공학박사 학위논문

Lightweight Offloading System For Edge Cloud Environments

엣지 클라우드 환경을 위한 연산 오프로딩 시스템

2020년 2월

서울대학교 대학원

전기 정보 공학부

정 혁 진

공학박사 학위논문

Lightweight Offloading System For Edge Cloud Environments

엣지 클라우드 환경을 위한 연산 오프로딩 시스템

2020년 2월

서울대학교 대학원

전기 정보 공학부

정 혁 진

Abstract

Lightweight Offloading System For Edge Cloud Environments

HyukJin Jeong
Department of Electrical and Computer Engineering
The Graduate School
Seoul National University

Emerging mobile applications, such as mobile cloud gaming [21] or cognitive assistance based on deep neural network (DNN) [26], require not only intense computations but also strict latency constraints. To meet the latency requirement, researchers have proposed *edge servers* [76] (also called *fog nodes* [7]), computation servers dispersed at the edges of the network, e.g., computation servers on Wi-Fi APs [77], small cell networks [56], or a computing cluster made up of mobile devices [54]. IEEE recently introduced OpenFog Reference Architecture [38], a generic architecture that distributes computing servers close to data sources (IoT sensors or actuators).

Offloading with edge servers is different from offloading with cloud servers. Edge servers are geographically widely distributed, and each of them covers a small region. Mobile clients can easily move across the boundary of edge servers, hence frequently being disconnected from an edge server and connecting to a new server. To achieve seamless mobile experience in this edge server environment, it is essential to rapidly deploy a service to an edge server and migrate the offloaded service from the previous server to the current server as the client moves.

The purpose of my dissertation is to build lightweight edge computing systems

which provide seamless offloading services even when users move across multiple edge servers. I focused on two specific application domains: 1) web applications and 2) DNN applications.

I propose an edge computing system which offload computations from web-supported devices to edge servers. The proposed system exploits the portability of web apps, i.e., distributed as source code and runnable without installation, when migrating the execution state of web apps. This significantly reduces the complexity of state migration, allowing a web app to migrate less than a few seconds. Also, the proposed system supports offloading of *webassembly*, a standard low-level instruction format for web apps [28], having achieved up to 8.4x speedup compared to offloading of pure JavaScript codes.

I also propose incremental offloading of neural network (IONN), which simultaneously offloads DNN execution while deploying a DNN model, thus reducing the overhead of DNN model deployment. Also, I extended IONN to support large-scale edge server environments by proactively migrating DNN layers to edge servers where mobile users are predicted to visit. Simulation with open-source mobility dataset showed that the proposed system could significantly reduce the overhead of deploying a DNN model.

Keywords: Cloud computing, Edge computing, Fog computing, Web application, Deep neural network

Student Number: 2014-21720

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Offloading Web App Computations to Edge Servers

Today, web is one of the most widely used mobile platforms. All modern smartphones and tablets are equipped with built-in web browsers, and many IoT devices operate based on web platforms such as Node.js, Tizen, or WebOS. These mobile web platforms run web apps whose program logics are typically written in JavaScript, a high-level language with rich language features. However, since JavaScript is not appropriate for fast execution, developers often use *webassembly* [28], a globally standardized low-level instruction format executable on web platforms such as browsers and Node.js, to implement performance-critical parts of web apps. The aim of this study is to build an edge computing framework that accelerates mobile web apps by offloading the execution of JavaScript and webassembly code from mobile device to edge servers.

One of the primary concerns of an edge computing framework is how to migrate the offloaded service between edge servers (or between edge and cloud) to handle user mobility. Since edge clouds by nature serve the clients in a close proximity, physical servers in the edge clouds have limited geographical service areas where mobile clients can easily move across the border. For seamless computation offloading, the offloaded service in the previous server has to be quickly migrated to the current server. Previous studies on edge computing take live migration of VM or container as a general solution for service migration [77] [25] [27] [57] [59], but their approach entails an intrinsic overhead to migrate the state of a virtual system (OS or language runtime), thus not efficient for offloading web app code. FaaS of cloud providers (AWS Lambda, Azure Functions) supports remote execution of JavaScript code in a *serverless* manner, but it has a limitation that it is only effective when executing short-lived, *stateless* jobs, which do not preserve data (such as global variables or closure variables) over the lifetime of each job [72] [10]; to persist any information beyond the execution of a stateless job,

the job needs to interact with other *stateful* components such as cloud storage, which inevitably introduces latency.

To make an edge computing framework for general, stateful web apps, I take an approach that dynamically migrates the runtime state of a computation thread between edge servers, and between edge and cloud. The overview of the proposed framework is depicted in Figure 1. A mobile application is programmed as a regular, self-contained web app runnable on any web-supported device. When there is an accessible edge server near the mobile device, the computation thread of the app is dynamically migrated to the edge server and executed there (Offloading). To migrate the thread at runtime, I capture the execution state of HTML5 web worker, a standard web API for multi-threading, and restore the worker with the same execution state at the destination machine; I call the web worker that can migrate to other devices as a *mobile web worker* (*mobile worker* in short). The migrated mobile worker can again move to other devices while the app is running, so it can move to an edge server close to the client and seamlessly serve the client without losing its execution state (Handoff). Even if the connection between the client and the mobile worker migrated to an edge server is broken (e.g., due to user mobility), the mobile worker can migrate from the edge to a cloud server and recover the connection with the client (Fallback).

