System Infrastructure for Mobile-Cloud Convergence

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Kiryong Ha

M.S., BioSystems Department, KAIST
B.S., Electrical Engineering and Computer Science, KAIST

Carnegie Mellon University
Pittsburgh, PA

December 2016

Keywords: Mobile Computing, Cloud Computing, Cloudlets, Edge Computing, Mobile Edge Computing, Virtual Machines, VM, Provisioning, VM Provisioning, VM Hand-off, migration, live migration, Offloading, Augmented Reality, Virtual Reality, Wearable Cognitive Assistance, Cloudlet Discovery, OpenStack

To my wife Dana
and my children Lynn and Joon

Acknowledgements

I would like to first thank my advisor Prof. Mahadev Satyanarayanan. I was truly fortunate to be advised by him. When I was an undergrad student in Korea, I dreamed of becoming a computer scientist reading his papers. So, it has been a great honor to work with him and to be guided by him. I have been sincerely inspired by his research insight and constant encouragement throughout the Ph.D. program. He has been not only a great academic advisor but also a wonderful life mentor. Also, I would like to thank my thesis committee members, Todd C. Mowry, Daniel P. Siewiorek, and Padmanabhan (Babu) Pillai. Todd has provided constructive feedback for my qualifying exam and my thesis. And it was lucky to work with Dan meeting him every week at the Elijah meeting and to serve as a teaching assistance for his class. Babu has been almost my co-advisor during my entire Ph.D. course. I really learned a lot from him about how to approach research problems and how to think like a researcher. It was very fortunate that I could have him as a mentor.

Interaction with many great people along the way have shaped me, honed my skill, and grown me as a researcher. It has been a great opportunity to be a member of Satya research group. Thanks especially to Yoshihisa Abe, Brandon Amos, Zhuo Chen, Tom Eiszler, Ziqiang Feng, Benjamin Gilbert, Jan Harkes, Wenlu Hu, Junjue Wang, and Wolfgang Richter. I enjoyed arguing, discussing, and brainstorming with all of them. Also, I loved the constructive feedback and the collaborative working culture of the group. I want to thank Chase Klingensmith for the group administrative assistance.

Further, it has been fortunate to collaborate with many wonderful colleagues and researchers throughout my Ph.D. courses. Thank you, Yuvraj Agarwal, Alok Shankar, Ishan Misra, Da-Yoon Chung, Changsoo Lee, Brandon Tyler, Ardalan Amiri Sani, Nigel Davies, Pieter Simoens, Yu Xiao, Doyeop Kim, YongJune Kim, Sangkil Cha, Minsuk Kang, Minhee Jeon, Hyoseung Kim, Dongyeop Kang, Dong-Bae Jun, HongJai Cho, Chansik Kim, Parag Dixit, Brandon Tyler, Grace Lewis, Soumya Simanta, Edwin Morris, Jeff Boleng, Rolf Schuster, Guenter Klas, Valerie Young, Lenin Ravindranath Sivalinga, Alec Wolfman, Sharad Agarwal, David Chu, Eduardo Cuervo, Victor Bahl, Taewook Oh, Sonia Bandersnatch, Kaushik Veeraraghavan, and Wonho Kim. I truly had a great time interacting and collaborating with these excellent people.

Last, but not the least, I would like to thank my family including my parents and parents-in-law. I could have gone through this long journey with their supports. And my lovely Lynn and Joon are the sole motivation for holding me to the highest of standards. Finally, my wife, Dana Kim, has been my closest friend and mentor. I would have never completed this without her. I am sincerely grateful to her for being with me and for keeping me honest and humble.

My research was primarily supported by graduate fellowships from Korean Foundation For Advanced Studies (KFAS) and Facebook. Additional support for this research was provided by the National Science Foundation (NSF) under grant numbers IIS-1065336 and CNS-1518865, Intel, Google, Vodafone, Verizon, Crown Castle, NVDIA, and the Conklin Kistler family fund. Any opinions, findings, conclusions or recommendations expressed in this dissertation are those of the author and should not be attributed to Carnegie Mellon University or the funding sources.

Abstract

The convergence of mobile computing and cloud computing enables new mobile applications that are both resource-intensive and interactive. For these applications, end-to-end network bandwidth and latency matter greatly when cloud resources are used to augment the computational power and battery life of a mobile device. This dissertation designs and implements a new architectural element called *a cloudlet*, that arises from the convergence of mobile computing and cloud computing. Cloudlets represent the middle tier of a 3-tier hierarchy, mobile device — cloudlet — cloud, to achieve the right balance between cloud consolidation and network responsiveness. We first present quantitative evidence that shows cloud location can affect the performance of mobile applications and cloud consolidation. We then describe an architectural solution using cloudlets that are a seamless extension of todays cloud computing infrastructure. Finally, we define minimal functionalities that cloudlets must offer above/beyond standard cloud computing, and address corresponding technical challenges.

Contents

List of Figures

Chapter 1

Introduction

The convergence of cloud computing and mobile computing has begun. Apple's *Siri* [8], which performs compute-intensive speech recognition in the cloud, hints at the rich commercial opportunities in this emerging space. On the user end, mobile devices are becoming smaller and smaller as a form of wearable device [43]. At the other end in the back-end server, cloud computing provides nearly infinite computing resources and scalability to mobile applications. Through context-aware real-time scene interpretation (including recognition of objects, faces, activities, signage text, and sounds), we can imagine new mobile applications that offer helpful guidance for everyday life much beyond what today's Siri can offer. These applications will be interactive and resource-intensive leveraging the power of the cloud.

However, one of the critical challenges in cloud-backed mobile computing is the end-to-end network responsiveness between the mobile device and associated cloud. When the use of cloud resources is in the critical path of user interaction, operation latencies can be no more than a few tens of milliseconds. Violating this bound results in distraction and annoyance to a mobile user who is already attention-challenged [5, 30]. Such fine-grained cloud usage is different from the coarse-grained usage models and SLA guarantees that dominate cloud computing today.

In fact, *centralization* in today's cloud computing makes it much harder to support that fine-grained cloud usage. This is reflected in the consolidation of compute capacity into a few large data centers. For example, Amazon Web Services spans the entire planet with just a handful of data centers located. The underlying value proposition is that centralization exploits economies of scale to lower the marginal cost of system administration and operations. These economies of scale evaporate if too many data centers have to be maintained and administered. But the aggressive global consolidation of data centers leads to a large separation between a mobile device and its cloud. End-to-end communication then involves many network hops and results in high latencies and low bandwidth. Limiting consolidation and locating small data centers much closer to mobile devices would solve this problem, but it would sacrifice the key benefit of

cloud computing. How do we achieve the right balance? Can we support latency-sensitive and resource-intensive mobile applications without sacrificing the benefits of cloud computing?

1.1 Thesis Statement

In this thesis, we design and implement a new architectural element called *a cloudlet*, that arises from the convergence of mobile computing and cloud computing. Cloudlets represent the middle tier of a 3-tier hierarchy, mobile device — cloudlet — cloud, to achieve the right balance between cloud consolidation and responsiveness. Specifically, we claim that:

Emerging mobile applications that are interactive and resource-intensive can be effectively supported by mobility-enhanced small-scale cloud datacenters called cloudlets that are located at the edge of the Internet.

The main contributions of this thesis are as follows:

1. We provide a measurement-driven quantitative analysis of emerging mobile applications, and show how cloudlets can help them.
2. We propose a cloudlet-based two-level cloud computing architecture that seamlessly extends today's cloud infrastructure.
3. We identify a set of minimal functionalities that cloudlets must offer above/beyond standard cloud computing, and address technical challenges in implementing them.

1.2 Thesis Overview

The remainder of this dissertation is organized as follows:

- In Chapter 2, we motivate the necessity of cloudlets by presenting quantitative evidence from a suite of five representative mobile applications. We provide quantitative analysis of the benefits of cloudlets by comparing performance of the application with Amazon AWS cloud.
- In Chapter 3, we show how a two-level architecture can achieve the right balance between cloud consolidation and network requirements. In this chapter, we propose three characteristics unique to cloudlets due to its decentralized architecture and identify three technical challenges; rapid provisioning, state handoff across cloudlets, and cloudlet discovery.
- In Chapter 4, we address the technical challenges of cloudlet provisioning. Since a mobile application relies on precisely-configured back-end server, it is difficult to support at global

scale across cloudlets in multiple domains. To address this problem, we describe just-in-time (JIT) provisioning of cloudlets using VMs. We introduce a technique called dynamic VM synthesis and apply a series of optimizations to aggressively reduce transfer cost and startup latency.

- In Chapter 5, we present an adaptive virtual machine handoff system to support user mobility in cloudlet context. We propose VM handoff as a technique for seamlessly transferring VM-encapsulated execution to a more optimal offload site as users move. In this work, we highlight the need for VM handoff to dynamically adapt to changing network condition and processing capacity.
- In Chapter 6, we explain how we discover and select a cloudlet for a mobile application. Because cloudlets are small data centers distributed at the edge of the Internet, a mobile device first has to discover, select and associate with the appropriate cloudlet among multiple candidates. We present our design choices in this resource discovery problem and introduce three important attributes of discovery in a cloudlet context.
- In Chapter 7, we present our efforts toward deploying cloudlet infrastructure. Since a cloudlet model requires additional deployment of hardware/software, it is important to provide a systematic way to incentivise the deployment. In this chapter, we show how we can we bootstrap the cloudlet deployment using an existing open eco-system.
- Finally, in Chapter 8, we conclude the dissertation and explain future work with a summary of contributions.

Chapter 2

Motivation

In mobile context, interactive and resource-intensive applications are emerging. Apple's *Siri* for the iPhone [8], which performs compute-intensive speech recognition in the cloud, hints at the rich commercial opportunities in this emerging space. Rapid improvements in sensing, display quality, connectivity, and computational capacity of mobile devices will lead to new cloud-enabled mobile applications that embody voice-, image-, motion- and location-based interactivity.

These new applications are pushing well beyond the processing, storage, and energy limits of mobile devices. When their use of cloud resources is in the critical path of user interaction, end-to-end operation latencies can be no more than a few tens of milliseconds. Violating this bound results in distraction and annoyance to a mobile user who is already attention-challenged. Such fine-grained cloud usage is different from the coarse-grained usage models and SLA guarantees that dominate cloud computing today.

In this chapter, we will provide the experimental evidence that these new applications force a fundamental change in cloud computing architecture. We describe five example applications of this genre in Section 2.1, and experimentally demonstrate in Section 2.2 that even with the rapid improvements predicted for mobile computing hardware, such applications will benefit from cloud resources. The remainder of the chapter explores the architectural implications of this class of applications. In the past, *centralization* was the dominant theme of cloud computing. This is reflected in the consolidation of dispersed compute capacity into a few large data centers. Aggressive global consolidation of data centers implies large average separation between a mobile device and its cloud. End-to-end communication then involves many network hops and results in high latencies. Section 2.3 quantifies this point using measurements from Amazon EC2. Under these conditions, achieving crisp interactive response for latency-sensitive mobile applications will be a challenge.

2.1 Example Mobile Applications

Beyond today's familiar desktop, laptop, and smartphone applications is a new genre of software to seamlessly augment human perception and cognition. Consider Watson, IBM's question-answering technology that publicly demonstrated its prowess in 2011 [108]. Imagine such a tool being available anywhere and anytime to rapidly respond to urgent questions posed by an attention-challenged mobile user. Such a vision may be within reach in the next decade. Free-form speech recognition, natural language translation, face recognition, object recognition, dynamic action interpretation from video, and body language interpretation are other examples of this genre of futuristic applications. Although a full-fledged cognitive assistance system is out of reach today, we investigate several smaller applications that are building blocks towards this vision. Five such applications are described below.

Face Recognition (FACE)

A most basic and fundamental perception task is the recognition of human faces. The problem has been long studied in the computer vision community, and fast algorithms for detecting human faces in images have been available for some time [111]. Identification of individuals through computer vision is still an area of active research, spurred by applications in security and surveillance tasks. However, such technology is also very useful in mobile devices for personal information management and cognitive assistance. For example, an application that can recognize a face and remind you who it is (by name, contact information, or context in which you last met) can be quite useful to everyone, and invaluable to those with cognitive or visual impairments. Such an application is most useful if it can be used anywhere, and can quickly provide a response to avoid potentially awkward social situations.

The face recognition application studied here detects faces in an image, and attempts to identify the face from a prepopulated database. The application uses a Haar Cascade of classifiers to do the detection, and then uses the Eigenfaces method [109] based on principal component analysis (PCA) to make an identification. The implementation is based on OpenCV [78] image processing and computer vision routines, and runs on a Microsoft Windows environment. Training the classifiers and populating the database are done offline, so our experiments only consider the execution time of the recognition task on a pre-trained system.

Speech Recognition (SPEECH)

Speech as a modality of interaction between human users and computers is a long studied area of research. Most success has been in very specific domains or in applications requiring a very limited vocabulary, such as interactive voice response in phone answering services, and hands-

free, in-vehicle control of cell phones. Several recent commercial efforts aim for general purpose information query, device control, and language translation using speech input on mobile devices [8, 55, 112].

The speech recognition application considered here is based on an open-source speech-to-text framework based on Hidden Markov Model (HMM) recognition systems [104]. It takes as input digitized audio of a spoken English sentence, and attempts to extract all of the words in plain text format. This application is single-threaded. Since it is written in Java, it can run on both Linux and Microsoft Windows. For this thesis, we ran it on Linux.

Object and Pose Identification (OBJECT)

A third application is based on a computer vision algorithm originally developed for robotics [105], but modified for use by handicapped users. The computer vision system identifies known objects, and importantly, also recognizes the position and orientation of the objects relative to the user. This information is then used to guide the user in manipulating a particular object.

Here, the application identifies and locates known objects in a scene. The implementation runs on Linux, and makes use of multiple cores. The system extracts key visual elements (SIFT features [67]) from an image, matches these against a database of features from a known set of objects, and finally performs geometric computations to determine the pose of the identified object. For the experiments in this chapter, the database is populated with thousands of features extracted from more than 500 images of 13 different objects.

Mobile Augmented Reality (AR)

The defining property of a mobile augmented reality application is the display of timely and relevant information as an overlay on top of a live view of some scene. For example, it may show street names, restaurant ratings or directional arrows overlaid on the scene captured through a smartphone's camera. Special mobile devices that incorporate cameras and see-through displays in a wearable eye-glasses form factor [38] can be used instead of a smartphone.

AR uses computer vision to identify actual buildings and landmarks in a scene, and label them precisely in the view [107]. This is akin to an image-based query in Google Goggles [40], but running continuously on a live video stream. AR extracts a set of features from the scene image, and uses the feature descriptors to find similar-looking entries in a database constructed using features from labeled images of known landmarks and buildings. The database search is kept tractable by spatially indexing the data by geographic locations, and limiting search to a slice of the database relevant to the current GPS coordinates. The prototype application uses a dataset of 1005 labeled images of 200 buildings as the relevant database slice. AR runs on

Application	Average request size	Response size
FACE	62 KB	$<$ 60 bytes
SPEECH	243 KB	$<$ 50 bytes
OBJECT	73 KB	$<$ 50 bytes
AR	26 KB	$<$ 20 bytes
FLUID	16 bytes	25 KB

Figure 2.1: Average request & response size of each application

Microsoft Windows, and makes significant use of OpenCV libraries [78], Intel Performance Primitives (IPP) libraries, and multiple processing threads.

Physical Simulation and Rendering (FLUID)

Our final application is used in computer graphics. Using accelerometer readings from a mobile device, it physically models the motion of imaginary fluids with which the user can interact. For example, it can show liquid sloshing around in a container depicted on a smartphone screen, such as a glass of water carried by the user as he walks or runs. The application backend runs a physics simulation, based on the predictive-corrective incompressible smoothed particles hydrodynamics (PCISPH) method [103]. We note that the computational structure of this application is representative of many other interactive applications, particularly "real-time" (i.e., not turn-based) games.

FLUID is implemented as a multithreaded Linux application. To ensure a good interactive experience, the delay between user input and output state change has to be very low, on the order of 100ms. In our experiments, FLUID simulates a 2218 particle system with 20 ms timesteps, generating up to 50 frames per second.

Figure 2.1 shows average request and response sizes for each application. All applications send requests with input data from the mobile device and receive back computed results based on the inputs. The average request size is tens of kilobyte for a captured image and several hundreds kilobytes for a recorded speech input. The response size is typically less than 100 bytes as the returned results are simple text strings. In the FLUID application, however, the requests are streams of sensed motion information using accelerometer data, so each request is just a few bytes. The response data is the state of the simulated world, so unlike the other applications, the responses here are much larger than the requests.

	Typical Server		Typical Handheld	
Year	Processor	Speed	Device	Speed
1997	Pentium® II	266 MHz	Palm Pilot	16 MHz
2002	Itanium®	1 GHz	Blackberry 5810	133 MHz
2007	Intel® Core™ 2	9.6 GHz (4 cores)	Apple	412 MHz iPhone
2011	Intel® Xeon® X5	32 GHz (2x6 cores)	Samsung Galaxy S2	2.4 GHz (2 cores)

Figure 2.2: Evolution of Hardware Performance (adapted from Flinn [31])

	Dell Latitude 2102	Samsung Galaxy S2
CPU	Intel® Atom™ N550	ARM Cortex-A9
	1.5 GHz per core, 2 cores (4 threads)	1.2 GHz per core, 2 cores
RAM	2 GB	1 GB
Storage	320 GB	16 GB
OS	Linux, Windows	Android

Figure 2.3: Dell Netbook Device Used in Experiments

2.2 Why Cloud Resources are Necessary

2.2.1 Mobile Hardware Performance

Handheld or body-worn mobile devices are always resource-poor relative to server hardware of comparable vintage [94]. Figure 2.2, adapted from Flinn [31], illustrates the consistent large gap in the processing power of typical server and mobile device hardware over a 15-year period. This stubborn gap reflects a fundamental reality of user preferences: Moore's Law has to be leveraged differently on hardware that people carry or wear for extended periods of time. This is not just a temporary limitation of current mobile hardware technology, but is intrinsic to mobility. The most sought-after features of a mobile device always include light weight, small size, long battery life, comfortable ergonomics, and tolerable heat dissipation. Processor speed, memory size, and disk capacity are secondary.

All the experiments in this chapter use a Dell Latitude 2102 as the mobile device. This small netbook machine is more powerful than a typical smartphone today (Figure 2.3), but it is representative of mobile devices in the near future.

2.2.2 Extremes of Resource Demands

At first glance, it may appear that today's smartphones are already powerful enough to support mobile multimedia applications without leveraging cloud resources. Some digital cameras and smartphones support built-in face detection. Android 4.0 APIs support tracking of multiple faces and give detailed information about the location of eyes and mouth [80]. Google's "Voice Ac-

Application	Condition 1	Condition 2	Condition 3
SPEECH	0.057 s	1.04 s	4.08 s
FACE	0.30 s	3.92 s	N/A

Figure 2.4: Average response time of applications on mobile device under different conditions (see Sect. 2.2.2)

tions for Android" performs voice recognition to allow hands-free control of a smartphone [42]. Lowe [66] describes many computer vision applications that run on mobile devices today.

However, upon closer examination, the situation is much more complex and subtle. Consider computer vision, for example. Its computational requirements vary drastically depending on the operational conditions. For example, it is possible to develop (near) frame-rate object recognition (including face recognition [84]) operating on mobile computers *if* we assume restricted operational conditions such as a small number of models (*e.g.*, small number of identities for person recognition), and limited variability in observation conditions (*e.g.*, frontal faces only). The computational demands greatly increase with the generality of the problem formulation. For example, just two simple changes make a huge difference: increasing the number of possible faces from just a few close acquaintances to the entire set of people known to have entered a building, and reducing the constraints on the observation conditions by allowing faces to be at arbitrary viewpoints from the observer.

To illustrate the great variability of execution times possible with perception applications, we perform a set of experiments using two of the applications discussed earlier. We run the SPEECH and FACE applications on the mobile platform, and measure the response times for a wide variety of inputs. Figure 2.4 shows the results. For the speech application, execution times generally increase with the number of words the algorithm recognizes (correctly or otherwise) in an utterance. Conditions 1, 2, 3 for this application correspond to sentences in which no words, 1–5 words, and 6–22 words are recognized, respectively. The response time varies quite dramatically, by almost 2 orders of magnitude, and is acceptable only when the application fails to recognize any words. When short phrases are correctly recognized, the response time is marginal, at just over 1 second, on average. For longer sentences, when the application works at all, it just takes too long. For comparison, Agus et al. [5] report that human subjects recognize short target phrases within 300 to 450 ms, and are able to tell that a sound is a human voice within a mere 4 ms.

In the case of the face recognition application, the best response times occur when there is a single, large, recognizable face in the image. These correspond to Condition 1 in Figure 2.4. It fares the worst when it searches in vain at smaller and smaller scales for a face in an image without any faces (Condition 2). Unfortunately, response time is close to the latter for images that only contain small faces. At close to 4-second average response time in these conditions,

Application	No Cloud median	No Cloud 99%	With Cloud median	With Cloud 99%
SPEECH	1.22 s	6.69 s	0.23 s	1.25 s
FACE	0.42 s	4.12 s	0.16 s	1.47 s

Figure 2.5: Response times with and without cloud resources.

	Ideal	Measured on campus					Measured off campus				
EC2 site	Latency (ms)	BW to/from Cloud (Mbps) Day 1	Day 2	Day 3	Latency (ms) median.	90%	BW to/from Cloud (Mbps) Day 1	Day 2	Day 3	Latency (ms) median.	90%
East	1.8	28 / 34	42 / 34	20 / 15	9.2	12.4	5.1 / 13.7	5.1 / 14.2	5.1 / 13.4	17.9	21.3
West	24.2	12 / 14	20 / 18	11 / 2.5	92.1	95.5	5.0 / 13.9	5.1 / 13.6	4.9 / 13.4	90.3	93.8
EU	36.8	3.6 / 0.9	13 / 0.4	7.6 / 0.9	99.3	103.0	4.9 / 13.8	5.0 / 11.8	4.8 / 13.3	112	115
Asia	102.5	10 / 0.5	2.4 / 0.2	3.0 / 0.4	265	272	4.6 / 9.4	4.6 / 9.2	4.4 / 9.7	277	286

Figure 2.6: Measured Network Quality to Amazon EC2 Sites from Carnegie Mellon University (Pittsburgh, PA) ("Ideal" is at speed of light)

this application is unacceptably slow. For comparison, recent experimental results on human subjects by Ramon et al. [87] show that recognition times under controlled conditions range from 370 milliseconds for the fastest responses on familiar faces to 620 milliseconds for the slowest response on an unfamiliar face. Lewis et al. [64] report that human subjects take less than 700 milliseconds to determine the absence of faces in a scene, even under hostile conditions such as low lighting and deliberately distorted optics.

Such data-dependent and context-dependent tradeoffs apply across the board to virtually all applications of this genre. In continuous use under the widest possible range of operating conditions, providing near real-time responses, and tuned for very low error rates, these applications have ravenous appetites for processing, memory and energy resources. They can easily overwhelm a mobile device.

2.2.3 Improvement from Cloud Computing

Performance improves considerably when cloud resources are leveraged. Figure 2.5 shows the median and 99th percentile response times for the SPEECH and FACE experiments of Figure 2.4 with and without use of cloud resources. For the speech case, we leverage an Amazon EC2 instance. For the face recognition application, we use a private cloud. Although variability in execution times still exists, the absolute response times are significantly improved. These experiments confirm that leveraging cloud resources can improve user experience for our example applications.

	Ideal	Measured on campus					Measured off campus				
EC2	Latency	BW to/from Cloud (Mbps)			Latency (ms)		BW to/from Cloud (Mbps)			Latency (ms)	
site	(ms)	Day 1	Day 2	Day 3	median.	90%	Day 1	Day 2	Day 3	median.	90%
East	38.5	4.7 / 5.2	4.7 / 5.2	5.6 / 5.5	89.4	101	0.8 / 3.3	1.9 / 4.7	0.6 / 3.1	106	123
West	54.3	5.4 / 3.5	5.4 / 3.5	3.6 / 3.6	159	208	0.5 / 2.4	1.4 / 2.8	0.7 / 2.6	182	201
EU	1.7	6.7 / 10.4	6.7 / 10.4	8.0 / 10.5	32.7	63.4	1.7 / 9.4	2.5 / 14.5	1.4 / 7.6	43.6	64
Asia	73.2	4.7 / 2.6	4.7 / 2.6	6.2 / 2.7	279	325	0.3 / 1.6	1.4 / 1.9	0.5 / 1.6	272	291

Figure 2.7: Measured Network Quality to Amazon EC2 Sites from Lancaster University (Lancaster, UK) ("Ideal" is at speed of light)

2.3 Effects of Cloud Location

In reality, "the cloud" is an abstraction that maps to services in sparsely scattered data centers across the globe. As a user travels, his mobile device experiences high variability in the end-to-end network latency and bandwidth to these data centers. We examine the significance of this variability for mobile multimedia applications. Response time for remote operations is our primary metric. Energy consumed on the mobile device is a secondary metric. Application-specific metrics such as frame rate are also relevant.

2.3.1 Variable Network Quality to the Cloud

In this chapter, we focus on Amazon EC2 services provided by several data centers worldwide. We use the labels "East," "West," "EU," and "Asia" to refer to the data centers located in Virginia, Oregon, Ireland and Singapore. We measured end-to-end latency and bandwidth to these data centers from a WiFi-connected mobile device located on our campuses in Pittsburgh, PA and Lancaster, UK. We also repeated these measurements from off-campus sites with excellent last-mile connectivity in these two cities. Figure 2.6 and 2.7 present our measurements, and quantify our intuition that a traveling user will experience highly variable cloud connectivity. There are also some surprises in the data.

One surprise is the amazingly good connectivity to EC2 East from our Pittsburgh, PA campus. From a wired connection, we measured 8 ms ping times and 200 Mbps transfer rates to this site. Such numbers are more typical of LAN connections than WAN transfers! We believe that this is due to particularly favorable network routing between our campus and the EC2 East site. This hypothesis is confirmed by the poorer off-campus measurements shown in Figure 2.6. Thus, our EC2 East on-campus results best serve to indicate what one can expect from a LAN-connected private cloud. Li et al. [65] report that average round trip time (RTT) from 260 global vantage points to their optimal Amazon EC2 instances is 73.68 ms. Therefore, the EC2 West numbers in

Figure 2.6 are more typical of cloud connectivity.

Another surprise is the great range of bandwidths observed, particularly the upload/download asymmetry and the significant variation between experiments. To mitigate this time-varying factor, we scheduled our experiments on weekday nights when conditions were stable and bandwidth consistently high. All experiments in the rest of the chapter were run under these conditions on campus in Pittsburgh.

2.3.2 Impact on Response Time

We next evaluate how cloud connectivity affects the applications described in Section 2.1. We consider six cases. The first, labeled "Mobile," runs the application entirely on the mobile device. Cloud connectivity is irrelevant, but the resource constraints of the mobile device dominate. In four other cases, the mobile device performs the resource-intensive part of each operation on one of the four Amazon data centers and blocks until it receives the result.

The sixth case, labeled "Cloudlet," corresponds to the theoretical best-case for data center location. With today's deployed wireless technology, this is exactly one WiFi hop away from a mobile device. This can only be approximated today in special situations: e.g., on a WiFi-covered campus, with access points connected to a private data center through a lightly-loaded gigabit LAN backbone. If naively implemented at global scale, Cloudlet would lead to a proliferation of data centers. Chapter 3 discusses how the consolidation benefits of cloud computing can be preserved while scaling out the cloudlet configuration.

Figure 2.8 compares the characteristics of the compute platforms used in our configurations. For Cloudlet, we create a minimal data center using a six-year old WiFi-connected server. The choice of this near-obsolete machine is deliberate. By comparing it against a fast mobile device and fast EC2 cloud instances, we have deliberately stacked the deck against Cloudlet. Hence, any wins by this strategy in our experiments should be considered quite meaningful.

FACE Figure 2.9 summarizes the response times measured for FACE under different conditions. Here, we test with 300 images that may have known faces, unknown faces, or no faces at all. Processing on the mobile device alone can provide tolerable response times for the easier images,

	Mobile	Cloudlet	Cloud (East, West, EU, Asia)
CPU	Intel® Atom™ N550 1.5 GHz, 2 cores, 4 threads	Intel® Xeon® E5320 1.86 GHz, 4 cores	Amazon X-Large Instance 20 Compute Units, 8 virtual cores
RAM	2 GB	4 GB	7 GB
VMM	none	KVM	Xen,VMware

Figure 2.8: Platform specifications

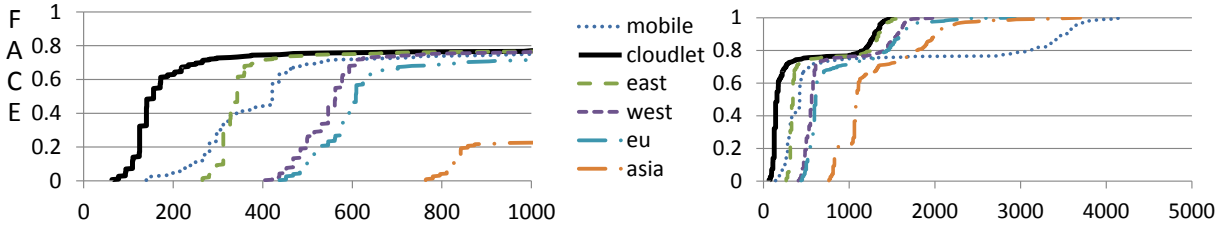


Figure 2.9: FACE: Cumulative distribution function (CDF) of response times in ms (300 images).

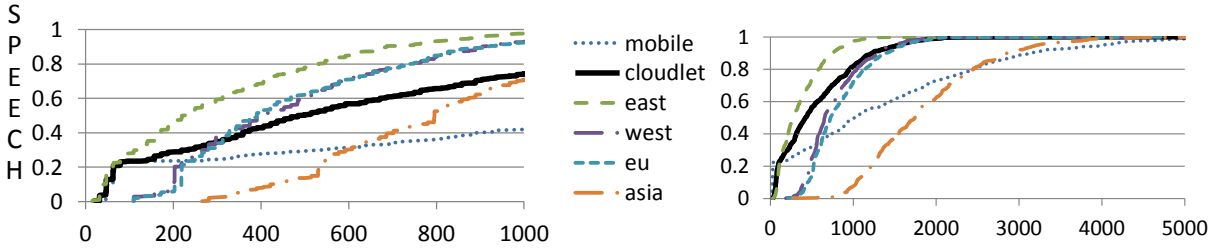


Figure 2.10: SPEECH: CDF of response times in ms (500 WAV files, each recording one sentence).

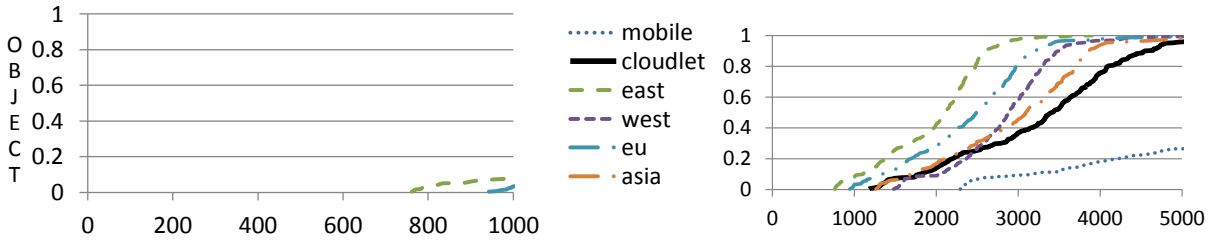


Figure 2.11: OBJECT: CDF of response times in ms (300 images).

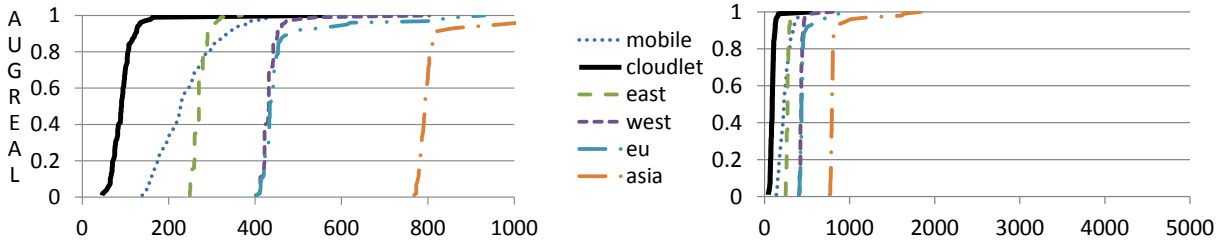


Figure 2.12: AR: CDF of response times in ms (100 images).

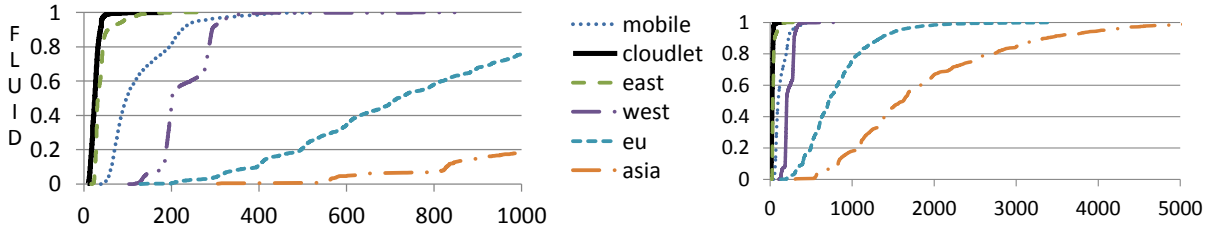


Figure 2.13: FLUID: CDF of response times in ms (10 minute runs, accelerometer data sampled every 20ms). (see also Figure 2.14)

	Normalized Simulation Speed	Displayed Frame Rate (FPS)
mobile	0.2	9.2
cloudlet	1.0	49.8
east	1.0	42.8
west	1.0	10.3
eu	1.0	3.6
asia	1.0	1.6

Figure 2.14: Simulation speed, frame rate for FLUID.

but is crushed by the heavy-tailed distribution of processing costs. Only cloudlet can provide fast response (<200ms) most of the time, and a tolerable worst case response time. Hence, cloudlet is the best approach to running FACE.

SPEECH Results for SPEECH are somewhat different (Figure 2.10). Here, the application generally requires significant processing for each query, and data transfer costs are modest. This changes the relative performance of the strategies significantly. As the response time is dominated by processing time, this favors the more capable but distant servers in the cloud over the weak cloudlet server. Processing without cloud assistance is out of the question. For SPEECH, using the closest EC2 data center is the winning strategy. To understand the effect of a more powerful cloudlet machine, we repeated that experiment with an Intel® Core™ i7-3770 desktop processor. The results shown in Figure 2.15 confirm that cloudlet now dominates the alternatives.

OBJECT Compared to the previous two applications, OBJECT requires significantly greater compute resources. Unfortunately, the processing load is so large that none of the approaches yield acceptable interactive response times (Figure 2.11). This application really needs more resources than our single VM instances or weak cloudlet server can provide. To bring response times down to reasonable levels for interactive use, we will either need to parallelize the application beyond a single machine/VM boundary and employ a processing cluster, or make use of GPU hardware to accelerate critical routines. Both of these potential solutions are beyond the scope of this chapter. Using the faster cloudlet machine (with an Intel® Core™ i7-3770 processor) does help significantly (Figure 2.16).

AR This application employs a low-cost feature extraction algorithm, and an efficient approximate nearest-neighbor algorithm to match features in its database. While these processing costs are modest, data transfer costs are high because of image transmission. Therefore, as shown

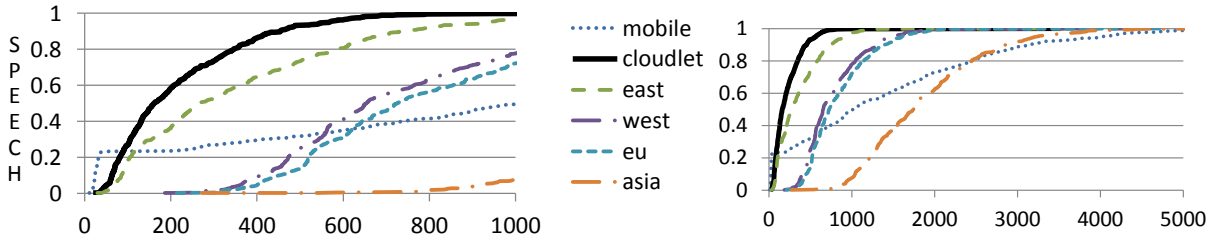


Figure 2.15: Experiments of Figure 2.10 repeated with faster cloudlet machine

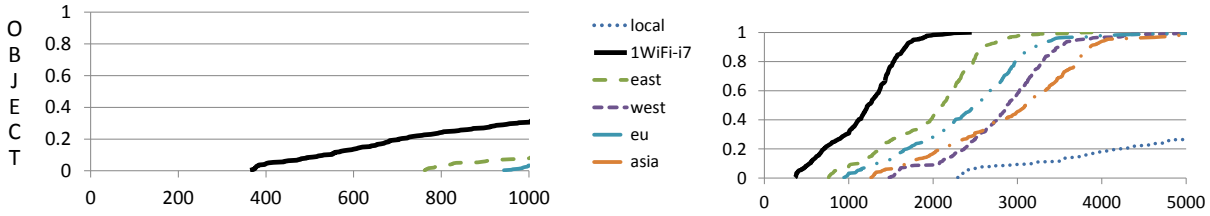


Figure 2.16: Experiments of Figure 2.11 repeated with faster cloudlet machine

in Figure 2.12, none of the EC2 cases is adequate for this application. They generally provide slower response times than execution on the mobile device. Cloudlet, on the other hand, works extremely well for this application, providing very fast response times (around 100ms) needed for crisp interactions. This is clearly the winning strategy for AR.

FLUID Response time for FLUID is defined as the time between the sensing of a user action (i.e., accelerometer reading), to when that input is reflected in the output. This largely reflects three factors: the execution time of a simulation step, network latency, and data transfer time for a frame from the simulation thread. As seen in Figure 2.13, local execution on the mobile device produces good response times, since all but the first factor are essentially zero. However, simulation speed and frame rate also need to be considered (Figure 2.14). The simulation runs asynchronously to the inputs and display, and tries to match simulated time with wall-clock time. Since the mobile device cannot execute the simulation steps fast enough, fluid motions are less than one fifth of realistic speeds. The cloud strategies do not have this issue, but due to bandwidth and network latencies, cannot deliver the results of the simulation fast enough to sustain the full frame rate. Only cloudlet and East can deliver both good responsiveness and high frame rates.

2.3.3 Impact on Energy Usage

Battery life is a key attribute of a mobile device. Executing resource-intensive operations in the cloud can greatly reduce the energy consumed on the mobile device by the processor(s), memory and storage. However, it increases network use and wireless energy consumption. Since peak

		Mobile	Cloudlet	East	West	EU	Asia
FACE	(W)	14.8	12.6	11.4	10.9	11.0	11.0
	(J/query)	16.4	5.4	6.6	8.5	9.5	14.3
SPEECH	(W)	16.1	14.5	14.5	14.4	14.4	14.4
	(J/query)	22.5	8.2	5.3	11.2	12.2	26.9
OBJECT	(W)	16.8	14.5	14.5	14.6	14.5	14.5
	(J/query)	107.0	48.2	28.9	41.5	35.3	43.8
AR	(W)	13.9	11.7	11.3	11.7	11.3	11.3
	(J/query)	3.3	1.1	3.1	5.1	5.2	9.4
FLUID	(W)	17.0	15.8	15.9	15.8	15.8	15.7
	(J/frame)	1.9	0.3	0.4	1.8	4.6	10.7

Figure 2.17: Energy consumption on mobile device

processor power consumption exceeds wireless power consumption on today's high-end mobile devices, this tradeoff favors cloud processing as computational demands increase. Network latency has recently been shown to increase energy consumption for remote execution by as much as 50%, even if bandwidth and computation are held constant [27, 31]. This is because hardware elements of the mobile device remain in higher-power states for longer periods of time.

Figure 2.17 summarizes energy consumption on our mobile device for the experiments described in Section 2.3.2. For each application, the first row shows the power dissipation in watts, averaged over the whole experiment. In all cases, this quantity shows little variation across data centers. Local execution on the mobile device incurs the highest power dissipation. Note that the netbook platform has a high baseline idle power dissipation (around 10W), so the relative improvement in power is likely to be larger on more energy-efficient hardware.

Average power dissipation only tells part of the story. Cloud use also tends to shorten the time to obtain a result. When this is factored in, the energy consumed per query or frame is dramatically improved. These results are shown in the second row for each application in Figure 2.17. In the best case, the energy consumed per result is reduced by a factor of 3 to 6. The strategies that exhibit the greatest energy efficiency are also the ones that give the best response times.

2.3.4 Summary and Discussion

The results of Sections 2.3.2 and 2.3.3 confirm that *logical proximity* to data center is essential for mobile applications that are highly interactive and resource intensive. By "logical proximity" we mean the end-to-end properties of high bandwidth, low latency and low jitter. Physical proximity is only weakly correlated with logical proximity because of the well-known "last mile" problem [106].

A cloudlet represents the best attainable logical proximity. Our results show that this extreme case is indeed valuable for many of the applications studied, both in terms of response time and energy efficiency. It is important to keep in mind that these are representative of a new genre of cognitive assistance applications that are inspired by the sensing and user interaction capabilities of mobile devices. Mobile participation in server-based multiplayer games such as Doom 3 is another use case that can benefit from logical proximity [11]. The emergence of such applications can be accelerated by deploying infrastructure that assures mobile users of continuous logical proximity to the cloud. The situation is analogous to the dawn of personal computing, when the dramatic lowering of user interaction latency relative to time-sharing led to entirely new application metaphors such as the spreadsheet and the WYSIWYG editor.

2.4 Enabling New Applications

In addition to the previous mobile applications, the cloudlet can be an enabling architectural element for futuristic applications. We have proposed new applications and frameworks that leverages cloudlets.

2.4.1 GigaSight

GigaSight [102] is a scalable Internet system for continuous collection of crowd-sourced video from head-up displays (HUDs) such as Google Glass. When equipped with a front-end camera, HUDs enable *near effortless capture of first-person viewpoint video*. Recording a video will merely require you to press a button on the shank of your glasses, rather than taking your smartphone out of your pocket and performing a number of touch screen interactions. Easy video capture will greatly increase the number of videos shared with the world (e.g. by uploading to YouTube). Integrating video capture with correlated sensor information such as gaze tracking, audio, geolocation, acceleration, and biodata (e.g., heartrate) is only a matter of time. GigaSight focuses on the gathering, cataloging, and access of first-person video from many contributors. By automatic tagging of this rich data collection, and by enabling deep content-based search of any subset of that data, it creates a valuable public resource much like the Web itself.

