. Brite: An approach to universal topology generation. In *MASCOTS '01* (2001), IEEE Computer Society.

[61] Mehraghdam, S., Keller, M., and Karl, H. Specifying and placing chains of virtual network functions. In *IEEE CloudNet* (2014).

[62] Montero, R. S., Rojas, E., Carrillo, A. A., and Llorente, I. M. Extending the cloud to the network edge. *Computer 50*, 4 (April 2017).

[63] Niu, Y., Li, Y., Jin, D., Su, L., and Vasilakos, A. V. A survey of millimeter wave communications (mmwave) for 5g: opportunities and challenges. *Wireless Networks 21*, 8 (Nov 2015), 2657–2676.

[64] Parsec gaming. https://parsecgaming.com.

[65] Puri, A., and Tripakis, S. *Algorithms for the Multi-constrained Routing Problem.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[66] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *ACM SIGCOMM 2013*.

[67] Qualcomm. Augmented and Virtual Reality: the First Wave of 5G Killer Apps.

[68] Rabbani, M. G., Esteves, R. P., Podlesny, M., Simon, G., Granville, L. Z., and Boutaba, R. On tackling virtual data center embedding problem. In *IFIP/IEEE IM 2013)*, pp. 177–184.

[69] Rangan, S., Rappaport, T. S., and Erkip, E. Millimeter-wave cellular wireless networks: Potentials and challenges. *Proceedings of the IEEE 102*, 3 (March 2014), 366–385.

[70] Rasmussen, C. E., and Williams, C. K. I. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning).* The MIT Press, 2005.

[71] Schrijver, A. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[72] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. Making middleboxes someone else's problem: network processing as a cloud service. In *SIGCOMM '12, Finland* (2012), pp. 13–24.

[73] Singh, S., Kulkarni, M. N., Ghosh, A., and Andrews, J. G. Tractable model for rate in self-backhauled millimeter wave cellular networks. *IEEE JSAC 33* (Oct 2015).

[74] Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2* (USA, 2012), NIPS'12, Curran Associates Inc., pp. 2951–2959.

[75] Trinh, T., Esaki, H., and Aswakul, C. Quality of service using careful overbooking for optimal virtual network resource allocation. In *The 8th Electrical Engineering/ Electronics, Computer, Telecommunications and Information Technology (ECTI) Association of Thailand - Conference 2011* (May 2011), pp. 296–299.

[76] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 363–378.

[77] Verma, A., Cherkasova, L., and Campbell, R. H. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing* (New York, NY, USA, 2011), ICAC '11, ACM, pp. 235–244.

[78] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 18:1–18:17.

[79] Wang, J., Pan, J., Esposito, F., Calyam, P., Yang, Z., and Mohapatra, P. Edge Cloud Offloading Algorithms: Issues, Methods, and Perspectives. *ACM Computing Surveys PP* (Oct 2018).

[80] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association, pp. 133–146.

[81] Waxman, B. M. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications 6* (Dec 1988), 1617–1622.

[82] Weib, M., Huchard, M., Stohr, A., Charbonnier, B., Fedderwitz, S., and Jager, D. S. 60-GHz Photonic Millimeter-Wave Link for Short- to Medium-Range Wireless Transmission Up to 12.5 Gb/s. *Journal of Lightwave Technology 26*, 15 (Aug 2008), 2424–2429.

[83] Wolsey, L. A. *Integer Programming*. John Wiley & Sons, 1998.

[84] Yu, M., Yi, Y., Rexford, J., and Chiang, M. Rethinking virtual network embedding: Substrate support for path splitting and migration. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 17–29.

[85] Zhang, Y., Beheshti, N., Beliveau, L., Lefebvre, G., Manghirmalani, R., Mishra, R., Patneyt, R., Shirazipour, M., Subrahmaniam, R., Truchan, C., and Tatipamula, M. StEERING: A software-defined networking for inline service chaining. In *IEEE ICNP* (2013).

Curriculum Vitae