This is the first in-depth study on offloading computations of stateful web apps written in JavaScript and webassembly to edge clouds. Previous studies on offloading web app (or JavaScript) computations [18] [102] [42] [68] [41] did not consider the case where a mobile client switches its offloading server during app execution, which would be common in edge cloud environments. Also, they did not deal with the offloading of webassembly functions, which are typically used to implement computation-intensive codes in web apps. A major contribution of this paper is the serialization algorithm that captures the complex state of HTML5 web worker, where JavaScript objects, webassembly functions, and native built-in objects are intermingled. I save the worker state as the form of JavaScript code named *snapshot*, which restores the contents of

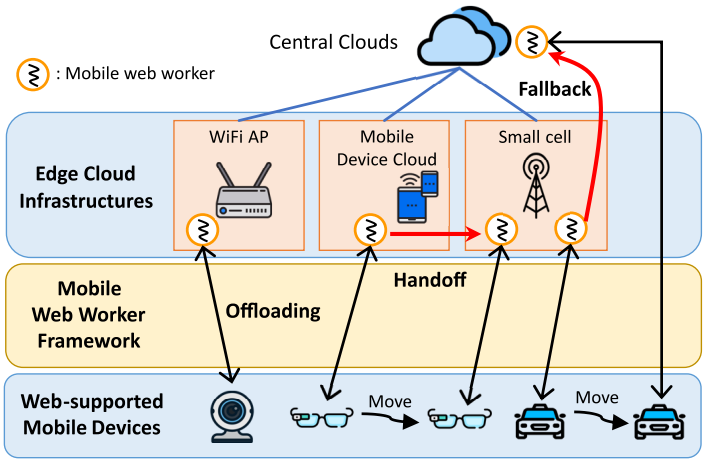


Figure 1: Overview of Proposed Framework.

JavaScript objects, instantiates webassembly functions, and reconstructs references between all runtime objects when executed.

I implemented the proposed system on popular web platforms: chromium browser and Node.js. In experiment with real hardware, our system migrated a web worker running a simple code within 150 ms. Even in a non-trivial application using a webassembly-version OpenCV library, a mobile web worker took a few seconds ($\sim$4 sec) for migration between edge servers, and between edge and cloud. Also, offloading webassembly code achieved up to 8.4x speedup, compared to offloading pure JavaScript code, reducing the performance gap between offloading with native codes and that with web app codes.

1.2 Offloading DNN Computations to Edge Servers

In recent years, Deep Neural Network (DNN) has shown remarkable achievements in the field of computer vision [51], natural language processing [86], speech recognition [24] and artificial intelligence [81]. Owing to the success of DNN, new applications using DNN are becoming increasingly popular in mobile devices. However, DNN is known to be extremely computation-intensive, such that a mobile device with limited

hardware has difficulties in running the DNN computations by itself. Some mobile devices may handle DNN computations with specialized hardware (e.g., GPU, ASIC) [62] [8], but this is not a general option for today's low-powered, compact mobile devices (e.g., wearables or IoT devices).

Current wisdom to run DNN applications on such resource-constrained devices is to *offload* DNN computations to central cloud servers. For example, mobile clients can send their machine learning (ML) queries (requests for execution) to the clouds of commercial ML services [66] [4] [20]. These services often provide servers where pre-trained DNN models or client's DNN models are installed in advance, so that the servers can execute the models on behalf of the client. More recently, there have been research efforts that install the same DNN models at the client as well as at the server, and execute the models partly by the client and partly by the server to trade-off accuracy/resource usage [30] or to improve performance/ energy savings [47]. Both approaches require the pre-installation of DNN models at the *dedicated servers*.

Unfortunately, the previous approaches are not appropriate for the generic use of *decentralized cloud infrastructures* (e.g., cloudlet [77], fog nodes [7], edge servers [76]), where the client can send its ML queries to any nearby generic servers located at the edge of the network (referred to as *cyber foraging* [75]). In this edge computing environment, it is not realistic to pre-install DNN models at the servers for use by the client, since I cannot know which servers will be used at runtime, especially when the client is on the move. Rather, on-demand installation by uploading the client's DNN model to the server would be more practical. A critical issue of the on-demand DNN installation is that the overhead of uploading the DNN model is non-trivial, making the client wait for a long time to use the edge server (see Section 2).

To solve this issue, I propose a new offloading approach, *Incremental Offloading of Neural Network* (IONN). IONN divides a client's DNN model into several partitions and determines the order of uploading them to the server. The client uploads the partitions to the server one by one, instead of sending the entire DNN model at once. The server

incrementally builds the DNN model as each DNN partition arrives, allowing the client to start offloading of DNN execution even before the entire DNN model is uploaded. That is, when there is a DNN query, the server will execute those partitions uploaded so far, while the client will execute the rest of the partitions, allowing collaborative execution. This incremental, partial DNN offloading enables mobile clients to use edge servers more quickly, improving the query performance.

As far as I know, IONN is the first work on partitioning-based DNN offloading in the context of cyber foraging. To decide the best DNN partitions and the uploading order, I introduce a novel heuristic algorithm based on graph data structure, which expresses the process of collaborative DNN execution. In the proposed graph, IONN derives the first DNN partition to upload by using a shortest path algorithm, which is expected to get the best query performance initially. To derive the next DNN partition to upload, IONN updates the edge weights of the graph and searches for the new shortest path. By repeating this process, IONN can derive a complete uploading plan for the DNN partitions, which ensures that the DNN query performance increases as more partitions are uploaded to the server and eventually converges to the best performance, expected to achieve with collaborative DNN execution.

I implemented IONN based on *caffe* DNN framework [44]. Experimental results show that IONN promptly improves DNN query performance by offloading partial DNN execution. Also, IONN processes more DNN queries while uploading the DNN model, making the embedded client consume energy more efficiently, compared to the simple all-at-once approach (i.e., uploading the entire DNN model at once).