There are many technical challenges at different levels to enable this system. One critical challenge is how to automatically remove privacy sensitive information from the personal video. Unfortunately, always-on video capture is much less deliberate and controlled than authoring text. You can't help capturing scenes, but specific objects/people in them may not be what you (or they) want published. It is therefore crucial to edit out frames and/or blur individual objects

in scenes. What needs to be removed is highly user-specific, but no user can afford the time to go through and edit video captured on a continuous basis. One therefore needs a process that continuously performs this editing as video is submitted for sharing. If a user is confident that the editing process accurately reflects his personal preferences, he is likely to share his captured video without further review. We refer to this user-specific lowering of fidelity as *denaturing*.

Denaturing has to strike a balance between privacy and value. At one extreme of denaturing is a blank video: perfect privacy, but zero value. At the other extreme is the original video at its capture resolution and frame rate. This has the highest value for potential customers, but also incurs the highest exposure of privacy. Where to strike the balance is a difficult question that is best answered individually, by each user. This decision will most probably be context-sensitive. From a technical viewpoint, state-of-the-art computer vision algorithms including face detection, face recognition, and object recognition should be applied to each frame.

Another challenge is high cumulative data rate of incoming videos from many users. Without careful design, this could easily overwhelm the capacity of metro area networks or the ingress Internet paths into centralized cloud infrastructure such as Google's compute engine or Amazon's EC2 sites. As of 2014, 1 hour of video is uploaded to YouTube each second [119], which is the equivalent of only 3600 users simultaneously streaming. When the usage of HUDs becomes mainstream, this number will rapidly increase. Verizon recently announced an upgrade to 100 Gbps links in their metro area networks [81], yet one such link is capable of supporting 1080p streams from only 12000 users at YouTube's recommended upload rate of 8.5 Mbps. Supporting a million users will require 8.5 Tbps.

GigaSight uses cloudlet, which is decentralized cloud computing infrastructure, to solve these challenges. The use of cloudlets in this case is based solely on bandwidth considerations and GigaSight can be considered as a *hybrid cloud architecture* that is effectively a CDN in reverse. Cloudlets receive users' streaming video data 24/7 and perform denaturing process in near real-time. Only meta-data about these videos (such as owner (anonymized), location of capture, start and end time of capture, cloudlets geolocation, and index terms) is stored in a global catalog in the cloud. In a small number of cases, based on popularity or other metrics of importance, some videos may be copied to the cloud for archiving or replicated in the cloud and other cloudlets for scalable access. But most videos reside only at a single cloudlet. How long they are kept around depends on the storage reclamation and replication policy.

2.4.2 Gabriel

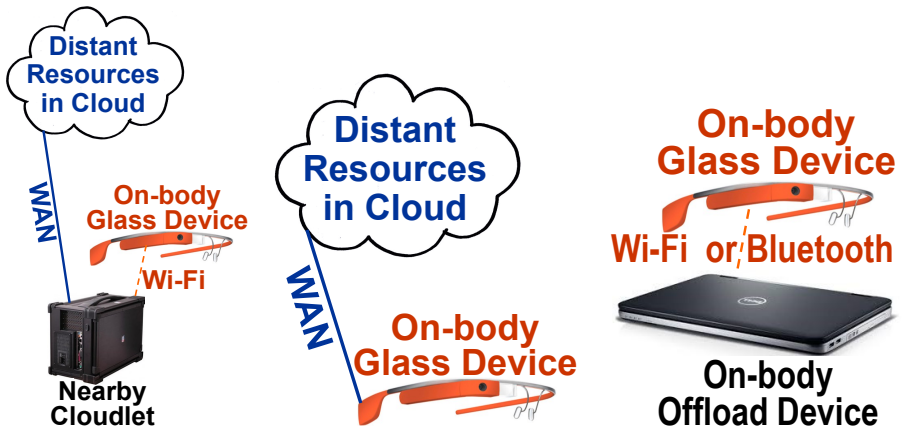
Gabriel [48] is an assistive system based on Google Glass devices. The possibility of using wearable devices for deep cognitive assistance (e.g., offering hints for social interaction via real-

time scene analysis) was first suggested nearly a decade ago [96, 98]. However, this goal has remained unattainable until now for three reasons. First, the state of the art in many foundational technologies (such as computer vision, sensor-based activity inference, speech recognition, and language translation) is only now approaching the required speed and accuracy. Second, the computing infrastructure for offloading compute-intensive operations from mobile devices was absent. Only now, with the convergence of mobile computing and cloud computing, is this being corrected. Third, suitable wearable hardware was not available. Although head-up displays have been used in military and industrial applications, their unappealing style, bulkiness and poor level of comfort have limited widespread adoption. Only now has aesthetically elegant, product-quality hardware technology of this genre become available. Google Glass is the most well-known example, but others are also being developed. It is the convergence of all three factors at this point in time that brings our goal within reach.

The Gabriel framework focuses on interactive cognitive assistance using Google Glass, which is the most widely available wearable device as of 2015. A Glass device is equipped with a first-person video camera and sensors such as an accelerometer, GPS[1], and compass. Although our prototype implementation works specifically on the Explorer version of Glass, our system architecture and design principles are applicable to any similar wearable device.

Our *Gabriel* system combines the first-person image capture and sensing capabilities of Glass with remote processing to perform real-time scene interpretation. In the system design of Gabriel, the unique demands of cognitive assistance applications create important constraints. First, we need crisp interactive response. Humans are acutely sensitive to delays in the critical path of interaction. Assistive technology that is introduced into the critical paths of perception and cognition should add negligible delay relative to the task-specific human performance figures cited above. Second, offloading is inevitable. The most sought-after features of a wearable device are light weight, small size, long battery life, comfort and aesthetics, and tolerable heat dissipation. System capabilities such as processor speed, memory size, and storage capacity are only secondary concerns. Offloading improves the speed of recognition, and lowers the energy cost to the mobile device. Third, coarse-grain parallelism is necessary. Human cognition involves the synthesis of outputs from real-time analytics on multiple sensor stream inputs. A wide range of software building blocks that correspond to these distinct processing engines exist today: face recognition [109], activity recognition [120] in video, natural language translation [12], OCR [41], and so on. These *cognitive engines* are written in a variety of programming languages and use diverse runtime systems. Some of them are proprietary, some are written for closed source operating systems, and some use proprietary optimizing compilers. Each is a natural unit of coarse-grain parallelism. In their entirety, these cognitive engines represent many

[1] Google Glass has hardware for GPS but it is not activated. Location is currently estimated with Wi-Fi localization.



(a) Normal (b) Cloud-Direct (c) Disconnected

Figure 2.18: Gabriel Offload Approaches

hundreds to thousands of person years of effort by experts in each domain. To the extent possible we would like to reuse this large body of existing code.

Cloudlet infrastructure plays a key role for these design constraints in Gabriel. It can serve powerful computing resources for cognitive engines without losing responsiveness. Moreover, it offers flexibility in computing environment supporting diverse set of programming language with different OSs and libraries by leveraging the VM abstraction. Our prototype implementation provide following features.

Low-latency Offloading

Gabriel achieves low-latency offload by using cloudlets. As a powerful, well-connected and trustworthy cloud proxy that is just one Wi-Fi hop away, a cloudlet is the ideal offload site for cognitive assistance. Wearable cognitive assistance can be viewed as a "killer app" that has the potential to stimulate investment in cloudlets. Figure 2.18(a) illustrates how offload works normally in Gabriel. The user's Glass device discovers and associates with a nearby cloudlet, and then uses it for offload. Optionally, the cloudlet may reach out to the cloud for various services such as centralized error reporting and usage logging. All such cloudlet-cloud interactions are outside the critical latency-sensitive path of device-cloudlet interactions. When the mobile user is about to depart from the proximity of this cloudlet, a mechanism analogous to Wi-Fi handoff is invoked. This seamlessly associates the user with another cloudlet for the next phase of his travels.

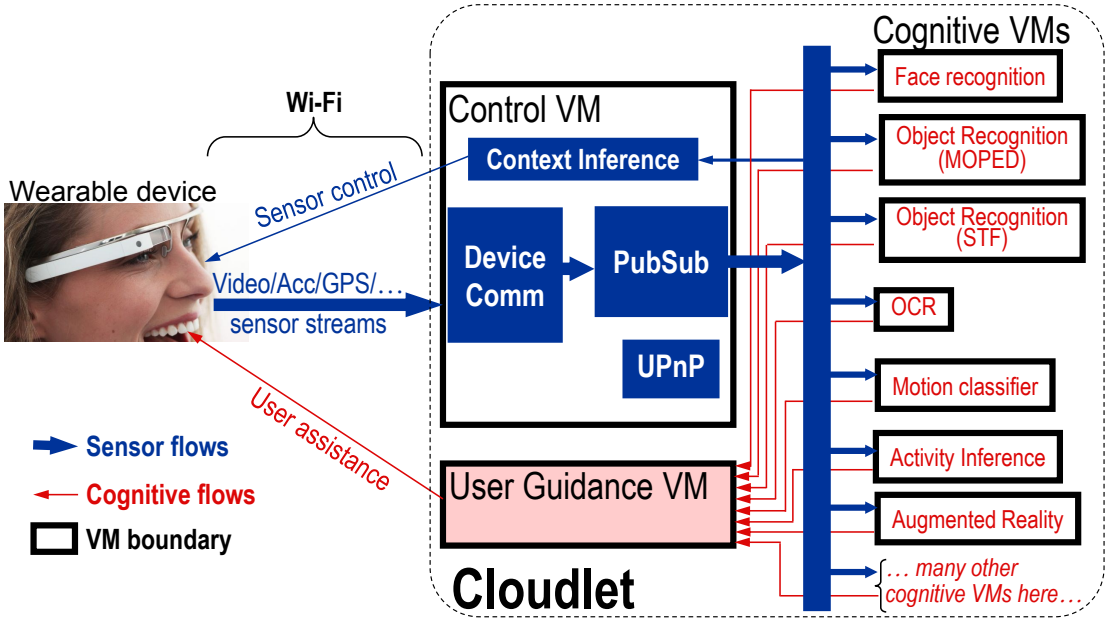


Figure 2.19: Back-end Processing on Cloudlet

Offload Fallback Strategy

When no suitable cloudlet is available, the obvious fallback is to offload directly to the cloud as shown in Figure 2.5(b). This incurs the WAN latency and bandwidth issues that were avoided with cloudlets. Since RTT and bandwidth are the issues rather than processing capacity, application-specific reduction of fidelity must aim for less frequent synchronous use of the cloud. This may hurt accuracy, but the timeliness of guidance can be preserved. When a suitable cloudlet becomes available, normal offloading can be resumed. An even more aggressive fallback approach is needed when the Internet is inaccessible. To handle these extreme situations, we assume that the user is willing to carry a device such as a laptop or a netbook that can serve as an offload device. As smartphones evolve and become more powerful, they too may become viable offload devices. The preferred network connectivity is Wi-Fi with the fallback device operating in AP mode, since it offers good bandwidth without requiring any infrastructure. Figure 2.5(c) illustrates offloading while disconnected.

VM Ensemble and PubSub Backbone

For the reasons explained earlier, a cloudlet must exploit coarse-grain parallelism across many off-the-shelf cognitive engines of diverse types and constructions. To meet this requirement, Gabriel encapsulates each cognitive engine (complete with its operating system, dynamically linked libraries, supporting tool chains and applications, configuration files and data sets) in its

own virtual machine (VM). Since there is no shared state across VMs, coarse-grain parallelism across cognitive engines is trivial to exploit. A cloudlet can be scaled out by simply adding more independent processing units, leading eventually to an internally-networked cluster structure. If supported by a cognitive engine, process-level and thread-level parallelism within a VM can be exploited through multiple cores on a processor — enhancing parallelism at this level will require scaling up the number of cores. A VM-based approach is less restrictive and more general than language-based virtualization approaches that require applications to be written in a specific language such as Java or C#.

Figure 2.19 illustrates Gabriel's back-end processing structure on a cloudlet. An ensemble of *cognitive VMs*, each encapsulating a different cognitive engine, independently processes the incoming flow of sensor data from a Glass device. A single *control VM* is responsible for all interactions with the Glass device. The sensor streams sent by the device are received and preprocessed by this VM. For example, the decoding of compressed images to raw frames is performed by a process in the control VM. This avoids duplicate decoding within each cognitive VM. A PubSub mechanism distributes sensor streams to cognitive VMs. At startup, each VM discovers the sensor streams of interest through a UPnP discovery mechanism in the control VM.

The outputs of the cognitive VMs are sent to a single *User Guidance VM* that integrates these outputs and performs higher-level cognitive processing. In this initial implementation of Gabriel, we use very simple rule-based software. As Gabriel evolves, we envision significant improvement in user experience to come from more sophisticated, higher-level cognitive processing in the User Guidance VM. From time to time, the processing in the User Guidance VM triggers output for user assistance. For example, a synthesized voice may say the name of a person whose face appears in the Glass device's camera. It may also convey additional guidance for how the user should respond, such as "John Smith is trying to say hello to you. Shake his hand."

Limiting Queuing Latency

In Gabriel's flexible and pluggable architecture, a set of components communicate using network connections. Each communication hop involves a traversal of the networking stacks, and can involve several queues in the applications and guest OSs, over which we have little control. The application and network buffers can be large, and cause many items to be queued up, increasing latency. To minimize queuing, we need to ensure that the data ingress rate never exceeds the bottleneck throughput, whose location and value can vary dramatically over time. The variation arises from fluctuations in the available network bandwidth between the Glass device and cloudlet, and from the dependence of processing times of cognitive engines on data content.

We have devised an application-level, end-to-end flow control system to limit the total number of data items in flight at a given time. We use a token-bucket filter to limit ingress of items

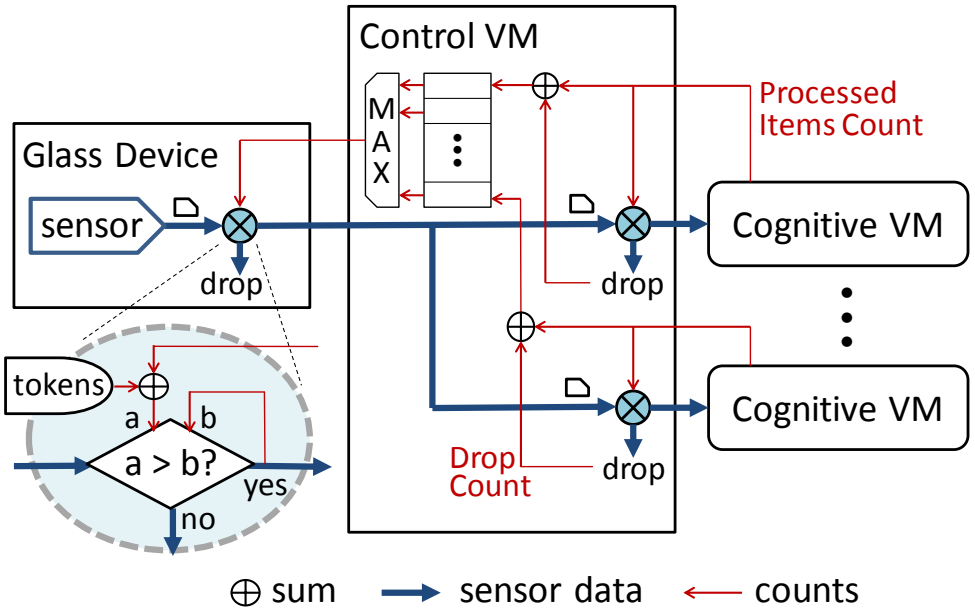


Figure 2.20: Two Level Token-based Filtering Scheme

for each data stream at the Glass device, using returned counts of completed items exiting the system to replenish tokens. This provides a strong guarantee on the number of data items in the processing pipeline, limits any queuing, and automatically adjusts ingress data rate (frame rates) as network bandwidth or processing times change.

To handle multiple cognitive engines with different processing throughputs, we add a second level of filtering at each cognitive VM (Figure 2.20). This achieves per-engine rate adaptation while minimizing queuing latency. Counts of the items completed or dropped at each engine are reported to and stored in the control VM. The maximum of these values are fed back to the source filter, so it can allow in items as fast as the fastest cognitive engine, while limiting queued items at the slower ones. The number of tokens corresponds to the number of items in flight. A small token count minimizes latency at the expense of throughput and resource utilization, while larger counts sacrifice latency for throughput. A future implementation may adapt the number of tokens as a function of measured throughput and latency, ensuring optimal performance as conditions change.

Chapter 3

Cloudlet Architecture

3.1 Two-level Hierarchical Architecture

In the previous chapter, we demonstrated the value of cloudlets for mobile computing. However, cloudlets work against cloud consolidation because there have to be many data centers at the edges of the Internet to ensure proximity everywhere. How can we reconcile these contradictory requirements?

We assert that the only practical solution to this problem is a hierarchical organization of data centers, as shown in Figure 3.1. Level 1 of this hierarchy is today's unmodified cloud infrastructure such as Amazon's EC2 data centers. Level 2 consists of *stateless data centers* at the edges of the Internet, servicing currently-associated mobile devices. This Level 2 data center is called a *cloudlet*. We envision an appliance-like deployment model for cloudlets. They are not actively managed after installation. Instead, soft state (e.g., virtual machine images and files from a distributed file system) is cached on their local storage from one or more Level 1 data centers. It is the absence of hard (durable) state at cloudlet that keeps management overhead low. Consolidation or reconfiguration of Level 1 data centers does not affect the cloudlets at Level 2. Adding a new cloudlet or replacing an existing one only requires modest setup and configuration. Once configured, a cloudlet can dynamically self-provision from Level 1 data centers. Physical motion of a mobile device may take it far from the cloudlet with which it is currently associated. When the distance becomes too great, a mechanism similar to wireless access point handoff can be executed to seamlessly switch association to a different cloudlet.

The hardware technology for cloudlet is already here today for reasons unrelated to mobile computing. For example, Myoonet has pioneered the concept of *micro data centers* for use in developing countries [74]. AOL has introduced indoor micro-data centers for enterprises [72] (Figure 3.2(c)) [72]. Today, these micro data centers are being used as Level 1 data centers in private clouds. By removing hard state and adding self-provisioning, they can be repurposed

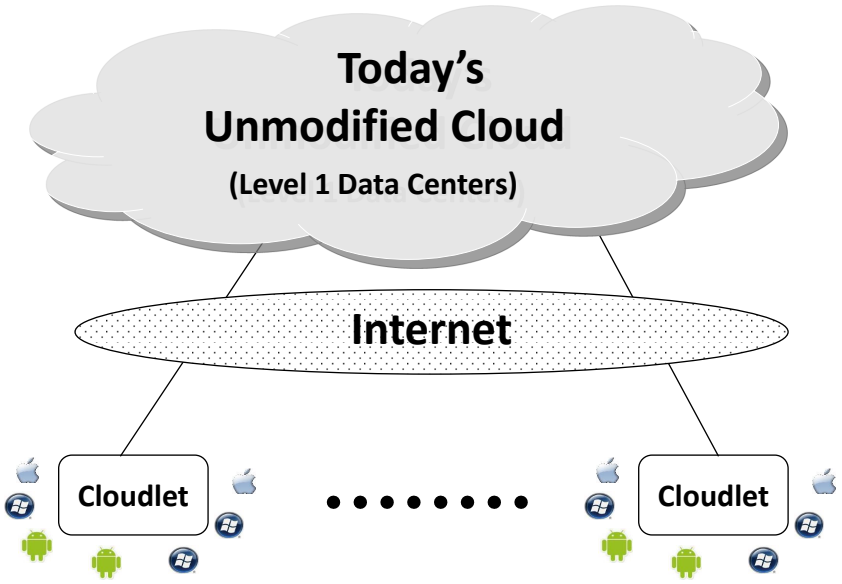
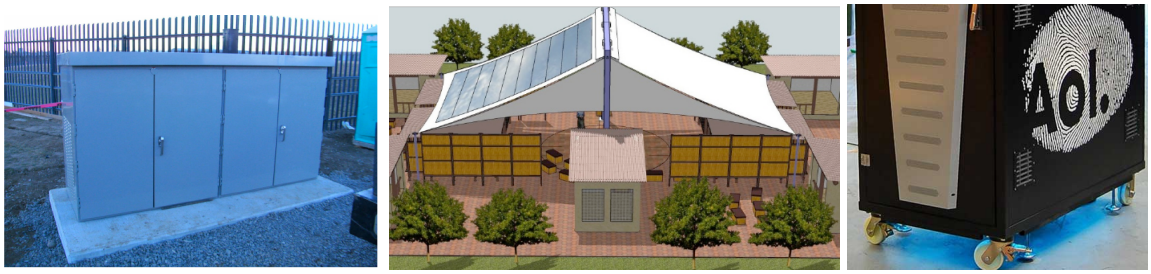


Figure 3.1: Two-level Hierarchical Cloud Architecture



(a) Outdoor (b) Solar Powered (c) Indoor

Figure 3.2: Unattended Micro Data Centers (Sources: [72, 74])

as cloudlets. In the future, one can envision optimized hardware for cloudlets. For example, with modest engineering effort, a WiFi access point could be transformed into a "nano," "pico," or "femto" cloudlet by adding processing, memory and storage. While much innovation and evolution will undoubtedly occur in the form factors and configurations of cloudlets, we identify four key attributes that any cloudlet implementation must possess:

- *Only soft state:* It does not have any hard state, but only cached state from cloud. It may also buffer data from a mobile device en route to a cloud.
- *Powerful and well-connected:* It is powerful enough to handle resource-intensive applications from multiple associated mobile devices. Battery life is not a concern because it is a stationary and wall-powered machine.
- *Close at hand:* It is easily deployable within one wireless hop (and LAN extension, if any) of associated mobile devices.

- *Builds on standard cloud technology:* It leverages and reuses cloud software infrastructure and standards (e.g. OpenStack [79]) as much as possible.

3.2 Technical Challenges Unique to Cloudlets

There is significant overlap in the requirements for cloud and cloudlet. At both levels, there is the need for: (a) strong isolation between untrusted user-level computations; (b) mechanisms for authentication, access control, and metering; (c) dynamic resource allocation for user-level computations; and, (d) the ability to support a very wide range of user-level computations, with minimal restrictions on their process structure, programming languages or operating systems. At cloud data center, these requirements are met today using the virtual machine (VM) abstraction. We believe, for the same reasons they are used in cloud computing today, that VMs are the right level of abstraction for cloudlets. Meanwhile, there are a few but important differentiators between cloud and cloudlet.

1. *Rapid provisioning*: The speed of provisioning matters at cloudlets. Today, cloud data centers are optimized for launching VM images that already exist in their storage tier. They do not provide fast options for instantiating a new custom image. One must either launch an existing image and laboriously modify it, or suffer the long, tedious upload of the custom image over a WAN. In contrast, cloudlets need to be much more agile in their provisioning. Their association with mobile devices is highly dynamic, with considerable churn due to user mobility. A user from far away may unexpectedly show up at a cloudlet (e.g., if he just got off an international flight) and try to use it for an application such as a personalized language translator. For that user, the provisioning delay before he is able to use the application impacts usability.
2. *VM migration across cloudlets (Hand-off)*: Once a user successfully uses a provisioned cloudlet, the next question is "What happens if a mobile device user moves away from the cloudlet he is currently using?" As long as network connectivity is maintained, the applications should continue to work transparently. However, interactive response will degrade as the logical network distance increases. In practice, this degradation can be far worse than physical distance may suggest. For example, when moving from a home Wi-Fi network to that of a neighbor down the street, communication to the first home's cloudlet will require two traversals of "last-mile" links connecting the homes to their ISPs. How can we address this effect of user mobility? If the offloaded services on the first cloudlet can be seamlessly transferred to the second cloudlet, end-to-end network quality can be maintained. We refer to this capability as *VM handoff*. VM handoff resembles live migration in cloud computing, but differs considerably in the details in terms of its metric

and constraints. For example, different from data center VM migration using a LAN, we need to migrate VM from one cloudlet to the other over a WAN.

3. *Cloudlet discovery*: Dynamic discovery of a cloudlet by a mobile client is an unique problem in cloudlet. Because cloudlets are small datacenters distributed geographically, a mobile device first has to discover, select and associate with the appropriate cloudlet among multiple candidates before it starts provisioning. These steps are unnecessary with a cloud because it is centralized. But in cloudlets, discovery and selection have to be carefully managed because the choice of a cloudlet can directly affect the provisioning time as well as future performance of the associated mobile application.

We believe these are the minimal functionalities that cloudlet must offer above/beyond standard cloud computing system to establish two-level architecture. In this thesis, we will address these challenges.

3.3 OpenStack++: Deploying Cloudlets

In addition to the technical challenges, we will also consider the pragmatic aspects of the cloudlet; deployment of cloudlet infrastructure. Since our cloudlet model requires reconfiguration or additional deployment of hardware/software, it is important to provide a systematic way to incentivise the deployment. And here we are facing a classic bootstrapping problem. We need practical applications to incentivize cloudlet deployment. However, developers cannot heavily rely on cloudlet infrastructure until it is widely deployed. How can we break this deadlock and bootstrap the cloudlet deployment?

The history of the Internet offers a hint. The Internet is an open ecosystem that uses a standard protocol suite (e.g. TCP/IP). Through this open standard, multiple vendors from low-level hardware companies to high level services providers are participating independently. However, no single vendor is bearing large risk for improving this ecosystem or dominating market. Instead, they are creating synergy by investing in their own business. In this ecosystem, innovation in one layer can stimulate others, resulting in additional investment. For example, wide use of Internet services such as email and web searching has encouraged ISPs to invest on their infrastructure. Those advances in infrastructure become a foundation for new Internet services like VoIP and social networks.

We will take a similar strategy with cloudlets. Many of today's server-based mobile applications are using cloud infrastructure to host their back-end server. We observe that an ecosystem in cloud computing is similar to that of Internet; various vendors from hardware to software are actively participating independently for their profit. For example in hardware, network vendors such as Cisco and Jupiter are deploying Software Define Network (SDN) routers/switches, and

blade server vendors like IBM and HP are reshaping their products [53]. Similarly in software, multiple hypervisors are rapidly developed to compete with each other, and various Linux vendors like RedHat and Canonical propose their own solutions for cloud computing. *OpenStack*, which is a free and open-source cloud computing software platform, provides openness in this emerging ecosystem [79]. It offers a suite of standard APIs, so each vendor in different layers can independently contribute without breaking compatibility. As of 2014, more than 150 companies have contributed to OpenStack.

We will leverage this open platform to expedite cloudlet deployment. That is, we will make our system work as OpenStack extensions, so that any individual or any vendior who uses OpenStack for their cloud computing can easily use cloudlets. We refer to this Cloudlet-enabled OpenStack as *OpenStack++*. Our task will include designing and implementing OpenStack++ APIs. We will also provide a client library and a web interface for the general OpenStack user.

Chapter 4

Rapid Just-In-Time Virtual Machine Provisioning

In this chapter, we will address the first technical challenges we mentioned in Chapter 3.2. The first challenge is cloudlet provisioning. We observed that many requirements can be met by using VMs for both cloud (Level 1) and cloudlet (Level 2). VMs guarantee bit-exact matches in the reconstructed state by simply resuming the entire machine. However, one of the major differences in using VM between cloud and cloudlet is the speed of provisioning. Cloudlets need to be much more agile in their provisioning because their association with mobile devices is highly dynamic, with considerable churn due to user mobility. A server-based mobile application relies on the precisely configured back-end server, so exact custom VM that encapsulates the back-end server needs to be provisioned at the cloudlet. However, it is unlikely that nearby cloudlets have the custom VM.

The large size of VM images complicates dynamic provisioning of cloudlets. At the same time, the presumption of *ubiquity* in mobile computing deprecates a static provisioning strategy. A mobile user expects good service for all his applications at any place and time. Wide-area physical mobility (e.g., an international traveler stepping off his flight) makes it difficult to always guarantee that a nearby cloudlet will have the precise VM image needed for offloading (e.g., natural language translation with customized vocabulary and speaker-trained voice recognition via the traveler's smartphone). The VM guest state space is simply too large and too volatile for static provisioning of cloudlets at global scale. A different provisioning challenge involves the deployment of new cloudlets for load balancing, hardware upgrades, or recovery from disasters. Dynamic self-provisioning of cloudlets will greatly simplify such deployments.

Rapid just-in-time provisioning of cloudlets is the focus of this chapter. We show how a cloudlet can be provisioned in as little as 10 seconds with a complete copy of a new VM image that is the back-end of an offloaded application such as face recognition, object recognition, or

augmented reality. The compressed sizes of these VM images can range from 400 MB for a stripped-down Linux guest, to well over 2 GB for typical Windows based images. The key to rapid provisioning is the recognition that a large part of a VM image is devoted to the guest OS, software libraries, and supporting software packages. The customizations of a base system needed for a particular application are usually relatively small. Therefore, if the *base VM* already exists on the cloudlet, only its difference relative to the desired custom VM, called a *VM overlay*, needs to be transferred. This concept of a VM overlay bears resemblance to copy-on-write virtual disk files [70] or VM image hierarchies [19], but extends to both disk and memory snapshots. Our approach of using VM overlays to provision cloudlets is called *dynamic VM synthesis*. Proof-of-concept experiments [98] showed provisioning times of 1–2 minutes using this approach. In this chapter, we present a series of optimizations that reduces this time by an order of magnitude.

Although motivated by mobile computing, dynamic VM synthesis has broader relevance. Today, public clouds such as Amazon's EC2 service are well-optimized for launching images that already exist in their storage tier, but do not provide fast options for provisioning that tier with a new, custom image. One must either launch an existing image and laboriously modify it, or suffer the long, tedious upload of the custom image. For really large images, Amazon recommends mailing a hard drive! Though developed for cloudlets, we show that dynamic VM synthesis can rapidly provision public clouds such as EC2.

4.1 Dynamic VM Synthesis

4.1.1 Basic Approach

The intuition behind dynamic VM synthesis is that although each VM customization is unique, it is typically derived from a small set of common base systems such as a freshly-installed Windows 7 guest or Linux guest. We refer to the VM image used for offloading as a *launch VM*. It is created by installing relevant software into a *base VM*. The compressed binary difference between the base VM image and the launch VM image is called a *VM overlay.* This idea of a binary difference between VM images to reduce storage and network transfer costs has been successfully used before [19, 70, 122]. We, therefore, extensively use this overlay concept for both VM disk and memory snapshots in this work.

At run-time, *dynamic VM synthesis* (sometimes shortened to "VM synthesis" or just "synthesis") reverses the process of overlay creation. Figure 4.1 shows the relevant steps. A mobile device delivers the VM overlay to a cloudlet that already possesses the base VM from which this overlay was derived. The cloudlet decompresses the overlay, applies it to the base to derive the launch VM, and then creates a VM instance from it. The mobile device can now begin perform-

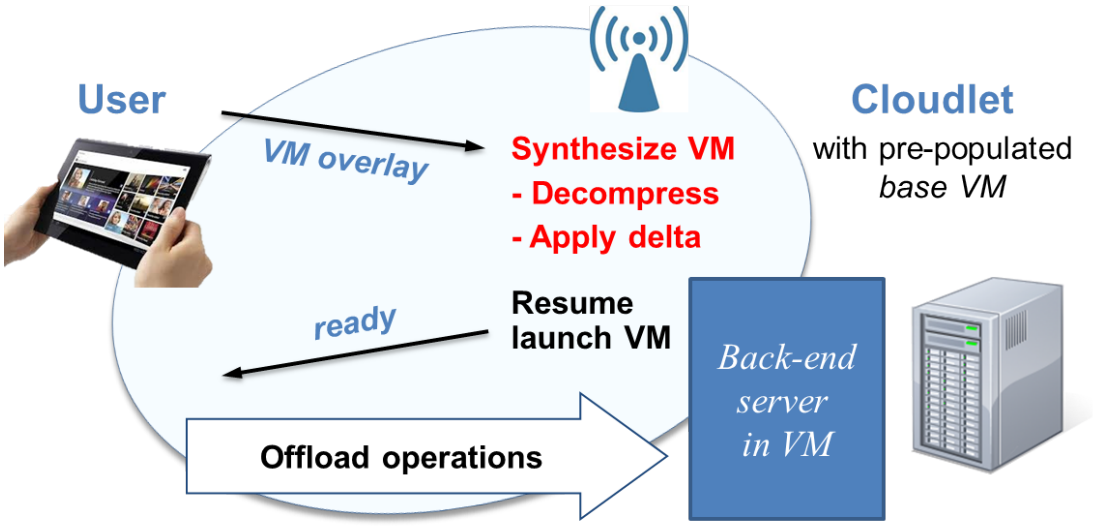


Figure 4.1: Dynamic VM Synthesis from Mobile Device

ing offload operations on this instance. The instance is destroyed at the end of the session, but the launch VM image can be retained in a persistent cache for future sessions. As a slight variant of this process, a mobile device can ask the cloudlet to obtain the overlay from the cloud. This indirection reduces the energy used for wireless data transmission, but can improve transfer time only when WAN bandwidth to the cloud exceeds local WiFi bandwidth. Also the cloud can lead the whole process of this variant.

Note that the cloudlet and mobile device can have different hardware architectures: the mobile device is merely serving as transport for the VM overlay. Normally, each offload session starts with a pristine instance of the launch VM. However, there are some use cases where modified state in the launch VM needs to be preserved for future offloads. For example, the launch VM may incorporate a machine learning model that adapts to a specific user over time. Each offload session then generates training data for an improved model that needs to be incorporated into the VM overlay for future offload sessions. This is achieved by generating a *VM residue* that can be sent back to the mobile device and incorporated into its overlay.

There are no constraints on the guest OS of the base VM; our prototype works with both Linux and Windows. We anticipate that a relatively small number of base VMs will be popular on cloudlets at any given time. To increase the chances of successful synthesis, a mobile device can carry overlays for multiple base VMs and discover the best one to use through negotiation with the cloudlet. Keep in mind that the VMs here are virtual appliances that are specifically configured for serving as the back-ends of mobile applications. Although these virtual appliances are generated on top of conventional operating systems such as Linux or Windows, they

App name	Install size (MB)	Overlay Size (MB) disk	memory	Synthesis time (s)
OBJECT	39.5	92.8	113.3	62.8
FACE	8.3	21.8	99.2	37.0
SPEECH	64.8	106.2	111.5	63.0
AR	97.5	192.3	287.9	140.2
FLUID	0.5	1.8	14.1	7.3

Figure 4.2: Baseline performance (8 GB disk, 1 GB memory)

are focused and dedicated to serve a particular mobile application, rather than general-purpose desktop environments that need a wider range of functionality.

It is useful to contrast dynamic VM synthesis with demand paging the launch VM from the mobile device or cloud using a mechanism such as the Internet Suspend/Resume system® [97]. Synthesis requires the base VM to be available on the cloudlet. In contrast, demand paging works even for a freshly-created VM image that has no ancestral state on the cloudlet. Synthesis can use efficient streaming to transmit the overlay, while demand paging incurs the overhead of many small data transfers. However, some of the state that is proactively transferred in an overlay may be wasted if the launch VM includes substantial state that is not accessed. Synthesis incurs a longer startup delay before VM launch. However, once launched, the VM incurs no stalls. This may be valuable for soft real-time mobile applications such as augmented reality.

It is also useful to contrast VM synthesis with launching the base VM and then performing package installations and configuration modifications to transform it into the launch VM. This is, of course, exactly what happens offline when creating the overlay; the difference is that the steps are now being performed at runtime on each association with a cloudlet. On the one hand, this approach can be attractive because the total size of install packages is often smaller than the corresponding VM overlay (e.g., Figure 4.2) and, therefore, involves less transmission overhead. On the other hand, the time delay of installing the packages and performing configuration is incurred at run time. Unlike optimization of VM synthesis, which is fully under our control even if the guest is closed-source, speeding up the package installation and configuration process requires individual optimizations to many external software components. Some of those may be closed-source, proprietary components. Of even greater significance is the concern that installing a sequence of packages and then performing post-installation configuration is a fragile and error-prone task even when scripted. Defensive engineering suggests that these fragile steps be performed only once, during offline overlay creation. Once a launch VM image is correctly created offline, the synthesis process ensures that precisely the same image is re-created on each cloudlet use. This bit-exact precision of cloudlet provisioning is valuable to a mobile user, giving him high confidence that his applications will work as expected no matter where he is in the

world. Finally, the installation approach requires the application to be started fresh every time. Execution state is lost between subsequent uses, destroying any sense of seamless continuity of the user experience.

4.1.2 Baseline Performance

We have built an instantiation of the basic VM synthesis approach, using the KVM virtual machine monitor. In our prototype, the overlay is created using the `xdelta3` binary differencing tool. Our experience has been that `xdelta3` generates smaller overlays than the native VM differencing mechanism provided by KVM. The VM overlay is then compressed using the Lempel-Ziv-Markov algorithm (LZMA), which is optimized for high compression ratios and fast decompression at the price of relatively slow compression [116]. This is an appropriate trade-off because decompression takes place in the critical path of execution at run-time and contributes to user-perceived delay. Further, compression is only done once offline but decompression occurs on each VM synthesis.

We test the efficacy of VM synthesis in reducing data transfer costs and application launch times on the VM back-ends of five mobile applications, explained in Chapter 2.1. In each case, user interaction occurs on a mobile device while the compute-intensive back-end processing of each interaction occurs in a VM instance on a cloudlet. These applications, written by various researchers and described in recent literature, are the building blocks of futuristic applications that seamlessly augment human perception and cognition. Three of the five back-ends run on Linux, while the other two run on Windows 7. These compute-intensive yet latency-sensitive applications are used in all the experiments reported in this chapter.

We first construct base VM images using standard builds of Linux (Ubuntu 12.04 server) and Windows 7. These VMs are configured with 8 GB of disk and 1 GB of memory. An instance of each image is booted and then paused; the resulting VM disk image and memory snapshot serve as *base disk* and *base memory* respectively. To construct a launch VM, we resume an instance of the appropriate base image, install and configure the application binaries, and launch the application. At that point, we pause the VM. The resulting disk image and memory snapshot constitute the launch VM image. As soon as an instance is resumed from this image, the application will be in a state ready to respond to offload requests from the mobile device — there will be no reboot delay.

The overlay for each application is the compressed binary difference between the launch VM image and its base VM image, produced using `xdelta3` and LZMA compression. The sizes of the overlays, divided into disk and memory components, are reported in Figure 4.2. For comparison, the sizes of the compressed application installation packages are also reported. Relative to VM image sizes, the VM synthesis approach greatly reduces the amount of data that must be

	Mobile	Cloudlet
Model	Dell™ Latitude 2120 Netbook	Dell™ Optiplex 9010 Desktop
CPU	Intel® Atom™ N550 1.5 GHz, 2 cores	Intel® Core® i7-3770 3.4 GHz, 4 cores, 8 threads (4 VCPUs for VM)
RAM	2 GB	32 GB (1 GB VM RAM)
Disk	250 GB HDD	1 TB HDD (8 GB VM disk)
Network	802.11a/g/n WiFi*	1 Gbps Ethernet
OS	Ubuntu 12.04 64bit (Kernel 3.2.0)	Ubuntu 12.04 64bit (Kernel 3.2.0)
VMM	—	QEMU/KVM-1.1.1†
Misc	Belkin N750 Router (802.11n, GigE)	

*2.4 GHz 802.11n used here; 38 Mbps measured average BW
†modified for some experiments, as described in Sect. 4.5

Figure 4.3: System configuration for experiments

transferred to create VM instances. Compared to the launch VM images (nominal 8 GB disk image plus memory snapshot), Figure 4.2 shows that overlays are an order of magnitude smaller. While they are larger than the install packages from which they were derived, VM synthesis eliminates the fragile and error-prone process of runtime package installation and configuration as discussed in Section 4.1.1. In fact, as we show later in Section 4.6.1, provisioning using the most optimized version of VM synthesis is faster than runtime installation and configuration.

The total time to perform VM synthesis is also reported in Figure 4.2. These times were measured using a netbook (client) and a virtual machine (server) hosted in a cloudlet described in Table 4.3. The client serves the application overlays to the cloudlet, which performs synthesis and executes the application VMs. For each application, the total time reported includes the time needed to transfer the overlay across WiFi, decompress it, apply the overlay to the base image, and resume the constructed application image. We note that the netbook used here is not significantly more capable than smartphones today, and achieves the same network bandwidth (38 Mbps) on 802.11n as the Samsung Galaxy 2 in our tests. Since most computation is done offline or on the cloudlet, and the data transfer is network limited, we do not expect significantly different results using a smartphone. However, for our prototype implementation, the netbook was convenient as it allowed us to use a full complement of x86 tools and libraries for the front-ends of our five applications. We use this configuration for all of the experiments in this chapter.

Although this baseline implementation of VM synthesis achieves bit-exact provisioning without transferring full VM images, its performance falls short for ad-hoc, on-demand use in mobile offload scenarios. Figure 4.2 shows that only one of the applications, FLUID, completes synthesis within 10 seconds. A synthesis time of 60 to 150 seconds is more typical for the other applications. That is too large for good user experience.

In the rest of this chapter, we present a multi-pronged approach to accelerating VM synthesis. We first reduce the size of the overlay using aggressive deduplication (Section 4.2) and by bridging the semantic gap between the VMM and guest OS (Section 4.3). We then accelerate the launch of the VM image by pipelining its synthesis (Section 4.4), and by optimistically launching before synthesis is complete (Section 4.5). The results presented in each of these sections shows the speedup attributable to that optimization.

4.2 Deduplication

Concept

Our first optimization leverages the fact that there are many sources of data redundancy in a VM overlay. Through deduplication we can eliminate this redundancy and thus shrink the overlay. A smaller overlay incurs less transmission delay and also consumes less energy on the mobile device for transmission. Deduplication is very effective at reducing redundant data, and has been used widely in a variety of fields. In the virtualization space, it has been applied to reduce memory footprints of concurrent VMs [113], and in accelerating VM migration [122]. It is particularly well suited to VM overlays, since the significant expense of deduplication is only incurred offline during overlay construction. The overhead of re-inflating deduplicated data during synthesis is trivial, especially because the cloudlet is a powerful machine that is not energy-constrained. From a number of sources, we can anticipate some duplication of data between the memory snapshot and the disk image of the launch VM. For example, at the moment the launch VM is suspended during overlay construction, the I/O buffer cache of the guest OS contains some data that is also present in its virtual disk. Additionally, data from some files on the virtual disk may have been read by the application back-end into its virtual memory during initialization. Further, depending on the runtime specifics of the programming language in which the application is written, there may be copies of variable initialization data both in memory and on disk. These are only a few of the many sources of data duplication between the memory snapshot and the disk image of the launch VM.