Furthermore, I propose *PerDNN*, a system that manages the offloading of DNN execution between mobile users and a number of inter-connected edge servers. PerDNN selects the best edge server to offload DNN execution using a partitioning algorithm based on runtime states, such as GPU statistics of edge servers and network conditions, as well as hyperparameters of DNN models. After edge server selection, PerDNN dynamically deploys the user's DNN layers to the edge server and executes the DNN

model collaboratively (partially at the client and partially at the server), to minimize the DNN execution time. To avoid the cold start that occurs when a user moves to a different edge server, PerDNN periodically predicts the next edge server to visit based on the user's recent trajectory, calculates a speculative partitioning plan between the client and the predicted server, and proactively migrates the server-side DNN layers of the plan to the next edge server. This allows the user to immediately start offloading DNN execution when visiting the predicted edge server. To reduce the network traffic, I select and migrate only a fraction among the server-side layers for the hot edge servers, which sharply cuts the network traffic with negligible performance degradation. To our knowledge, PerDNN is the first study to 1) exploit GPU information for DNN partitioning and 2) perform real-time proactive caching in the context of edge computing.

To evaluate PerDNN, I simulated edge computing scenarios where more than a hundred of users offload DNN computations to edge servers dispersed in a smart city while they are on the move. For simulation, I used two open source mobility datasets collected from Beijing [104] and KAIST [71], and execution profiles of real embedded boards and desktop servers. Using a linear SVR model, which showed the best accuracy among various trajectory prediction algorithms, PerDNN predicted the next move of the clients and proactively transmitted their DNN layers to the edge servers around the predicted location. It removed 70$\sim$90% of cold starts and achieved 58$\sim$97% higher DNN query throughput when mobile users change their offloading servers, compared to a baseline with no proactive transmission. Also, I could reduce 43$\sim$67% of the peak backhaul traffics needed for proactive migration by migrating a fraction of a DNN model for crowded servers, with 1$\sim$2% of performance loss.

Chapter 2. Seamless Offloading of Web App Computations

2.1 Motivation: Computation-Intensive Web Apps

Nowadays, web apps are facing new demands for complex, feature-rich applications. For example, there exist many projects [17] to implement a game engine based on HTML5/JavaScript to run 3D games in web apps. A game engine repetitively calculates the physics of the virtual world, such as collision, fluttering of objects, and gravity, and renders the game objects in every frame, hence requiring a high computation power. There are also attempts to execute deep neural network, which is actively used in various fields but requires intense computations, on web platforms (e.g., Tensorflow.js). Those frameworks rely on advanced web features, such as WebGL for GPU support, web workers for multi-threading, or webassembly for efficient execution of low-level instructions, to deal with the heavy computations.

Despite the use of advanced web features, it is still difficult to run such compute-intensive web apps on resource-constrained mobile device. I conducted a quick experiment to observe the performance of a web app in a mobile device and a server. The mobile device was ODROID XU4 [74], a popular embedded board equipped with a quad-core ARM big.LITTLE CPU (2.0 GHz/1.5 GHz), Mali-T628 mobile GPU, and 2 GB of memory. The server was a desktop PC with intel i7 CPU (3.6 GHz), GTX 1050 GPU, and 16 GB of memory. Using ammo.js [50], a physics engine library, I implemented a web app that simulates the movements of 3D cubes falling from the air. The app was implemented in two different ways: one whose calculation code is written in JavaScript (js) and another that runs the code in webassembly (wasm). I measured the frames per second (fps) of the app while varying the number of cubes to change the computation loads.

Figure 2 shows the result. When the number of cubes is 50, all configurations

showed 60 fps, which is the maximum fps of the app. As the number of cubes increases, i.e., the amount of computations increases, the client's fps quickly dropped in both js and wasm, compared to the server's fps. Assuming that the engine is used to implement a 3D game, the client will fail to produce 30∼60 fps, a typical fps supported by modern games, even with the webassembly code when the number of cubes (≈game objects) is larger than 500. On the other hand, the server produced 30∼40 fps with 1,000 cubes. The result shows a compelling reason to offload computations to a remote server when running such a complex web app in resource-constrained mobile devices.

Another important observation in Figure 2 is that the performance of webassembly is remarkably better than JavaScript, such that ∼10 higher fps is achievable by using webassembly instead of JavaScript when the server simulates 1,000 cubes. The fast execution performance of webassembly results from its low-level assembly-like format, which can take advantage of modern hardware capabilities [28]. A recent report [32] has also demonstrated that webassembly has better performance than JavaScript in computation-intensive benchmarks. These results imply that webassembly is better than JavaScript when implementing compute-intensive codes, and therefore offloading webassembly codes must be supported for better performance when offloading web app computations.

A simple way to execute JavaScript/webassembly code in a remote server is to use a typical cloud solution, which encapsulates the code and its dependencies with a VM and deploys the VM to edge servers. Existing studies on offloading JavaScript code [68] [96] [22] can perhaps adopt the cloud solution by encapsulating their server-side components with the VM. However, the VM-based approach has a large migration overhead, thus not appropriate for the edge cloud environment where mobile users can frequently change a connected server; the overhead of VM migration is further discussed in Section 2.4.2.