Separately, we can also expect some duplication of data between the overlay and the base VM (which is already on the cloudlet). Recall that the baseline implementation in Section 4.1.1 creates a VM overlay by constructing a binary delta between a launch VM and the base VM from which it is derived. This binary delta may contain duplicate data that has been copied or relocated within the memory or disk image. Indeed, the baseline system cannot take advantage of the fact that many parts of memory should be identical to disk because they are loaded from disk originally, e.g., executables, shared libraries, etc. An efficient approach to capturing this begins with a list of modifications within the launch VM and then performs deduplication to

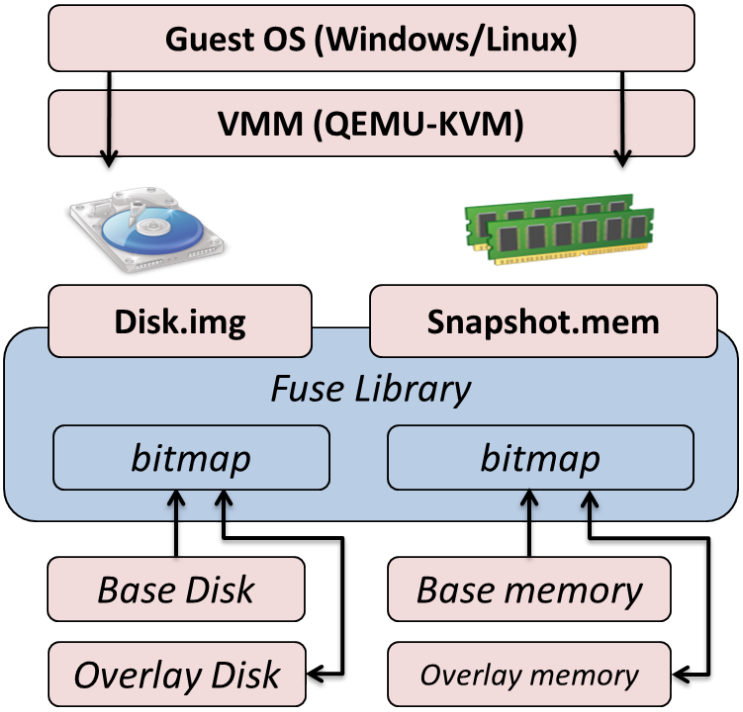


Figure 4.4: FUSE Interpositioning for Deduplication

further reduce this list to the minimal set of information needed to transform a base VM into the launch VM. If we could find this minimal set, then we could construct smaller VM overlays.

Implementation

The choice of the granularity at which comparisons are performed is a key design decision for deduplication. Too large a granularity will tend to miss many small regions that are identical. Very small granularity will detect these small regions, but incur large overhead in the data representation. Our choice is a chunk size of 4 KB because it is a widely-used page size for many popular operating systems today. For example, current versions of Linux, Mac OS X, and Windows all use a 4 KB page size. An additional benefit of deduplicating at this granularity is that most operating systems use Direct Memory Access (DMA) for I/O, which means the disk is accessed with memory page size granularity. Thus, the 4 KB chunk size is likely to work well for both memory and disk deduplication.

To discover the portions of disk and memory modified during the process of creating a launch VM, we introduce a shim layer between the VMM and the backing files for virtual disk and memory using FUSE, as shown in Figure 4.4. During the installation and configuration steps of launch VM construction, the shim layer exposes I/O requests from the VMM to the virtual disk file and memory snapshot file. On every write to either the virtual disk or memory snapshot, we

redirect the write to the corresponding overlay file and mark a bitmap indicating this chunk has changed. When reads occur at a later point in time, we consult this bitmap to determine if the read should be serviced from the original base files, or from the new overlay files. As in [76], we have found that FUSE has minimal impacts on virtual disk accesses, despite the fact that it is on the critical read and write paths from the VM to its disk. However, memory operations would become prohibitively expensive with this additional component. We therefore do not use FUSE to capture memory changes. Rather, we capture the entire memory snapshot only after we finish customizing the launch VM. We then interpret this memory snapshot, and compare it with *base memory* to obtain the modified memory chunks and corresponding bitmap.

We reuse this FUSE shim layer at VM synthesis time to avoid the data copying that would be required to explicitly merge the overlay virtual disk/memory with the base to reconstruct the launch VM. Instead, we redirect VM disk/memory access to either the overlay or the base image based on the bitmap. This approach to just-in-time reconstruction of a launch image has been used previously in systems such as ISR [58] and the Collective [19], though only for VM disk.

Once we have a list of modified disk and memory chunks, we perform deduplication by computing SHA-256 hashes [36] of their contents. We use these hashes to construct a unique set of pages which are not contained within the base VM and must be included in the transmitted overlay. We construct the set of unique modified disk and memory chunks using five comparison rules: (1) compare to base VM disk chunks, (2) compare to base VM memory chunks, (3) compare to other chunks within itself (within modified disk or modified memory respectively), (4) compare to a zero-filled chunk, and (5) compare between modified memory and modified disk. These five comparison rules capture various scenarios that are frequent sources of data redundancy, as discussed in Section 4.2.

For each unique chunk, we compare it to the corresponding chunk (same position on disk or in memory) in the base VM. We use the `xdelta3` algorithm to compute a binary delta of the chunk and transmit only the delta if it is smaller in size than the chunk. The idea behind this is that even if the hashes do not match, there may still be significant overlap at a finer byte granularity which a binary delta algorithm can leverage.

Evaluation

Figure 4.5 shows the benefit of deduplication for the overlay of each application. For any deduplication between memory and disk, we choose to only retain the duplicated chunks within memory. For deduplication purposes, it does not matter if the canonical chunk resides within disk or memory; we chose the memory snapshot as the canonical source of chunks.

Averaged across the five applications, only 22% of the modified disk and 77% of the modified memory is unique. The biggest source of redundancy is between modified memory and

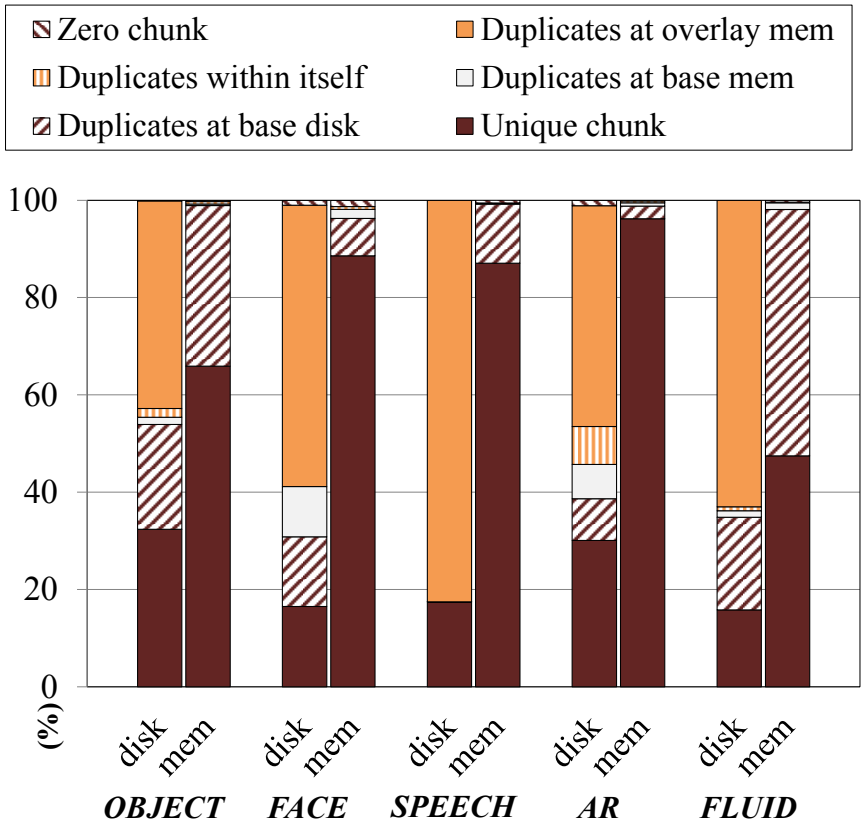


Figure 4.5: Benefit of Deduplication

modified disk: each application exhibits greater than 58% duplication, with SPEECH exhibiting 83% duplication. The base disk is the second biggest source of duplication. On average, 13% of the modified disk and 21% of the modified memory are identical with chunks in the base disk. We analyzed files associated with the duplicated chunks for the OBJECT application. Our findings are consistent with our intuition: most of the associated files in the modified disk are shared resources located within the `/usr/shared/`, `/usr/lib/`, and `/var/lib/` directories, and a large portion of the files are shared libraries such as `libgdk-x11`, `libX11-xcb`, and `libjpeg`. The overlay memory shows similar results, but it also includes copies of executed binaries such as `wget`, `sudo`, `xz`, `dpkg-trigger`, and `dpkg-deb` in addition to shared libraries.

4.3 Bridging the Semantic Gap

Concept

The strong boundary enforced by VM technology between the guest and host environments is a double-edged sword. On the one hand, this strong boundary ensures isolation between the host, the guest, and other guests. On the other hand, it forces the host to view each guest as a black box, whose disk and memory contents cannot be interpreted in terms of higher-level abstractions such as files or application-level data structures. This challenge was first recognized by Chen and Noble [20]. Various attempts to bridge the semantic gap between VMM and the guest include VM introspection for intrusion and malware detection [37, 54] and memory classification [17] for improving prefetcher performance.

The semantic gap between low-level representations of memory and disk, and higher-level abstractions is also problematic when constructing VM overlays. For example, suppose a guest application downloads a 100 MB file, and later deletes it. Ideally, this should result in no increase in the size of the VM overlay. However, the VMM will see up to 200 MB of modifications: 100 MB of changed disk state, and 100 MB of changed memory state. This is because the file data moves through the in-memory I/O buffer cache of the guest OS before reaching the disk, effectively modifying both memory state and disk state. When the file is deleted, the guest OS marks the disk blocks and corresponding page cache entries as free, but their (now garbage) contents remain. To the VMM, this is indistinguishable from important state modifications that need to be preserved. Deduplication (described in Section 4.2) can cut this state in half, but we would still unnecessarily add 100 MB to the overlay.

Ideally, only the state that actually matters to the guest should be included in the overlay. When files are deleted or memory pages freed, none of their contents should be incorporated into the overlay. In essence, we need semantic knowledge from the guest regarding what state needs to be preserved and what can be discarded. When constructing the launch VM, a user (or application developer) installs a back-end application server on the base VM. This installation process typically involves several steps including downloading installation packages, creating temporary files, and moving executable binaries to target directories. Also, it is likely that all unneeded files will be deleted after finishing the installation process. We note that there is nothing unusual about this procedure for constructing custom VMs; it is identical to how custom VMs are typically generated in Amazon EC2, for example. We wish to fully leverage the user's intent when producing the overlay. We discard chunks containing semantically unnecessary footprint of the installation process by bridging the semantic gap between the VMM and guest in a manner that is transparent to guests.

In separate sections below, we show how this semantic gap can bridged for disk and memory

state.

Implementation: Disk

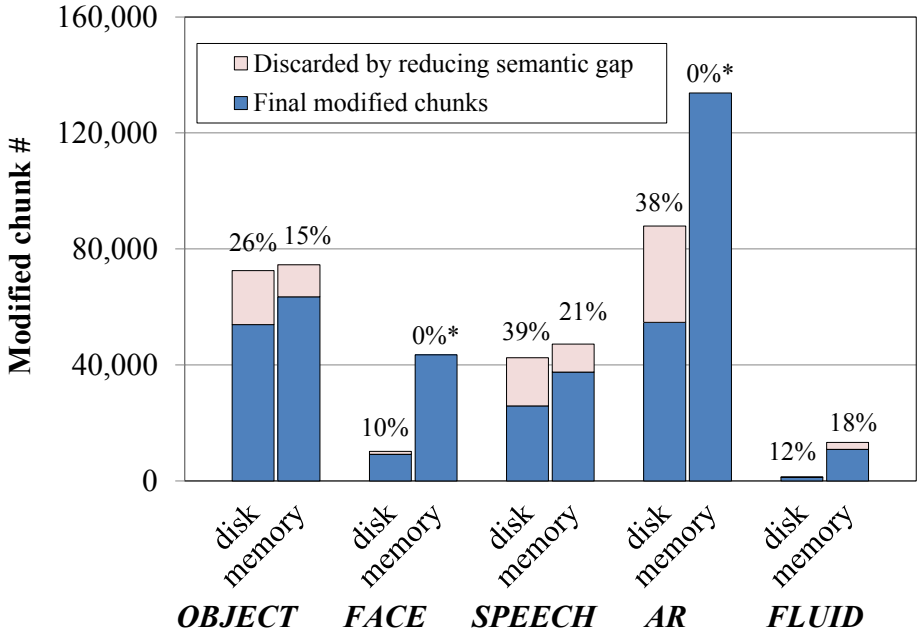
To accurately account for disk blocks that are garbage, we need either (1) a method of communicating this information from the guest OS to the host, or (2) a method of scanning the contents of the file system on the virtual disk to glean this OS-level information. The first approach requires guest support, and may not be possible for every guest OS. The second approach requires no guest support, but does require an understanding of the on-disk file system format. Both approaches may be used in tandem to cross-check their results.

1. **Exploiting TRIM support:** The TRIM command in the ATA standard enables an OS to inform a disk which sectors are no longer in use. This command is important for modern devices such as Solid State Drives (SSDs) which implement logic to aggressively remap writes to unused sectors. Wear-leveling algorithms and garbage collection inside of SSDs use this knowledge to increase write performance and device life.

   The TRIM command provides precisely the mechanism we desire — an industry-standard mechanism for communicating semantic information about unused sectors from an OS to the underlying hardware. We can exploit this mechanism to communicate free disk block information from the guest OS to the host to reduce VM overlay size. We modify the VMM (KVM/QEMU) to capture TRIM events and to log these over a named pipe to our overlay generation code. When generating the overlay, we merge this TRIM log with a trace of sector writes by timestamp to determine which blocks are free when the VM is suspended; these blocks can be safely omitted from the overlay. To make use of this technique, we simply need to ensure that TRIM support is enabled in the guest OS. As TRIM is an industry standard, it is supported by almost all modern operating systems, including recent Linux distributions and Windows 7.

2. **Introspecting the file system:** An alternative approach is to use knowledge of the on-disk file system format to directly inspect the contents of a virtual disk [56, 91] and determine which blocks are currently unused. Many file systems maintain lists of free blocks forming a canonical set of blocks which should not be included in an overlay. In the worst case, the entire file system can be crawled to determine which blocks are in use by files within the file system. Although this approach is file-system-specific, it avoids the need to communicate information from a running guest, or to carefully trace TRIM and write events.

   We identify free disk blocks using the tool described by Richter et al. [91]. This tool



*Our implementation cannot determine free memory pages for Windows guests.
*Percentage represents the fraction of discarded chunks.

Figure 4.6: Savings by Closing the Semantic Gap

reads and interprets a virtual disk image and produces a list of free blocks. It supports the `ext2/3/4` family of Linux file systems and the `NTFS` file system for Windows.

Implementation: Memory

It is difficult to determine which memory pages are considered free by a guest OS. Although the VMM can inspect the page tables, this is not sufficient to determine if a page is in use because unmapped pages are not necessarily free [17]. Inspecting page contents is also not good enough, because free pages normally contain random data and are not zeroed.

To bridge this gap, there are two natural approaches: (1) communicate free page information from the guest OS to the host, or (2) interpret memory layout data structures maintained by the guest OS. Unfortunately, there is no standard way of accomplishing the first approach (i.e., no memory counterpart to TRIM support), so we focus our efforts on the second approach. In order to obtain the list of free memory pages we first introduce a tiny kernel module into Linux guests. This module exposes the memory addresses of two data structures for memory management through the `/proc` file system in the guest. We suspend the VM, and feed these addresses and the memory snapshot to an offline scanning program. This scanning program reads the memory snapshot and parses the memory management data structures at the specified addresses to identify the free pages.

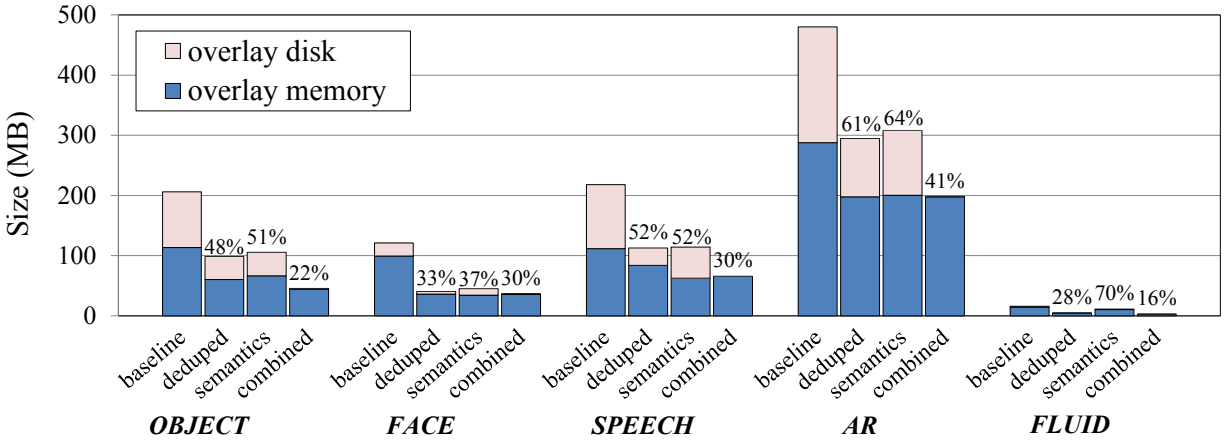


Figure 4.7: Overlay Size Compared to Baseline (Percentage represents relative overlay size compared to baseline)

Since our approach requires modifying the guest OS, it is not usable on closed-source OSs such as Windows. Further, in-memory data formats tend to be highly volatile across OS releases, and to evolve much more rapidly than file system formats. Even an open-source kernel such as Linux will require significant maintenance effort to track these changes.

Other techniques could be employed to infer free pages without the need for guest support. For example, a VMM could monitor memory accesses since the guest's boot and keep track of pages that have been touched. This would avoid guest modification at the cost of lower fidelity—some of the pages reported as used could have been touched, but later freed. Perhaps with the advent of Non-Volatile Memories (NVMs), which provide persistent storage with memory-like, byte-addressable interfaces, there may be a need to introduce a standardized TRIM-like feature for memory. Such support would make it possible to bridge the memory semantic gap in an OS-agnostic way in the future.

Evaluation

For the disk semantic gap, our experiments show that the TRIM and introspection approaches produce nearly identical results. Just a few additional free blocks are found by the introspection approach that were not captured by TRIM. We therefore present only the results for the TRIM approach.

Figure 4.6 shows how much we gain by closing the disk semantic gap. For each application we construct the VM image by downloading its installation package, installing it, and then deleting the installation package. We therefore expect our approach to find and discard the blocks that held the installation package, reducing overlay size by approximately the installation package size. Our results confirm this for all of the applications except one: for FACE, the semantically

discarded disk blocks together were smaller than the installation package. On investigation, we found that this was due to the freed blocks being reused post-install. On average, across the five applications, bridging the disk semantic gap allows 25% of modified disk chunks to be omitted from the overlay.

Figure 4.6 also shows the savings we can achieve by discarding free memory pages from the VM overlay. We can discard on average 18% of modified memory chunks for the Linux applications OBJECT, SPEECH, and FLUID. Since our implementation is limited to Linux, we cannot reduce the memory overlays for the two Windows-based applications (FACE and AR).

Combining deduplication and bridging of the semantic gap can be highly effective in reducing the VM overlay size. Figure 4.7 shows VM overlay size with each optimization individually represented, and also combined together. The "baseline" represents VM overlay size using the approach described in Section 5.3.3. The bar labeled "deduped" is the VM overlay with deduplication applied; "semantics" is the VM overlay with semantic knowledge applied (only disk for Windows applications); and, "combined" is the VM overlay size with both optimizations applied. On average compared to the baseline implementation, the deduplication optimization reduces the VM overlay size to 44%. Using semantic knowledge reduces the VM overlay size to 55% of its baseline size. Both optimizations applied together reduce overlay size to 28% of baseline.

The final overlay disk almost disappears when we combine both optimizations. This is because a large portion of disk chunks are associated with installation packages. Recall that to install each application, we first download an installation package in the VM and remove it later when it finishes installation. This installation file is already compressed, so further compression does little. In addition, this newly introduced data is less likely to be duplicated inside the base VM. Therefore, applying semantic knowledge removes most of the unique chunks not found by deduplication. For example in AR, $25,887$ unique chunks remained after deduplication, but 96% of them are discarded by applying semantic knowledge.

4.4 Pipelining

Concept

There are three time-consuming steps in VM synthesis. First, the VM overlay is transferred. Next, the VM overlay is decompressed. Finally, the decompressed VM overlay is applied to the base VM (i.e., `xdelta3` in reverse). These steps are serialized because we need the output of the preceding step as input to the next one, as shown in Figure 4.8. This serialization adds significantly to the VM start latency on a cloudlet. If we could begin the later steps before the

Figure 4.8: Baseline VM Synthesis

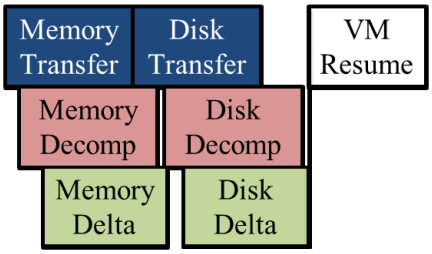


Figure 4.9: Pipelined VM Synthesis

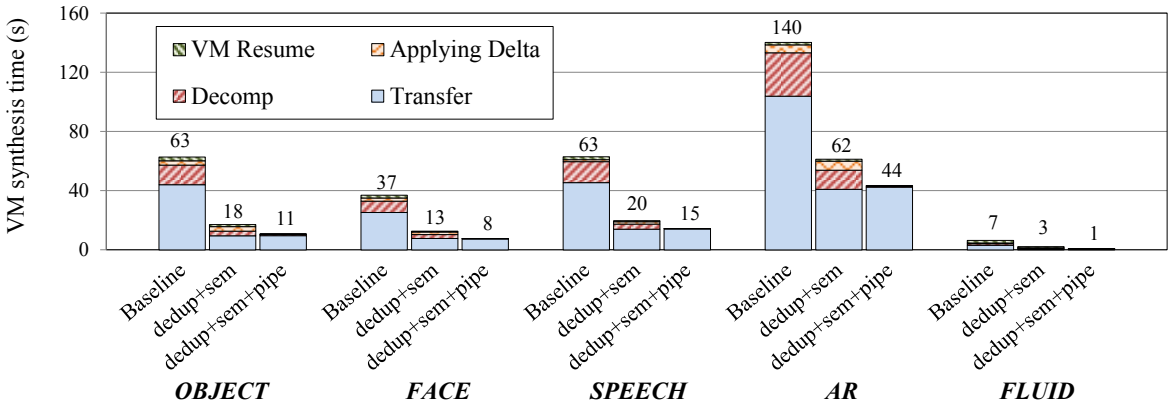


Figure 4.10: VM Synthesis Acceleration by Pipelining

preceding ones complete, we could shrink the total time for synthesis as shown in Figure 4.9.

Implementation

The implementation follows directly from the pipelining concept. We split the VM overlay into a set of segments and operate on each segment independently. The VM synthesis steps can now be pipelined. The decompression of a segment starts as soon as it is transferred, and happens in parallel with the transfer of the next segment. Likewise, the application of an overlay segment to the base VM proceeds in parallel with the decompression of the next segment. Given sufficiently small segment size, the total time will approach that of the bottleneck step (typically the transfer time), plus any serial steps such as VM instance creation and launch.

Evaluation

Figure 4.10 compares the performance of the baseline synthesis approach to an optimized one that combines deduplication, semantic gap closing, and pipelining. The results confirm that once pipelining is introduced, transfer time becomes the dominant contributor to the total synthesis time. The synthesis time shown in Figure 4.10 includes all of the time needed to get the VM to the point where it is fully resumed and ready to accept offload requests from the mobile device. Two applications now launch within 10 seconds (FACE and FLUID), while two others launch within 15 seconds (OBJECT and SPEECH). Only AR takes much longer (44 seconds), but this is because its overlay size and, therefore, transfer time remains high. On average, we observe a 3x–5x speedup compared to the baseline VM synthesis approach from Section 5.3.3.

4.5 Early Start

Concept

We have shown that the optimizations described in the previous sections greatly reduce the size of overlays and streamline their transfer and processing. For several of the applications, the optimized overlay size is close to the size of the install image. Hence, there is little scope for further reducing size to improve launch times. Instead, we consider whether one really needs to transfer the entire overlay before launching the VM instance. This may not be necessary for a number of reasons. For example, during the overlay creation process, the guest OS was already booted up and the application was already launched at the point when the VM was suspended. Any state that is used only during guest boot-up or application initialization will not be needed again. As another example, some VM state may only be accessed during exception handling or other rare events and are unlikely to be accessed immediately after VM instance creation.

The potential benefits of optimism can be significant. Table 4.11 shows the percentage of chunks in the overlay that are actually accessed by the five benchmark applications between VM launch and completion of the first request. A substantial number of chunks are not used immediately. We can speed up VM launch by transferring just the needed chunks first, synthesizing only those parts of the launch VM, and then creating the VM instance. The transfer of the missing parts of the overlay and synthesis of the rest of the launch VM can continue in the background until it is completed.

Implementation

We explored a number of alternatives in translating the concept of early start into a viable implementation. One option is to profile the resume of the launch VM, and order the chunks in

	OBJECT	FACE	SPEECH	AR	FLUID
% chunks	17.4	56.9	26.8	65.2	27.1
% size	30.6	63.0	33.0	87.9	50.3

Percentage of the overlay accessed between VM launch and completion of first request, in terms of chunks and compressed overlay size.

Figure 4.11: Percentage of Overlay Accessed

the overlay accordingly. When offloading, the VM is resumed concurrently with the synthesis operations. If the VM attempts to access chunks that have not yet been synthesized, it will be blocked until the chunk becomes available. If the order of chunks is correct, the VM can begin running significantly before the VM synthesis completes.

Unfortunately, it is difficult to get this order perfectly right. In our early experiments, multiple profiling runs produced slightly different chunk access patterns. In particular a small number of chunks may be accessed early in one run, but not at all in another. With a large number of chunks, it is unlikely that every chunk that is needed early in an actual VM resume will have been picked up in the profiling. More likely, one or more of these chunks will be missed in profiling, and will be placed near the end of the overlay. Getting even one chunk wrong can force the VM to wait for all chunks to be transferred and VM synthesis to complete.

Alternatively, we can avoid trying to predict the chunk access order by using a demand fetching approach, as done in [58], [93], and many subsequent efforts. Here, the VM is started first, and the portions of the overlay needed to synthesize accessed chunks are fetched on demand from the mobile device. Unfortunately, this approach, too, has some issues. Demand fetching individual chunks (which can be very small due to deduplication and delta encoding) requires many small network transfers, with a round trip penalty imposed on each, resulting in poor effective bandwidth and slow transfers. To alleviate this, we can cut the overlay into larger segments comprised of many chunks, and perform demand fetching at segment granularities. This will help amortize the demand fetching costs, but leaves open the question of sizing the segments. Smaller segments let one fetch more closely just the needed chunks. Larger segments, in addition to being more bandwidth friendly, can achieve better compression ratios, but will be less selective in transferring just what is needed. Finally, how chunks are grouped into segments can also significantly influence performance. For example, if needed chunks are randomly distributed among segments, one will likely need to transfer the entire overlay to run the VM.

Our implementation uses a hybrid approach that combines profiling and demand paging, similar to VMTorrent [90] though applied to VM overlays rather than VM images. We make a reasonable attempt to order the chunks according to a profiled access pattern computed offline. We then break the overlay into segments. During offload, we start the VM and begin streaming the segments in order, but also allow out-of-order demand fetches of segments to preempt the

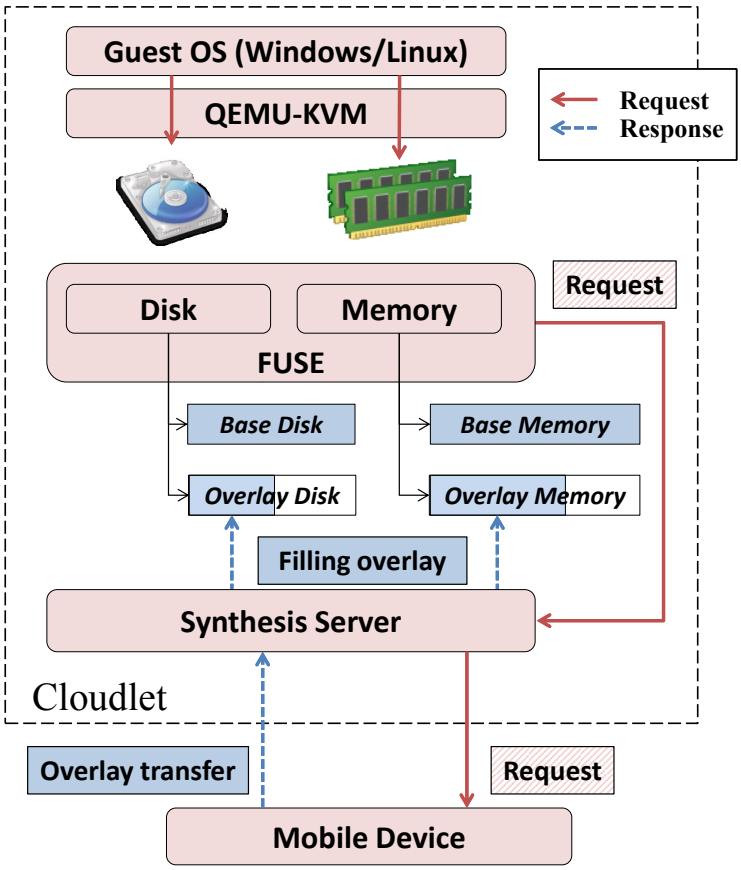


Figure 4.12: System Implementation for Early Start

original ordering. Thus, we use demand fetching to retrieve chunks that were not predicted by the profiling, but unlike [90], we simultaneously bulk-stream segments in a work-conserving manner to quickly transfer and synthesize all chunks. While this approach bears some resemblance to classic prefetching with out-of-band handling of demand misses, these concepts are being applied to an overlay rather than a VM image.

Figure 4.12 illustrates our implementation of this hybrid approach. A critical issue is that all of the widely used VMMs, including KVM, Xen, VirtualBox, and VMware, require the entire memory snapshot before resuming a VM, hindering early start. So, we first modify the VMM (KVM in our case) to resume a VM without first reading in the entire memory snapshot. Rather, it now memory maps the snapshot file, so portions are implicitly loaded when accessed. We then implement a FUSE file system that hosts the VM disk image and memory snapshot. This routes disk accesses of the VMM to our user-level code that can perform just-in-time VM synthesis on the accessed chunks (disk or memory). Our code consults a small bitmap that indicates whether the particular chunk needs to be served from the base image or the overlay. If the overlay is needed, then a local cache of processed chunks is checked. If the chunk is not available, a

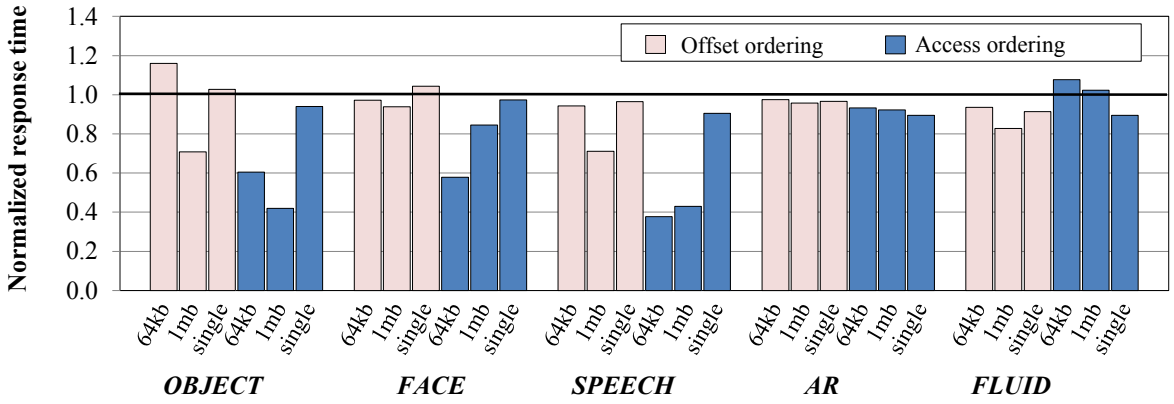


Figure 4.13: Normalized First Response Time for Early start

demand-fetch of the needed overlay segment is issued to the mobile device. Concurrently, in the background, the code processes the stream of overlay segments as it is received from the mobile device. With this implementation, only the small bitmap assigning chunks to overlay or base image needs to be transferred before the VM is launched.

Evaluation

We evaluate our early start approach with a few different combinations of segment size and chunk order. We test with small (approx. 64KB) and medium (approx. 1MB) sized segments, as well as with just a single segment comprising the entire overlay. (The latter effectively disables demand-fetching). We also test with chunks sorted by access-order (based on a single profiling run of each application) and offset-order (with memory before disk chunks). We slightly modify the ordering so that duplicate chunks are contained within the same segment, avoiding any need for pointer-chasing between segments when handling deduplication.

With early-start, the VM synthesis time itself is less relevant. Rather, the metric we use is the first response time of the application. This is measured by having the mobile device initiate the synthesis of the application VM, and then repeatedly attempt to send it queries. The response time is measured from the initial VM start request to when the first reply returns to the client, thus including the overlapped transfer time, synthesis time, and application execution time. Figure 4.13 compares performance of early start for different chunk ordering policies and segment sizes. The values are normalized to the first response time when the VM begins execution once VM synthesis (including all of the other optimizations discussed previously) completes. Access ordering alone does not help significantly due to the inaccuracies of the single profiling runs. However, access-ordered chunks with demand fetching with 64 KB or 1 MB segments can significantly reduce first response times. We see up to 60% reduction for *SPEECH* and *OBJECT*. For *FLUID*, the response time without early start is already so short that

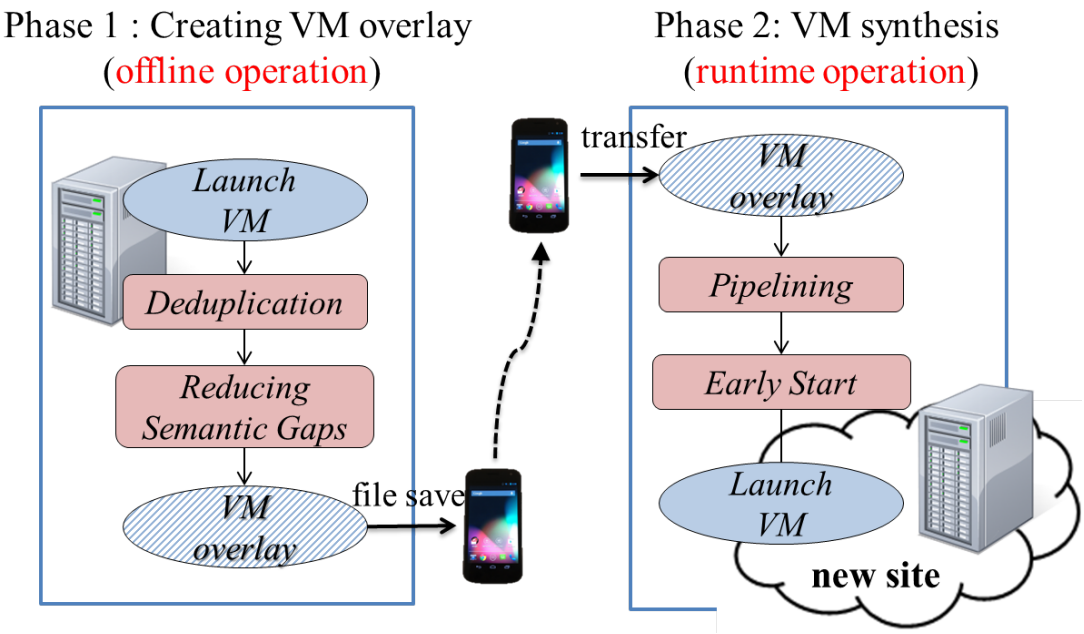


Figure 4.14: Fully Optimized VM Synthesis

small fluctuations due to compression and network affect the normalized response time adversely. *AR*, which requires 90% of the data in its overlay, does not benefit much from early start.

4.6 Final Results and Discussions

4.6.1 Fully Optimized VM Synthesis

We have shown that all of the various techniques for improving VM synthesis described in this chapter work quite well on their own. One may wonder: how effective are these techniques when combined? In this section we evaluate a complete, fully-optimized implementation of VM synthesis incorporating all of the improvements described in this work. The figure of merit here is the total latency as perceived by the user, from the beginning of the application offload process to when the first reply is returned. This first response metric is dependent on the time consumed by the offloading operation and launch of the application VM.

Figure 4.14 illustrates the fully-optimized process of VM synthesis as we intend it to be used with mobile devices and cloudlet infrastructure. We first construct minimal application overlays via deduplication and by preserving only the semantically meaningful chunks. We store and serve these overlays to the cloudlet from the mobile device (a netbook in our experiments). To minimize the time required to perform VM synthesis, we pipelined the synthesis steps, and then applied early start on the cloudlet.

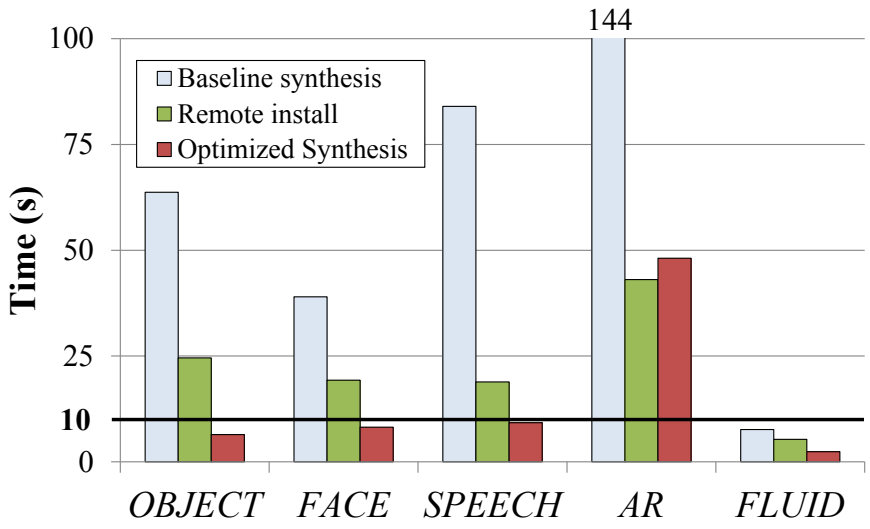


Figure 4.15: First response times

Figure 4.15 shows the improvement in first-response times with our fully-optimized VM synthesis over the baseline version described in Section 5.3.3. In this experiment, we used an overlay segment size of 1 MB, which provides a good tradeoff between demand fetch granularity and good compression. Overall, we improved performance of VM synthesis by a factor of 3 to 8 across these applications. Except for *AR*, the first responses for all of the other applications come within 10 s. A combination of multiple factors causes significantly longer synthesis times for the *AR* application. First, it has the largest installation size among all five applications and a significant portion of the installation is a database file that is less likely to be deduplicated. In addition, we could not close the memory semantic gap for *AR* since our implementation cannot determine free memory pages for Windows guests. Further, as depicted in Table 4.11, *AR* requires almost all of the overlay to serve the initial request, and, thus, it does not benefit from the early start optimization.

We also compare our results to the first-response time for a remote installation approach to running a custom VM image. This involves resuming a standard VM, uploading and installing the application packages, and then executing the custom applications. In Section 4.1.2, we have already dismissed this approach on qualitative grounds; in particular, even scripted install can be fragile, the resulting configuration is not identical every time, and the application is restarted every time so execution state is not preserved. The only redeeming quality of this approach is that the install packages tend to be smaller than the baseline VM overlays, potentially making the remote install faster. Here, we use highly optimized application packages that are self-contained (including needed libraries, or statically-compiled binaries), and fully-scripted installation to show the remote install approach at its fastest.

As we can see from Figure 4.15, however, our optimized VM synthesis approach produces

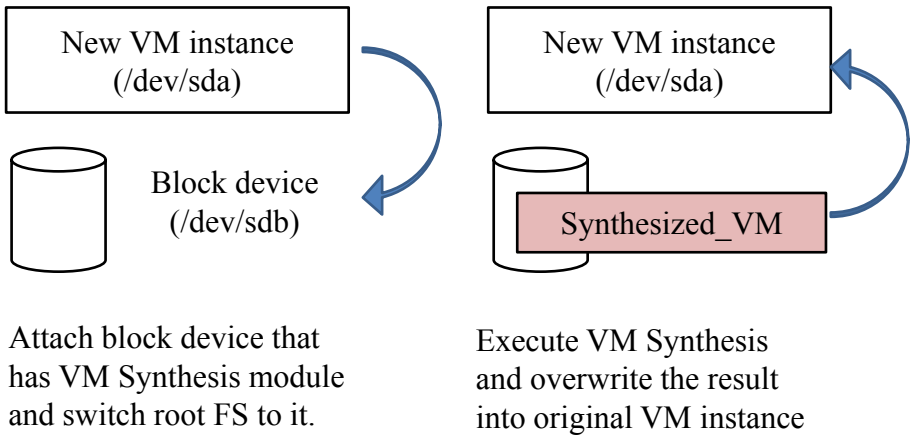


Figure 4.16: VM Synthesis for EC2

significantly better first-response times than remote install in all but one case. In that case, the two approaches are basically a tie. Thus, our optimized VM synthesis approach can achieve very fast offload and execution of custom application VMs on cloudlets, yet maintain strong guarantees on their reconstructed state.

4.6.2 Improved WiFi Bandwidth

All of the experiments in this chapter were conducted using 802.11n WiFi at 2.4 GHz (38 Mbps measured average bandwidth). We expect these times to improve in the future as new wireless technologies and network optimizations are introduced, increasing the bandwidth of WiFi networks. In other words, VM synthesis time is now directly correlated to network bandwidth. While WAN bandwidth improvements require large infrastructure changes, mobile bandwidth to the wireless AP at a cloudlet only requires localized hardware and software changes. New WiFi standards such as 802.11ac promise up to 500 Mbps and are actively being deployed [117]. Recent research [45] also demonstrates methods of increasing bandwidth up to 700% with software-level changes for WiFi networks facing contention. Thus, both industry and research are focused on increasing WiFi bandwidth. This directly translates into faster VM synthesis. Based on our measurements, until actual transfer times improve by 3x, the transfer stage will remain the bottleneck (assuming the cloudlet processor remains constant). Beyond this, we will need to parallelize the decompression and overlay application stages across multiple cores to benefit from further improvements in network bandwidth.

4.7 VM Synthesis on Amazon EC2

In this work, we have presented VM synthesis as a technique to rapidly offload customized application VMs to cloudlet infrastructure near a mobile device. However, the technique is much more general than this, and can help whenever one wishes to transfer VM state across a bottleneck network. In particular, VM synthesis can significantly speed up the upload and launch of a custom VM on commercial cloud services across a WAN. Here, we describe our VM synthesis solution for Amazon's public EC2 cloud.

The normal cloud workflow to launch a customized VM involves three steps: (1) construct the VM image, including installing custom software and libraries, and making requisite configuration changes; (2) upload the VM image to the cloud, a step largely limited by the client-to-cloud bandwidth; and (3) launch and execute a VM instance based on the uploaded VM image, a step that depends on the cloud provider's backend scheduling and resources. VM synthesis promises to speed up the second step by reducing the amount of state uploaded to a cloud.

Today, no cloud supports VM synthesis as a primitive operation. In our EC2 implementation, we perform VM synthesis entirely within a running VM instance. EC2 does not allow external access to the disk or memory image of an instance, so we cannot manipulate the saved state of a paused instance to effect synthesis. We also cannot generate a data file, treat it as a VM image, and launch an instance based on it. We work around these limitations by performing VM synthesis within a live instance, which modifies its own state and then reboots into the custom VM environment. Assuming that the base VM image and synthesis tools have already been installed in an EC2 block device, synthesis proceeds in the eight steps are as follows.

1. Create a new EC2 VM instance from an existing Amazon VM image,
2. Attach a cloud block device with VM synthesis tools and base VM image,
3. Change the root file system of the instance to the attached block device,
4. Perform **VM synthesis** over the WAN to construct the modified VM disk,
5. Mount the modified VM disk,
6. Synchronize / copy the modified file system with the instance's original,
7. Detach block device,
8. Reboot with the customized file system.

Steps 1-3 occur on the left hand side of Figure 4.16, while steps 4-8 occur on the right hand side. We do not handle the memory portion of a VM in EC2 because we do not have access to the raw memory image. This requires an unnecessary reboot and wasted time in synchronizing file systems. If EC2 had a VM synthesis primitive, the memory image and VM disk could be directly exposed by their infrastructure and only step 4 would remain; the VM overlay would be

	10 Mbps		100 Mbps	
	Synthesis	Amazon	Synthesis	Amazon
Synthesis Setup	44 s	—	46 s	—
Uploading†	36 s	607 s	8 s	204 s
Post-processing	96 s	139 s	97 s	105 s
Total	180 s	746 s	154 s	310 s

†Upload time for VM synthesis includes all synthesis steps (overlay transfer, decompression, and applying delta).

Figure 4.17: Time for Instantiating Custom VM at Amazon EC2

transmitted, applied, and then the VM could be directly resumed without reboot.

We compare the time it takes to perform VM synthesis to the time required in the normal cloud workflow to deploy and execute a custom VM with the OBJECT application. The results are shown in Table 4.17. For VM synthesis, synthesis setup corresponds to steps 1-3, uploading to step 4, and post-processing corresponds to steps 5-8. With the normal Amazon workflow, there is no analog to synthesis setup. However, after upload, Amazon takes time to provision resources and to boot a VM within EC2; this is included in the total post-processing time. We present results for two WAN bandwidths, 10 Mbps and 100 Mbps, in Table 4.17. In both cases, VM synthesis wins over the normal cloud workflow with a 4x improvement in the 10 Mbps case and a 2x improvement in the 100 Mbps case. The normal cloud workflow is bottlenecked on bandwidth because it must upload the full 514 MB compressed VM image, but VM synthesis reduces this to a much more compact 42 MB VM overlay. It is important to note here that pre- and post-processing for VM synthesis are artificially inflated because of the lack of native VM synthesis support and the convoluted mechanisms we needed to employ to work around limitations imposed by EC2.

4.8 Related Work

Offloading computation has a long history in mobile computing, especially to improve application performance and battery life [32, 77, 92]. The broader concept of cyber foraging, or "living off the land" by leveraging nearby computational and data storage resources, was first articulated in 2001 [95]. In that work, the proximity of the helper resources, known as "surrogates," to the mobile device was intuitively assumed, but how to provision them was left as future work. Since then, different aspects of cyber foraging have been explored by a number of researchers. Some of these efforts have looked at the tradeoffs between different goals such as execution speed and energy usage based on adaptive resource-based decisions on local versus remote execution [33, 34]. Other efforts have looked at the problem of estimating resource usage of a

future operation based on past observations, and used this estimate to pick the optimal execution site and fidelity setting [46, 75]. Many researchers have explored the partitioning of applications between local and remote execution, along with language-level and runtime tools to support this partitioning [10, 22, 27].

Since 2008, offloading computation from a mobile device over the Internet to a cloud computing service such as Amazon EC2 [100] has become possible. But, cloud computing places surrogates far away across a multi-hop WAN rather than nearby on a single-hop WLAN. A 2009 position paper [98] introduced VM-based surrogate infrastructure called "cloudlets." Proximity of offload infrastructure was deemed essential for deeply immersive applications where crisp interactive response requires end-to-end latency to be as low as possible. Recent application studies [24, 49] have confirmed the need for proximity of offload infrastructure when a mobile device runs interactive and resource intensive applications.

Aspects of the VM overlay concept can be seen in copy-on-write (COW) mechanisms. Sapuntzakis et al. [93] showed how COW could be applied hierarchically to VMs to create an efficient representation of a family of virtual appliances. Their following work, the Collective [19], advanced this approach and proposed a cache-based system to cope with various network conditions. Similarly, QCOW2, a widely used virtual disk file format, uses a read-only base image and stores modified data in a separate file [70]. Strata [85] combined union file system and package management semantics to easily create and deploy virtual appliances and to dynamically compose them. VM overlays, as articulated in [98], extend the base and modifications concept to both VM disk and memory state, and focuses on a mobile, cyber-foraging use case.

Some aspects of the optimization techniques proposed in this work have been individually investigated in other domains. Deduplication has been widely adopted in file systems, network storage, and virtualization. In file systems, it is used to reclaim storage space by detecting duplicated files or blocks. LBFS (Low Bandwidth File System) [73] is an example of a network file system that uses deduplication to reduce bandwidth demand. It introduced the use of Rabin fingerprints for defining content-based chunk boundaries that are edit-resistant. REBL (Redundancy Elimination at the Block Level) [60] applied deduplication along with compression and delta-encoding to achieve effective storage reduction. It introduced the concept of superfingerprints to reduce the computational effort of deduplication. Deduplication has also been used in the virtual machine space. Waldspurger removed duplicated memory pages and shared identical memory pages across multiple virtual machines to conserve memory on the host machine [113]. Several recent works have also used deduplication to reduce the cost of VM migration both within datacenters [122] and across WANs [118]. As our work uses deduplication in an offline stage, we can apply it aggressively across both disk and memory images.

Demand fetching of VM disk state was introduced by Kozuch et al. [58] and Sapuntzakis et

al. [93]. Both leveraged the fact that only a small portion of a VM disk is typically accessed in a session. Post-copy migration [51] applied demand fetching of VM memory to live VM migration to reduce network transmission costs and the total migration time. Post-copy migration immediately started the VM at the target destination instead of pre-copying a VM's memory state over mulitiple iterations. SnowFlock [61] combined demand fetching and packet multicasting to provide highly efficient and scalable cloning of VMs. It started VM execution on a remote site with only critical metadata and performed memory-on-demand, where clones lazily fetch portions of VM state as it is accessed. VM image Distribution Network (VDN) [83] used demand fetching of content-addressed chunks in the datacenter. VMTorrent [90] enabled scalable VM disk streaming by combining block prioritization, profile-based execution prefetch, and on-demand fetch. Our work also uses profiled prefetching and demand-fetching.

The significance of the semantic gap between VMMs and guest OSes was first articulated by Chen and Noble [20]. Later, Garfinkel and Rosenblum [37] coined the term virtual machine introspection and developed an architecture focusing on analyzing memory. Another effort to bridge this gap is VMWatcher [54], which enabled malware detection by introducing a technique called guest view casting to systematically reconstruct internal semantic views of a VM, such as files, processes, and kernel modules, in a non-intrusive manner. Kaleidoscope [17] exploited x86 architectural information (e.g. page table entries) to classify VM memory into sets of semantically-related regions and used this for better prefetching and faster cloning of a VM into many transient fractional workers. Our work bridges the semantic gap to minimize VM overlay size by identifying freed pages and blocks, and, to the best of our knowledge, is the first to use TRIM support for this purpose.

4.9 Chapter summary

Beyond today's familiar desktop, laptop and smartphone applications is a new genre of software seamlessly augmenting human perception and cognition. Supporting the compute-intensive and latency-sensitive applications typical of this genre requires the ability to offload computation from mobile devices to widely dispersed cloud infrastructure, a.k.a., cloudlets. Physical dispersion of cloudlets makes their provisioning a challenge. In this chapter, we have shown how cloudlets can be rapidly and precisely provisioned by a mobile device to meet its exact needs just before use. We have also shown that although our solution, dynamic VM synthesis, was inspired by the specific demands of mobile computing, it also has broader relevance to public cloud computing infrastructure.

Chapter 5

Adaptive Virtual Machine Hand-off

Once a user successfully provisions application's back-end server at cloudlet, the next question is "What happens if a mobile device user moves away from the cloudlet he is currently using?" As long as network connectivity is maintained, the applications should continue to work transparently. However, interactive response will degrade as the logical network distance increases. In practice, this degradation can be far worse than physical distance may suggest. For example, when moving from a home Wi-Fi network to that of a neighbor down the street, communication to the first home's cloudlet will require two traversals of "last-mile" links connecting the homes to their ISPs. How can we address this effect of user mobility? If the offloaded services on the first cloudlet can be seamlessly transferred to the second cloudlet, end-to-end network quality can be maintained. We refer to this capability as *VM handoff*. VM handoff resembles live migration in cloud computing, but differs considerably in the details such as metric and constraints.

5.1 Introduction

Seamless handoff is a core concept in cellular networks. Without any disruption to ongoing voice or data transmission, a user is able to freely move over a significant geographic area that is spanned by many base stations of limited individual coverage. In this chapter, we describe an analogous mechanism for cloud offload. Since virtual machine (VM) encapsulation is used for safety, isolation, resource allocation, and provisioning of multi-tenant cloudlets, we refer to our mechanism as *VM handoff*. As a user moves, his VM is seamlessly transferred from cloudlet to cloudlet or between cloud and cloudlet in order to preserve low end-to-end latency. This mechanism resembles live migration of VMs in data centers, but differs in at least three important ways.

First, the two mechanisms are optimized for very different performance metrics. The figure of merit in VM handoff is total time to completion, since degraded end-to-end latency persists

	5%	10%	50%	90%	95%
Home A & B	18.5	19.2	26.4	77.8	133.6
Home B & C	36.4	37.2	44.9	87.2	98.0
Home C & A	38.8	39.3	44.9	75.1	92.6

(a) Latency Distribution (milliseconds)

	5%	10%	50%	90%	95%
Home A & B	0.5	0.6	1.9	2.3	2.3
Home B & C	0.5	0.7	0.8	0.9	0.9
Home C & A	0.5	0.5	0.8	0.9	0.9

(b) Upload Bandwidth Distribution (Mbps)

These are the observed distributions of latency and upload bandwidth over a one-week period in November 2014 between three homes that are located within a one square mile area. All the homes have broadband Internet connectivity provided by ISP1 for A and B, and ISP2 for C.

Figure 5.1: Network Quality Between Homes

until the end of the operation. Live migration, on the other hand, aims at short duration of the very last step (called "down time"), during which the VM instance is suspended. The total time to completion is a secondary consideration. As we show in Section 5.5.1, this difference in optimization metric can result in an order of magnitude difference in total completion time. Second, the economics of cloudlet deployments force it to accept whatever network and computing resources exist across dispersed cloudlets. Unlike the data center assumptions of live migration, VM handoff cannot rely on the presence of a dedicated high-bandwidth network. Hence, our system needs to tolerate high variability in bandwidth and compute capacity due to workloads from other users even during the course of a single handoff. By dynamically adapting to these changing conditions, VM handoff can offer significant performance improvement as shown in Section 5.5.4. Third, VM handoff leverages the presence at the destination of a *base VM* from which its image was derived and uses this in combination with delta provisioning of VM states as originally proposed by Chapter 4.

The interactive response of such latency-sensitive applications will degrade as the logical network distance increases. As discussed in the next section, this degradation can be far worse than physical distance may suggest. VM handoff can mitigate this effect of user mobility while remaining transparent to applications.

5.2 Why VM Handoff?

In practice, how important is VM handoff to user experience? Consider the scenario of a user visiting his neighbor who lives just down the street. The user is running a mobile application that is latency-sensitive and is using the services of a cloudlet in his home. When he reaches the second home, he associates with a Wi-Fi access point there but continues to use the original cloudlet. Because of two traversals of last-mile links connecting the homes to their ISPs, the user is likely to see significant degradation of latency and bandwidth to his cloudlet.

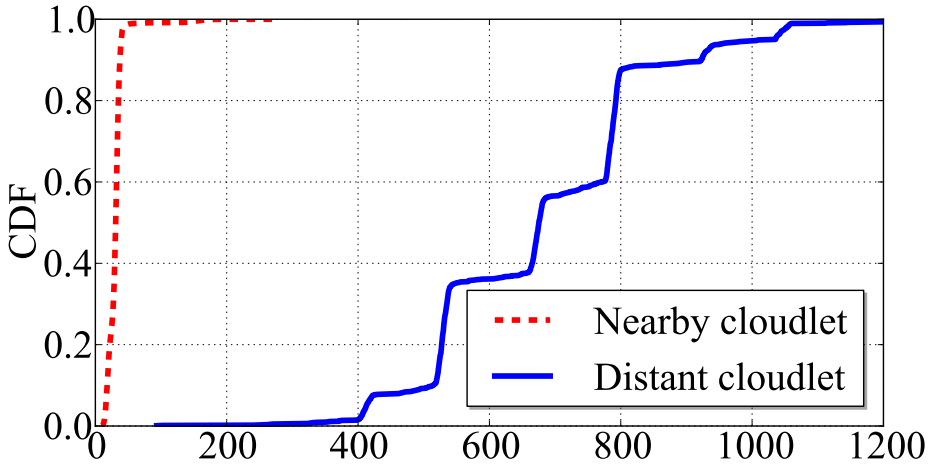
To validate this intuition, we measured network quality over a one-week period between the homes of three collaborators that are located within the same neighborhood in a city. As the results in Figure 5.1 show, end-to-end latency and bandwidth are poor in spite of physical proximity. For example, even though homes A and B connect to the same ISP, the median latency between them is 26.4 milliseconds. This is consistent with measurements reported by Sundaresan et al [106]. Homes that are connected to different ISPs can expect even worse network quality between them. For example, homes B and C are just one block apart, but the median latency between them is more than 40 milliseconds. Without VM handoff, even modest user mobility may result in unacceptable degradation of network quality to the associated cloudlet.

Degraded network quality translates into slower response times for latency-sensitive applications. To illustrate this, we evaluate the performance of three representative mobile applications. In our experiments, user interaction occurs on a mobile device while the compute-intensive processing of each interaction occurs in a back-end server encapsulated in a VM instance on a cloudlet. In Figure 5.2, the label "nearby cloudlet" corresponds to the case where the mobile device and cloudlet are both located at home C. In this configuration, the mobile device has one-hop Wi-Fi connectivity to its cloudlet. The label "distant cloudlet" corresponds to the case where the mobile device is at C, but its associated cloudlet is at A. In all cases, a nearby cloudlet yields significantly lower response times. For example, the median response time of `FACE` is 104 ms when it is associated to the nearby cloudlet, but it is 882 ms when using the distant cloudlet. Since this performance difference is solely due to differing network conditions, it confirms the importance of VM handoff.

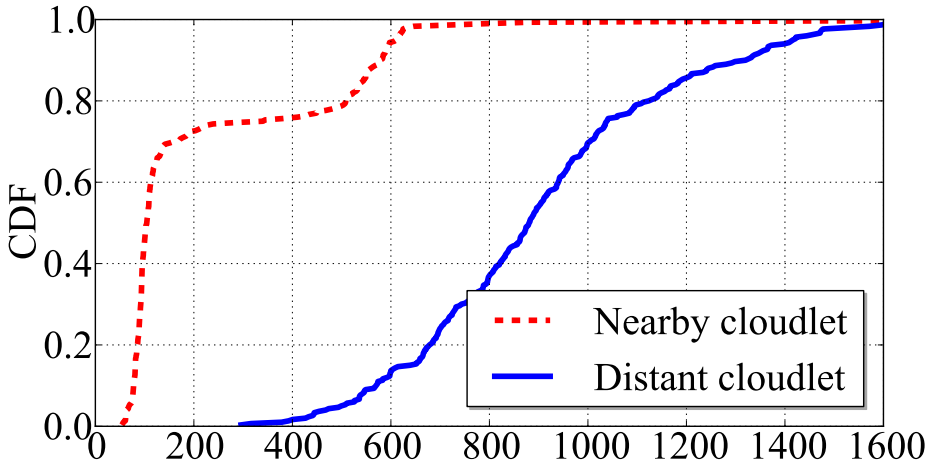
5.3 Background and Related Work

5.3.1 Live Migration for Data Centers

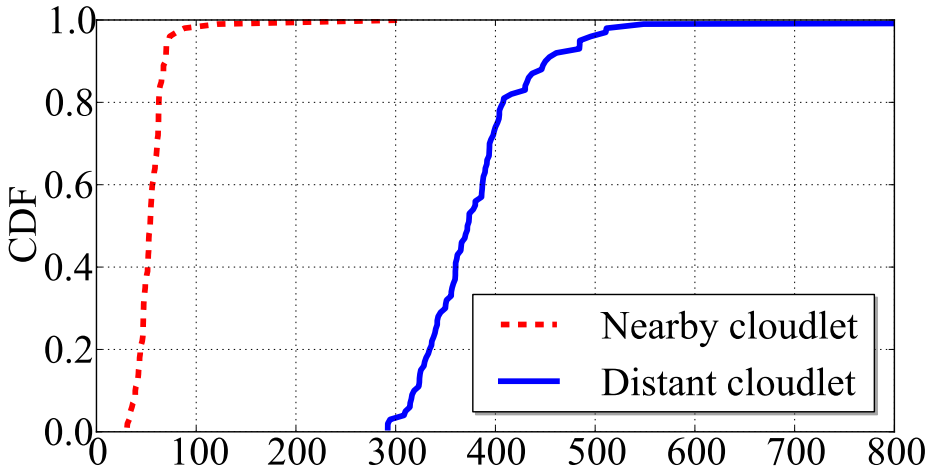
Live migration [23] is the de facto standard for VM migration in data centers. Its goal is to minimize "down time" during which a migrating VM instance is suspended. To achieve this goal, live migration allows the VM instance to continue execution at the source host while transferring



(a) Fluid Graphics (10 minutes run)



(b) Face Recognition (300 requests)



(c) Augmented Reality (100 requests)

Figure 5.2: CDF of Response Times (milliseconds)

VM	Total time	Down time	State transfer size
OBJECT	127 Min (0.03)	1.45 s (0.1)	8.42 GB (0.004)
MAR	159 Min (394)	7.44 s (0.3)	10.56 GB (0.43)

(a) No Base Image at Destination (`no-share`)

VM	Total time	Down time	State transfer size
OBJECT	12 Min (0.07)	1.54 s (0.3)	0.80 GB (0.004)
MAR	52 Min (20.4)	7.63 s (0.8)	3.45 GB (1.35)

(b) Using Base Image at Destination (`incremental`)

VM for both OBJECT and MAR are configured with 8GB disk and 1GB memory. The OBJECT guest operating system is Ubuntu Linux 12.04, while that of MAR is Windows 7. Average of 3 runs is reported with standard deviation in parentheses.

Figure 5.3: QEMU-KVM Live Migration on WAN

modified memory state in the background to the destination host. During the transfer, which may take many seconds to tens of seconds in typical usage, additional memory state may be modified by the executing VM instance. The entire process is repeated for multiple iterations (typically of shorter and shorter durations) until the very last step. In that final step, the VM instance is suspended at the source, all its remaining modified state is sent to the destination, and execution is resumed there. Through this convergent series of iterations, live migration minimizes down time to as little as hundreds of milliseconds on LAN. The entire migration process typically takes between tens of seconds to a few minutes.

As originally described, live migration does not transfer disk state; it only transfers memory state. This is acceptable in a data center environment because the source and destination hosts can be assumed to share disk storage through a mechanism such as a SAN (storage area network) or NAS (network-attached storage) device. A number of efforts [7, 15, 69, 121] have extended the basic live migration mechanism to work across data centers that are connected by a WAN. These efforts address the transfer of both memory state and disk state during migration, since shared-storage mechanisms do not work well over WANs.

5.3.2 Using Live Migration for VM Handoff

One may wonder whether VM live migration can be used with cloudlets because it is designed for seamless migration of computing states across machines. To verify feasibility, we conducted experiments using QEMU-KVM 1.1.1 in Ubuntu Linux 12.04. This production-quality VMM

implementation is widely used in OpenStack data centers for cloud computing [79]. As representative cloudlet workloads, we use `MAR`, the augmented reality application that was described in Section 5.2, and `OBJECT`, an object recognition application. The back-end server of `OBJECT` uses the MOPED algorithm [25] on an image sent by the mobile device, and returns the bounding box and identity of recognized objects. We performed live migration between two cloudlets that were connected by a stable 10 Mbps, 50 ms RTT network provided by a Linktropy emulator on a gigabit Ethernet.

QEMU-KVM hypervisor provides a mechanism to migrate both memory states and disk states, called *live block migration*. In live block migration, one can either transfer entire disk state to the destination or only transmit incremental disk state assuming the source and destination share base disk states. Figure 5.3(a) presents the results for the expected worst case, which transfers the entire disk state to the destination. The results show a total migration time of over two hours for both images, but a down time of just a few seconds. Two hours is a very long time for a mobile user to suffer degraded network access to his cloudlet. The fact that down time is just a few seconds is little consolation. Most mobile users would gladly accept much longer down times if it resulted in a faster switch to a nearby cloudlet.

Figure 5.3(b) presents the results when the base VM images into which `OBJECT` and `MAR` were installed are available at the destination cloudlet. One would expect the total state transfer, and hence migration time, to decrease significantly. This is indeed true for `OBJECT`, which completes the entire migration in just 12 minutes. Unfortunately, `MAR` behaves quite differently. The total migration time still takes more than 50 minutes with high variation. It turns out that this unexpected behavior is due to the timing of background activity in the Windows 7 guest. This background activity generates modified memory state at rate high enough to inordinately prolong live migration. And it is reflected in the large total volume of state transferred. Yet, down time remains less than 10 seconds for `MAR` in Figure 5.3(b).

In principle, one could re-tune live migration parameters to eliminate this pathological behavior. However, this could hurt down time in situations where that metric matters. Optimizing for the right context-sensitive metric is a non-trivial problem. Classic live migration is a fine choice for data centers, but inappropriate in a cloudlet context. Other possibilities include applying the concept of partial VM migration [13], and leveraging content similarity from VM images distributed across multiple nodes [62, 82, 88, 89]. These approaches are complementary to VM handoff, which focuses on achieving near-ideal post-handoff performance without reliance on external infrastructure.

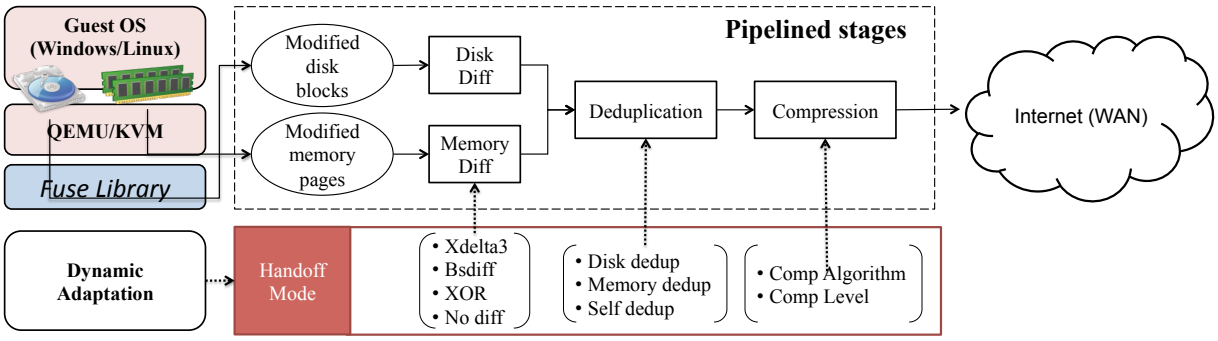


Figure 5.4: Overall System Diagram for VM Handoff

5.3.3 Dynamic VM Synthesis

Our implementation of VM handoff is inspired by VM synthesis described in 4. However, rather than overlay creation being a one-time offline operation, a series of overlays are generated afresh at runtime on the source cloudlet during the course of a single VM handoff. The time for overlay creation, which was ignored in earlier work because it was an offline operation, now becomes a significant limiting factor. In addition, the tuning parameters (such as compression algorithm) used in overlay creation are dynamically re-optimized at run-time in order to reflect the current relative costs of network transmission and cloudlet computation. Thus, although VM handoff borrows concepts from VM synthesis, it represents a substantial new mechanism in its own right.

5.4 Design and Implementation

Our design of VM handoff reflects the three considerations mentioned in Section 5.1: (a) optimizing for total handoff time rather than down time; (b) dynamically adapting to WAN bandwidth and cloudlet load; and (c) leveraging existing VM state at cloudlets. Figure 5.4 illustrates the overall design. A pipeline of processing stages is used to efficiently find and encode the differences between current VM state at the source, and already-present VM state at the destination. This delta encoding is then deduplicated and compressed (using parallelization wherever possible), and then transferred. The algorithms and parameters used in these stages are chosen to match current processing resources and network bandwidth. We describe these details below.

5.4.1 Tracking Changes

Our system is mostly implemented as modules separate from the hypervisor. This allows the system to be more flexible in controlling the resource usage for handoff, minimizes modifications to the hypervisor, and potentially allows support for different hypervisors, though our current implementation uses QEMU/KVM. On the other hand, it is more difficult to track VM disk and memory state changes from outside the hypervisor.

To efficiently track changes to VM disk state, our system uses the Linux FUSE interface to implement a user-level filesystem on which the VM disk image is stored. All of the running VM's disk accesses are passed through the FUSE layer, which can accurately track modified blocks with little performance impact to the running VM. At the start of handoff, our system can then immediately capture all VM disk blocks that differ from those in the corresponding standard base VM image. As in live migration, the VM can continue to run, and any further disk modifications will be tracked for subsequent iterations of transmission.

For tracking VM memory modifications, a FUSE-like approach would incur too much overhead on every memory write. Instead, we capture the memory snapshot at handoff, and determine the changed blocks in our code. To get the memory state, we rely on QEMU/KVM's live migration mechanism. We use a Unix socket to send a command to QEMU/KVM to start migrating state. This will cause it to mark all VM pages as read-only to trap and track any further modifications, and to start a complete transfer of the memory state. Rather than sending this state over the network, our system redirects this through a pipe to our processing stages that filter out unmodified pages, and heavily compress the remaining data before transmission using various techniques. As this capture of memory snapshot is based on a mechanism intended for live migration, QEMU/KVM will then iterate this process, sending pages that were modified over the duration of the previous iteration of modified pages. To limit repeated transmission of a set of rapidly changing pages, our system can regulate the start of these iterations, limit how many iterations are performed, or can eliminate them completely by pausing the VM before starting the memory snapshot.

5.4.2 Reducing Data Size

As the network bandwidth is often the bottleneck in VM handoff, our system tries to aggressively reduce the data volume transmitted across the network. We implement a pipeline of processing stages to delta-encode, deduplicate, and compress data before it hits the network.

We study the effectiveness of these processing stages in reducing data size on four applications that are representative of cloudlet workloads. The behavior of these applications, `OBJECT`, `FACE`, `MAR`, and `FLUID` are explained in Figure 5.2 and 5.3. Figure 5.5 shows that the system

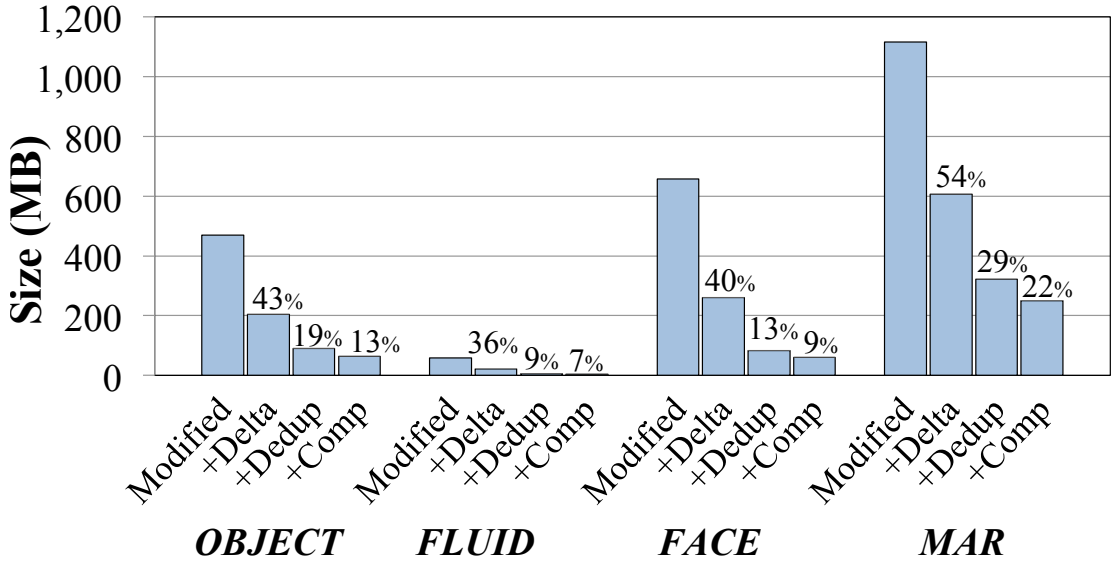


Figure 5.5: Cumulative reductions in size of VM state transferred

can significantly reduce the volume of data transferred to between 1/5 and 1/10 of the total modified data blocks. However, the processing costs of these operations may result in CPU, rather than network transfer, becoming the bottleneck. To avoid this, our system can use different algorithms and settings to balance the processing requirements and data size reductions. Details of these options are discussed below.

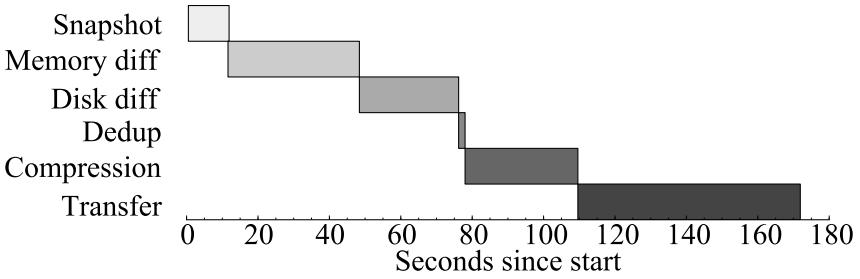
Delta encoding of modified pages and blocks: The streams of modified disk blocks and all VM memory pages are fed to two delta encoding stages (*Disk diff* and *Memory diff* stages in Figure 5.4). The data streams are split into 4KB (page/block size) chunks, and are compared to the corresponding chunks in the base VM. We use a SHA-256 hash [36] to make these comparisons. The hash values are preserved for later use. Chunks that are identical to those in the base VM are omitted. For each modified chunk, we use a binary delta algorithm to encode the difference between the chunk and the corresponding one in the base VM image, and only transmit the delta if it is smaller in size than the chunk. The idea here is that small or partial modifications are common and there may be significant overlap between the modified and original block when viewed at finer granularities. Our system can be dynamically configured to use either `xdelta3`, `bsdiff4`, or `xor` to perform the binary delta encoding, or to simply pass through modified chunks without delta encoding. As both the hash computations and the delta encoding steps are compute intensive, we parallelize these on multiple processing threads.

Deduplication: The streams of modified disk and memory chunks, along with the computed hash values, are merged and passed to the deduplication stage. Deduplication has been widely used and is very effective in reducing redundant data in a variety of contexts. Here, deduplication is particularly effective, as multiple copies of the same data commonly occur in a running VM. For example, multiple copies of the same data may reside in kernel and user-level buffers, or on disk and OS page caches. For each modified chunk, we compare the hash value to those of (1) all base VM disk chunks, (2) all base VM memory chunks, (3) a zero-filled chunk, (4) all prior chunks seen by this stage. The last is important to capture multiple copies of new data in the system, in either disk or memory. This stage filters out the duplicates that are found, replacing them with pointers to identical chunks in the base VM image, or that were previously emitted. As the SHA-256 hash used for matching was already computed in the previous stage, deduplication primarily reduces to fast hash lookups operations, so this stage can be easily run as a single thread.

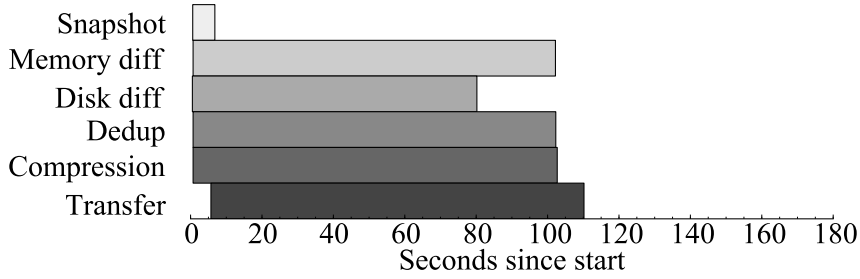
Compression: Compression is a final stage of processing before the VM modification data is sent to the network. In this stage, we attempt to squeeze the data further by applying one of several off-the-shelf compression algorithms, including `GZIP` (deflate algorithm) [28], `BZIP2` [18], and `LZMA` [116]. These algorithms vary in the data compression achieved and processing speed. GZIP provides relatively modest compression ratios, but uses very little processing time. LZMA is optimized for high compression ratios and fast decompression, but at the price of slow compression. BZIP2 falls in the middle in terms of compression rate and processing requirements. As these compression algorithms (particularly LZMA) work best on large blocks of data, this stage aggregates the modified chunk stream into approximately 1 MB segments before applying compression. Finally, as this is a CPU intensive stage, we run multiple instances of the compression algorithms in separate execution threads, sending segments to the threads in round-robin fashion. This lets us take advantage of all available cores, and parallelizes well, without requiring multi-threaded implementations of the compression algorithms.

5.4.3 Pipelined Execution

Our system pipelines the execution of these processing stages, so all of them are active simultaneously, and data is streamed through the various processing steps. This has two main advantages over a serialized implementation, where all data is processed through a particular stage before starting the next stage. First, it allows downstream stages to start before the preceding ones complete. In particular, we can start transferring data on the network quickly, in parallel to the processing stages. Increasing network bandwidth utilization is critical for VM handoff between cloudlets, where network bandwidth is a scarce resource that should not be wasted or left un-



(a) Serial Processing



(b) Pipelined Processing

Memory/Disk diff stage use `xdelta3` algorithm and compression stages uses LZMA compression level 5.

Figure 5.6: Serial versus Pipelined Processing

sused. Pipelining helps ensure migration data begins to reach the network as quickly as possible. Secondly, it requires less memory to buffer the intermediate data generated by the individual stages, as they are consumed quickly by downstream ones. Note that the total processing time is not significantly affected by the pipelining. This is because the total amount of computation is roughly the same, and even in the serialized case, our multi-threaded stages can make good use of multiple processing cores.

The example measurements in Figure 5.6 illustrate the benefit of a pipelined implementation. In the serial case, each stage has to be completed before the next stage can begin. In the pipelined case, all stages are available to begin processing data very soon after initialization. Figure 5.6 shows that pipelining produces a roughly 5% reduction in total processing time. (Serial processing takes 109.6 s and pipelined processing takes 102.7 s from its start to finishing compression). However, total handoff time is reduced by 36% (171.9 s $\rightarrow$ 110.2 s) because network transfers are overlapped with execution of the processing stages.

We also measure how well our pipelined system scales as the available CPU resources increase at the cloudlet. Figure 5.7 shows VM handoff time for different workloads with differing

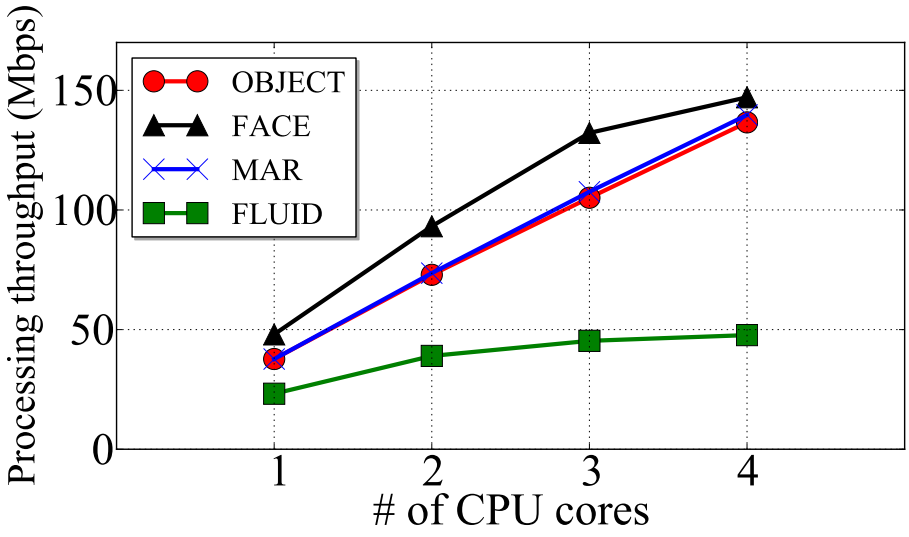


Figure 5.7: Processing Scalability of the System

number of CPU cores. Except for `FLUID`, the processing throughput increases as we use more cores. Much of the gain comes from parallel processing of the input data, since we use multiple execution threads for the stages that are CPU-intensive such as binary delta and compression. For `FLUID`, the total volume of data processed is too small to significantly benefit from multiple cores.

5.4.4 Dynamic Adaptation

In the example handoff performance data from Figure 5.6, due to the particular choices of algorithms and parameters used, the system is clearly processing bound. The processing takes 109.6 s, and dominates the total handoff time, while network transfer only requires 62.3 s. Here, it would have been better to select compression algorithms to reduce the processing requirements, even at the expense of data transfer size in order to reduce the overall handoff time.

If we knew exactly how much compression could be achieved and exactly how long this would take to transfer the modified VM state for all algorithms, and had guarantees on the available bandwidth, we could find a static configuration of processing stages to optimize the handoff time. However, we cannot know all of these in advance, as this is highly dependent on the actual data that needs to be transferred. Furthermore, network bandwidth can fluctuate significantly over time, as can available processing resources. Thus, selecting the best processing parameters *a priori* is not practical, and in any case, a static configuration may not remain the best choice as conditions change over the duration of handoff. Furthermore, the best static configuration for one workload might not work well for the other workloads because processing time and compression ratio vary depending on the workloads. Instead, our system employs continuous monitoring of

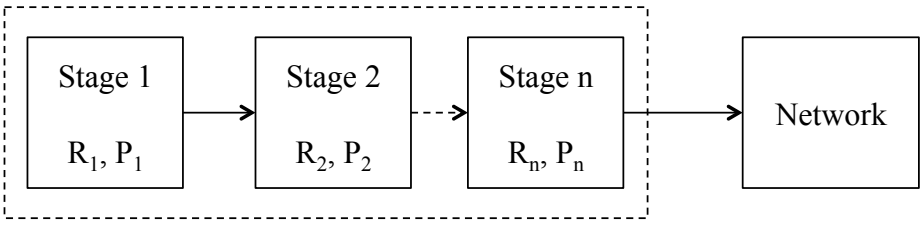


Figure 5.8: Pipeline modeling

handoff performance, and uses this information to dynamically adapt the processing stage settings to reduce handoff time, as detailed below. In the rest of this chapter, we refer to a set of selected algorithms and parameters for the processing stages as an *operating mode*.

Pipeline Performance Model

In order to develop an algorithm to select the right operating mode, we first develop a simplified model of our processing pipeline. The goal of this modeling is to calculate the system throughput by identifying the bottlenecks. Our pipelined system has two potential bottlenecks: processing and network transmission. On one hand, the handoff operation will be bottlenecked by the processing speed if the system is configured to aggressively reduce migration data size using processing-intensive algorithms and settings. On the other hand, the speed of handoff may be limited by the available network bandwidth if the pipeline does less processing in order to more fully utilize network bandwidth. In this modeling, we characterize system throughput with respect to these potential bottlenecks.

Figure 5.8 shows a simplified model of our pipeline. The processing is a sequence of stages $1 - n$, each of which takes input data and outputs a transformed, smaller version of the data. For each stage i, we define:

$$P_i = \text{processing time}, \; R_i = \frac{\text{output size}}{\text{input size}} \quad \text{at stage i}$$

These values are defined for a particular operating mode (set of selected algorithms and parameters for the processing stages). From these, we can compute the processing time as:

$$Time_{processing} = \sum_{1 \le i \le n} P_i \tag{5.1}$$

The time to transfer on the network is computed as:

$$Time_{network} = \frac{Size_{migration}}{\text{Network bandwidth}} \tag{5.2}$$

(where $Size_{migration} = Size_{\text{modified\_VM}} \times (R_1 \times \cdots \times R_n)$)

Since our pipeline overlaps processing and network transmission, from (5.1) and (5.2), the total total handoff time is:

$$Time_{handoff} = max(Time_{processing}, Time_{network}) \tag{5.3}$$

In the implementation, we use processing throughput and network transmission throughput instead of processing time and network transfer time, because calculating total transmission time requires undetermined information such as total input size. Therefore, the total system throughput is

$$Thru_{system} = min(Thru_{processing}, Thru_{network}) \tag{5.4}$$

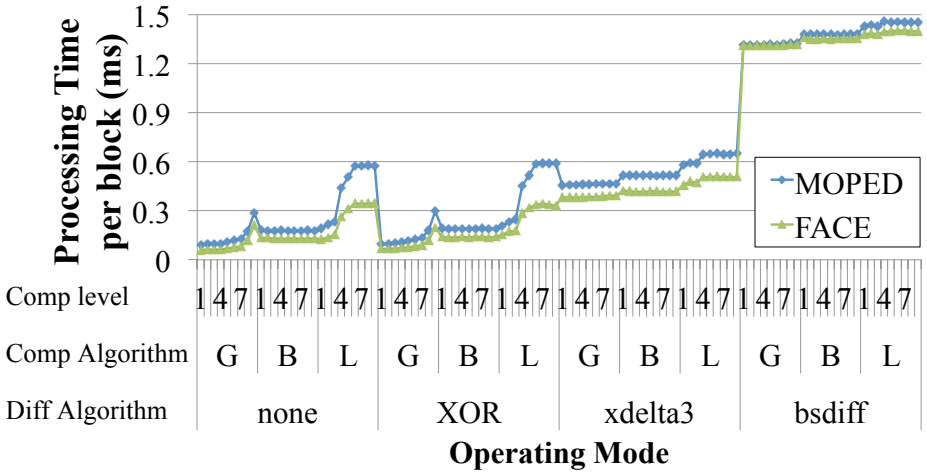
where,

$$\begin{aligned} Thru_{processing} &= \frac{1}{\sum_{1 \le i \le n} P_i} \\ Thru_{network} &= \frac{\text{Network Bandwidth}}{(R_1 \times \cdots \times R_n)} \end{aligned} \tag{5.5}$$

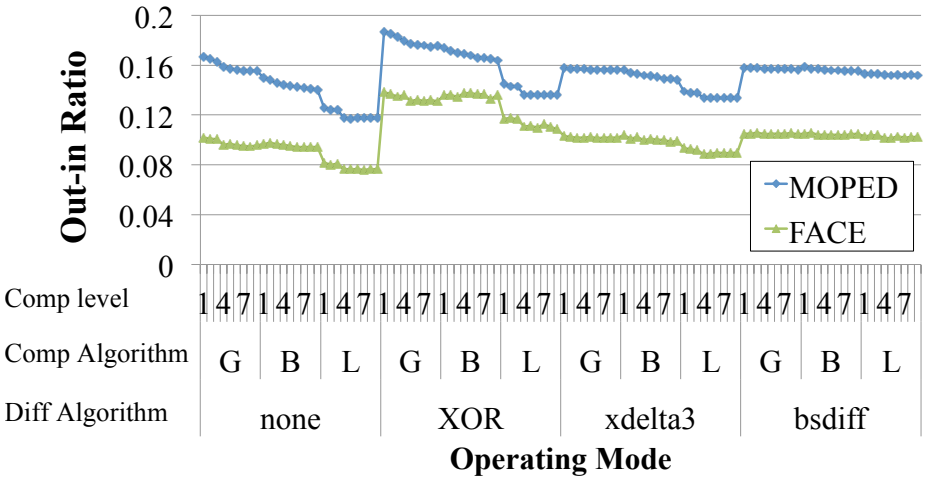
Intuitively, (5.5) shows our pipelined system is bottlenecked by either processing speed or network transmission speed. It also indicates that we can calculate the system throughput if we measure processing time (P) and out-in ratio (R).