Another cloud solution for offloading web app computations is FaaS services such as AWS Lambda or Azure Functions. They save a *deployment package*, which contains

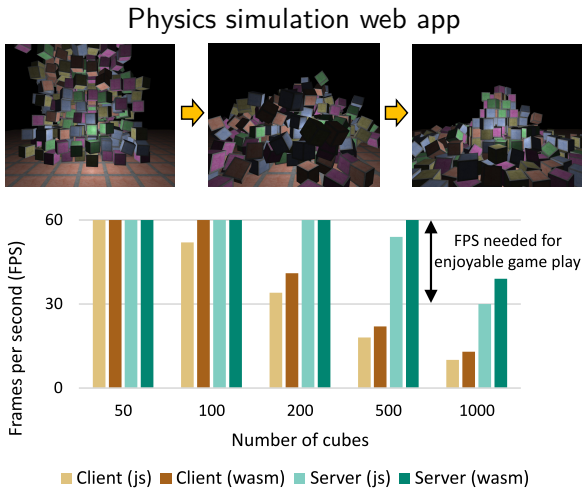


Figure 2: Frames per second (FPS) of physics simulation web app.

the code of JavaScript functions and its dependencies, in the cloud and deploys the container to an edge server using the package. When a client connects to a different edge server, a new container is created based on the deployment package saved in the cloud, so the runtime state of the container running on the previous server is not preserved. For example, the physics simulation app in Figure 2 needs to maintain the current locations of cubes in order to compute their next locations, but these location data will be lost whenever a new container is created as the physical server is changed. To preserve the runtime state beyond the lifetime of the container, the state has to be saved in the external stateful component such as cloud storage, which leads to additional latency and programming complexity [72]. A developer might be able to obviate the use of stateful components by implementing the app backend in a stateless manner, by transmitting the whole cube locations along with each request to the server (i.e., the server does not maintain any state), but it will cause substantial data transmissions if there are many state variables.

In this dissertation, I tackle all issues mentioned above by using a concept of *mobile web worker*, which is the extension of HTML5 web worker to support migration across remote devices. The runtime state of the mobile web worker is automatically captured

and restored at another server when the offloading server is changed, so stateful codes, written in either JavaScript or webassembly (or both), can be offloaded without external stateful components while the client is on the move. Also, the migration of a mobile web worker is much more lightweight than VM migration, because it only migrates the web app state, not the whole virtual system state.

2.2 Mobile Web Worker System

In this section, I briefly review HTML5 web worker first and then explain the mobile web worker system.

2.2.1 Review of HTML5 Web Worker

HTML5 web worker (web worker) is a standard web API to execute JS or webassembly code in the background of web app [96]. App developers can create a web worker in JS code by calling a constructor named *Worker* with an argument of JS code which initializes the worker, e.g., loading libraries, creating user-defined objects, and registering event handlers. When *Worker* is called, a new thread (worker thread) is created and executes the initialization code in its own global scope. After the web worker is initialized, it communicates with the main thread of the app based on *message passing* interfaces. The worker and the main thread can send a message to each other by calling *postMessage* function, and in response to the message, they invoke callback functions saved in the *onmessage* variable. After a worker finishes its job, the main thread can remove the worker by calling a function named *terminate*.

Computation-intensive codes are typically executed on web workers, because the main thread, which is responsible for user interaction, should not be blocked by executing long-running codes. For example, for the physics simulation app in Figure **??**, I made a web worker perform complex physics calculations, while the main thread just renders the results on the screen. So, the main thread could spend more time on waiting

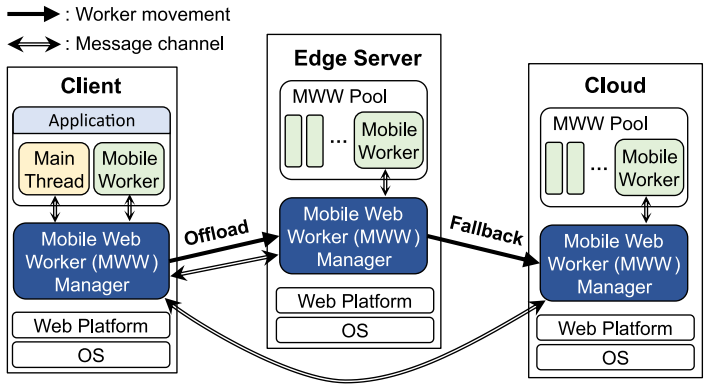


Figure 3: Mobile Web Worker System Architecture.

user inputs, making the app more responsive. Since interactivity is important for many mobile apps, computations are commonly run in web workers. As such, web workers are a natural target to offload to a more powerful server.

2.2.2 Mobile Web Worker System

Figure 3 depicts the architecture of mobile web worker system in the edge cloud environment with three components: a mobile client, an edge server, and a cloud server. The system introduces a *mobile web worker (MWW) manager*, which is installed in all devices to handle the creation, the migration, and the termination of web workers.

In the system, a mobile client runs a regular, unmodified web app whose computation-intensive codes are executed in a web worker. When the client detects accessible edge servers, MWW manager finds the best server to execute the worker, which minimizes the *latency* between when the main thread sends a request to the worker and when it receives the result from the worker. In typical web worker applications, the latency can be defined as follows:

Execution latency = input transmission time + processing time + output transmission time

where *input transmission time* indicates the time to send the input data from the main thread to the worker. *Processing time* is the time to execute callback functions saved in the worker. *Output transmission time* is the time to send the output from the worker to the main thread. *Input/output transsmission time* can be easily obtained by dividing the size of input/output data by the network speed, which can be measured at runtime. The size of the output data depends on applications, so I assumed the function of output size is given by developers, e.g., for the physics simulation app in Figure **??**, the output size is the multiplication of the number of cubes and the data size to represent the location/rotation of each cube.