Adaptation Heuristic

Applying this model to our system, we develop a heuristic to dynamically adapt the operating mode. The goal of the heuristic is to select the operating mode that maximizes the system throughput $Thru_{system}$. We profile the values of P and R for various workloads and operating settings. However, as discussed earlier, the actual computational demands and compression ratios achieved vary greatly depending on the actual content of the modified VM data. Hence, using the profiled P and R values directly may be highly misleading. Figure 5.9 illustrates this with measured P and R values for varying operating modes of two different workloads, `FACE` and `OBJECT`. Each data point in the figure is P or R per block (i.e. modified memory page or disk blocks) of a particular mode. As expected, the P and R values differ significantly for the two workloads even with the same compression settings. However, the ratio of P (or R) values between different operating modes are relatively stable between the different workloads. In other words, the trends of P and R in varying operating modes are similar across different workloads. The intuition behind this is that although one workload may be much harder than another, it impacts the various compression algorithms to a similar degree, and the relative performance re-



(a) Processing time (P) in different operating modes



(b) Out-in ratio (R) in different operating modes

Figure 5.9: Trends of P and R across workloads (G: `Gzip`, B: `Bzip2`, L: `LZMA`)

mains roughly similar. We confirm this on the 4 very different workloads used here: the absolute values of P and R of each workload are different, but their trends are similar. The robustness of this similarity across workloads from different developers and which span different operating systems, libraries, and databases suggests that it is a stable metric upon which base adaptation. Our heuristic uses ratios of P (or R) from the profiled data, rather than the absolute values, to determine which operating modes will likely minimize handoff time. Each iteration of the heuristic proceeds as follows:

1. First, measure the current P ($P_{current}$) and R ($R_{current}$) values of the running workload with the current operating mode ($M_{current}$). The system also measures the current network bandwidth by observing the rate of data segment acknowledgments received from

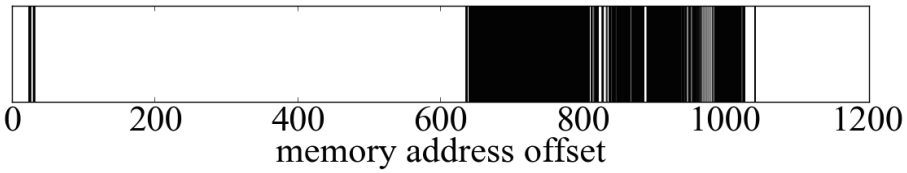


Figure 5.10: Modified Memory Regions (Black)

the handoff destination.

2. From the profile, find the profiled value P ($P_{profile}$) and R ($R_{profile}$) of the matching operating Mode, M. Then, compute the scaling factor for P and R.

$$Scale_P = \frac{P_{current}}{P_{profile}}, Scale_R = \frac{R_{current}}{R_{profile}}$$

3. Apply these scaling factors to "adjust" the profiled values for the current workload. Then, our heuristic calculates processing throughput ($Thru_{processing}$) and network transmission throughput ($Thru_{network}$) using (5.5), for each operating mode.
4. Finally, select an operating mode that maximizes the system throughput according to (5.4).

This heuristic can react to changes in the networking bandwidth, available processing resources (due to other loads on the cloudlet), or to changes in the compressibility of the VM modifications. In practice, the P values are actually measured in terms of processing time per input data block. The total possible combinations of settings (number of operating modes) is fairly small, so we can exhaustively enumerate them with little processing effort. The adaptation loop is repeated every 100 ms to let the system react quickly to any changes, or quickly discover any mispredictions. An updated operating mode will last for at least 5 s to provide hysteresis and give the system enough time to let effects of the change propagate throughout the pipeline, and reflect in stable measurements of P and R values.

5.4.5 Workload Distribution

The actual load placed on the processing pipeline and network are directly related to the number of modified pages and blocks that are transferred. For disk blocks, our change tracking mechanisms ensure only the modified disk blocks are delivered to the processing pipeline. For the memory image, however, the entire snapshot, including both modified and unmodified pages are processed. The relative loads on the network and processor, as well as our P and R values, vary depending on the ratio of modified and unmodified pages arriving at the pipeline.

Unfortunately, operating systems often manage memory space such that allocations, and

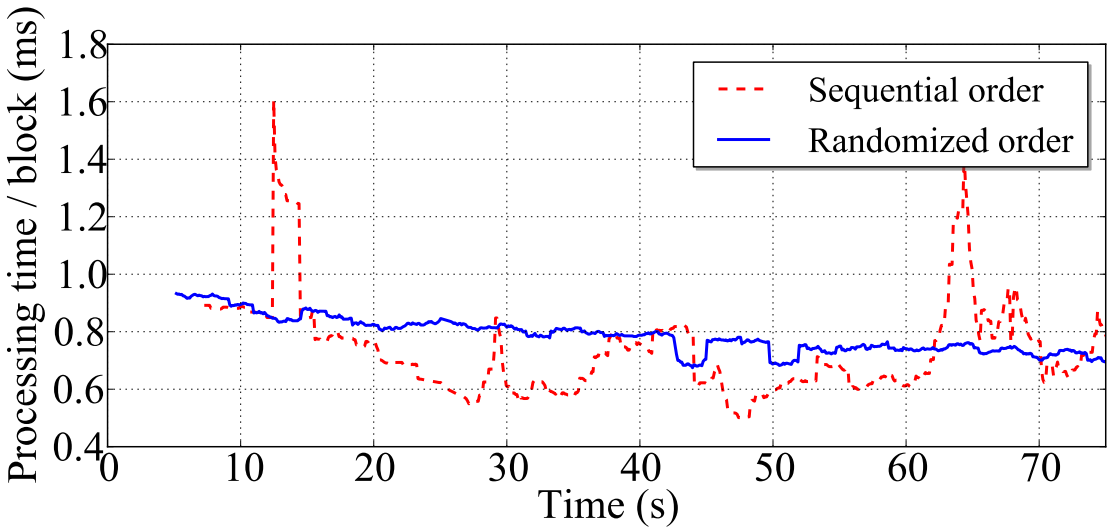


Figure 5.11: Randomized versus Sequential memory order (1-second moving average)

therefore modifications, tend to be clustered. Figure 5.10 illustrates the set of modified pages for a VM by physical address. Clearly, the modifications are non-uniform, and highly clustered. Sending this memory snapshot to our processing pipeline would result in a highly bursty workload. This is problematic for two reasons. First, long sequences of unmodified pages could drain later pipeline stages of useful work and may idle the network, wasting this precious resource. Long strings of modified pages could result in high processing loads, requiring lower compression rates to keep the network fully utilized. Both of these are detrimental to our goal of minimizing transfer time.

To address this problem and have a balanced workload during the process of VM handoff, we use a technique called *workload distribution*. Workload distribution randomizes the order of page frames passed by the hypervisor to our processing pipeline. This way, even if the memory snapshot has a long series of unmodified pages at the beginning of physical memory, all of our pipeline stages will quickly receive work to do, and neither processing nor network resources are left idling for long. More importantly, the ratio of modified and unmodified pages arriving at the processing pipeline remains stable compared to when we simply pass pages in the sequential, address order. Figure 5.11 shows how workload distribution moderates the processing time per block, eliminating both spikes and troughs. The spikes correspond to CPU-bound conditions, causing underutilization of network, while the troughs result in underutilization of CPU resources due to network bottlenecks. Workload distribution avoids the extremes and helps our system efficiently use both resources. Note that in this figure, no adaptation is performed (i.e., static mode is used), so the overall average processing times are the same for both plots. Furthermore, no network transfer is actually performed, so effects of network bottlenecks are not shown here.

5.4.6 Iterative Transfer for Liveness

VM handoff has to strike a delicate balance between brief service disruption and extended service degradation. If total handoff time were the sole metric of interest, the approach of suspending, transferring, and then resuming the VM would be optimal. Unfortunately, even with all our optimizations, this is likely to disrupt offload service too long (many minutes on a slow WAN) for good user experience. At the other extreme, if reducing the duration of service disruption were the sole metric of interest, classic live migration would be optimal. However, as shown earlier, this may extend degraded service unacceptably. The reality of mobile user experience is that neither extreme is acceptable.

Our solution to this problem is to borrow the concept of iterative transfer from live migration, but to embed it in the very different context of adaptive VM state transfer. As in live migration, the VM instance continues to run and accrue further changes which are transferred in subsequent iterations. However, unlike live migration, which focuses solely on volume of data transfer to drive the process, VM handoff is sensitive to multiple factors: data volume, processing speed, compression ratio achieved, and current bandwidth. Our system uses an input queue threshold to trigger the next iteration and uses the duration of an iteration to capture all factors affecting the speed of migration. If the duration of the iteration is sufficiently short, then our system suspends the VM and completes the handoff operation. We have empirically set the input queue threshold to 10 MB, and the interval to 2 seconds.

5.5 Evaluation

In this section, we evaluate the performance of our system for different workloads under various network and computing conditions. Specifically, we investigate our VM handoff system by answering the following questions:

- Is our system able to provide short VM handoff time on a slow WAN? How does it perform compared to classic live migration? (Section 5.5.1)
- How does it compare to classic live migration or other approaches like `Docker`? (Section 5.5.2)
- How effectively does our system select operating modes at a given condition? How well does this adaptation compare to the static modes? (Section 5.5.3)
- Is the implementation complexity of dynamic adaptation necessary? How dynamically does our system adapt its pipeline to varying conditions? (Section 5.5.4)

In our experiments, we emulate WAN-like conditions using the Linux Traffic Control (`tc` [68] tool), on physical machines that are connected by gigabit Ethernet. We configure bandwidth in

		1 CPU core		2 CPU cores	
BW (Mbps)		Handoff Time(s)	Down Time(s)	Handoff Time(s)	Down Time(s)
OBJECT	5	113.9	15.8 (6 %)	111.6	17.2 (7 %)
	10	66.9	7.3 (42 %)	58.6	5.5 (5 %)
	15	52.8	5.3 (12 %)	43.6	5.5 (31 %)
	20	49.1	6.9 (12 %)	34.1	2.1 (22 %)
	25	45.0	7.1 (30 %)	30.2	2.1 (26 %)
FLUID	5	25.1	4.0 (5 %)	17.3	4.1 (6 %)
	10	24.6	3.2 (29 %)	15.7	2.5 (4 %)
	15	23.9	2.9 (38 %)	15.6	2.2 (14 %)
	20	23.9	3.0 (38 %)	15.4	2.0 (19 %)
	25	24.0	2.9 (43 %)	15.2	1.9 (20 %)
MAR	5	494.4	24.0 (4 %)	493.4	24.5 (10 %)
	10	257.9	13.7 (25 %)	250.8	12.6 (13 %)
	15	178.2	8.8 (19 %)	170.4	9.0 (17 %)
	20	142.1	7.1 (24 %)	132.3	7.3 (20 %)
	25	121.4	7.8 (22 %)	109.8	6.5 (22 %)
FACE	5	247.0	24.3 (3 %)	245.5	26.5 (7 %)
	10	87.4	15.1 (10 %)	77.4	14.7 (24 %)
	15	60.3	11.4 (8 %)	48.5	6.7 (15 %)
	20	46.9	7.0 (14 %)	36.1	3.6 (12 %)
	25	39.3	5.7 (25 %)	31.3	4.1 (17 %)

Average of 5 runs and relative standard deviations (RSDs, in parentheses) are reported. For handoff times, the RSDs are always smaller than 9 %, generally under 5 %, and omitted for space. For down time, the deviations are relatively high, as this can be affected by workload at the suspending machine.

Figure 5.12: Overall System Performance

a range from 5 Mbps to 25 Mbps, according to the average bandwidths observed over the Internet [6, 115], and use a fixed latency of 50 ms. To control computing resource availability, we use *CPU affinity masks* to assign a fixed number of CPU cores to our system. Our cloudlet machines (both handoff source and destination) each have an Intel® Core™ i7-3770 processor (3.4 GHz, 4 cores, 8 threads) and 32 GB main memory. To measure VM down time, we synchronize time between the source and destination machines using NTP. For difference encoding, our system selects from `xdelta3`, `bsdiff`, `xor`, or null. For compression, it uses the `gzip`, `bzip2`, or `LZMA` algorithms at compression levels (1–9). We use `OBJECT`, `FACE`, `FLUID`, and `MAR`, described earlier, as VM handoff workloads. For each workload, a back-end server is running and ready to serve mobile clients.

5.5.1 Overall Performance

Figure 5.12 presents the overall performance of VM handoff over a range of network bandwidths. *Handoff time* is the total duration from the start of VM handoff until the VM resumes on the destination cloudlet. A user may see degraded application performance during this period. *Down time,* which is included in handoff time, is the duration for which the VM is suspended. Even at 5 Mbps, handoff time is just a few minutes and down time is just a few tens of seconds for all workloads. These are consistent with user expectations under such challenging conditions. As WAN bandwidth improves, handoff time and down time both shrink. At 15 Mbps using two cores, VM handoff completes within one minute for all of the workloads except `MAR`, which is an outlier in terms of size of modified memory state (over 1 GB, see Figure 5.5). The other outlier is `FLUID`, where the modified state is so small that there is not enough time for our adaptation mechanism to adjust behavior. Consequently, the numbers at different bandwidths are very similar.

How does VM handoff perform compared to off-the-shelf techniques? Figure 5.13(a) contrasts the performance of VM handoff and QEMU-KVM live migration over WAN (10 Mbps). It shows both variants of live migration supported by QEMU-KVM: `no share`, where all disk and memory state are transferred; and `incremental`, where the destination has a copy of the original disk, so only memory and modified blocks are transferred. VM handoff clearly outperforms KVM's no-share migration. Even with the incremental migration mode (which has a conceptually similar assumption to our approach), VM handoff improves total migration time by an order of magnitude. This is due to the aggressive use of deduplication and compression while maintaining balance between processing rate and network transfer rate.

5.5.2 Performance Comparison

We also compare VM handoff with `Docker` [29]. Docker provides process-level containers on top of a Linux kernel. It is becoming popular as a light-weight alternative to a full VM. Docker does not natively support migration of a running container, but with Checkpoint/Restore in Userspace (`CRIU`) [26], a form of migration can be achieved. This involves suspending the container and copying memory and disk state to the destination. So unlike live migration or VM handoff, down time is equal to total migration time. Figure 5.13(b) compares VM handoff to Docker migration for two Linux applications (`Docker-CRIU` only works for Linux apps). For `OBJECT`, VM handoff is two times faster even though it deals with whole VM state, rather than just a process and its dependencies, due to the aggressive optimizations used. For `FLUID`, the total state is so small that the VM handoff optimizations do not really kick in, and Docker has the edge in overall time, but a longer down time.

VM	Approach	Total time	Down time	Transfer size (GB)
OBJECT	Handoff	1 Min	5.5 s	0.06
	KVM(no-share)	127 Min	1.5 s	8.42
	KVM(incremental)	12 Min	1.5 s	0.80
FLUID	Handoff	16 Sec	2.5 s	0.007
	KVM(no-share)	124 Min	3.4 s	8.2
	KVM(incremental)	≈ 4 Min	2.4 s	0.234
MAR	Handoff	¡ 5 Min	12.6 s	0.27
	KVM(no-share)	159 Min	7.4 s	10.56
	KVM(incremental)	52 Min	7.6 s	3.45
FACE	Handoff	¡ 2 Min	14.7 s	0.08
	KVM(no-share)	147 min	6.0 s	9.9
	KVM(incremental)	104 Min	5.50 s	6.9

(a) Comparison with QEMU-KVM live migration

VM	Approach	Total time	Down time	Transfer size
OBJECT	Handoff	58.6 s	5.5 s	61 MB
	Docker	118 s	118 s	98 MB
FLUID	Handoff	15.7 s	2.5 s	7.0 MB
	Docker	6.9 s	6.9 s	6.5 MB

(b) Comparison with Docker

Figure 5.13: Comparison with Off-the-shelf Techniques at 10 Mbps BW, 2 CPU cores

5.5.3 Operating Mode Selection

VM handoff uses dynamic adaptation to select an operating mode for the given network conditions, processing resources, and workload. Figure 5.14 illustrates this, showing processing time and compression ratios achieved under varied bandwidth for the OBJECT workload. "CPU time" is the absolute CPU usage, in seconds, by VM handoff. "Compression ratio" is the ratio of

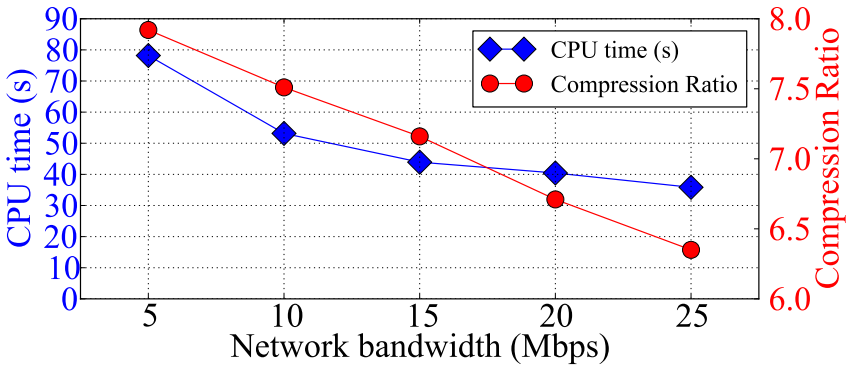
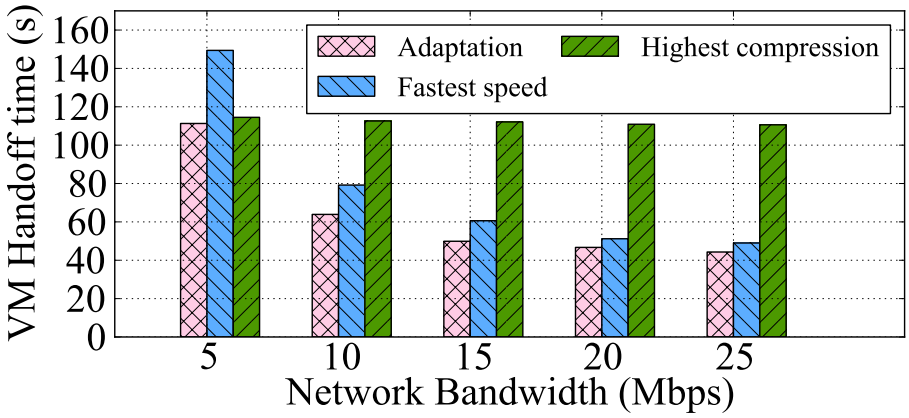
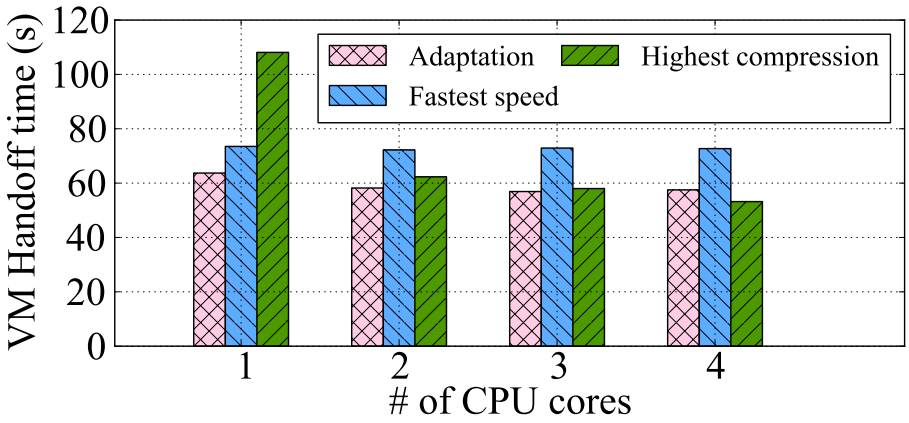


Figure 5.14: Performance Detail of OBJECT using 1 CPU core and Varying Network



(a) Varying network bandwidth with fixed 1 CPU core



(b) Varying CPU cores with fixed 10 Mbps BW

Figure 5.15: Adaptation VS Static Modes (`OBJECT`)

the input data size (i.e., modified VM state) to the output data size (i.e., final data shipped over the network). Our adaptation mechanism uses more CPU cycles to aggressively compress VM state when bandwidth is low, thus reducing the volume of data transmitted. At higher bandwidth, our system selects an operating mode consuming fewer CPU cycles, to avoid making processing a bottleneck. The average CPU usage remains high (between 80% and 90%) even though absolute CPU usage, in seconds, drops as the network bandwidth increase, indicating that the mechanism successfully balances computing speed and network transfer speed while using all available resources.

This trend in CPU time and compression rate is consistent across all applications except `FLUID`, which has so little modified state that there is not enough time for adaptation to be effective. Generally, our mechanisms are more effective when larger amounts of data are involved, so `OBJECT` and `FACE` (with 500 - 800 MB modified size) show median improvements, while `MAR` ($>$ 1.3 GB) and `FLUID` ($<$ 70 MB) are two extreme ends. Therefore for the following

experiments, we only present the results for the `OBJECT` workload. To avoid any experimental bias caused by optimization for a specific workload, we generate an adaptation profile (See section 5.4.4) using `FACE` and test using `OBJECT`.

Comparison with Static Modes

How effective is the adaptive approach over picking a static configuration? To evaluate this, we first compare against two distinctive modes in the spectrum of the operating modes: `highest compression` and `fastest speed`. For `fastest speed`, each stage is tuned to use the least processing resources to achieve fast processing of migration data. In `highest compression`, we exhaustively run all the possible combinations and choose the option that minimizes the data transfer size. Note that the most CPU-intensive mode might not be the highest compression mode; some configurations can incur high processing costs, yet fail to achieve high compression rates.

Figure 5.15(a) compares VM handoff time of the two static modes and adaptation. As expected, `fastest speed` performs best with high bandwidth, but works poorly with limited bandwidth. Except for the highest bandwidth tests, it is network-bound, so performance scales linearly with bandwidth. In contrast, `highest compression` minimizes the handoff time when bandwidth is low, but is worse than the other approaches at higher bandwidth. This is because its speed becomes limited by computation, and bandwidth is not fully utilized. It is largely unaffected by bandwidth change except in the very lowest bandwidth setting where network becomes a bottleneck. Unlike the two static cases that perform well only in certain bandwidth ranges, adaptation always yields good performance. In the extreme cases such as 5 Mbps and 25 Mbps, where the static modes have their best performance, the adaptation is as good as these modes. In the other conditions, it outperforms the static modes.

Figure 5.15(b) shows the handoff time for adaptation and the two static modes for differing numbers of CPU cores, with fixed 10 Mbps network bandwidth. `Fastest speed` shows constant handoff time regardless of available computing power, because it is not limited by processing, but by network transfer. `Highest compression` improves as we assign more CPU cores. Again, adaptation is better than or similar to the best performance of the static operating modes in all of the conditions.

Exhaustive Evaluation of Static Modes

We have shown that adaptation performs better than two distinctive static modes. Note that it is not trivial to determine *a priori* whether either of these static modes, or one from the many different possible modes, would work well for a particular combination of workload, bandwidth, and processing resources. For example, our adaptation heuristic selects 15 different operating

BW	Approach	Handoff time	Down time
5	Adaptation	113.9 s (3 %)	15.8 s (6 %)
	Best static	111.5 s (1 %)	15.9 s (12 %)
	Top 10%	128.3 s (2 %)	20.7 s (9 %)
10	Adaptation	66.9 s (6 %)	7.3 s (42 %)
	Best static	62.0 s (1 %)	5.0 s (11 %)
	Top 10%	72.1 s (1 %)	4.8 s (3 %)
20	Adaptation	49.1 s (8 %)	6.9 s (12 %)
	Best static	45.5 s (3 %)	8.1 s (15 %)
	Top 10%	48.5 s (1 %)	4.9 s (11 %)
30	Adaptation	37.0 s (4 %)	2.6 s (47 %)
	Best static	34.3 s (2 %)	2.1 s (8 %)
	Top 10%	48.5 s (1 %)	4.8 s (3 %)

Relative standard deviations are reported in parentheses.

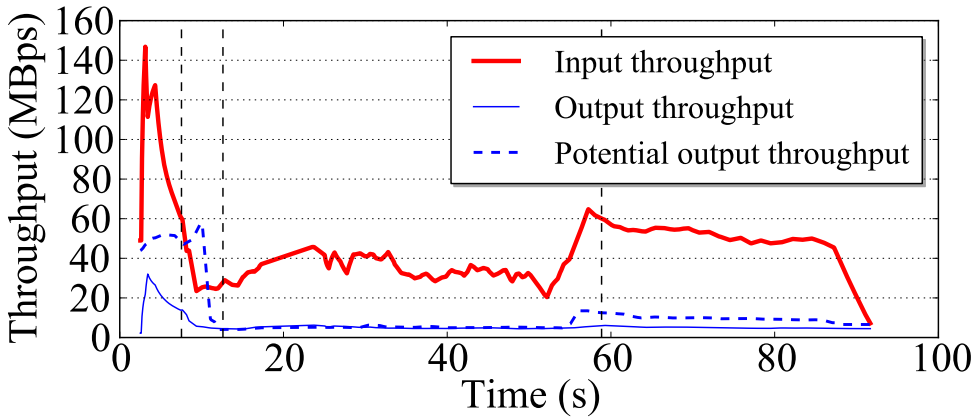
Figure 5.16: Performance Comparison Between Adaptation and Static Modes (`OBJECT`, 1 core)

modes for the `OBJECT` workload as bandwidth is varied between 5 Mbps and 30 Mbps. Furthermore, the selections of the best static operating mode at particular resource levels is unlikely to be applicable to other workloads, as the processing speed and compression ratios are likely to be very different.

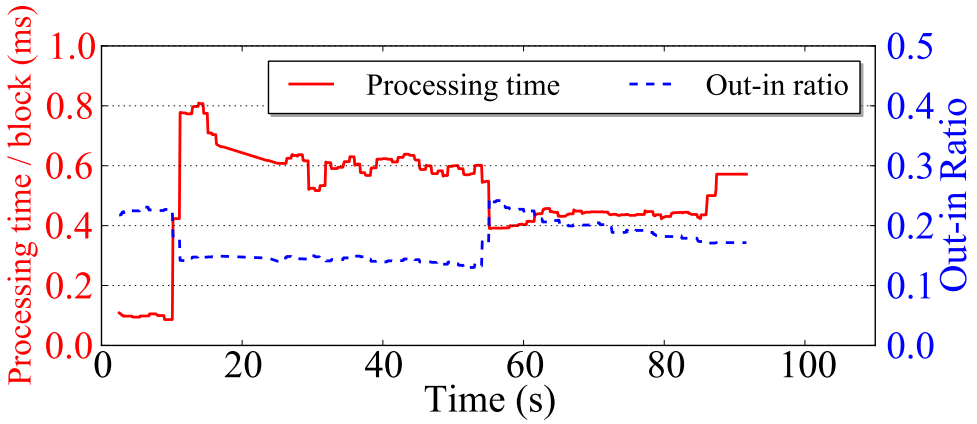
In spite of this, suppose we could somehow find the best operating mode for the workload and resource conditions. How well does our adaptation mechanism compare to this optimal static operating mode? To answer this question, we exhaustively measure the VM handoff times for all possible operating modes. Figure 5.16 compares the best static operating mode with adaptation for the `OBJECT` workload at varying network bandwidth. To compare adaptation results with the top tier of static operating modes, we also present the 10th percentile performance among the static modes for each condition. The adaptation results are nearly as good as the best static mode for each case. Specifically, the adaptation results always rank within the top 10 among the 108 possible operating modes in most of the cases (ranked top 19th at 20 Mbps network bandwidth). Therefore, our adaptation mechanism manages to select good operating modes across a wide range of conditions, with little loss compared to the best static operating mode for each condition.

5.5.4 Dynamics of Adaptation

Our VM handoff system uses dynamic adaptation to both select an ideal operating mode for a static set of of resources, and also to adjust the modes as conditions change. To evaluate how well this process works, we study traces of execution under both static conditions and varying conditions.



(a) System throughput trace



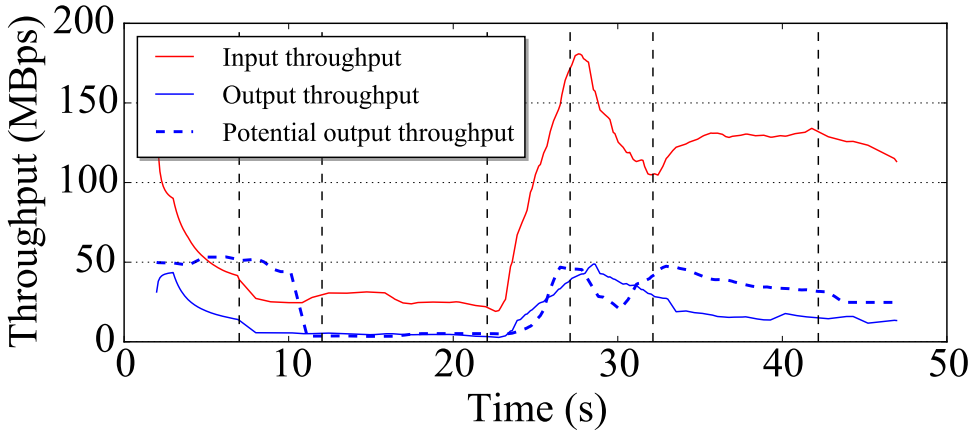
(b) P and R trace

Figure 5.17: Adaptation Trace Using 5 Mbps and 1 CPU core (`OBJECT`)

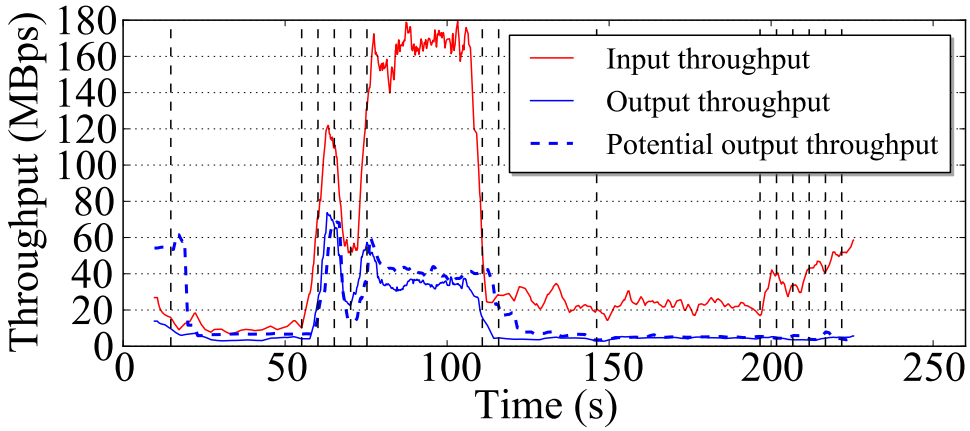
Adapting to Available Resources

We first demonstrate how well our system adapts to the available resources and network bandwidth. Figure 5.17(a) is an execution trace of our system, showing various throughputs achieved at different points in the system: output throughput, potential output throughput, and input throughput. Output throughput is the actual rate of data output generated by the processing pipeline to the network (solid blue line). Ideally, this line should stay right at the available bandwidth level, which indicates that the system is fully utilizing the network resource. If the rate stays above the available bandwidth level for a long time, the output queues will fill up and the processing stages will stall. If it drops below that level, the system is processing bound and cannot fully utilize the network. In the figure, we see that the output rate closely tracks the available network bandwidth (5 Mbps).

The second curve in the figure represents the potential output rate (blue dashed line). This shows what the output rate would be given the current configurations of the pipeline stages,



(a) `OBJECT`: BW change from 5 → 30 Mbps at 20 s



(b) `MAR`: BW change from 5 → 30 → 5 Mbps

Figure 5.18: Adaptation for Varying Network BW

if it were not limited by the network bandwidth. When using more expensive compression, the potential output rate drops. Ideally, this curve stays above the bandwidth line (so we do not under utilize the network), but as low as possible, indicating the system is using the most aggressive data reduction techniques without being CPU-bound. Here, too, we see that the system keeps this metric close to the network bandwidth limit.

The final curve is input throughput, which is the actual rate at which the modified memory and disk state emitted by QEMU/KVM is consumed by the pipeline (thick red line). This is the metric that ultimately determines how fast the handoff completes, and it depends on the actual output rate and the data compression. The system maximizes this metric, given the network and processing constraints.

The vertical dashed lines in the trace indicate the points at which the current operating mode is adjusted. As described in Section 5.4.4, the system bases the decision on measurements of P and R values (depicted in Figure 5.17(b)) made every 100 ms. A decision is made every 5 seconds

to let the effects of changing modes propagate through the system before the next decision point. In Figure 5.17(a), our heuristic updates the operating mode 3 times during the VM handoff.

During the first 10 seconds of the trace, we observe high peaks of input and output throughput. During this period, the empty network buffers in the kernel and the inter-stage buffers in our pipeline absorb large volumes of data without hitting the network. Thus, the transient behavior is not limited by the network bottleneck. However, once the buffers fill, the system immediately adapts to this constraint.

The first decision, which occurs at approximately 10 seconds, changes the compression algorithm from its starting mode (GZIP level 1) to a much more compressive mode (LZMA level 5), adapting to low bandwidth. The effects of the mode change are evident in the traces of P and R (Figure 5.17(b)), where processing time per block suddenly increases, and the out-in ratio drops after switching to the more expensive, but more compressive algorithm. In general, when the potential output is very high (with excess processing resources), the operating mode is shifted to a more aggressive technique that reduces the potential output rate closer to the bandwidth level, while increasing the actual input rate. Our system manages to find a good operating mode for this trace at this first decision point; the following two changes are only minor updates to the compression level.

Adapting to Changing Conditions

Finally, we evaluate our adaptation mechanism in a case where conditions change during handoff. Figure 5.18(a) shows a system throughput trace for `OBJECT`, where network bandwidth is initially 5 Mbps, but increases to 35 Mbps at 20 seconds. We use Linux `tc` to emulate BW change.

The monitoring module in our implementation observes these changes, and our adaptation system reacts quickly, ensuring a good operating mode is used throughout the trace. At the first decision point, the mechanism selects high processing, high compression settings (LZMA, level 9) to deal with the very low network bandwidth. The output rate is limited by network, but the input rate is kept higher due to the greater level of compression. When the bandwidth increases at 20 s, our system switches to the most light-weight operating mode (GZIP, level 1, No diff) to avoid being processing bound. Mode changes other than these two are minor changes such as compression level change.

We evaluated our system with more complicated condition changes using `MAR`, which provides longer handoff times and allows us to test multiple bandwidth changes. In this case, bandwidth starts at 5 Mbps, but increases to 35 Mbps at 50 seconds, and reverts back to 5 Mbps at 100 seconds. Figure 5.18(b) shows how the various system throughputs change over time. Similar to the `OBJECT` trace, at the first decision point, our system selects high processing, high

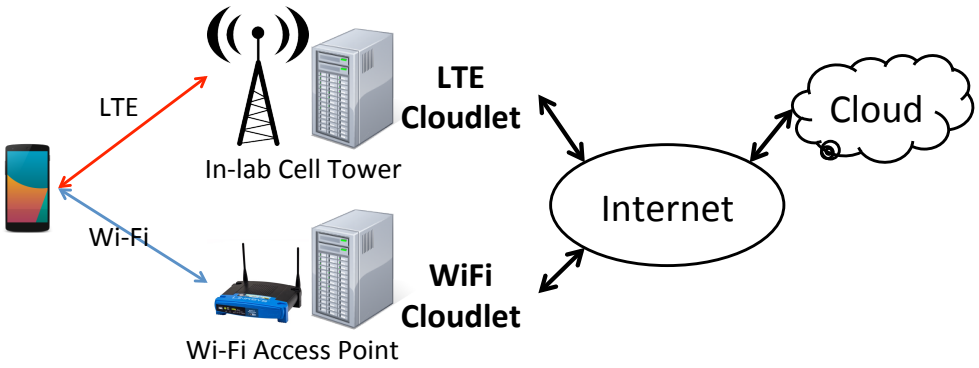


Figure 5.19: Network Setup for LTE-cloudlet WiFi-cloudlet

compression settings (LZMA, level 9) to deal with the very low network bandwidth. At 58 s, a major decision is made to switch back to GZIP compression to avoid being processing bound (as potential output is below the new network throughput). After a few minor mode changes, the system settles on a mode that fully utilizes the higher bandwidth (GZIP, level 7). Finally, a few seconds after bandwidth drops at time 100, our system once again switches to high compression (LZMA, level 9). The other mode changes are minor changes in compression level, which do not significantly affect P or R. Throughout the trace, the system manages to keep output throughput close to the network bandwidth, and potential output rate not much higher, thus maximally using processing resources.

When the network BW changes, no single static mode can do well – the ones that work well at high network bandwidth are ill-suited for low bandwidth, and vice versa. We verify that by comparing our result with all possible static operating in this varying network condition. For the `MAR` experiment, the best static operating mode completes VM handoff in 282 s, which is 31 s slower than our dynamic adaptation result, 251 s.

5.6 Experimental Deployment

We have created a deployment in our lab that includes cloudlets attached to the WiFi network and to an in-lab LTE base station. This testbed allows us to investigate the scenario of a user leaving his home, where he was using a local WiFi cloudlet, and needs to switch to one in the cell tower to maintain low-latency offload.

Our in-lab LTE network operates under a license for experimental use from the FCC, and is based on a Nokia eNodeB whose transmission strength is attenuated to 10 mW. The eNodeB is configured to filter traffic, redirecting matching packets to a machine that serves as the LTE cloudlet. Therefore, mobile devices connected to this base station communicate with the LTE

	Application	Total time	Downtime
LTE to WiFi	OBJECT	62.6 s (2.2)	6.8 s (0.6)
	FLUID	16.5 s (1.9)	2.3 s (0.6)
	MAR	255.0 s (0.3)	16.6 s (0.3)
	FACE	79.4 s (3.0)	18.2 s (4.5)
WiFi to LTE	OBJECT	64.1 s (0.6)	7.0 s (0.2)
	FLUID	17.9 s (2.7)	3.4 s (1.0)
	MAR	255.7 s (3.7)	15.9 s (2.7)
	FACE	84.2 s (2.0)	18.5 s (5.6)

10 Mbps WAN with 1 CPU core. Average of 3 runs is reported with standard deviation in parentheses.

Figure 5.20: VM handoff between `Cellular-cloudlet` and `WiFi-cloudlet`

cloudlet with minimal latency. Our WiFi-cloudlet is directly connected to the WiFi AP, so that it is just one hop away from a WiFi-connected mobile device. Figure 5.19 illustrates this configuration. To emulate the WAN between the cellular and WiFi networks and for easy comparison with Figure 5.12, we use Linux `tc` to regulate traffic to 10 Mbps, with 50 ms latency between the two cloudlets. Figure 5.20 shows performance measurement of VM handoff between the LTE and WiFi cloudlets. As expected, the total VM handoff time and down time are consistent with the previous results shown in Figure 5.12. When a user switches network between cellular and WiFi, the backend server can be migrated to a new cloudlet in a few minutes.

5.7 Chapter summary

The notion of cloudlet is getting broader acceptance [14, 16, 44]. However, even modest user mobility can result in significant network degradation for computation offloading. In this work, we propose VM handoff as a technique to preserve low latency across cloudlets. To minimize VM handoff time, various compression techniques are pipelined into a parallel processing. We present the first fine-grain adaptive mechanism for migrating VMs over WAN and show how dynamic adaptation can play an important role in order to cope with varying WAN bandwidth and cloudlet load. We also show this mechanism can be implemented in today's Internet speed improving VM handoff time at least one order of magnitude compared to the conventional live migration. Our experimental deployment using LTE-cloudlet confirms that VM handoff is applicable and practical on both LTE and WiFi network.

Chapter 6

Cloudlet Discovery

The dynamic discovery of a cloudlet by a mobile client is a unique problem in cloudlets. Because cloudlets are small data centers distributed geographically, a mobile device first has to discover, select and associate the appropriate cloudlet among multiple candidates before it starts using the cloudlet. These steps are unnecessary with a cloud because it is centralized. But in cloudlets, discovery and selection have to be carefully managed because the choice of a cloudlet can directly affect the provisioning time as well as the performance of the offloaded mobile application.

In principle, cloudlet discovery and selection are not very different from a regular resource discovery and selection, which is widely studied topic. Therefore, our cloudlet discovery system employs many existing ideas to perform efficient resource discovery in cloudlet context and focuses on practical aspects rather than presenting novel research challenges. The rest of the chapter is organized as follows. We first explain design requirements in Section 6.1. Then, we show our system design to meet the requirements and present a prototype implementation in Section 6.2 and Section 6.3. We provide a *Cloudlet Discovery API* in Section 6.4 and validate it using `WiFi-cloudlet` and `LTE-cloudlet` in Section 6.4.2.

6.1 Design Requirements

6.1.1 Supporting Disconnected Operation

The hierarchical organization of cloudlets explained in Chapter 3.1 was derived solely from the considerations of performance and consolidation. As a bonus, it also improves high availability of the cloudlet. Once a cloudlet has been provisioned for an associated mobile device, WAN network failures or cloud data center failures are no longer disruptive. This achieves disconnected operation, a concept originally developed for distributed file systems [57]. To provide the full benefit of the disconnected operation, our cloudlet discovery should be able to discover and select

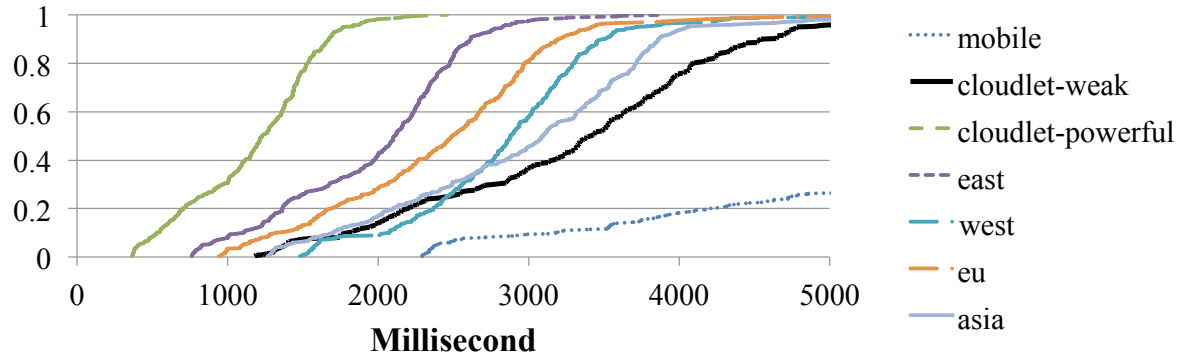


Figure 6.1: Cumulative distribution function (CDF) of response times in ms for `OBJECT`

a cloudlet without a public Internet connection.

The improved availability of the two-level architecture applies even to mobile applications that are not latency-sensitive. Any mobile application that uses the cloud for remote execution can benefit. Although not widely discussed today, the economic advantages of data center consolidation come at the cost of reduced autonomy and vulnerability to cloud failure. These are not hypothetical worries, as shown by the day-long outage of Siri in 2011 [86, 99], the multi-hour weather-related outage of Amazon's data center in Virginia in June 2012 [71], and the extended Christmas Eve 2012 outage of Netflix's video streaming service due to an Amazon failure [59]. As users become reliant on mobile applications, they will face inconvenience and frustration when a cloud service for a critical application is unavailable. These concerns are especially significant in domains such as military operations and disaster recovery.

Another aspect of the cloudlet discovery is how immediately it can find a newly created cloudlet. A cloudlet is a distributed element that is not controlled by a central authority. This means anyone can create a new cloudlet for one's own use or to share with others. The instant visibility of a newly created cloudlet is important to support cloudlet's distributed nature. In a hostile environment such as military or disaster recovery, this ability is critical to rapidly associated with the new cloudlet.

6.1.2 Application Aware Cloudlet Discovery

When a mobile device requests a cloudlet for offloading, different applications will have different preferences according to the application's characteristics. Most of the applications explained in Chapter 2 involve an interactive response. For these applications, network proximity between a mobile device and an associated cloudlet is the most crucial factor for selecting a cloudlet. However, some applications can be less sensitive to the network proximity, but being more affected by the power of the cloudlet's hardware such as CPU clock speed and memory size. Figure 6.1 shows a cumulative distribute function of response times for `OBJECT` when different offload-

	Cloudlet-Weak	Cloudlet-Powerful	Cloud (East, West, EU, Asia)
CPU	Intel® Xeon® E5320 1.86 GHz, 4 cores	Intel® Core® i7-3770 3.40 GHz, 4 cores	Amazon X-Large Instance 20 Compute Units, 8 virtual cores
RAM	4 GB	4 GB	7 GB
VMM	KVM	KVM	Amazon Xen

Figure 6.2: Hardware Configuration for Offloading Comparisons

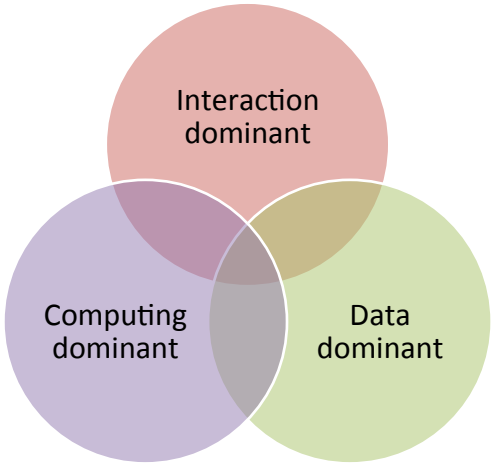


Figure 6.3: Cloudlet Discovery based on Application Characteristics

ing sites such as a cloudlet, Amazon East, Amazon West, Amazon EU, and Amazon Asia are used for offload computing. For all cases, the client and server are exactly identical and only the network proximity varies. Both `cloudlet-weak` and `cloudlet-powerful` are located just one hop away from a mobile device, but the hardware specification is different between two as shown in Figure 6.2. Although `cloudlet-weak` (black bold line) provides much better network connectivity to a mobile device (blue dashed line), it is slower than any Amazon data centers because the dominant component of the response time is the processing speed of the offloaded computation. In contrast, `cloudlet-powerful` (green dashed line) improves the response time dramatically and is superior to all Amazon data centers. This is because the hardware configuration (e.g. CPU clock speed) of the `cloudlet-powerful` is powerful so that it processes the input data much faster in addition to the improvement coming from the network proximity. The cloudlet's computing power is a dominant factor for the performance for this application.

Similarly, various factors can influence the performance of an application. Figure 6.3 shows a broad application classification in terms of performance factors related to a cloudlet context. The interaction dominant applications require tight network proximity. `Fluid` and Cloud gaming applications are good examples in this category. Low latency between a mobile device and associated offloading site is critical in this case and cloudlets by definition are helpful in this case.

An application is compute dominant if its performance is primarily determined by the computing speed. Many computer vision applications belong to this category (e.g. `OBJECT`). The location of a cloudlet is less important for applications in this category. Lastly, data dominant applications need large data or a subset of large data for processing a request. Map-based applications are a good example of this kind. If an application uses a very large data set in an unpredictable manner, there is a little chance for a cloudlet to optimize data placement. A cloudlet just has to fetch data as needed from a cloud, or if this is too expensive, restrict the application to running in the central cloud. However, many data-intensive applications typically use a subset of large data, and exhibit locality in access. In many cases, we can predict the likely data required from context, and hoard [57] this data at the cloudlet. For example, in a map application, physical location is highly correlated with accessed map data, so the geographical region around the cloudlet can be locally cached.

An extreme case of the application-specific cloudlet discovery is that a popular application provides its own discovery service. It is plausible that an expert of a highly successfully application knows the dominant performance factors of the application, and provides the best cloudlet discovery service directly to its customers. For example, a provider of cloud gaming can make a contract with providers of cloudlet infrastructure, and provides its own discovery service for its customer.

6.1.3 Discovery without Modifying Mobile Applications

So far, we assume that a mobile device needs to explicitly trigger a discovery request to associate with a cloudlet. And the return message contains a connection information to a back-end server hosted at the selected cloudlet. This implies that the mobile application needs to be changed to connect to the back-end server using the IP address returned from the discovery procedure. This is not a significant change because the application just needs to support configurable IP addresses for the back-end server. However, any modification on a mobile application can set a high bar for the developer. Sometimes it is not possible to modify legacy applications. Therefore, the cloudlet discovery system should also provide a way to support unmodified mobile applications.

6.2 System Design

Our design of cloudlet discovery is strongly influenced by the requirements described in the early section. We present an overview of the system and explain how we met these requirements. It is important to note that exact choice of each approach is configurable depending on the usage of discovery systems.

6.2.1 Global and Local Search

Since cloudlets are nearby offloading sites located at the edge of the Internet, it is natural to use `Zero-configuration` networking such as UPnP, Apple Bonjour, and Avahi to automatically find local cloudlets. `Zero-configuration` is a convenient way to find nearby cloudlets without asking for a user's manual intervention. However, the limitation of the local resource discovery is its coverage. These protocols rely on IP multicast (SSDP protocol for UPnP and mDNS for Apple Bonjour and Avahi) [21], so that the packets are often filtered by a network router. In practice, not every router can route IP multicasting packets and some of them intentionally filter out the multicasting packets for policy reason. In such cases, a mobile device can miss valid cloudlets. To compensate, we combine a local resource discovery with global search using a directory server. For global search, a mobile device connects to a pre-configured directory server to get promising cloudlets. In our system, both approaches, global and local discovery, are triggered in parallel.

6.2.2 Two-level Search

The dominant performance factor of offloading can vary depending on an application's characteristics. Many target applications are affected by the network proximity to a cloudlet. However, an application's performance can be bounded by other factors such as processing speed and data access. To support application-specific performance factor at the system level, we consider the following attributes for cloudlet discovery.

- **Network proximity**: Latency and throughput between a mobile device and a cloudlet. Low latency and high throughput are desirable for the interactive applications.
- **Resource availability**: Hardware specification of cloudlets to serve user's workload such as CPU clock speed, free memory size, network connectivity to a central cloud, and etc. Also, unique Hardware features such as GPU.
- **Cache states**: How much data for application execution is cached at the cloudlet. The cache states are particularly useful when a user follows a repeated routine in his daily life. Cache states can decide which cloudlet is more useful when multiple idle cloudlets with similar configurations are available.
- **Authentication**: Privilege (e.g. credential) to access cloudlet.
- **Miscellaneous**: Pricing for using cloudlet, and etc.

It is important to note that the performance aspects of an application are not determined by a single factor; rather the application is likely to have multiple performance attributes at the same time to different degrees. For example, the performance of `SPEECH` is largely determined by the

computing speed but it also requires good network condition to provide a crisp response upon a user's voice input. Likewise, MAR uses a large amount of data (e.g. database storing landmarks) to process a request and it also needs short latency to instantly overlay the result on a mobile device's screen. Therefore, our discovery and selection of cloudlet should be able to consider various attributes together.

As mentioned, the choice of attributes will be highly depending on an application. It is also interesting to note that two applications that provide similar functionality can have different performance attributes depending on how they interact with a user. For example, if a face recognition application is used for tagging faces in a photo, a user may not care about a few hundred milliseconds delay. However, if the face recognition is used for an augmented reality application, network proximity to an associated cloudlet is critical for performance.

How do we take different aspects of an application into account in the discovery process? One naive approach is to describe the application's details in the cloudlet discovery query. A directory server interprets the request and finds matching cloudlets from the database. However, a major problem of this approach is the overhead for saving and matching detailed attributes for every cloudlet. Although some cloudlet information such as a total number of cores and memory size are static, other information is transient. For example, resource information such as CPU usage and free memory size fluctuate over time. And cache information of the cloudlet changes frequently as VMs are accessing data for execution. Frequent uploads of cloudlets' dynamic states will cause not only a scalability problem but also a privacy issue. Since cloudlets are distributed elements, it is undesirable to upload details of each cloudlet to a single centralized authority.

To support application-specific cloudlet discovery, we design our system to take a two-level search for cloudlet discovery. We first find a list of promising cloudlets using a local/global search based on network proximity. Then, we contact a small set of cloudlet that is authorized to use in parallel to collect details of each cloudlet. The cloudlet's dynamic information is retrieved at this second stage. Scalability is not an issue and no sensitive information is leaked exposed globally because the information is shared only between a cloudlet and an authorized mobile device. In addition, the second step of connecting to all candidate cloudlet will be in parallel, so that it will add relatively small delay in cloudlet discovery.

6.2.3 Extensible Discovery Attributes

We envision cloudlets will be deployed everywhere as an infrastructure. For example, a user uses a cloudlet at home and migrates the state of a back-end server from home-cloudlet to a cell tower cloudlet when he leaves. At many different places such as airport, school, and work, cloudlet will provide value to nearby users. Accordingly, a cloudlet discovery system should be flexible

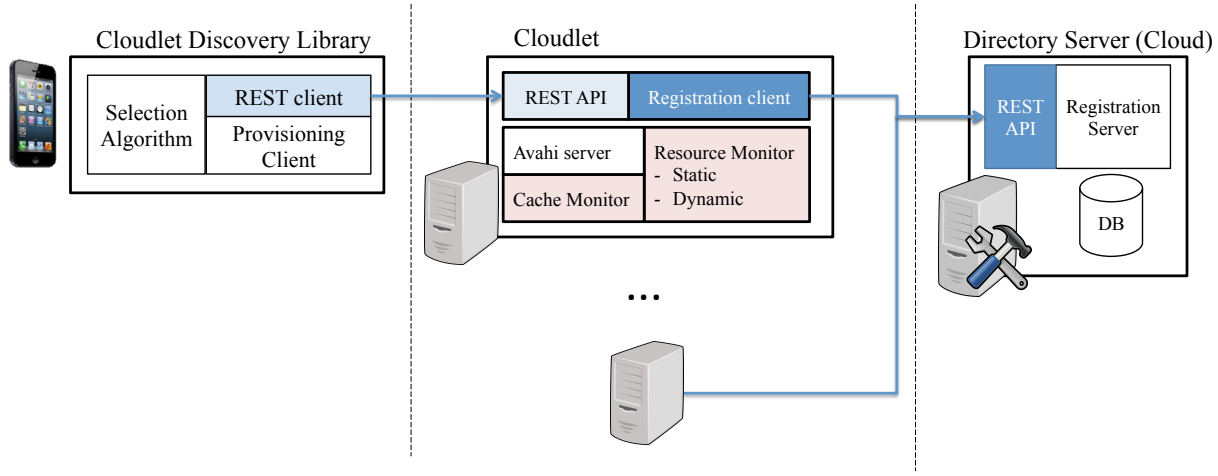


Figure 6.4: Overall Implementation of the Cloudlet Discovery System

enough to cover a wide range of usage scenarios. In other words, discovery attributes should be extensible to adapt to a new environment.

The difference between WiFi networks and cellular networks shows how preferred attributes of the discovery changes. Different from *WiFi-cloudlet*, *Cellular-cloudlet* is likely to work under a control of a network operator, because cloudlets in cellular networks need to be installed in the operator's infrastructure logically close to a cell-tower. In most of the cases, the association between a mobile device and a cloudlet is one-to-one mapping because there is only one cell-tower providing networking for a mobile device at a given moment. In this scenario, the selection of cloudlet for network proximity is straightforward. However, other aspects of discovery such as privilege, pricing model, and roaming condition start playing important roles.

To support a wide range of cloudlet deployment scenarios, cloudlet discovery attributes should be extensible such that a new attribute can be easily added and weights for each attribute are simple to adjust. In addition, the mechanism for adding/deleting attributes follows the industry standard, so that it can conveniently support 3rd-party algorithms for cloudlet discovery.

6.3 Implementation

We have built a prototype implementation of cloudlet discovery system that closely follows the design description in the previous section. A prototype uses Android mobile devices and x86-based workstations for cloudlet and cloud. Our cloudlet is running a Ubuntu 12.04 LTS server using Intel Core i7-3770 with 32 GB memory. As shown in Figure 6.4, the overall system is composed of three parts; a mobile device, a cloudlet, and a cloud. We explain each component of the system at the following sections.

Description	Method	API
Register a cloudlet	POST	URL : /api/v1/Cloudlet/ URL params - (option) ip_address: IP address of a cloudlet HTTP body (JSON) - status: "RUN" - rest_api_url: URL of cloudlet s rest API - rest_api_port: port # of of cloudlet s rest API - features: cloudlet s SW/HW specific features - meta: misc in key/value including static resource information
Unregister a cloudlet	PUT	URL : /api/v1/Cloudlet/ HTTP body (JSON) - status: "TER"
Update cloudlet status	PUT	URL : /api/v1/Cloudlet/ HTTP body (JSON): same as register method
Discover cloudlets	GET	URL: /api/v1/Cloudlet/search/ URL params: - (option) n: number of maximum cloudlet to search - (option) ip_address: IP address of a client device - (option) latitude: latitude of a client device - (option) longitude: latitude of a client device

Figure 6.5: Example of a Discovery Query to a Directory Server

6.3.1 Cloud Component (Central Directory Server)

A cloud component is a directory server that acts as a centralized authority storing basic information of cloudlets. This part is responsible for a global cloudlet search discussed in Section 6.2.1. Each cloudlet sends a registration request to the directory server when it starts and the cloudlet keeps updating its status using a heartbeat message. For communication between a cloudlet and the directory server, we use an RESTful API [114], which is a widely used approach for Web services. Each cloudlet will make an HTTP POST message to the directory server for an initial registration and use HTTP PUT for heartbeat messages. The RESTful API is also used for a mobile client to get cloudlet information. A mobile client will send HTTP GET message with credential information and parameters to search promising cloudlets. Full APIs are listed in Figure 6.5.

6.3.2 Cloudlet Component

The cloudlet part of the discovery system has two interfaces; one is an HTTP client module to communicate with the directory server in the cloud and the other is an HTTP server module to receive a query from a mobile device. A client module sends a registration message and heartbeat messages to the directory server. A server module receives a request from a mobile device using RESTful APIs. The server part is designed to support the application-aware two-level search discussed in Section 6.2.2. It returns detailed information of the cloudlet such as cache states and dynamic resource information upon a mobile device request.

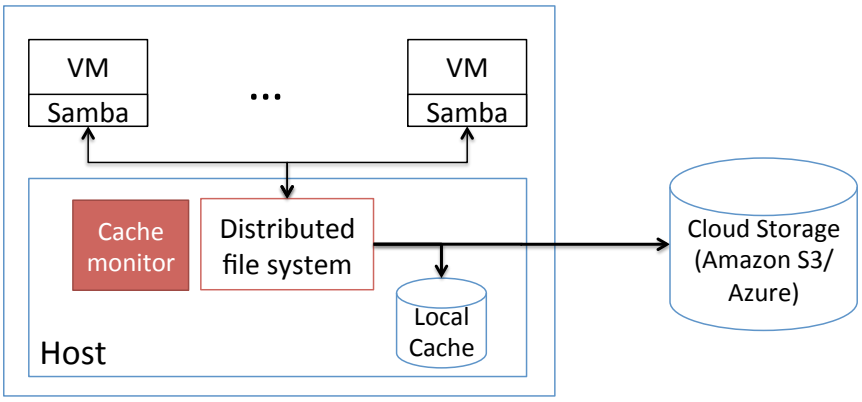


Figure 6.6: Overview of Cache Monitoring Daemon

In addition to the communication modules, cloudlets maintain two monitoring daemons; a resource monitor and a cache monitor. The resource monitor is responsible for checking static and dynamic hardware status. Static resource information such as a total number of CPU cores, CPU clock speed, and total memory size is updated to the directory server once when the cloudlet registers. And dynamic resource information such as CPU usage and free memory pages is directly passed to a mobile device whenever the mobile device sends a query.

Cache monitor keeps track of the cache states of a cloudlet. Figure 6.6 shows an overview of the cache-monitoring module. To cache data accessed at back-end servers running inside of VMs, we connect VMs and a host machine via `Samba`. As shown in the figure, each VM has a `Samba`-mounted directory that is mapped to a directory in a host machine. And this directory is managed by the distributed file systems such as Coda, NFS, and Ceph. In this way, a cloudlet can cache the back-end program's data using a distributed file system. There are a few practical issues in the current design of connecting VMs with a host machine, such as the namespace in the distributed file system and access control. In our prototype implementation, we show feasibility with a simplified implementation. The cache monitor interacts with the distributed file system to control cache policy and maintain cached files/data for each application. Based on the caching information, the monitor can answer a query from a mobile device.

The `Avahi` server is hosted in the cloudlet for local discovery discussed in Section 6.2.1. `Avahi` is a zero-configuration networking that implements the multicast DNS/DNS-SD protocol. It is widely used in *Linux*-like operating systems and provides a set of language bindings [9]. Using `Avahi`, a mobile device can discover cloudlets within a broadcasting domain without connecting to the cloud.

Query to a Directory Server	Result

<pre>{
  "application":{
    "required-RTT": 30,
    "app-id": "moped",
    "required-files":
      ["moped/**/*xml"]
  }
}</pre>
	<pre>{
	  'app_cache_files': [
	    'moped/kitchen/set1/coke.moped.xml',
	    'moped/kitchen/set1/fuze_boPle.moped.xml',
	    'moped/kitchen/set1/juicebox_front.moped.xml'
	  ],
	  'cpu_clock_speed_mhz': 3392.0,
	  'ip_address': '128.2.210.197',
	  'latitude': '40.4439',
	  'longitude': '-79.9561',
	  'mod_time': '2014-04-18T16: 27: 17',
	  'rest_api_port': 8022,
	  'rest_api_url': '/api/v1/resource/',
	  'status': 'RUN',
	  'total_cpu_num': 8,
	  'total_cpu_usage_percent': 5.1,
	  'total_free_memory_mb': 28377,
	  'total_mem_mb': 32129
	}</pre>

Figure 6.7: Example of a Discovery Query to a Directory Server

6.3.3 Mobile Device Component

A mobile device has a HTTP client to connect to both a directory server and a cloudlet. The workflow for discovering and selecting a cloudlet are as follows.

1. An application or a background service at a mobile device connects to a directory server using the *cloudlet discovery library* to find a list of promising cloudlets. At the same time, the `Avahi` client at a mobile device finds local cloudlets.
2. For every candidate cloudlet, a mobile device sends a query to each cloudlet in parallel to get detailed information such as resource availability and file cache state.
3. The cloudlet library chooses the best cloudlet for the mobile application based on the collected cloudlet information.
4. (Optional) *A provisioning library* performs provisioning of a back-end server of the mobile application to the selected cloudlet if needed. After provisioning, the cloudlet returns an IP address of the back-end server to the mobile application.
5. The mobile application starts and sends offloading requests to the back-end server using the returned IP address.

It is important to note the workflow can vary to handle special-cases. For example, if there is only one candidate cloudlet (e.g., pre-selected by the directory server), then the selection process will be skipped.

We have implemented a `cloudlet discovery library` for mobile clients and it

poses two queries during the discovery process; one to a directory server and the other to all candidate cloudlets. A query to the directory server is a simple HTTP GET message asking for nearby cloudlets. The second query is sent to every candidate cloudlet with application specific questions. Figure 6.7 shows an example query to a cloudlet. The query contains 1) application ID, 2) maximum allowed round trip latency, and 3) required files to run the application to check a cloudlet's cache status. These requirements are written in a JSON and the discovery APIs allows a user to extend it by adding key-value pairs. The return message from the cloudlet has an answer for the inquiry. For example, it lists all cached files that match with the request. It also forwards an IP address of a back-end server if it is already provisioned. Otherwise, it will return a URL to call the REST API for provisioning. The cloudlet selection algorithm makes a final choice using all information from cloudlets.

6.3.4 Achieving Application Transparency

Our implementation so far has required a slight modification on a mobile application to accept a new IP address of a back-end server. However, the cloudlet discovery should also support legacy applications or closed-source applications transparently. We show that we can perform cloudlet discovery without modifying a mobile application using a level of indirection technique such as 1) Domain Name System (DNS) and 2) HTTP redirection.

DNS-based approach

DNS is a hierarchical distributed naming system for resources and services connected to the Internet. It translates a domain name such as `elijah.cs.cmu.edu` or `google.com`, which is easy to be memorized by humans, to the IP addresses consumed by the underlying network systems. An example address resolution process is as follows. In DNS, a domain name server is responsible for translating a specific domain name in question by a sequence of queries starting with the right-most domain label. The address resolution process starts with a query to a root server and a root server will respond with a referral to more authoritative servers. For example, a query for `myapp.findcloudlet.org` will be first referred to the "org" servers. And the next query will be sent to one of the returned "org" servers and it will respond the address of the next level authoritative server, `findcloudlet.org`. This process is iteratively repeated until the client receives the IP address(es) of the domain. This hierarchical architecture can provide scalability, but it can increase response time because of the round-trip latency of multiple requests. A local DNS servers caches DNS query results for a period of time configured in the domain name record.

The translation from a domain name to IP address(es) is a level of indirection in address-

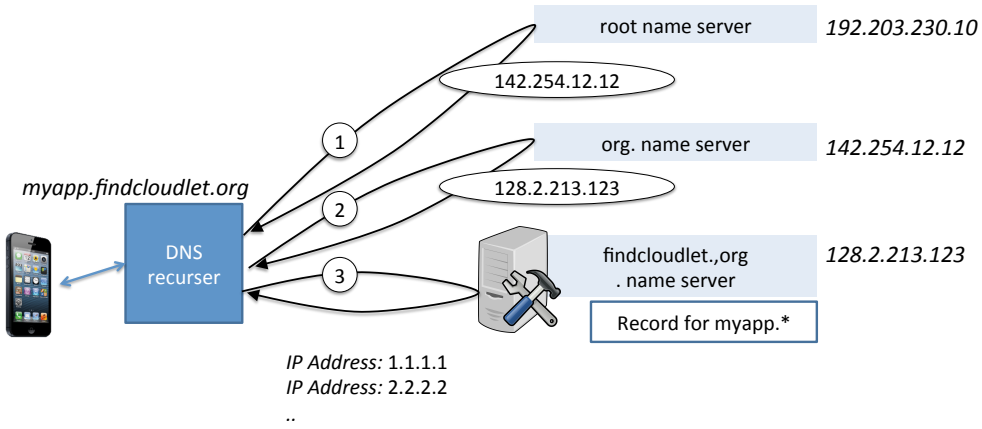


Figure 6.8: Steps for domain address `myapp.findcloudlet.org` resolution

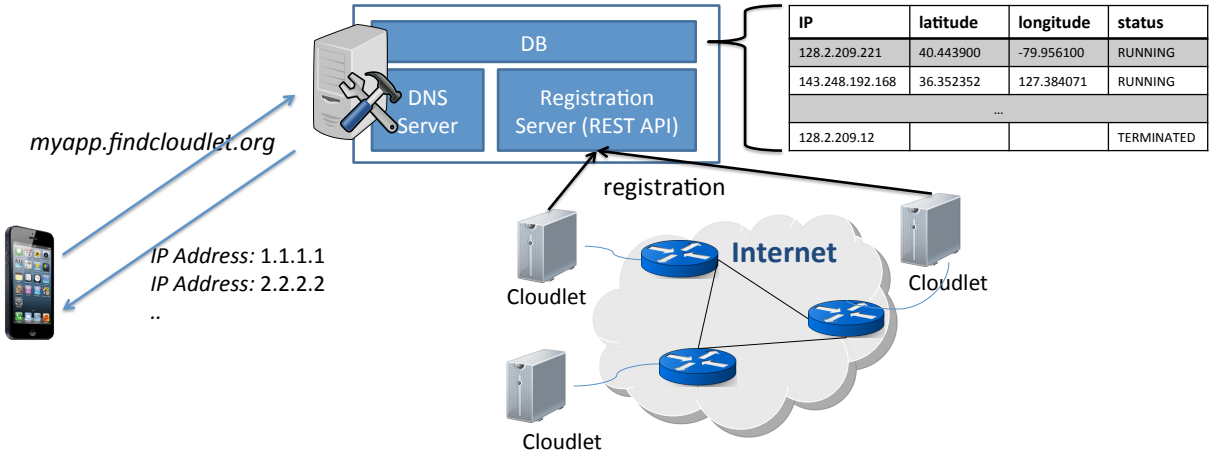


Figure 6.9: Overview of Transparent Cloudlet Discovery Using DNS

ing network endpoint, which we are going to leverage. Figure 6.8 shows steps for the translation with example domain, `myapp.findcloudlet.org`. If we own the domain name `findcloudlet.org`, we can register our own domain name server that is responsible for translating any sub-domain under findcloudlet.org. We can make the cloudlet discovery process transparent to a mobile device by returning the IP address of a selected cloudlet when a mobile device tries to connect to a server with domain name, `myapp.findcloudlet.org`. A closed-source mobile application will just connect to a back-end server using a predefined domain name, `myapp.findcloudlet.org`, but it will be redirected to the selected cloudlet.

In this case, the cloudlet discovery module works closely with a DNS server. To find and return an IP address of the best cloudlet upon the DNS query, every working cloudlet should first register itself to a centralized directory server so that the domain name server can get all cloudlet information. This structure is aligned well with the two-level-search explained in Section 6.2.2. Figure 6.9 shows an overview of transparent cloudlet discovery procedure using domain name

Figure 6.10: Example of HTTP 302 Redirection Response

server. Once a DNS server receives a DNS query, it first estimate the geographical location of a client device using geo-IP database. Then, it returns a list of promising cloudlets that are located near the mobile device. To make a newly created cloudlet visible to the mobile device, we set a very short expiration time at time to live (TTL) value to avoid caching at local DNS server.

The DNS-based approach does not require any modifications on a server-side program or a client-side program. However, there are some limitations. First, approximated location of the client from geo-IP database is not based on the mobile device's IP address but using a local DNS server's IP address that typically resolves domain name on behalf of a mobile device's request. Though the local DNS server is likely to be located close to the mobile device, it can be far away if a mobile user specifies the DNS server manually. Google and Yahoo have submitted a draft to IETF to propose a new option of DNS requests that recursive servers could include their own client's IP address to the upstream authoritative server [39]. It essentially allows a DNS server to get an IP address of a mobile device. The second limitation is lack of details in a cloudlet discovery request. Since we leverage the DNS protocol for the cloudlet discovery, the discovery protocol is limited by what DNS standard allows. As a result, cloudlet discovery will be purely based on the estimated geographical location of a mobile device. Other attributes such as a cloudlet's hardware specification or cache state cannot be considered in this approach.

HTTP redirection

HTTP redirection is a standard Web technique for making a web page available under more than one URL address. It is originally designed to support various scenarios such as preventing a broken link when Web pages are moved and URL shortening. HTTP redirection can be done

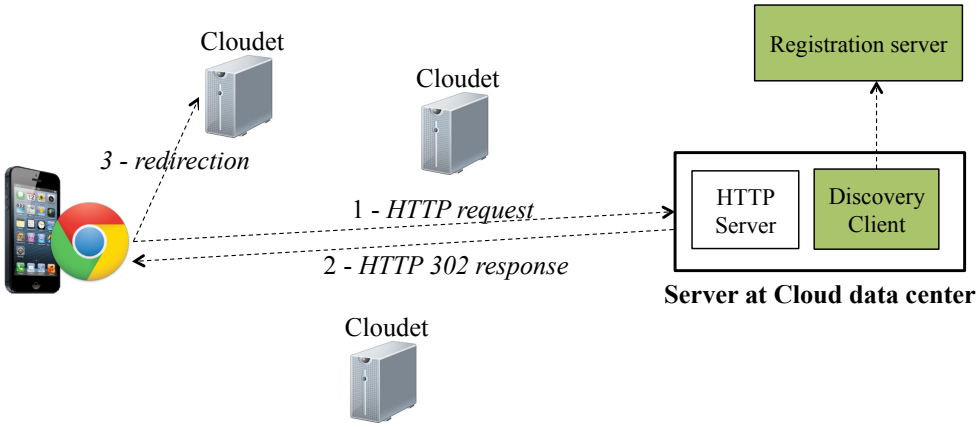


Figure 6.11: Overview of Cloudlet Discovery using HTTP redirection

using HTTP status code 3xx in HTTP protocol. HTTP/1.1 protocol defines several status codes for redirection and the following are some examples [52].

- **300 Multiple Choices**: Indicates multiple options for the resource that the client may follow.
- **301 Moved Permanently**: This and all future requests should be directed to the given URL.
- **302 Found**: Originally ”temporary redirect” in HTTP/1.0. Superseded by 303 and 307 in HTTP/1.1 but preserved for backward compatibility.
- **303 See Other**: Forces a GET request to the new URL even if original request was POST.

Figure 6.10 shows an example of HTTP response for a 302 redirection message. When a mobile application makes a connection to a HTTP server, the server will return HTTP redirection message including the redirection destination at the header. As a result, a HTTP client library will automatically make another HTTP connection to a new server using the received URL. The cloudlet discovery process leverages this redirection to re-route HTTP connection from an original server to a new HTTP server running at cloudlet. Figure 6.11 shows steps for cloudlet discovery using HTTP redirection mechanism. Similar to the DNS-based approach, every cloudlet first registers itself to a directory server before a mobile application makes a connection. When a HTTP request arrives at the HTTP server, it will consult to the registration server to find the best cloudlet based on the mobile device’s IP address. Different from the DNS-based approach, however, HTTP-based cloudlet discovery knows the exact IP address of the mobile device.

This discovery process is transparent to the application because the redirection happens at HTTP client library. So it does not require any modification on the client program. However, the HTTP server needs to be aware of cloudlets and should perform HTTP redirection. It is possible to avoid modification to the HTTP server if we perform the discovery operation at the

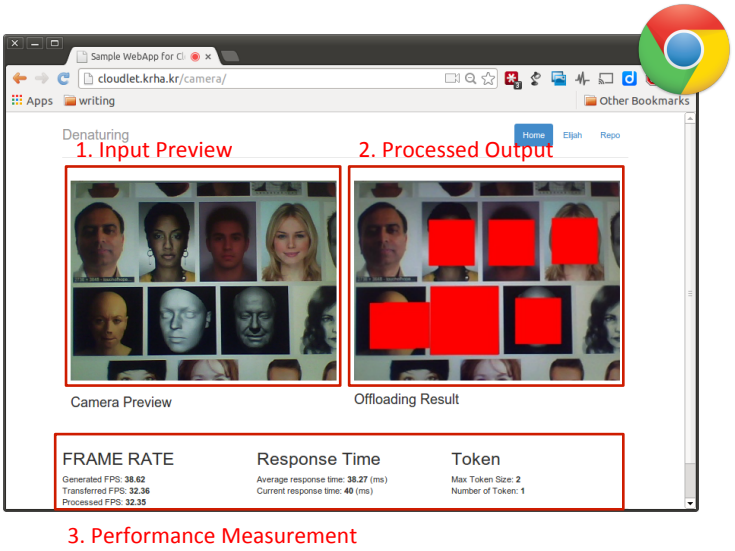


Figure 6.12: Example Web Application to validate HTTP-redirection-based cloudlet discovery

load balancer layer, last before the HTTP server. Then the load balancer will redistribute HTTP traffic to geographically dispersed edge nodes.

We have implemented an example Web application to show the feasibility of this approach. This Web application detects faces from a camera captured image and returns a frame in which all faces are removed. This is an example of denaturing for privacy protection explained in Chapter 2.4.1. A web client continuously transfers camera-captured frames to the Web Server. And the Web server returns new frames with face removed. The front-end is written in *HTML5* to access a camera at the Web browser and *WebSocket* is used for data transmission. We use the *Bootstrap library* to properly lay out the web page at both mobile screen and desktop screen [110]. The server-side hosts static web content using an *Apache* web server and dynamic content using *Jetty* (Java Serverlet Container). The WebSocket server is implemented using the Jetty framework [35] and it performs face detection on each image.

The process of cloudlet discovery for the example Web application exactly follows steps shown in Figure 6.11. The original server is located at the Amazon Asia data center. A web request from Pittsburgh is redirected to a cloudlet located at Pittsburgh via HTTP redirection. Since the application is interactively sending and receiving frames, the application response time improves greatly similar to what was shown in Chapter 2.1.

6.4 Cloudlet Discovery APIs and Validation

We provide cloudlet discovery APIs for the users to conveniently and programmatically perform the cloudlet discovery. As we shown in the steps of cloudlet discovery in Section 6.3.3, a mobile

user/device starts a discovery process right before using a cloudlet. Alternatively, 3rd-party entity can also generate a discovery message on behalf of a mobile user. For example in DNS-based approach in Section 6.3.4, an authoritative DNS server for a back-end server's domain name takes responsibility to discover the best cloudlet. The cloudlet discovery APIs provide a simple way to access a cloudlet. In this section, we illustrate details of the cloudlet discovery APIs and show how the cloudlet discovery works from the programmer's perspective. Also, we validate our cloudlet discovery system by showing the actual use of the APIs.

6.4.1 The Core of Discovery APIs

The main method of the cloudlet discovery APIs is `discover`. The following information is passed to the method as parameters to reflect various aspects discussed in system design.

- **IP address of a directory server**: For a global search, the IP address of a directory server is required.
- **Application information**: As mentioned in Section 6.1.2, the selection of the cloudlet depends on the application characteristics. As a result, application specific conditions such as maximum allowed RTT and minimum CPU clock speed are needed.
- (Optional) **Mobile device information**: Client device information can be valuable. For example, networking communication type, either WiFi or LTE, can be critical information for deciding between a `cellular-cloudlet` and a `WiFi-cloudlet`.
- (Optional) **Cloudlet selection algorithm**: To support 3rd-party cloudlet selection mechanism, a function pointer can be accepted as a parameter. The function should receive a list of cloudlets and return one cloudlet.

Figure 6.13 shows Python specification for the `discover` method. Cloudlet discovery APIs are available in Python, Java, and C programming languages and the following URLs shows API documentation, respectively.

- **Python**: https://libcloudlet.readthedocs.org/en/latest/
- **Java**: https://libcloudlet.readthedocs.org/en/latest/
- **C**: https://libcloudlet.readthedocs.org/en/latest/

It is important to note that taking a directory server's address as a parameter gives flexibility in cloudlet discovery by allowing vendor-specific directory server. One can pass either a generic directory server or a 3rd-party directory server provided by the 3rd-party vendor. For example in `cellular-cloudlet`, if a network operator controls all cloudlets and knows the best one for a user (See at Section 6.2.3), it can assign a particular cloudlet to a mobile user without going

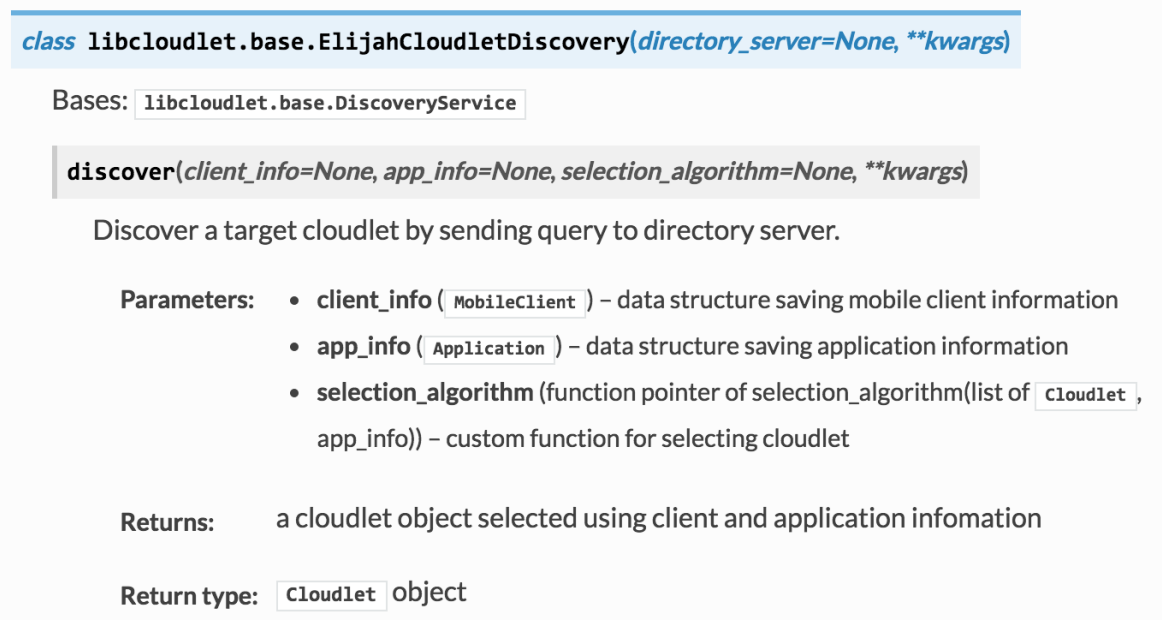


Figure 6.13: Example of Cloudlet Discovery APIs in Python

though the second level search. This can be done by returning a single cloudlet at the first level search when a mobile user uses a vendor-provided directory server address at the `discover` method.

6.4.2 Validating Cloudlet Discovery System using APIs

In this section, we validate our cloudlet discovery system by showing how the API works. We illustrate how each component of cloudlet discovery system interacts with each other upon the request of cloudlet discovery. The cloudlet discovery starts by calling the *discover* method. Figure 6.14 shows highlighted parts of the code. It first connects to a directory server to get a list of promising cloudlets, and then connects each cloudlet to retrieve detail information. Finally, using the cloudlet selection algorithm, it returns one cloudlet for a given application.

The first level search using a directory server: A mobile client poses a query to a directory server using simple HTTP GET message. It can optionally pass a geographical information of the device by appending parameters in the HTTP URL such as *http://findcloudlet.org/api/v1/Cloudlet/search/?latitude=40.4439?longitude=-79.9561*. If no information is given by the mobile device, the directory server estimates the location using a Geo-IP mapping. Then the server returns cloudlets based on the geographical proximity. The returned JSON message is converted into class objects.

```
class ElijahCloudletDiscovery(DiscoveryService):
  ...
  def discover(self, client_info=None, app_info=None,
               selection_algorithm=None, **kwargs):
    # first level search to get cloudlet list from a directory server
    cloudlet_list = self._list_cloudlets(self.directory_server, app_info)
    ...
    # second level search to each cloudlet
    self._get_cloudlet_details(cloudlet_list, app_info)
    ...
    # select the best one
    cloudlet = selection_algorithm(cloudlet_list, app_info)
    ...

    return cloudlet
```

Figure 6.14: API Code for *discovery* method

The second level search connecting for each cloudlet: Using the returned cloudlet list (cloudlet objects), the library makes HTTP connections to all cloudlets in parallel. Figure 6.15 shows *_get_cloudlet_details* method launching multiple threads and each of them calls Cloudlet object's *get_info* method to retrieve cloudlet details.

Selecting a cloudlet: Finally, the library selects and return one cloudlet using the *select_cloudlet* method. This method received a list of cloudlet objects and (optionally) application information

```
@staticmethod
def _get_cloudlet_details(cloudlet_list, app_info):
    thread_list = list()
    for cloudlet in cloudlet_list:
        new_thread = CloudletQueryingThread(cloudlet, app_info)
        thread_list.append(new_thread)
    for th in thread_list:
        th.start()
    for th in thread_list:
        th.join()

class Cloudlet(object):
  ...
  def get_info(self, app_info):
    ep = urlparse(self.REST_endpoint)
    params = json.dumps({'application': app_info.__dict__})
    headers = {"Content-type": "application/json"}
    with closing(HTTPConnection(ep.hostname, ep.port, timeout=1)) as conn:
      conn.request("GET", "%s" % end_point[2], params, headers)
      data = conn.getresponse().read()
      json_data = json.loads(data)
      setattr(self, app_info.get_appid(), json_data)
  ...
```

Figure 6.15: API Code for the Second Level Search

as parameters. The selection is usually a process of finding one cloudlet that meets a set of conditions such as maximum allowed network latency, minimum CPU clock speed, and size of cached data as shown in Figure 6.16. It can be also solved by linear optimization. It is also important to note that this code can be a custom method.