The *processing time* depends on numerous factors such as hardware specification, device workloads, and application logics, so estimating the processing time of general applications is extremely difficult, which is beyond the scope of this dissertation. Instead, I made an application-specific model based on the input size of the task. For the physics simulation app in Figure **??**, the input size was defined as the number of cubes. In the offline phase, I created the dataset of processing time by running a web worker in each edge server with different input sizes, and then performed linear regression on the dataset. The regression model of each device is saved in the corresponding MWW manager, so MWW managers can estimate the processing time on line. Note that processing time estimation has been extensively studied, e.g., using program features [53], device workloads [10], or executed algorithms [47], so it would be possible to apply the previous studies according to situation.

To determine the best server with the minimum expected latency, a client sends a query along with the input size of its worker to nearby edge servers. The MWW manager of each server estimates the *processing time* based on its regression model and sends it back to the client. The client calculates the *execution latency* by adding *input/output transmission time* to the estimated *processing time*, and migrates the worker to the server with the minimum execution latency (Offload in Figure 3). If the estimated execution latency of all servers is longer than local execution time (e.g., due to bad

network conditions), the worker will be executed in the client device, i.e., a client offloads only those workers that can achieve positive offloading performance gains.

To migrate the web worker at runtime, MWW manager captures the worker's runtime state and transmits it to the server. The server restores the worker state on a new worker thread, which was pre-allocated and waiting in the thread pool named *MWW Pool* (the detailed process of web worker migration will be further explained in Section 2.3). After the worker state is restored, the MWW manager in the server reconstructs the message channel between the worker and the main thread by establishing a web socket connection with the client (depicted as double lines in Figure 3); for this, the mobile worker carries the IP address and the port number of the client-side MWW manager. The main thread and the offloaded worker can communicate with each other using the standard web worker interfaces (*postMessage*, *onmessage*), because I modified those interfaces to send and receive messages through the web socket connection when the worker is not in the client device. After being restored at the server, the web worker sends an acknowledgement message to the client to wipe out the worker in the client.

The migrated web worker can move again to other devices for better performance or to handle unstable connectivity. MWW manager periodically determines the execution location of the worker, so that the worker can migrate to a better server with less latency, if any. When the client is disconnected from the edge server, the worker migrates to a predetermined *fallback server* with stable connectivity, e.g., a cloud server, and re-establishes the connection with the client, which recovers the offloading status from abrupt disconnection (Fallback in Figure 3). If the worker cannot access any fallback server, e.g., due to internet failure, its state will be lost, so the worker will need to be restarted. To reduce such a case, I can use a more aggressive method, for example, if the client is predicted to visit a place with no internet connectivity (such as airplane or basement) based on mobility prediction algorithms [16] [98], the worker can come back to the client in advance to avoid being terminated. In this way, mobile web worker system can handle various mobile scenarios by migrating a worker between the client,

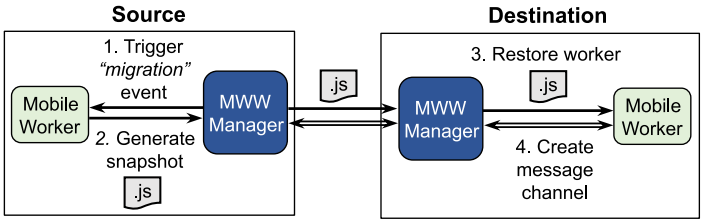


Figure 4: Overview of web worker migration.

the edge, and the cloud.

2.3 Migrating Web Worker

This section explains the detailed process of migrating a web worker. Figure 15 depicts a high level overview of web worker migration, where a web worker is migrated from a source to a destination; the source and destination can be any two devices (client, edge, or cloud server). MWW manager in the source initiates the migration of a web worker by raising a 'migration' event (step 1). In response to the event, the worker captures its runtime state as the form of JavaScript code named *snapshot*, which restores the saved worker state when executed (step 2). MWW manager in the destination restores the worker state by executing the snapshot on a new worker thread, which is the same way to create a standard web worker (step 3). Lastly, MWW manager in the destination restores the message channel between the newly created worker and the main thread in the client (step 4).

A major problem of web worker migration is how to generate a *snapshot* code that restores the worker state of a saved point. There exist a substantial amount of work on the code-format snapshot in the field of web app (or JavaScript app) migration [55] [64] [52] [18], but most of them have only focused on how to handle the complex language features of JavaScript (such as closure) when migrating the app state. When it comes to the migration of *native data* such as webassembly functions or built-in objects, little attention has been paid so far. In this section, I describe a comprehensive algorithm

for generating a snapshot to restore the web worker state, which consists of native data (webassembly functions and built-in objects) as well as JavaScript objects.

2.3.1 Runtime State of Web Worker

To capture the snapshot of a web worker, I need to first understand the runtime state of a web worker. Figure 20 illustrates the web worker state of the physics simulation app in Figure **??** right after the worker is initialized, i.e., after a worker thread is created and the user code given as the argument of *Worker* constructor is executed. A web worker has its own scope, named *worker global scope*, where *built-in objects* and *user-defined data* are saved. *Built-in objects* include standard JavaScript built-in objects (*Array*, *String*, etc) and web worker interfaces (such as *onmessage* and *postMessage*), which are automatically generated when the worker thread is created. *User-defined data* indicates any object (either JavaScript or webassembly) created when executing the user code. In the test app, the user code defines global variables 'Ammo' and 'MainLoop'. 'Ammo' is a JavaScript object created by ammo.js physics engine library, which provides webassembly functions for physics calculation. 'MainLoop' is a JavaScript function that calculates the location of cubes in each frame using the webassembly functions. The user code also specifies *event information*, the bindings between events and function objects, so that the corresponding function (event handler) is invoked when the target event is raised. 'MainLoop' function is the event handler registered for a 'timer' event, so it is repeatedly called after a certain time period.

Webassembly (wasm) functions have a unique structure different from normal JavaScript objects. Wasm functions are loaded from a *wasm file*, typically generated from other high level languages. Developers compile function codes written in a high level language (C, C++, Rust, .Net) using a wasm compiler such as *emscripten* [101]. The wasm compiler generates platform-independent wasm code and saves it as a wasm file (.wasm), which is deployed together with a web app. To use wasm functions in web apps, the wasm file has to be compiled and instantiated with a JavaScript

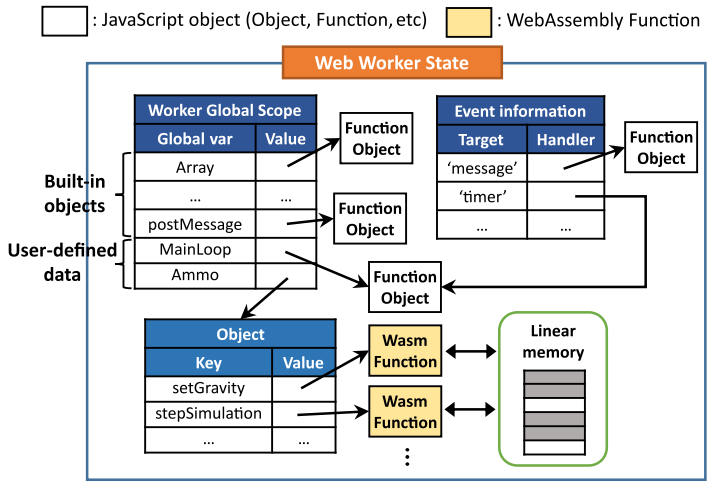


Figure 5: Runtime state of the web worker in the test app.

interface such as *instantiate*[1]. When *instantiate* is called, *importObject* is given as an argument, which specifies *linear memory* and JavaScript functions used by wasm functions. *Linear memory* is a contiguous memory space which can be used by wasm functions. In the test app, the whole state of the virtual world (cubes, gravity, collision model, etc) for simulation is stored in the linear memory, so the linear memory has to be migrated together with the mobile worker. The imported JavaScript functions also have to be restored after migration so that wasm functions can call them on the destination machine.

2.3.2 Snapshot of Mobile Web Worker

Snapshot must capture the aforementioned states of a web worker (user-defined data, built-in objects, and event information) at the source device and restore them at the destination when executed. In this section, I explain how I generate the snapshot code for each data.

[1]There are other interfaces for wasm compilation, such as *compile* or *instantiateStreaming*, but they all work similarly, so I do not explain all of them in this dissertation.

Saving User-defined Data

To capture user-defined data, I recursively traverse user-defined global variables in the worker global scope and generate a JavaScript code to restore the variables and their values. For example, if a worker has a global variable 'foo' which references a JavaScript object that has a property 'a' with the value of 1, the restoration code will be like "var foo = {'a':1};". In fact, there exist many complex issues when generating the snapshot code to handle the language features of JavaScript, such as closure, but those issues have been tackled by a plenty of previous works [55] [64] [52] [18], so I adopted their ideas to generate a snapshot code for JavaScript objects. More specifically, from [64] and [52], I adopted an *object reference array* to restore aliased references and a *scope tree* to restore nested closure.

Wasm functions contain native codes compiled at runtime, so they should not be directly migrated as normal JavaScript objects. For example, if I send wasm functions compiled at an ARM client to an x86 server, the wasm functions will not run properly. To support migration across different platforms, MWW manager sends the wasm file to the destination along with the snapshot and makes the snapshot compile the wasm file when executed. To generate such a snapshot code, MWW manager intercepts the arguments of wasm interface (*instantiate*), a JavaScript function used to compile a wasm file. The arguments are restored at the destination and passed to *instantiate*, so that the wasm functions can be instantiated at the destination machine. For this, MWW manager generates a snapshot code which restores the saved arguments, fetches the wasm file, and calls *instantiate* with the arguments to compile the wasm file.

A minor issue when compiling wasm functions is that *instantiate* needs to be called with a linear memory as the argument. Since the size of a linear memory is quite large (16$\sim$128 MB in the test apps), the compilation of wasm functions will be delayed considerably if I start compilation after the whole linear memory is transmitted. To quickly restore wasm functions, I call *instantiate* with a *dummy linear memory*, which has the same size as the transmitted linear memory but is filled with zeros, and copy the

real data onto the dummy memory when the linear memory arrives.

After compiling wasm functions, the snapshot has to restore the references between JavaScript objects and wasm functions. For the example in Figure 20, I need to create a reference from the 'setGravity' property of the 'Ammo' object to the corresponding wasm function, so that the wasm function can be called in JavaScript code, e.g., "Ammo.setGravity(...);". To do so, I traverse user-defined global variables and save a list of all variables and object properties that reference wasm functions. Also, I save the names (identifiers) of wasm functions referenced by those variables (or properties). Using these information, I generate a code that assigns wasm functions to the corresponding variables and properties.

Figure 6 shows a simplified snapshot code generated by MWW mananger. The code restores the user-defined data of the web worker state in Figure 20 when executed. The snapshot first restores JavaScript objects and global variables (line 2∼10); references of all JavaScript objects are saved in an array named *obj_ref* (abbreviation for *object reference array*) to solve the aliasing problem [64]. Next, the snapshot creates an argument object for *instantiate* (*wasm_args*), including a dummy linear memory (line 14∼17). The wasm file (ammo.wasm) is fetched and compiled with the created argument object (line 20∼23). After the wasm file is compiled, references from JavaScript objects to wasm functions are restored (26∼28); obj_ref[0] indicates the 'Ammo' object. When the linear memory, which was asynchronously sent with the snapshot code, arrives, the contents of the linear memory is copied to the dummy linear memory.