6.4.3 System Flexibility (Supporting 3rd-Party Provider)

We design our communication protocol as simple as possible following the industry standard to allow a 3rd-party entity to easily modify and extend it. We use JSON and RESTful interface between a mobile device and a cloud as well as between a cloudlet to a cloud. The simplicity in design not only gives system extensibility but also lowers the bar in integrating with other system. As an example, we bind our cloudlet discovery system with LDAP (light-weight directory access protocol) server, which is a widely used directory server. We start from the assumption that a cloudlet provider uses LDAP to maintain its cloudlets. And we show that we can easily

```
class ElijahCloudletSelection(object):
  ...
  @staticmethod
  def select_cloudlet(cloudlet_list, app_info):
    if len(cloudlet_list) == 1: return cloudlet_list[0]

    # filter out unmet cloudlets
    filtered_cloudlet = []
    for cloudlet in cloudlet_list:
      # check CPU min
      cloudlet_info = cloudlet[app_info.get_appid()]
      cloudlet_cpu = cloudlet_info[ResourceInfoConst.CLOCK_SPEED]
      required_clock = app_info[AppInfoConst.REQUIRED_CPU_CLOCK]
      if required_clock and cloudlet_cpu >= required_clock:
        filtered_cloudlet.append(cloudlet)
      # check rtt
      ...

    # check cache
    max_cache_score, max_cache_cloudlet = 0.0, None
    for cloudlet in filtered_cloudlet:
        cloudlet_info = cloudlet[app_info.get_appid()]
        cache_score = cloudlet_info[ResourceInfoConst.APP_CACHE_SCORE]
        if cache_score and cache_score > max_cache_score:
            max_cache_score, max_cache_cloudlet = cache_score, cloudlet

    # check application preference
    weight_rtt = getattr(app_info, AppInfoConst.KEY_WEIGHT_CACHE)
    weight_cache = getattr(app_info, AppInfoConst.KEY_WEIGHT_CACHE)
    weight_resource = getattr(app_info, AppInfoConst.KEY_WEIGHT_CACHE)
    ...

    return selected_cloudlet
```

Figure 6.16: API Code for Cloudlet Selection

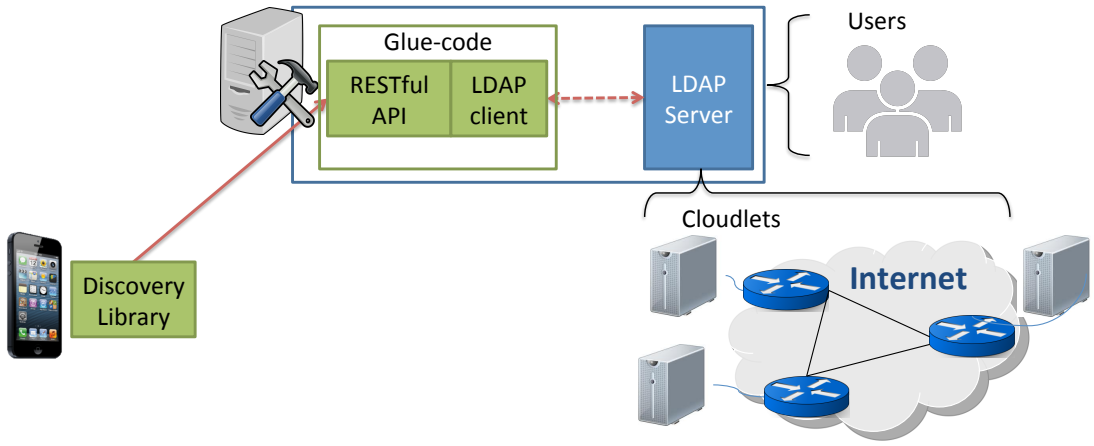


Figure 6.17: Overview of Cloudlet Discovery System binded with LDAP

bind our cloudlet discovery system with the LDAP server using a small glue logic. Figure 6.17 shows overview of the discovery system combined with LDAP. Instead of retrieving cloudlet information directly from the RESTful server (and its database), a glue logic acts as a LDAP client and extracts relevant cloudlet information from the LDAP server. Then it converts the results into a JSON after filtering out unnecessary information. This glue logic is short and straightforward so that it is less than 100 lines of code in our implementation.

6.5 Conclusion

Cloudlet discovery is a process of discovering and selecting the best cloudlet for a mobile application among all dispersed cloudlets at the edge of the network. Different from centralized cloud, cloudlet discovery has to be carefully managed because the choice of a cloudlet greatly affects the performance of the application. We have implemented a prototype cloudlet discovery system that supports application specific selection and disconnected operation using industry standard communication and messaging. In addition, we expand our system to support close-source mobile applications using DNS and HTTP redirection. We have focused on achieving flexibility and extensibility of the system to cover wide range of cloudlet deployment models.

Chapter 7

Deploying Cloudlet Infrastructure

In this chapter, we focus on the pragmatic aspects of cloudlet research; deployment of a cloudlet infrastructure. Since a cloudlet model requires reconfiguration or additional deployment of hardware/software, it is important to provide a systematic way to incentivise the deployment. And here we are facing a classic bootstrapping problem. We need practical applications to incentivize cloudlet deployment. However, developers cannot heavily rely on a cloudlet infrastructure until it is widely deployed. How can we break this deadlock and bootstrap the cloudlet deployment?

The history of the Internet offers a hint. The Internet is an open ecosystem that uses a standard protocol suite (e.g. TCP/IP). Through this open standard, multiple vendors from low-level hardware companies to high-level services providers independently participate. However, no single vendor is bearing large risk for improving this ecosystem. Instead, they are creating synergy by investing in their own business. In this ecosystem, innovation in one layer can stimulate others, resulting in additional investment. For example, the wide use of the Internet services such as email and web searching has encouraged Internet service providers (ISPs) to invest in this infrastructure. Infrastructure advances have become a foundation for new Internet services like

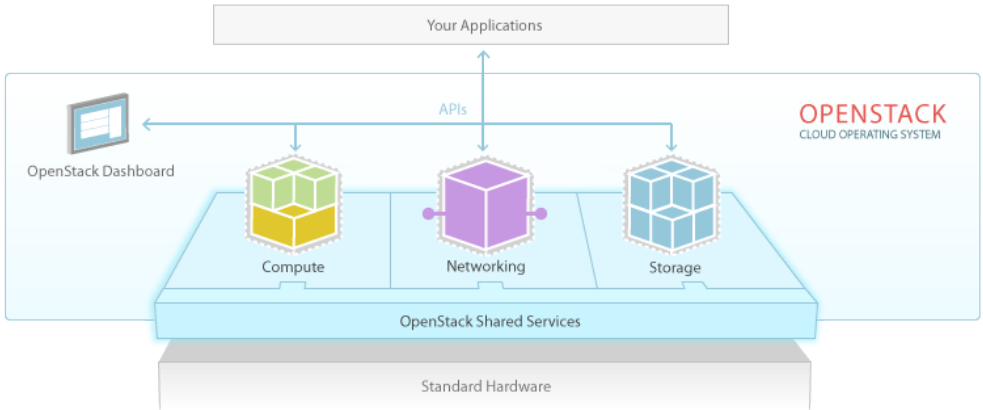


Figure 7.1: OpenStack Software Overview Diagram

Code name	Category	Description
Nova	Compute	Provision and manage large pools of on-demand computing resources
Swift	Object Storage	A scalable redundant storage system to save objects and files
Cinder	Block Storage	Persistent block-level storage devices for use with OpenStack compute instances
Neutron	Networking	A system for managing networks and IP addresses
Horizon	Dashboard	Self-service, role-based web interface for users and administrators
Keystone	Identity	Multi-tenant authentication system that ties to existing stores (e.g. LDAP) and Image Service
Glance	Image Service	Discovery, registration, and delivery services for disk and server images

Figure 7.2: OpenStack core modules

voice over IP (VoIP) and social networks.

We will take a similar strategy with cloudlets. Many of today's server-based mobile applications use the cloud as a back-end server. We observe that the ecosystem in cloud computing is similar to that of the Internet; hardware to software vendors are actively and independently participating for profit. For example, in hardware, network vendors such as Cisco and Juniper are deploying Software Defined Network (SDN) routers/switches, and blade server vendors like IBM and HP are reshaping their products [53]. Similarly in software, multiple hypervisors are rapidly developed to compete with each other, and various Linux vendors like RedHat and Canonical propose their own solutions for cloud computing.

OpenStack, a free and open-source cloud computing software platform, provides openness in the emerging cloud software ecosystem [79]. It offers a suite of standard APIs so vendors can independently contribute to different layers without breaking compatibility. It began in 2010 as a joint project of Rackspace Hosting and NASA. Currently it is managed by OpenStack Foundation, a non-profit corporate entity established in 2012. OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed through a dashboard that gives administrators control. It has a modular architecture with code names for its components. The major modules and their description are listed in Figure 7.2. As of 2015, more than 500 companies have joined the project, including AT&T, AMD, Canonical (Ubuntu), Cisco, Citrix, Dell, EMC, HP, Huawei, IBM, Intel, Red hat, and Yahoo [4]. In every single business category, the top three vendors support or participate in OpenStack. As this large number of participants indicates, OpenStack is becoming the *de facto* standard open-source cloud computing platform.

We will leverage this open platform to expedite cloudlet deployment. That is, we will make

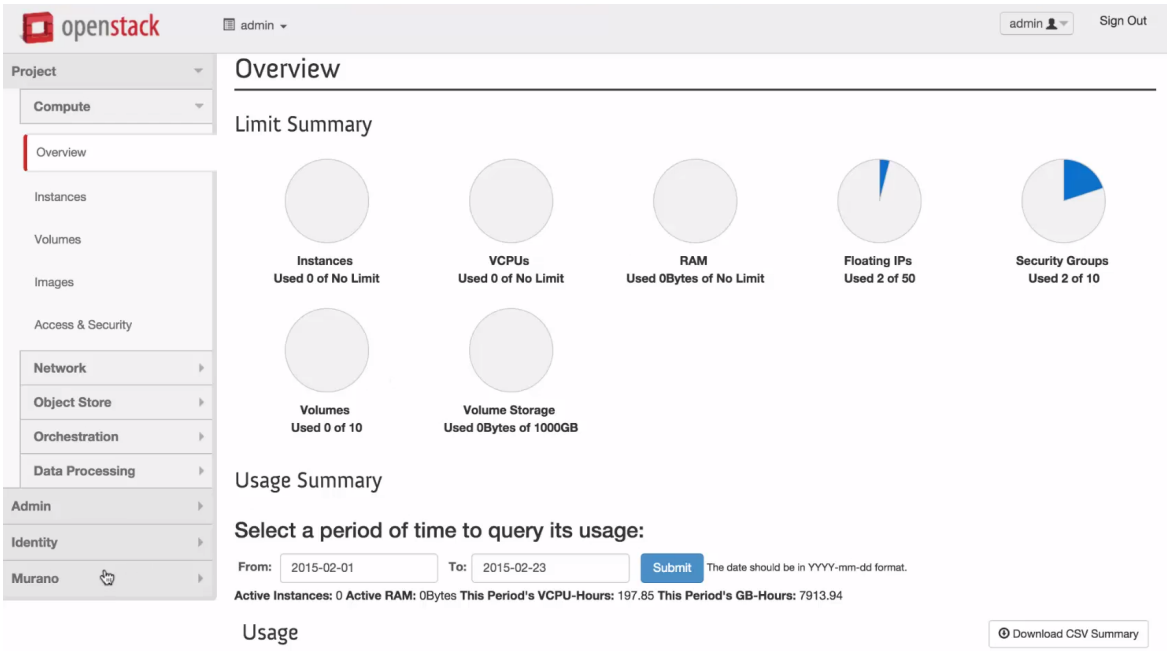


Figure 7.3: OpenStack Dashboard Example

our work as OpenStack extensions, so that any individual or any vendor who uses OpenStack for their cloud computing can easily use cloudlets. We refer to this Cloudlet-enabled OpenStack as *OpenStack++*. This project focuses on the design and implementation of cloudlet features of the OpenStack++ API. We will also provide a client library and web interface for the OpenStack users. It is worth noting that OpenStack-based approach is our initial effort toward the cloudlet deployment, and we ultimately want to have open APIs for cloudlet functionalities.

7.1 Design

OpenStack has a 6-month time-based release cycle, which is a fairly short period, and a new release usually introduces significant changes not only in external API but also in internal APIs. Therefore, in order to keep track of their release cycle with minimal effort on the cloudlet binding code, We are taking a modular approach for the OpenStack integration by extending the original code rather than modifying the code directly. In addition, we maintain a standalone cloudlet executable, which runs without an OpenStack cluster along with OpenStack++, as explained in Section 7.1.2. A standalone executable and an OpenStack++ cluster share a cloudlet library for the core functionality.

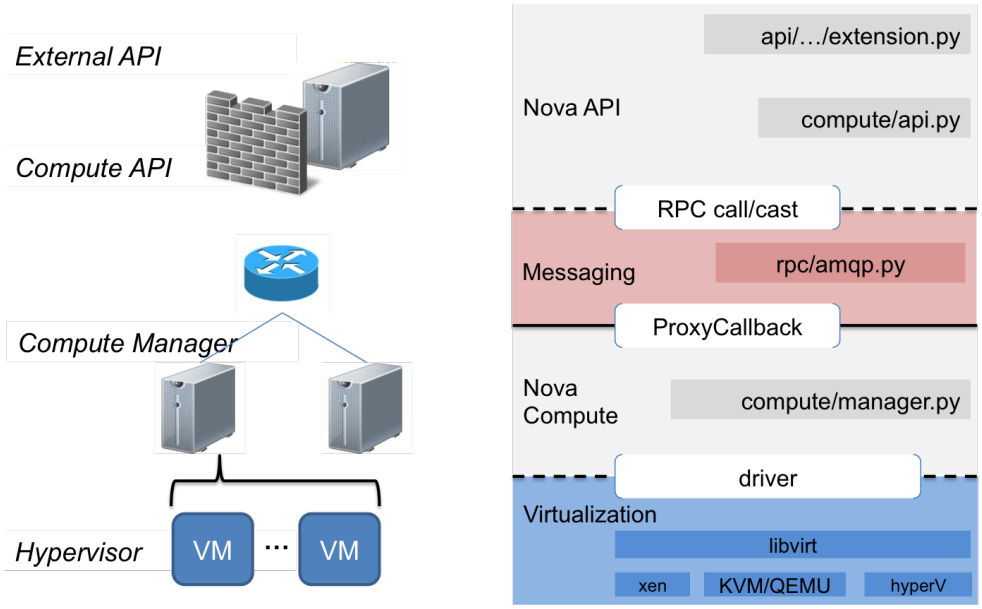


Figure 7.4: OpenStack API call hierarchy

7.1.1 Modular Approach using OpenStack Extension

OpenStack provides an extension mechanism to add new features to support innovative approaches. This allows developers to experiment and develop new features without worrying about the implications to the standard APIs. Since the extension is queryable, a user can first send a query to a particular OpenStack cluster to check the availability of the cloudlet features. Figure 6 shows the OpenStack API call hierarchy. APIs for extensions are provided to the users by implementing an Extension class. An API request from the user will arrive at the extension class and a set of internal APIs will be called to accomplish desired functionality. Some of the internal API calls will be passed to a corresponding compute node via the messaging layer if necessary. Then, the API manager at the compute node will receive the message and handle it by sending commands to the hypervisor via a driver. Finally, the driver class will return the result and pass it to the user following the reverse call sequence.

The cloudlet extensions follow the same call hierarchy. Once a user sends a request via a RESTful interface, the message will be propagated to the matching compute node. Then the hypervisor driver performs the given task. Here's an example command flow for creating a VM overlay. The command is applied to a running virtual machine and generates a VM overlay which extracts the difference between the running VM and the base VM. To define a new action for creating a VM overlay, a cloudlet extension class is declared following the OpenStack extension rule. The user-issued API request first arrives at the extension class, and then is passed to a corresponding compute node via API and message layer. At the compute node, the message

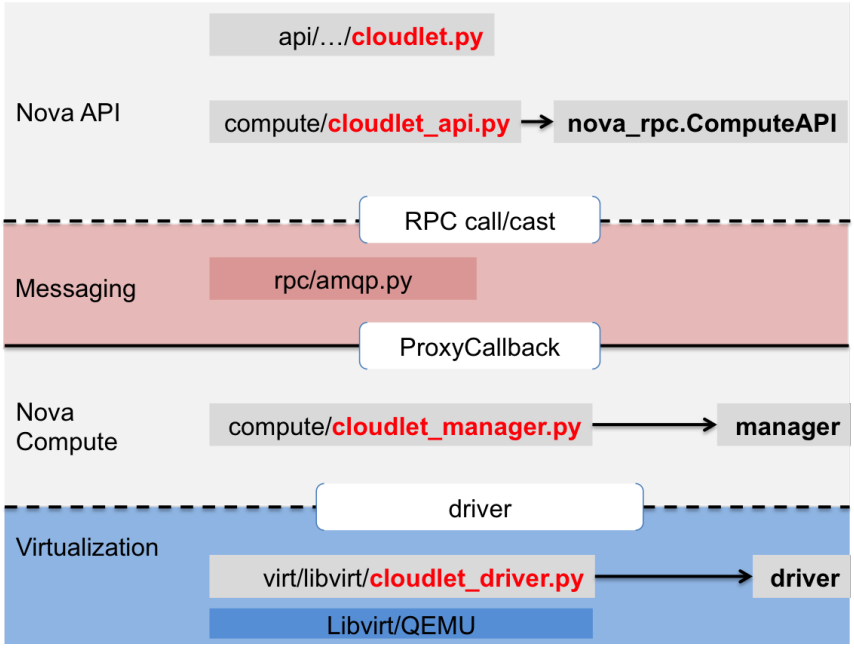


Figure 7.5: Classes/Files for Cloudlet API call hierarchy

is then handled by a cloudlet hypervisor driver, which interacts with a target virtual machine. Finally, the cloudlet hypervisor driver will create a VM overlay using the VM snapshot.

To support a specific API, the API manager (manager.py) and hypervisor driver (driver.py) should be able to handle the message in addition to the cloudlet extension. This requires modifications to the original files/classes of OpenStack (Figure 7.5). Modifying original OpenStack code, however, will cause significant maintenance overhead, especially because OpenStack is frequently updated. Instead, we create a new class for both the API manager and hypervisor driver inheriting a matching class in OpenStack and save each of them as a new file. Fortunately, OpenStack provides a way to use a custom class for the API manager and hypervisor driver via a configuration file. As shown in Figure 7.5, cloudlet specific code is placed in separate files such as cloudlet_api.py, cloudlet_manager.py, and cloudlet_driver.py. This makes management overhead much lower because the cloudlet feature can be added by simply placing those files into OpenStack directory and changing a configuration file.

7.1.2 Support for both OpenStack and a Standalone Executable

It is important to support a standalone version of cloudlet execution in addition to the OpenStack extended version. There are two rationales for maintaining a standalone executable along with OpenStack++. First, OpenStack is designed for providing end-to-end cloud computing services, so it is not trivial to install and maintain OpenStack itself. For those users who want to use

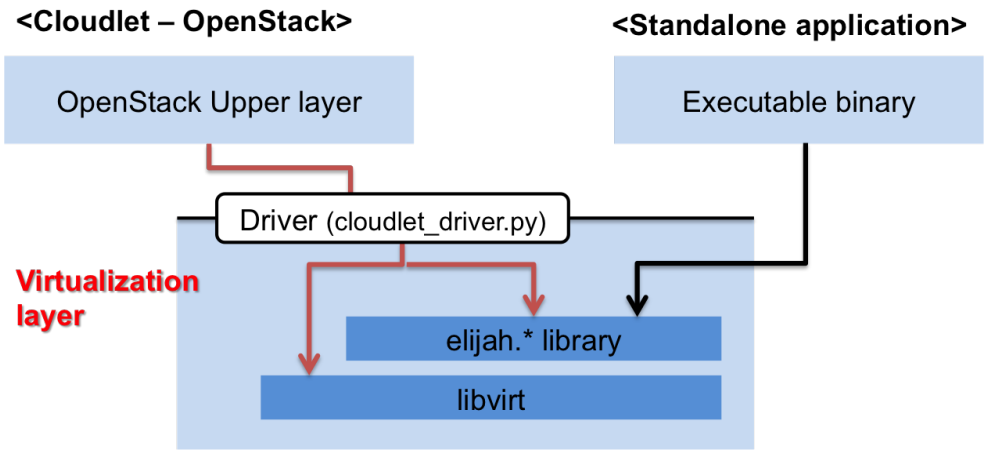


Figure 7.6: Supporting both OpenStack and Standalone

only cloudlet features in a simple way, a standalone executable will be a clean solution. Second, a standalone executable is much easier to debug and simple to assess the performance. Since OpenStack is a complex system, the standalone version provides a straightforward way to debug the system. To support both approaches effectively, we created a cloudlet library that both OpenStack and standalone code can use. This library is packaged as a python library because OpenStack uses python. Figure 7.6 shows a high-level diagram of how the cloudlet library is used.

7.2 Implementation

OpenStack++ implements the following features.

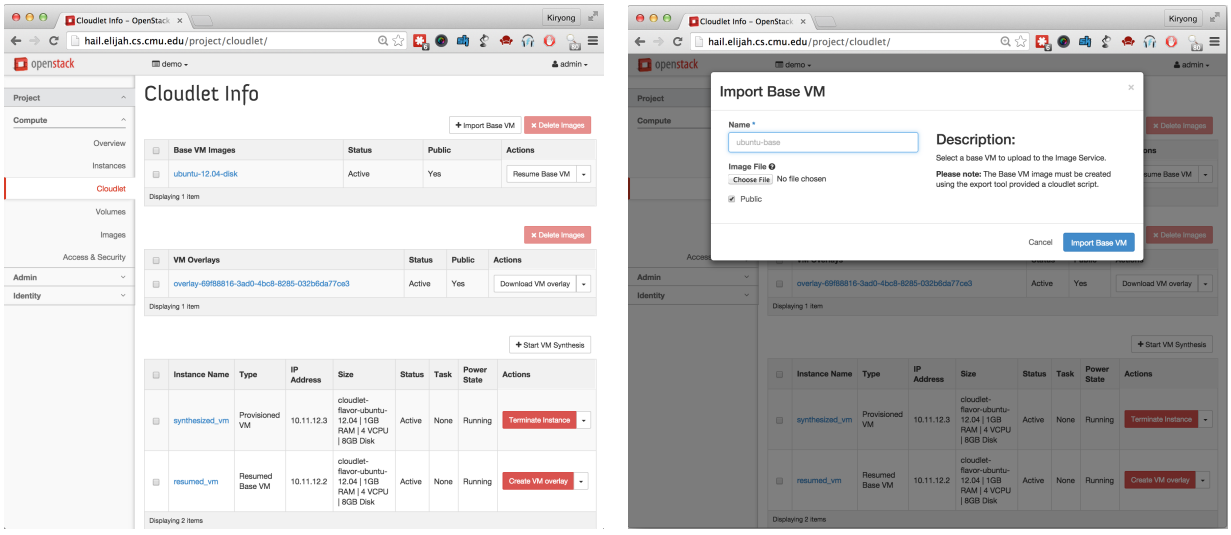
1. **Import Base VM**: Import a Base VM from a file to the Glance image storage. we assumed that each Cloudlet has a set of prepopulated Base VMs, and this is a function for importing a Base VM.
2. **Resume Base VM**: Resume one of the base VMs to make a customized VM and to create a new VM overlay for the customized VM.
3. **Create VM overlay**: Create a VM overlay from a running VM instance.
4. **VM synthesis**: Provisioning a VM instance at a OpenStack++ cluster using a VM overlay.
5. **VM handoff**: Migrating a VM instance to a different OpenStack++ cluster.

Among those features, *Resuming Base VM* and *Creating VM overlay* are off-line operations that the developers use to create a VM overlay of the back-end server. That means those two operations will not be used by mobile users, but used by the application developers. *Importing Base*

Cloudlet features	OpenStack interpretation	Output in OpenStack
Import Base VM	Save VM snapshot to the Glance image storage	New VM image
Resume Base VM	Resume a VM from memory and disk snapshot	New VM instance
Create a VM overlay	Incremental snapshot of the VM	1) Disk incremental snapshot 2) Memory incremental snapshot
VM synthesis	Recover the VM from the incremental snapshot (a.k.a VM overlay)	New VM instance
VM handoff	Migrate a running VM instance from one OpenStack cluster to another	1) Terminate VM at source OpenStack 2) create new VM instance at destination OpenStack

Figure 7.7: Cloudlet features as an analogy for OpenStack

VM is for pre-provisioning the base VM and it is also an off-line operation which the administrator of the OpenStack++ cluster uses after the installation of a new OpenStack. The remaining two features, *VM synthesis* and *VM handoff*, are the run-time operations that are executed during the mobile application's offloading. Figure 7.7 shows how these operations are interpreted from the perspective of OpenStack. For example, resuming a base VM and performing VM synthesis can be considered a process of instantiating a new virtual machine on OpenStack.



(a) Dashboard (b) Import Base VM

Figure 7.8: Screenshot of Cloudlet Dashboard and Importing Base VM

```
krha@hail:~$ nova image-show ubuntu-12.04-diskhash
+---------------------------+--------------------------------------------------------------------------+
| Property                  | Value                                                                    |
+---------------------------+--------------------------------------------------------------------------+
| OS-EXT-IMG-SIZE:size      | 118018164                                                                |
| created                   | 2015-06-30T21:24:10Z                                                     |
| id                        | 666f1d7e-d546-4bf7-a22a-cfa9ee253e75                                     |
| metadata base_sha256_uuid | abda52a61692094b3b7d45c9647d022f5e297d1b788679eb93735374007576b8         |
| metadata cloudlet_type    | cloudlet_base_disk_hash                                                  |
| metadata image_location   | snapshot                                                                 |
| metadata image_type       | snapshot                                                                 |
| metadata is_cloudlet      | True                                                                     |
| minDisk                   | 8                                                                        |
| minRam                    | 1024                                                                     |
| name                      | ubuntu-12.04-diskhash                                                    |
| progress                  | 100                                                                      |
| status                    | ACTIVE                                                                   |
| updated                   | 2015-06-30T21:24:26Z                                                     |
+---------------------------+--------------------------------------------------------------------------+
krha@hail:~$
```

(a) Metadata for a Base VM's memory snapshot file

Custom Properties

Image Type	snapshot
base_resource_xml_str	<domain type='kvm'> <name>cloudlet-d71b7be54b104dee82c9038fddbbf39c</name> <uuid>d71b7be5-4b10-4dee-82c9-038fddbbf39c</uuid> <memory unit='KiB'>1048576</memory> <currentMemory unit='KiB'>1048576</currentMemory> <vcpu placement='static'>4</vcpu> <os> <type arch='x86_64' machine='pc-1.0'>hvm</type> <boot dev='hd'/> </os> <features> <acpi/> </features> <cpu mode='custom' match='exact'> <model fallback='forbid'>core2duo</model> <topology sockets='1' cores='4' threads='1'/> </cpu> <clock offset='utc'/> <on_poweroff>destroy</on_poweroff> <on_reboot>restart</on_reboot> <on_crash>destroy</on_crash> <devices> <emulator>/usr/local/bin/cloudlet_qemu-system-x86_64</emulator> <disk type='file' device='disk'> <driver name='qemu' type='raw'/> <source file='/home/cloudlet/cloudlet/image/pci_hotplug/precise.raw'/> <target dev='hda' bus='ide'/> <address type='drive' controller='0' bus='0' target='0' unit='0'/> </disk> <disk type='file' device='cdrom'> <driver name='qemu' type='raw'/> <source file='/var/lib/cloudlet/conf/ovftransport.iso'/> <target dev='hdc' bus='ide'/> <readonly/> <address type='drive' controller='0' bus='1' target='0' unit='0'/> </disk> <controller type='ide' index='0'> <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/> </controller> <interface type='user'> <mac address='52:54:00:9f:a8:da'/> <model type='virtio'/> <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/> </interface> <input type='mouse' bus='ps2'/> <graphics type='vnc' port='-1' autoport='yes' listen='127.0.0.1' keymap='en-us'> <listen type='address' address='127.0.0.1'/> </graphics> <video> <model type='cirrus' vram='9216' heads='1'/> <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/> </video> <memballoon model='virtio'> <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/> </memballoon> </devices> <seclabel type='none'/></domain>
base_sha256_uuid	abda52a61692094b3b7d45c9647d022f5e297d1b788679eb93735374007576b8
cloudlet_base_disk_hash	666f1d7e-d546-4bf7-a22a-cfa9ee253e75
cloudlet_base_memory	8f7bcdd1-72b8-448e-b704-0054ee2d133c
cloudlet_base_memory...	8c49ceca-5e02-4415-8e3a-1a7f919892ad
cloudlet_type	cloudlet_base_disk
image_location	snapshot
is_cloudlet	True

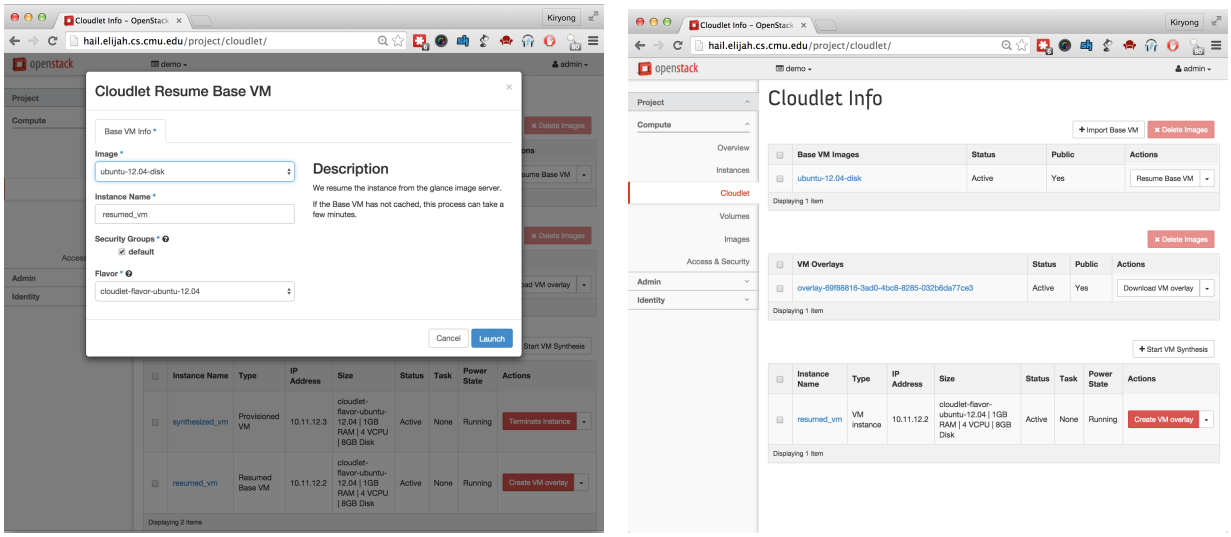
(b) Metadata for a Base VM's disk image file

Figure 7.9: Glance metadata of Base VM

7.2.1 Import Base VM

We presume that every cloudlet caches a set of base VMs. The OpenStack++ administrator imports base VMs after installing OpenStack++ at a cloudlet by using the *import Base VM* operation. Figure 7.8-(a) shows a screenshot of the Cloudlet panel in the OpenStack dashboard. The first table shows a list of base VMs on this OpenStack cluster. The second table displays a list of VM overlays saved in the Glance storage, where OpenStack saves virtual machine images. The last table presents running VMs, each of which is either a resumed base VM or a synthesized VM. The *Import Base VM* button is at the right corner of the first table, which an administrator can use to import a new base VM. Figure 7.8-(b) shows the UI for import base VM. Input file path for a base VM and base VM's name are required.

From OpenStack's viewpoint, importing a base VM is equivalent to saving new blobs at a Glance storage. Hence, instead of creating a new API, we reuse existing Glance APIs to



(a) UI for resuming a Base VM

(b) Finishing Base VM Resume

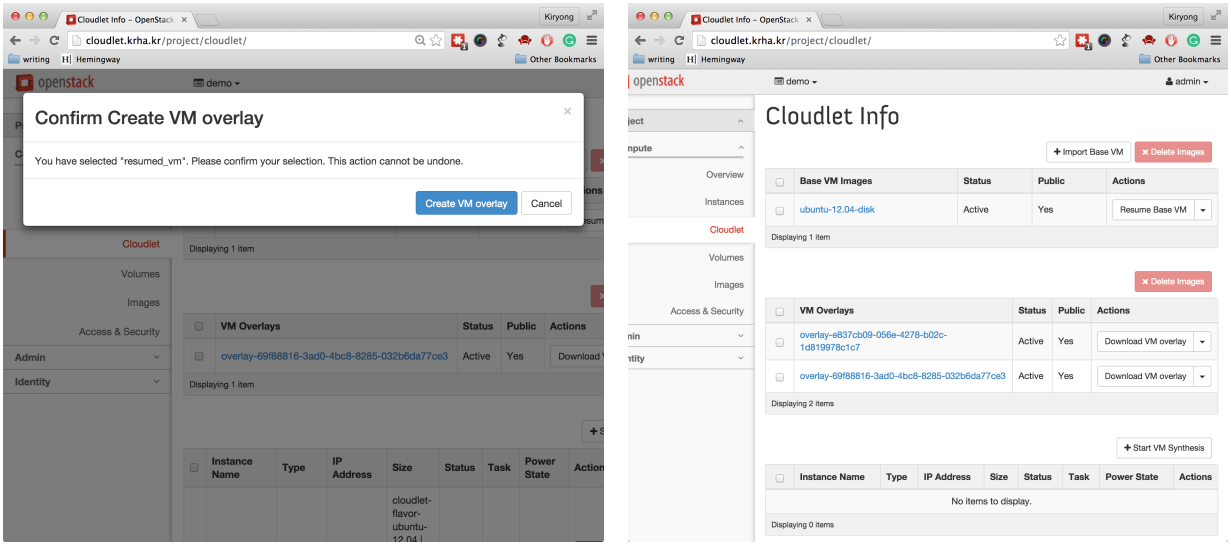
Figure 7.10: Screenshot of 'Resume Base VM'

accomplish this operation. The only difference is that multiple files should be saved to import a base VM because a single base VM is a zipped file composed of four files internally; a base disk image, a base memory snapshot, a hash value list for the disk image, a hash value list for the memory snapshot. A received base VM file is first decompressed into four files, and then saved to the Glance storage one by one. To indicate that these are all related to the Cloudlet's base VM, we marked each file with a metadata using file type keyword as shown in Figure 7.9-(a). Further, since OpenStack treats each glance image as an independent entity, the disk image saves the UUIDs of all other associated files to connect all the participating files. In addition, we save VM's libvirt configuration as a part of the metadata of the Base VM's disk image for the later use. Figure 7.9-(b) shows an example of metadata of a disk image.

7.2.2 Resume Base VM

For a front-end mobile application, a developer prepares a back-end server that will run at a cloudlet. The installation process of the back-end server typically includes preparing dependent libraries, downloading/setting executable binaries, and changing OS/system configurations. A developer can perform these operations using a resumed base VM. Figure 7.10 shows a screenshot for resuming a base VM. At the first table of base VM list, *Resume Base VM* button will resume the selected base VM. The resumed instance will be displayed in the third table.

For OpenStack, resuming a base VM is analogous to instantiating a new VM instance using a VM snapshot. Therefore, instead of devising a new API, I modified the original API for



(a) Start VM overlay creation

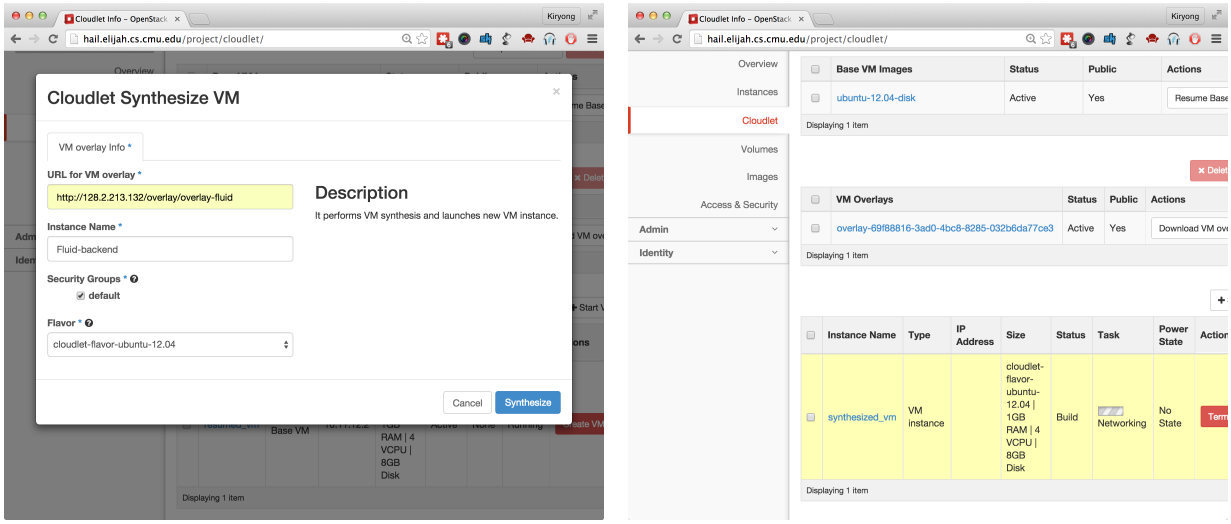
(b) Finish VM overlay creation

Figure 7.11: Screenshots of 'Creating VM overlay'

launching a new VM instance. The original API will handle all error checking conditions such as permission, quota, and resource availability. After passing all condition checking, the message will finally arrive at the compute node to launch a new VM. At the code level, this message will arrive at the hypervisor driver class and be passed to the underlying virtualization. To handle resuming a Base VM task at the hypervisor driver, we built a cloudlet hypervisor driver class, *CloudletDriver*, that inherits the original *LibvirtDriver*. Upon a new message, CloudletDriver examines the metadata of the associated virtual disk image. If the virtual disk image has a cloudlet flag, then CloudletDriver resumes the selected base VM instead of starting a new VM instance from boot. Resuming a Base VM is different from the existing VM resume mechanism of OpenStack in that a user can resume multiple VM instances of the base VM simultaneously and the resumed VM is considered as a new VM instance.

7.2.3 Create VM overlay

After resuming the selected base VM, a developer can install a desired back-end server on it. A developer is supposed to start creating a VM overlay after finishing all the installation and after launching the back-end server process. Overlay creation will start by simply clicking a *Create VM overlay* button next to the VM instance row as shown in Figure 7.11-(a). This operation will apply optimizations to generate a minimal VM overlay and finally save the VM overlay in Glance storage as listed in the second table in Figure 7.11-(b). One can download the VM overlay using *Download* button.



(a) UI for VM provisioning

(b) VM provisioning

Figure 7.12: Screenshot of 'VM provisioning'

For *creating VM overlay* operation, we introduce a new API. Creating VM overlay can be classified as the same category of API call as reboot VM and resize VM, which applies a specific action to a running VM instance. Therefore, we used the existing Action URL but declared a new action type using the OpenStack Extensions mechanism. The extension defines a new Action for creating a VM overlay and passes this command to the virtualization driver (a.k.a. Cloudlet-Driver) via the internal API class. For the internal API, a cloudlet API class, *CloudletAPI*, that inherits *nova_rpc.ComputeAPI* was added. we avoid modifying the original OpenStack code or files, by class inheritance and by using a new file.

At this point, OpenStack++ cluster is ready to serve mobile applications. OpenStack++ is installed and Base VMs are imported by the service provider (Administrator) using *Import Base VM*. Developers prepare a VM overlay that contains the back-end server of the mobile application. This VM overlay file or an URL of the VM overlay will be distributed to the mobile users. Then, the mobile applications can dynamically provision the back-end server at an OpenStack++ cluster by either directly transmitting a VM overlay file saved at the mobile device or by passing an URL of the VM overlay to the OpenStack++ cluster.

7.2.4 VM Provisioning

VM provisioning is a run-time operation for a rapid provisioning of an application's back-end server to a nearby Cloudlet. Using the VM provisioning, a mobile user can launch a back-end server at an arbitrary cloudlet using a VM overlay. Figure 7.12-(a) shows the UI for the VM provisioning. A mobile user is asked to input a URL for the VM overlay. Instance flavor

Figure 7.13: Example of VM provisioning request message

will be automatically selected by reading the metadata of the associated base VM. Once VM provisioning is finished, a new VM will launch and the relevant information is displayed in the third table of Figure 7.12-(b).

Similar to the resuming base VM in Section 7.2.2, VM provisioning launches a new VM instance at the OpenStack cluster. Therefore, I leverage OpenStack's original VM creation API. In OpenStack, to create a new VM instance, an OpenStack user sends a HTTP POST message to a specific URL, *https://openstack-addr/v2.1/servers* typically using a client program. The detailed configurations for the new VM such as name, disk image, and flavor are described in JSON payload of a HTTP message. To differentiate a VM synthesis request from a regular VM creation request, we add a special keyword, *'overlay_url'*, to the metadata of the message. Figure 7.13 shows an example of a VM provisioning message. To specify a location of VM overlay, an *'overlay_url'* is appended in the metadata.

This message is finally handled at cloudlet hypervisor driver, *CloudletDriver*, like the resuming base VM operation. At the code level, CloudletDriver inherits the Libvirt hypervisor driver, *LibvirtDriver*, overriding the VM spawning method. At the VM spawning method, it checks the metadata to find a keyword *'overlay_url'*. If the request has the overlay_url metadata, it will perform VM synthesis using the given URL of VM overlay.