Saving Built-in Data

Built-in objects are automatically created when a worker global scope is initialized both in the source and destination device, so I do not generate a code to restore them from scratch. Instead, I check if the built-in objects were modified after they had been created and generate an update code for the modified parts. To identify the modified built-in objects, I save the initial state of built-in objects. I recursively traverse built-in

```
(function(){
    // Restore JavaScript objects
    let obj_ref = new Array();
    obj_ref[0] = {...};
    obj_ref[1] = {...};
    ...

    // Restore global variables
    self.Ammo = obj_ref[0];
    self.MainLoop = obj_ref[1];

    // Start to restore wasm functions
    // Step 1. Create argument of instantiate
    let wasm_args = {}
    wasm_args['global'] = {...};
    ...
    wasm_args['memory'] = new
         WebAssembly.Memory(...); // dummy
         linear memory

    // Step 2. Fetch and compile wasm file
    fetch('ammo.wasm').then(response) =>
        response.arrayBuffer()
    ).then(bytes =>
        WebAssembly.instantiate(bytes,
             wasm_args)
    ).then(result => {
        // Step 3. Restore references from
             JavaScript to wasm functions
        obj_ref[0]['setGravity'] =
             result.instance.exports['
             setGravity'];
        obj_ref[0]['stepSimulation'] =
             result.instance.exports['
             stepSimulation'];
        ...
    });
})();
```

Figure 6: Snapshot code generated by MWW manager. The code restores the web worker state depicted in Figure 20.

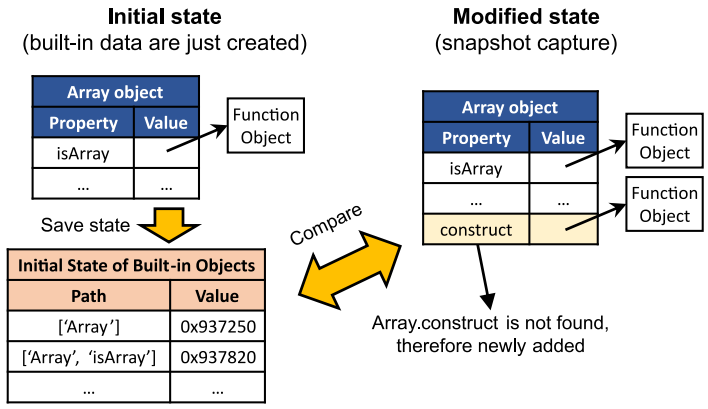


Figure 7: Example of the modified built-in *Array* object.

objects and save the paths and values of them as soon as the worker thread is created. When capturing a snapshot, I again traverse the global variables of built-in objects and compare the current value of each property with the value saved in the corresponding path. If a property was changed (added, removed, or modified), I generate a snapshot code that restores the modified parts when executed.

Figure 22 illustrates how to detect modifications on the built-in *Array* object. As soon as the *Array* object is created, I save the path and value of each property of the *Array* object, as shown in the left. The mobile worker uses a library named *Sugar.js* [46], which provides syntactic sugar by adding a new method “construct" to the *Array* object. When capturing a snapshot, I traverse the properties of the *Array* object and compare the value of each property with the saved value. ‘Array.construct’ is not found on the saved state (since it is newly added), so I generate a code that adds the method to the Array object, e.g., “Array.construct = obj_ref[∼];", and insert the code into the point of the snapshot where global variables are restored (line 11 in Figure 6). If a property of a built-in object was deleted, I generate a code that removes the property using *delete* operator.

Saving the initial state of built-in objects incurs an additional overhead when a mobile worker is created. The time to traverse all built-in objects is not negligible (∼92 ms in the client board, ODROID XU4), so I save the initial state of the built-in

objects as a file and load the file when a mobile worker is created, rather than repeatedly traversing built-in objects. The file is implemented as a JavaScript code that restores the paths and values of all built-in objects. For instance, *Array* object and *Array.isArray* method are saved as "builtin.path = ['Array', 'Array.isArray']; builtin.value = [Array, Array.isArray];". The file will automatically restore the paths and values of built-in objects when loaded in web apps.

It is worth noting that previous works on web app migration did not concern the modification of built-in objects at all [64] or partially tracked the modification on the *prototype* of built-in objects using an obsolete function (Object.observe) [52]. My approach can detect modifications in any property other than *prototype* and does not use obsolete functions, thus compatible with any library and browser.

Saving Event Information

Event information is maintained by the web platform, so I can collect it by accessing the web platform, as [64] did. MWW manager generates a code to restore the event information using a JS function, *addEventListener*, and attaches the code at the bottom of the snapshot. For a timer event, I calculate the time remaining until the timer is triggered, and create a timer event that will be triggered after that time.

2.3.3 End-to-End Migration Process

Figure 23 illustrates the entire process of web worker migration in the proposed system. The migrated web worker is assumed to be created with a JavaScript file named 'worker.js'. The worker creates built-in objects, saves their initial state, and executes the 'worker.js'. After initialization, the worker performs its job while communicating with the main thread. In the destination, a worker is created with 'base.js', which is a JavaScript file that only has an event handler that executes the snapshot code when the code is delivered to the worker; I call this worker as a *pre-built worker*. The pre-built worker stands by in the MWW pool until a snapshot code is delivered.