7.2.5 VM Handoff

VM handoff will migrate a running VM instance from one OpenStack cluster to another. Since it involves two independent OpenStack clusters, the operation starts from the assumption that a user has a permission to access both clusters. In other words, the source OpenStack cluster needs permission to call the API of the destination OpenStack cluster. To get the permission of the destination OpenStack cluster, the handoff UI at the source OpenStack cluster asks for credential information of the destination as shown in Figure 7.14. Although the Web UI does not save any credential information, one can use a client program that accepts an auth-token instead of

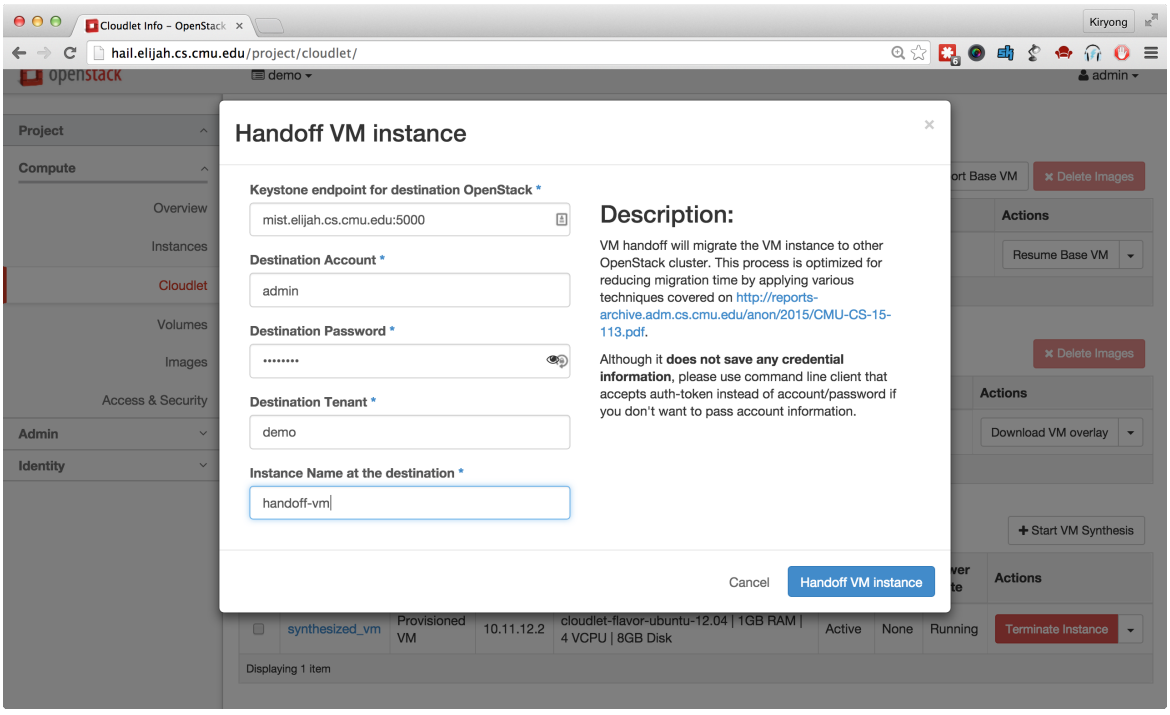


Figure 7.14: Screenshots of 'VM Handoff' configuration

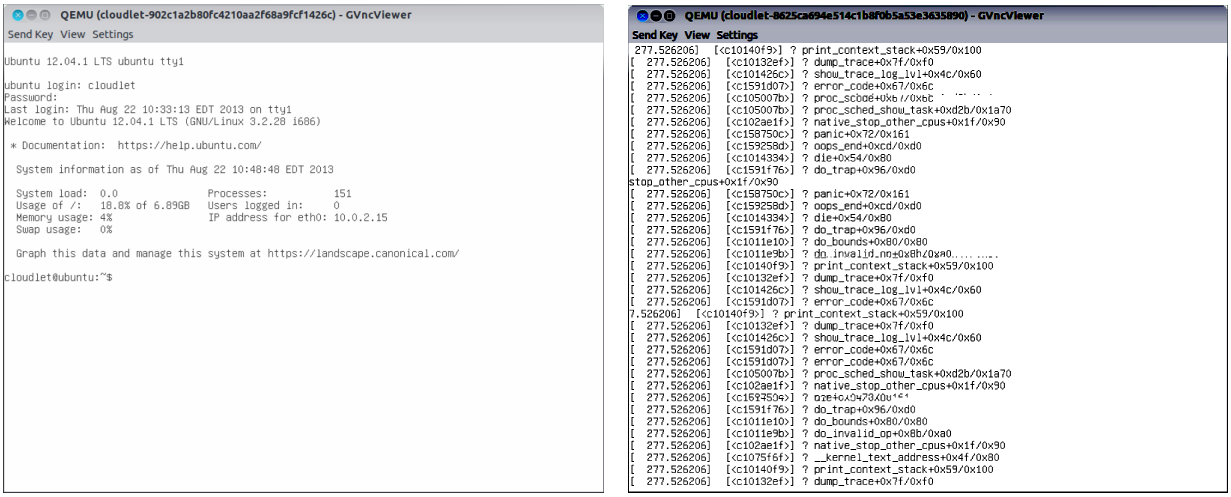
```
{
    "cloudlet-handoff": {
        "handoff_url": "network://dest-openstack.com/"
        "dest_token": "akwqub1As8a61jsakaAj1Saa11"
    }
}
```

Authentication-token for the destination

Figure 7.15: Example of VM handoff request message

account/password. In cases where a mobile user uses an OpenStack++ cluster to run a back-end server program, a client program will replace the role of UI to programmatically trigger VM handoff.

Similar to creating the VM overlay in Section 7.2.3, VM handoff applies an action on a running VM instance. Accordingly, we create a new API extending the Action URL using OpenStack Extension. In the JSON payload of the HTTP POST message, *https://openstack_addr/v2.1/servers/server_id* the handoff command and detail descriptions are added as shown in Figure 7.15.



(a) Success to Resume

(b) Fail to Resume (Kernel Panic)

Figure 7.16: Challenges on CPU Flag Compatibility

7.3 Challenges

While porting the cloudlet open source code to OpenStack, there have been many design and implementation challenges. Some are related to complying with OpenStack practices and some are about implementation issues such as library compatibility. Though not every challenge is tightly coupled to the research, it is worth to report some of those challenges because they are relevant to cloudlet deployment.

7.3.1 Portability of the VM

The first challenge is about resuming a suspended virtual machine. Technically, VM provisioning resumes a VM instance at the target OpenStack++ cluster, which is suspended at a different site. This causes several portability issues in the VM. Since a VM has a relatively narrow interface to run on a hypervisor compared to the high-level approach such as a process, it is known to be relatively easy to migrate from one place to another. That is why VM migration is used and is stable in today's data center. However, in our case, different from the data center VM migration, host machines can be highly heterogeneous and the source and destination machine can have a different networking environment. Those changes will introduce subtle issues in the VM portability. There are two major portability problems in the OpenStack++ porting; 1) CPU compatibility and 2) Stale networking state.

CPU compatibility: To get full performance from a host machine, a virtual machine usually

inherits CPU flags from the host machine. That means if a source host machine, where VM is suspended, supports a wider range of CPU features than a destination host machine, where VM is resumed, then a resumed OS will crash when it tries to use a CPU feature that is not available at the destination. This happens because 1) suspend/resume is agnostic to the guest OS and 2) the guest OS checks CPU flags only once at the boot-time. Unfortunately, QEMU/KVM does not strictly check CPU flags when resuming a VM, either. Figure 7.16 shows an example of guest OS failure (kernel panic) when a VM is resumed at a host machine with insufficient CPU features.

To handle this CPU incompatibility issue, we use a pre-defined CPU model for the base VM. *Libvirt* library defines a set of CPU models for the virtual machine [2]. Among the wide range of CPU models including *pentiumpro, coreduo, n270, core2duo, qemu64, Conroe, Penryn, Nehalem, Westmere, SandyBridge*, and *Haswell*, we choose *Core2duo* because it covers a reasonable range of CPU flags (even more than *qemu64*), but is common enough to support old machines. If one has a more managed environment where minimal CPU features can be forced on all cloudlet machines, then a more decent CPU model like *SandyBridge* will be helpful to maximize performance. In my implementation, we enforce the *Core2duo* CPU model and VM provisioning and VM handoff will not start if the host machine does not support it rather than fail in run-time.

Staleness in networking configuration: A virtualized network interface card (NIC) is attached to the virtual machine using emulated hardware interfaces like PCI. This virtual NIC has unique hardware configurations such as the MAC address, and a guest OS loads those configurations at boot-time assuming that it won't be changed until the next booting. Then the guest OS will setup networking via this NIC. For example, the guest OS can have a private IP address with NAT or it can access the Internet directly using a public IP address. Various network configurations are possible in OpenStack using *Neutron* [3]. However, the NIC and networking information configured at a source OpenStack cluster are not valid at a destination OpenStack cluster where the VM is resumed. This is applied to 1) resuming a base VM, 2) VM provisioning, and 3) VM handoff, because they are all technically resuming a memory state of the VM. Figure 7.17 shows a diagram of broken networking when a VM is provisioned with the memory state. The resumed VM originally has a NIC with MAC address 11-22-33-44-55, but OpenStack networking assigned a new NIC card for this newly instantiated VM. This new NIC card has MAC address, aa-bb-cc-dd-ee, and the underlying OpenStack network module uses this MAC address to configure networking. For example, a router will forward a network packet designated to the VM using MAC aa-bb-cc-dd-ee, but it won't be delivered to the application because the guest OS thinks that its MAC is 11-22-33-44-55.

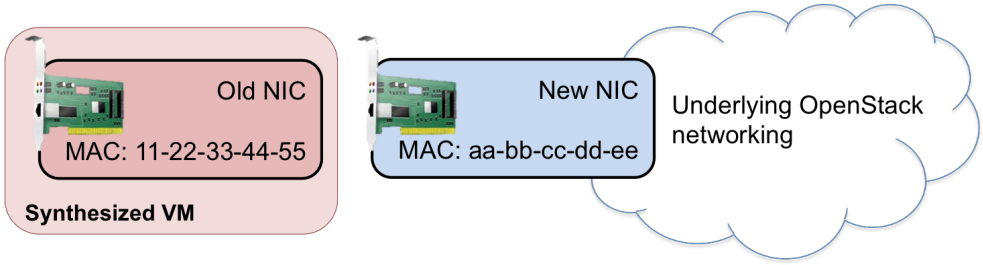


Figure 7.17: Example of networking staleness in synthesized VM

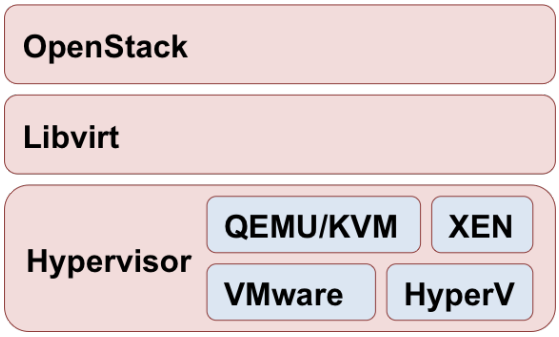


Figure 7.18: Layers in Cloud Computing software

To overcome this inconsistency and to enable networking for the VM, we detach the old virtual NIC and attach a new virtual NIC given by the new OpenStack via Hot PCI Plugin [101]. Since it is an industry standard, most modern OSes support it. It is also supported by the KVM/QEMU hypervisor using VT-d techniques [1]. After resuming the VM successfully (either resumed from VM provisioning or VM handoff mechanism), I detach the old virtual NIC of the VM and attach a new NIC configured by the OpenStack networking. This way, the operating system understands the reattachment of the PCI device, and accordingly updates stale networking configurations. In theory, this won't affect the applications' networking because applications are running on the TCP/IP layer and are agnostic to the underlying network level.

7.3.2 Modification on hypervisor (QEMU/KVM)

Figure 7.18 shows the layers of cloud computing software. At the bottom, a hypervisor is responsible for creating and running virtual machines. Commercial products and open source projects including VMWare ESX, Microsoft Hyper-V, and QEMU/KVM are the examples at this layer. One level above, *Libvirt* is an open source management tool for managing virtual machines. It supports KVM, Xen, VMware ESX and other virtualization hypervisors. Libvirt provides a set of APIs for the orchestration of hypervisors. This is useful because each hypervisor has slightly different syntax for a similar function. Libvirt provides a high-level abstraction hiding complex-

ity and diversity of various hypervisors. At the top layer, OpenStack tries to provide a complete end-to-end software system providing 1) resource management for computing (e.g. virtual machines), networking, storage, 2) authentication and permission, and 3) high-level API for easy use and so on.

In this hierarchy of cloud computing software, OpenStack++ patches the original OpenStack to enable cloudlet features. However, to get the best performance for both provisioning and handoff, the cloudlet code modifies the QEMU/KVM hypervisor, which can cause a problem for merging OpenStack upstream. This is because OpenStack and QEMU/KVM are maintained by independent organizations. Therefore, the modifications to the QEMU/KVM hypervisor can be an obstacle for OpenStack upstream merging because this modification is not an official release from the QEMU/KVM community. The right approach is 1) first merge cloudlet's modified QEMU/KVM to QEMU/KVM upstream, 2) wait for the new release of cloudlet-enabled QEMU/KVM, 3) merge OpenStack++ to OpenStack upstream with the requirement of cloudlet-enabled QEMU/KVM. This is not a research challenge or an implementation issue but matters in practice when pursuing OpenStack upstream merging.

In order to have a better understanding, we list the necessity of modified QEMU/KVM in the cloudlet implementation. For VM provisioning relevant issues, the QEMU/KVM modification helps by improving the memory snapshot format. Memory snapshot in QEMU/KVM is an internal data structure and its format is poorly maintained in terms of compatibility. As a result, a QEMU memory snapshot saved using a certain version of QEMU/KVM hypervisor might not be resumable at a different version of QEMU/KVM hypervisor. In addition, the original QEMU/KVM compresses each memory page if possible, and that prevents the cloudlet code from performing deduplication and randomly accessing a specific memory page. Further, cloudlets modify QEMU/KVM to support *early start* optimization which allows a VM instance to start without a full memory snapshot. Early start optimization uses on-demand fetching of the memory snapshot to speed up VM provisioning.

For VM handoff, a behavior of the original VM live migration has been changed. While the VM handoff is proceeding, new dirty memory pages of the running VM are not sent to the network immediately but accumulated under the control of VM handoff code. This is different from the original behavior of live migration where dirty memory pages are immediately sent. This change is designed to save network bandwidth by avoiding transmission of frequently modified memory pages (hot region) repeatedly. Also, the VM handoff module will achieve adaptive VM handoff that balances computation speed and network transmission speed.

Modifications to QEMU/KVM are inevitable to speed up provisioning and handoff, but it can be an obstacle for OpenStack upstream merging from a practical standpoint. To minimize the influence of modified QEMU/KVM at OpenStack, we used it only for the cloudlet related

tasks. That is, modified and unmodified QEMU/KVM coexist at OpenStack++ and OpenStack's original tasks use unmodified QEMU/KVM as before. Details of optimizations for provisioning and handoff are described at [49] and [50], respectively.

7.4 Chapter summary

Cloudlets are becoming widely accepted from academia and industry. However, the development of cloudlets faces a classic bootstrapping problem. It needs practical applications to incentivize cloudlet deployment while developers cannot heavily rely on cloudlet infrastructure until it is widely deployed. To provide a systematic way to incentivise cloudlet deployment, we implemented OpenStack++ that extends OpenStack, an open source ecosystem for cloud computing. With OpenStack++, any individual or any vendor who uses OpenStack for their cloud computing can easily use the cloudlet structures. For this work, we designed and implemented OpenStack++ APIs and ported the cloudlet open source project to OpenStack. In addition, we provided a web interface and a client program for OpenStack users.

Chapter 8

Conclusion and Future Work

This dissertation proposes a new architecture for cloud-mobile convergence. It proposes, and shows the feasibility of, cloudlets; mobility-enhanced small-scale cloud datacenters that are located at the edge of the Internet. We show that cloudlets can enable resource-intensive and interactive mobile applications by greatly improving end-to-end network bandwidth and latency without sacrificing the benefits of cloud computing. In this chapter, we conclude the dissertation with a summary and contribution, and discuss research directions and challenges opened in this area.

8.1 Contributions

This dissertation presents the following thesis.

Emerging mobile applications that are simultaneously interactive and resource-intensive can be effectively supported by mobility-enhanced small-scale cloud data centers called cloudlets that are located at the edge of the Internet.

To demonstrate the thesis, we first provide a measurement-driven quantitative analysis of emerging mobile applications, and show how cloudlets can help them. Then we propose a cloudlet-based two-level computing architecture that seamlessly extends today's cloud infrastructure. We identify three functionalities that cloudlets must offer above/beyond standard cloud computing; cloudlet discovery, rapid just-in-time provisioning, and VM handoff across cloudlets. Finally, we present a systematic way to accelerate cloudlet deployment using OpenStack. We summarize our contributions in the following sections.

8.1.1 Quantitative Analysis of Emerging Mobile Applications

It is expected that network latency between a mobile device and an associated data center can affect the performance of cloud-backed mobile applications. However, the magnitude of this impact is not clear. End-to-end response time is influenced by multiple factors contributing to the application's performance; not only network conditions, but also other factors such as relative computing power between a mobile device and a cloud, and application characteristics. Further, it is fair to ask whether offloading is necessary at all given that mobile hardware is rapidly improving. Here we provide measurement-driven quantitative analysis of the benefit of the cloudlet using five representative mobile applications and confirm the necessity of cloudlets.

From the experiments in Section 2.3.2 and 2.3.3, we confirm that network proximity resulting in high bandwidth, low latency, and low jitter to the associated data center is essential for the target mobile applications. A cloudlet represents the best attainable network proximity by its location at the edge of the Internet. Our results show that a cloudlet is indeed valuable for many of the studied applications, in terms of both response time and energy usage.

8.1.2 Cloudlet Discovery

Since cloudlets are small data centers dispersed geographically, a mobile device first has to discover, select and associate with the appropriate cloudlet among many choices. These steps are unnecessary with a cloud because it is centralized. But in cloudlets, discovery and selection have to be carefully managed because the choice of a cloudlet can directly affect the provisioning time as well as the performance of the offloaded mobile applications.

We have designed and implemented a cloudlet discovery system that supports application-specific cloudlet selection and disconnected operation using local/global search and two-level search (Section 6.2.1 and 6.2.2). In addition, we expand our system to support close-sourced mobile applications by using DNS and HTTP redirection. We have focused on achieving flexibility and extensibility of the system to cover wide range of cloudlet deployment scenarios.

8.1.3 Rapid Just-in-Time Provisioning

A cloudlet needs rapid provisioning because its association with mobile devices is highly dynamic, with considerable churn due to user mobility. A user may unexpectedly show up at a cloudlet (e.g., if he just got off an international flight) and try to use it for an application such as a personalized language translator. For that user, the provisioning delay before he is able to use the application impacts usability.

Since cloud offload relies on precisely-configured back-end software, it is difficult to support at global scale across cloudlets in multiple domains. To address this problem, we describe just-

in-time (JIT) provisioning of cloudlets. Using a suite of five representative mobile applications, we demonstrate a prototype system that is capable of provisioning a cloudlet with a non-trivial VM image in approximately 10 seconds. This speed is achieved through dynamic VM synthesis and a series of optimizations to aggressively reduce transfer costs and startup latency.

8.1.4 VM handoff across Cloudlets

Once a user successfully uses a provisioned cloudlet, the next question is "What happens if a mobile user moves away from the cloudlet he is currently using?" As long as network connectivity is maintained, the applications should continue to work transparently. However, interactive response will degrade as the logical network distance increases. In practice, this degradation can be far worse than physical distance may suggest. For example, when moving from a home Wi-Fi network to that of a neighbor down the street, communication to the first home's cloudlet will require two traversals of "last-mile" links connecting the homes to their ISPs.

We have proposed *VM handoff* as a technique for seamlessly transferring VM-encapsulated execution to a more optimal cloudlet. To minimize VM handoff time, various compression techniques are pipelined into a parallel processing. We present the first fine-grain adaptive mechanism for migrating VMs over WAN and show how dynamic adaptation can play an important role in order to cope with varying WAN bandwidth and cloudlet load. We also show this mechanism can be implemented in today's Internet speed improving VM handoff time at least one order of magnitude compared to the conventional live migration. Our experimental deployment using LTE-cloudlet confirms that VM handoff is applicable and practical on both LTE and WiFi network.

8.1.5 Cloudlet Deployment

Finally, we present a systematic way to accelerate deployment of cloudlet infrastructure. Since the cloudlet model requires reconfiguration or additional deployment of hardware/software, it is important to provide a systematic way to incentivise the deployment. However, cloudlet development faces a classic bootstrapping problem. It needs practical applications to incentivize the deployment, but application developers cannot heavily rely on cloudlet infrastructure until it is widely deployed.

To break this deadlock and bootstrap the cloudlet deployment, we leverage *OpenStack*, which is an open eco-system for cloud computing. We design and implement all cloudlet features to work as OpenStack extensions, so that any individual or any vendor who uses OpenStack for their cloud computing can easily adopt cloudlets. We refer to this Cloudlet-enabled OpenStack as *OpenStack++*. We also provide a client library and web interface for OpenStack++.

8.2 Future Work

The cloudlet is a new architectural element that opens many new research topics across multiple layers of the system. In this dissertation, we design and implement the system functionalities of the cloudlet in order to achieve two-level architecture for cloud computing. Based on this low-level systems work, we would like to introduce two distinctive research directions: 1) advance system support for cloudlet infrastructure and 2) edge computing.

8.2.1 Advanced System Support for Cloudlets

Multi-Tenant Support

Cloudlets are designed to be served as a shared infrastructure like cloud data centers. Therefore, it is presumed that they will provide safety and strong isolation between untrusted computations from different users. We use virtual machine abstraction to achieve this goal for the same reason as it is used in public clouds like Amazon AWS. However, even with VM encapsulation, we cannot prevent performance interference between VMs running on the same physical machine. This is because the VMs can suffer from resource contention at various places such as the last level cache, memory access, and I/O devices. This is well known problem in cloud data centers and exacerbated in the cloudlets for two reasons. First, a cloudlet is designed to support interactive mobile applications and these applications are sensitive to the delay and jitters in response time. Unlike large-scale web services that are hosted by a cloud data center, performance degradation or jitter caused by the resource contention can be critical in the cloudlet context. Second, a cloudlet by definition is a small-scale data center that typically has many fewer reserved resources than a cloud data center. Accordingly, there is more likely to be resource interference. Also, because of the cloudlet's limited computing resources, it is important to maximize resource utilization, which might make the system prone to resource contention.

Efficient multi-tenant support for a cloudlet is not only related to the optimization of VM placements but is also associated with other cloudlet functionalities such as cloudlet discovery and VM handoff. For example, if we can predict that a new upcoming workload could cause resource contention when it runs with current VMs, then the cloudlet discovery system can take a preventive approach and suggest a different cloudlet in the discovery step. Another approach involves migrating a VM that causes resource contention to another cloudlet using VM handoff. In this case, we migrate VM not because of the user's mobility but in order to balance workloads across cloudlets. When addressing this research topic, we should focus on how to optimize the placement of cloudlet workloads while globally maximizing resource utilization without sacrificing response time.

Manageable Cloudlets

Because of their distributed nature, the management cost of cloudlets is likely to be much higher than that of centralized cloud data centers. To reduce this cost, we use an alliance-like deployment model for cloudlets. Within this model, cloudlets are not actively managed after installation. Instead, soft state elements from a cloud (e.g., virtual machine images as well as files from a distributed file system) are cached on their local storage. The resulting absence of hard (durable) state in the cloudlet that keeps management overhead low.

Although this approach enables us to minimize management cost with the cloudlet's software stack, the management cost for the underlying software (e.g. OpenStack++) and hardware remains considerable. Because updates within this low layer are required less frequently than updates of high-level applications, theoretically, management can be simple. However, in practice, if one wants to deploy and maintain a large number of cloudlets, management of this low layer cannot be ignored. Hence, we should think carefully about how we can efficiently manage cloudlet infrastructure to minimize management requirements.

Lowering Response Time at the Last Hop of the Network

Last-hop network connectivity has been dramatically improved over the past decade. Initial smartphones, in the early 2000s, shipped with 802.11b WiFi, which had up to 11 Mbps throughput. By contrast, today's smartphone is equipped with 802.11ac WiFi that can transmit up to 1.69 Gbps (with a 4-Antenna AP and a 2-Antenna client). Similarly, cellular networking has been upgraded from 2G to 4G LTE, and now operators are showing demos of 5G network supporting up to 25 Gbps [63].

However, all these efforts are focused on increasing throughput but not on reducing network latency. For cloudlet applications, we should put more effort into reducing latency at the last hop of the network. For instance, we need to analyze network interference caused by multiple devices and study on how to minimize it. Network interference is a challenging problem, especially in unlicensed frequency (e.g., WiFi). A cellular network also sacrifices network latency to maximize utilization. For example, the cellular operator typically quantizes network transmission slots into discrete buckets based on the next transmission slot designated for the device.

The offloading response time cannot be degraded only by the last-hop network but can also be slowed down by other components of the system. Mobile devices are frequently on and off the network device in order to save energy, and this causes delays in network transmission. And scheduling policy on an operating system for both the mobile-side and server-side can produce additional latency. In this research, we need end-to-end analysis of processing delay that includes the application framework, operating systems, and network stacks. Based on the time breakdown

of each of these components, we can locate inefficiency within the components.

8.2.2 Computing at the Edge of the Internet

Today, many of "things" in the Internet (i.e., IoT) are located at the edge of the network and generate a large volume of data. These are usually sensor devices that are distributed at local sites for monitoring and measurement purposes. The data generated is not usually transmitted to a central cloud but is stored at the local site for following reasons. First, the volume of data is too large to transfer over a wide area network (WAN) or the speed of data generation is faster than the network transmission speed. Second, data can be summarized using representative values so that transmission of raw data is unnecessary. For example, one does not want to upload temperature data to a cloud data center for every 10 ms. Instead hourly or daily averages are sufficient for most of the applications. Legal issues with regard to data movement/placement or privacy concerns can also prevent a cloud data center from collecting dispersed data.

With the proliferation of data generated at the edge of the Internet, an efficient system for edge data analysis is needed. The edge analysis system should be able to perform not just a set of preconfigured operations but also fully customized operations based on users' specific requests. Consider video recording, which comprises one of the most common type of data in IoT devices. Given the richness of the video contents, the recorded data can often end up being valuable for some totally unintended reasons. For example, a CNN news item [47] reported the arrest of a thief who could be seen stealing in the background of the video clip. Normally, a tiny patch in the background would be ignored, but in this case, the context of a robbery made it relevant. To capture such unexpected aspects of the data, the edge analytics system should fully support custom query. Further, because the result of the analysis can be used by operations and real-time decision algorithms, minimizing the response time of this analysis becomes crucial.

How to minimize the response time required for analyzing highly distributed data is a challenging task. Followings are the research questions to be answered in this study.

- how to avoid laggard cloudlets in order to reduce response time,
- how to efficiently exchange (intermediate) data between cloudlets,
- how to handle repeated or similar queries efficiently.

Bibliography

[1] How to assign devices with VT-d in KVM. http://www.linux-kvm.org/page/How_to_assign_devices_with_VT-d_in_KVM.

[2] Libvirt: Domain XML format. https://libvirt.org/formatdomain.html.

[3] Neutron - OpenStack. https://wiki.openstack.org/wiki/Neutron,.

[4] OpenStack Wikipedia. https://en.wikipedia.org/wiki/OpenStack,.

[5] T.R. Agus, C. Suied, S.J. Thorpe, and D. Pressnitzer. Characteristics of human voice processing. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, Paris, France, June 2010.

[6] AKAMAI. State of the Internet. http://www.akamai.com/dl/akamai/akamai-soti-q114.pdf.

[7] Sherif Akoush, Ripduman Sohan, Bogdan Roman, Andrew Rice, and Andy Hopper. Activity based sector synchronisation: Efficient transfer of disk-state for wan live migration. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 22–31, 2011.

[8] Apple. Apple Siri. https://www.apple.com/ios/siri/.

[9] Avahi Project. Avahi. http://www.avahi.org/.

[10] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, San Juan, Puerto Rico, 2007. ACM. ISBN 978-1-59593-614-1. doi: 10.1145/1247660.1247692. URL http://doi.acm.org/10.1145/1247660.1247692.

[11] Sean K. Barker and Prashant Shenoy. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of ACM Multimedia Systems*, Phoenix, AZ, February 2010.

[12] Adam L. Berger, Vincent J. Della Pietra, and Stephen A. Della Pietra. A maxi-

mum entropy approach to natural language processing. *Comput. Linguist.*, 22(1):39–71, March 1996. ISSN 0891-2017. URL http://dl.acm.org/citation.cfm?id=234285.234289.

[13] Nilton Bila, Eyal de Lara, Kaustubh Joshi, H. Andrés Lagar-Cavilla, Matti Hiltunen, and Mahadev Satyanarayanan. Jettison: Efficient idle desktop consolidation with partial vm migration. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 211–224, 2012.

[14] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, Helsinki, Finland, 2012.

[15] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 169–179, 2007.

[16] Gabriel Brown. Converging Telecom & IT in the LTE RAN. White Paper, Heavy Reading, February 2013.

[17] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. Kaleidoscope: Cloud micro-elasticity via VM state coloring. In *Proceedings of the Sixth Conference on Computer Systems*, Salzburg, Austria, 2011. ACM. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966471. URL http://doi.acm.org/10.1145/1966445.1966471.

[18] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[19] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*, Boston, MA, 2005.

[20] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=874075.876409.

[21] Stuart Cheshire. Multicast DNS. http://www.multicastdns.org/.

[22] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth ACM European Conference on Computer Systems*, Salzburg, Austria, 2011. ACM.

ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966473. URL http://doi.acm.org/10.1145/1966445.1966473.

[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, 2005.

[24] S. Clinch, J. Harkes, A. Friday, N. Davies, and M Satyanarayanan. How close is close enough? Understanding the role of cloudlets in supporting display appropriation by mobile users. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications*, Lugano, Switzerland, Mar. 2012.

[25] Alvaro Collet, Manuel Martinez, and Siddhartha S. Srinivasa. The MOPED framework: Object Recognition and Pose Estimation for Manipulation. *The International Journal of Robotics Research*, 2011.

[26] CRIU. Checkpoint/Restore in Userspace. https://criu.org/Main_Page.

[27] E. Cuervo, A. Balasubramanian, D. Chok, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, June 2010.

[28] P. Deutsch. DEFLATE Compressed Data Format Specification, 1996. http://tools.ietf.org/html/rfc1951.

[29] Inc Docker. Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com/.

[30] Stephen R Ellis, Katerina Mania, Bernard D Adelstein, and Michael I Hill. Generalizeability of Latency Detection in a Variety of Virtual Environments. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 48, 2004.

[31] Jason Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing via Opportunistic Offload*. Morgan & Claypool Publishers, 2012.

[32] Jason Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In *Proceedings of the Seventeenth ACM Symposium on Operating systems Principles*, 1999.

[33] Jason Flinn, Dushyanth Narayanan, and Mahadev Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=874075.876398.

[34] Jason Flinn, SoYoung Park, and Mahadev Satyanarayanan. Balancing performance, en-

ergy, and quality in pervasive computing. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, 2002. IEEE Computer Society. ISBN 0-7695-1585-1. URL http://dl.acm.org/citation.cfm?id=850928.851899.

[35] Eclipse Foundation. Jetty - servlet engine and http server. http://www.eclipse.org/jetty/.

[36] Gallagher, P. Secure Hash Standard (SHS), 2008. http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.

[37] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proceedings Network and Distributed Systems Security Symposium*, San Diego, California, USA, 2003.

[38] Gizmag. Lumus glasses let you watch video, and the real world, 2012. http://www.gizmag.com/lumus-see-through-video-glasses/20840/.

[39] Google. Client IP information in DNS requests. http://tools.ietf.org/html/draft-vandergaast-edns-client-ip-01,.

[40] Google. Google Goggles. http://http://www.google.com/mobile/goggles,.

[41] Google. tesseract-ocr. http://code.google.com/p/tesseract-ocr/,.

[42] Google. Voice Actions for Android, 2011. http://www.google.com/mobile/voice-actions/.

[43] Google. Glass Glass. https://www.google.com/glass/start/, 2014.

[44] Kira Greene. AOL Flips on 'Game Changer' Micro Data Center. http://blog.aol.com/2012/07/11/aol-flips-on-game-changer-micro-data-center/, July 2012.

[45] Arpit Gupta, Jeongki Min, and Injong Rhee. WiFox: Scaling WiFi performance for large audience environments. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, Nice, France, 2012. ACM. ISBN 978-1-4503-1775-7. doi: 10.1145/2413176.2413202. URL http://doi.acm.org/10.1145/2413176.2413202.

[46] Selim Gurun, Chandra Krintz, and Rich Wolski. NWSLite: A light-weight prediction utility for mobile devices. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services*, Boston, MA, USA, June 2004.

[47] Jamie Guzzardo. CNN report: New Jersey family's picture catches theft in the

making. http://www.cnn.com/2010/CRIME/08/24/new.jersey.theft.photo/index.html?hpt=C1, AUG 2010.

[48] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards Wearable Cognitive Assistance. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys 2014)*, 2013.

[49] Kiryong Ha, Padmanabhan Pillai, Grace Lewis, Soumya Simanta, Sarah Clinch, Nigel Davies, and Mahadev Satyanarayanan. The impact of mobile multimedia applications on data center consolidation. In *Proceedings of the IEEE International Conference on Cloud Engineering*, San Francisco, CA, Mar. 2013.

[50] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. Adaptive vm handoff across cloudlets. Technical Report CMU-CS-15-113, CMU School of Computer Science, 2015.

[51] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 51–60, Washington, DC, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508301. URL http://doi.acm.org/10.1145/1508293.1508301.

[52] IETF. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://tools.ietf.org/html/rfc7231.

[53] InfoWord. HP blade server targets cloud computing.

[54] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, Virginia, USA, 2007. ACM. ISBN 978-1-59593-703-2.

[55] Jibbigo. Jibbigo Voice Translator Apps. http://www.jibbigo.com.

[56] Richard W.M. Jones. Libguestfs tools for accessing and modifying virtual machine disk images, Dec. 2012. http://http://libguestfs.org/.

[57] James J. Kistler and Mahadev Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[58] Michael Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, NY, USA, 2002. IEEE Computer Society. ISBN 0-7695-1647-5. URL http://dl.acm.org/citation.cfm?id=832315.837557.

[59] Danielle Kucera. Amazon apologizes for Christmas Eve outage affecting Netflix, December 2012. http://articles.washingtonpost.com/2012-12-31/business/36103607_1_amazon-web-services-christmas-eve-outage-largest-online-retailer.

[60] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M Tracey. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247415.1247420.

[61] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, Nuremberg, Germany, 2009. ACM.

[62] Hsu-Fang Lai, Yu-Sung Wu, and Yu-Jui Cheng. Exploiting neighborhood similarity for virtual machine migration over wide-area network. In *Proceedings of the 2013 IEEE 7th International Conference on Software Security and Reliability*, SERE '13, pages 149–158, 2013.

[63] Stephen Lawson. Ericsson working with AWS to make carriers more agile, Feburary 2016. http://www.pcworld.com/article/3036024/ericsson-working-with-aws-to-make-carriers-more-agile.html.

[64] Michael B. Lewis and Andrew J. Edmonds. Face detection: Mapping human performance. *Perception*, 32:903–920, 2003.

[65] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th annual conference on Internet measurement*, pages 1–14. ACM, 2010.

[66] David Lowe. The Computer Vision Industry , 2010. http://people.cs.ubc.ca/~lowe/vision.html.

[67] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[68] Martin A. Brown. Traffic Control HOWTO. http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html.

[69] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. Xvmotion: Unified virtual machine migration over long distance. In

Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14, pages 97–108, 2014.

[70] Mark McLoughlin. The QCOW2 image format, Sep. 2008. http://people.gnome.org/~markmc/qcow-image-format.html.

[71] Robert McMillan. (Real) Storm Crushes Amazon Cloud, Knocks out Netflix, Pinterest, Instagram. *Wired*, June 2012.

[72] R. Miller. AOL Brings Micro Data Center Indoors, Adds Wheels. http://www.datacenterknowledge.com/archives/2012/08/13/aol-brings-micro-data-center-indoors-adds-wheels, August 2012.

[73] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating Systems principles*, Banff, Alberta, Canada, 2001. ACM. ISBN 1-58113-389-8. doi: 10.1145/502034.502052. URL http://doi.acm.org/10.1145/502034.502052.

[74] Myoonet. Unique Scalable Data Centers, December 2011. http://www.myoonet.com/unique.html.

[75] Dushyanth Narayanan and M. Satyanarayanan. Predictive Resource Management for Wearable Computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, San Francisco, CA, 2003.

[76] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going back and forth: Efficient multideployment and multisnapshotting on clouds. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, San Jose, California, USA, 2011. ACM. ISBN 978-1-4503-0552-5. doi: 10.1145/1996130.1996152. URL http://doi.acm.org/10.1145/1996130.1996152.

[77] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[78] OpenCV. OpenCV Wiki. http://opencv.willowgarage.com/wiki/.

[79] OpenStack. OpenStack - Open Source Cloud Computing Software. http://www.openstack.org/.

[80] Ryan Paul. First look: Android 4.0 SDK opens up face recognition APIs, October 2010. http://arstechnica.com/gadgets/news/2011/10/first-look-android-40-sdk-opens-up-face\discretionary{-}{}{}recognition-apis.ars.

[81] PC world. http://www.pcworld.com/article/255519/verizon_to_offer_100g_links_resilient_mesh_on_optical_networks.html, 2012.

[82] Chunyi Peng, Minkyong Kim, Zhe Zhang, and Hui Lei. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *Proceedings of INFOCOM 2012*, INFOCOM 2012, pages 181–189, 2012.

[83] Chunyi Peng, Minkyong Kim, Zhe Zhang, and Hui Lei. VDN: Virtual machine image distribution network for cloud data centers. In *Proceedings of the 32nd IEEE International Conference on Computer Communications*, Orlando, FL, USA, 2012. IEEE.

[84] P. Jonathon Phillips, W. Todd Scruggs, Alice J. O'Toole, Patrick Flynn, Kevin W. Bowyer, Cathy L. Schott, and Matthew Sharpe. FRVT 2006 and ICE 2006 Large-Scale Results. Technical Report NISTIR 7408, National Institute of Standards and Technology, March 2007.

[85] Shaya Potter and Jason Nieh. Improving virtual appliance management through virtual layered file systems. In *Proceedings of the 25th International Conference on Large Installation System Administration*, pages 3–3, Boston, MA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2208488.2208491.

[86] Sarah Jacobsson Purewal. Siri Goes Down For a Day; Apple Says Network Outages Are Possible. *PCWorld*, November 2011.

[87] Meike Ramon, Stephanie Caharel, and Bruno Rossion. The speed of recognition of personally familiar faces. *Perception*, 40(4):437–449, 2011.

[88] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. Vmtorrent: virtual appliances on-demand. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 453–454, 2010.

[89] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. Vmtorrent: Scalable p2p virtual machine streaming. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 289–300, 2012.

[90] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. VMTorrent: Scalable P2P virtual machine streaming. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, Nice, France, 2012. ACM. ISBN 978-1-4503-1775-7. doi: 10.1145/2413176.2413210. URL http://doi.acm.org/10.1145/2413176.2413210.

[91] Wolfgang Richter, Mahadev Satyanarayanan, Jan Harkes, and Benjamin Gilbert. Near-

real-time inference of file-level mutations from virtual disk writes. Technical Report CMU-CS-12-103, Carnegie Mellon University, School of Computer Science, Feb. 2012.

[92] Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2(1), Jan. .

[93] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, 2002.

[94] Mahadev Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada, 1996.

[95] Mahadev Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, 2001. ISSN 1070-9916. doi: 10.1109/98.943998.

[96] Mahadev Satyanarayanan. Augmenting Cognition. *IEEE Pervasive Computing*, 3(2):4–5, April-June 2004.

[97] Mahadev Satyanarayanan, Benjamin Gilbert, Matt Toups, Niraj Tolia, Ajay Surie, David R. O'Hallaron, Adam Wolbach, Jan Harkes, Adrian Perrig, David J. Farber, Michael A. Kozuch, Casey J. Helfrich, Partho Nath, and H. Andres Lagar-Cavilla. Pervasive personal computing in an Internet Suspend/Resume system. *IEEE Internet Computing*, 11(2), 2007.

[98] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), October-December 2009.

[99] Antony Savvas. Firms will flee cloud if lessons from Siri and RIM outages not learned. *CFO World*, November 2011.

[100] Amazon Web Services. Overview of Amazon Web Services, Dec. 2010. `http://d36cz9buwru1tt.cloudfront.net/AWS_Overview.pdf`.

[101] PCI SIG. PCI Hot-Plug Specification Revision 1.1. `http://goo.gl/2MW0mb`, 2001.

[102] Simoens, P., Xiao, Y., Pillai, P., Chen, Z., Ha, K., Satyanarayanan, M. Scalable Crowd-Sourcing of Video from Mobile Devices. In *Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys 2013)*, Taipei, Taiwan, June 2013.

[103] B. Solenthaler and R. Pajarola. Predictive-corrective incompressible SPH. *ACM Trans.*

Graph., 28(3):40:1–40:6, July 2009. ISSN 0730-0301. doi: 10.1145/1531326.1531346. URL http://doi.acm.org/10.1145/1531326.1531346.

[104] Sphinx-4. Sphinx-4: A speech recognizer written entirely in the java programming language. http://cmusphinx.sourceforge.net/sphinx4/.

[105] Siddhartha Srinivasa, David Ferguson, Casey Helfrich, Dmitry Berenson, Alvaro Collet Romea, Rosen Diankov, Garratt Gallagher, Geoffrey Hollinger, James Kuffner, and J Michael Vandeweghe. Herb: a home exploring robotic butler. *Autonomous Robots*, 28 (1):5–20, January 2010.

[106] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. Broadband Internet Performance: A View From the Gateway. In *Proceedings of ACM SIGCOMM 2011*, Toronto, ON, August 2011.

[107] Gabriel Takacs, Maha El Choubassi, Yi Wu, and Igor Kozintsev. 3D mobile augmented reality in urban scenes. In *Proceedings of IEEE International Conference on Multimedia and Expo*, Barcelona, Spain, July 2011.

[108] Clive Thompson. What is I.B.M.'s Watson? *New York Times Magazine*, June 2011. http://www.nytimes.com/2010/06/20/magazine/20Computer-t.html.

[109] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[110] Twitter. Bootstrap - The world's most popular mobile-first and responsive front-end framework. http://getbootstrap.com/.

[111] Paul Viola and Michael Jones. Robust Real-time Object Detection. In *International Journal of Computer Vision*, 2001.

[112] Vlingo. Voice to Text Applications Powered by Intelligent Voice Recognition. http://www.vlingo.com.

[113] Carl A. Waldspurger. Memory resource management in VMWare ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, 2002. ACM. ISBN 978-1-4503-0111-4. doi: 10.1145/1060289.1060307. URL http://doi.acm.org/10.1145/1060289.1060307.

[114] Wikipedia. Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer,.

[115] Wikipedia. List of countries by Internet connection speeds. http://en.wikipedia.org/wiki/List_of_countries_by_Internet_connection_speeds,.

[116] Wikipedia. Lempel-Ziv-Markov chain algorithm, 2008. http://en.wikipedia.org/w/index.php?title=Lempel-Ziv-Markov_chain_algorithm&oldid=206469040.

[117] Wikipedia. List of 802.11ac Hardware, 2012. http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.

[118] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus van der Merwe. CloudNet: Dynamic pooling of cloud resources by live WAN migration of virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Newport Beach, California, USA, 2011. ACM. ISBN 978-1-4503-0687-4. doi: 10.1145/1952682.1952699. URL http://doi.acm.org/10.1145/1952682.1952699.

[119] YouTube statistics. http://www.youtube.com/t/press_statistics, 2012.

[120] Ming yu Chen and Alex Hauptmann. MoSIFT: Recognizing Human Actions in Surveillance Videos. Technical Report CMU-CS-09-161, CMU School of Computer Science, 2009.

[121] Weida Zhang, King Tin Lam, and Cho Li Wang. Adaptive live vm migration over a wan: Modeling and implementation. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing*, CLOUD '14, pages 368–375, 2014.

[122] Xiang Zhang, Zhigang Huo, Jie Ma, and Dan Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, Heraklion, Greece, 2010. IEEE.