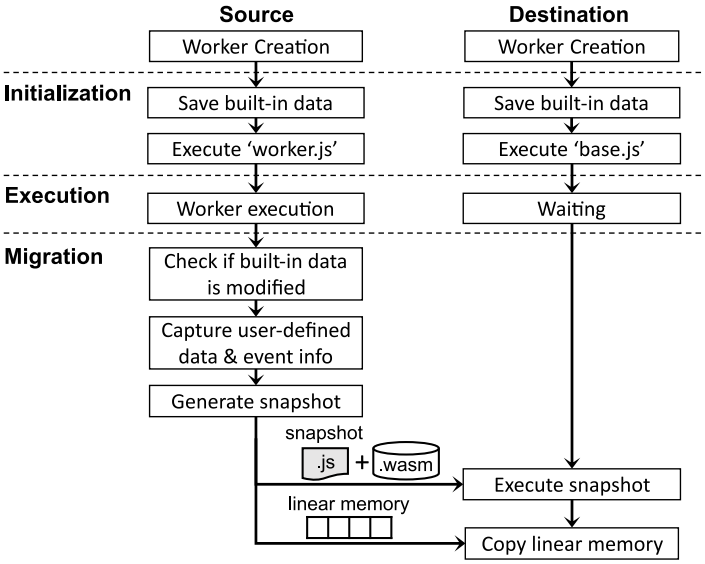


Figure 8: End-to-end web worker migration.

In the migration phase, MWW manager checks if built-in objects were modified and captures user-defined data and event information. MWW manager generates a snapshot code and sends it to the destination along with the wasm file. The linear memory is also transmitted asynchronously. The snapshot is delivered to one of the pre-built workers in the MWW pool. The pre-built worker executes the snapshot to restore the web worker state, and copies the linear memory to the dummy linear memory when the linear memory arrives. If the linear memory arrives faster than the snapshot, MWW manager in the destination waits for the arrival of the snapshot to keep the restoration order.

2.4 Evaluation

2.4.1 Experimental Environment

I implemented the proposed mobile worker system in chromium browser (for a client) and Node.js (for edge and cloud). I slightly modified the browser and Node.js to collect the runtime data maintained internally by the web platform, such as closure and event information, when capturing a snapshot. The number of lines modified in each web

platform was ∼200, and the rest of the system was implemented in JavaScript, which implies that it is not very burdensome to implement the proposed system on existing web platforms. There is a method that can collect the runtime state without modifying the web platform by instrumenting the app code [55] [18], but I did not take the approach because the instrumented code is significantly slower than the original app code [18]. Another minor implementation issue was that HTML5 web worker is not officially supported in Node.js. To implement a mobile worker in Node.js, I made a pseudo-web worker using the Node.js *subprocess* module. I implemented the interfaces and the built-in objects of a web worker as a Node.js package, making the Node.js process mimic the behavior of the standard web worker.

The client board (Odroid-XU4) has a quad-core ARM big.LITTLE CPU (2.0GHz/ 1.5GHz) with 2GB memory. I have two edge servers: *source* and *destination*. The source server is equipped with quad-core CPU (3.6 GHz, i7-7700) and 16 GB memory, and the destination server is equipped with quad-core CPU (3.0 Ghz, 17-5960x) and 32 GB memory. Cloud server has 8 vCPU (2.0GHz, Xeon) with 32 GB of memory, leased from Google Cloud. Upload and download speed between two edge servers were set to 42 Mbps and 118 Mbps, respectively (average fixed broadband speed of United States in April 2019 [39]), using a Linux traffic controller (tc-netem). Upload and download speed between the client and edge servers was set to 10 Mbps and 36 Mbps (average mobile network speed of United States in April 2019 [39]). Network bandwidth between the edge and cloud was ∼250 Mbps.

I conducted experiments with three web apps, all of which perform intense computations in the web worker while the main thread interacts with a user. Table 7 shows the description of each app. *physics* is the test app used in Figure **??**, which renders 3D cubes on the screen based on Ammo physics engine. *opencv* performs face detection using Haar cascade in opencv.js, which is a web version of OpenCV library [65]. *filter* applies a Gaussian blur filter on the image using a library named web-dsp [14]. All wasm files used in the test apps were compiled from C++ code by using *emscripten*

App	App description	Library
Physics Simulation (physics)	Main thread renders 3D cubes on the screen using the locations and rotations of cubes calculated by the web worker.	Ammo.js
OpenCV Face Detection (opencv)	Main thread sends an image (258x196) to the worker. The worker performs face detection algorithm and returns the locations of faces to the main thread.	opencv.js
Image Filtering (filter)	Main thread sends an image (640x480) to the worker. The worker applies Gaussian blur filter to the image and sends the blurred image to the main thread.	web-dsp

Table 1: Test web applications

[101], a popular wasm compiler.

2.4.2 Migration Performance

This section evaluates the *migration time* of a web worker, which directly affects the performance of handoff and fallback in edge clouds. I measured the migration time as the time between when the source device triggers a “migration" event and receives a message from the migrated worker; the worker sends the message as soon as the linear memory is restored (the linear memory data is copied to the dummy memory). I measured the migration time in three different migration patterns explained in section 2.2.2: *offload* from mobile device to an edge server, *handoff* from an edge server to another edge server, and *fallback* from an edge server to a cloud server.

Figure 18 shows the migration time composed of three components: *capture*, *restore*, and *others*. *Capture* indicates the time to capture a snapshot at the source device. *Restore* is the time between when the destination starts to execute the snapshot and finishes copying the linear memory to the dummy memory; this includes the time to execute a snapshot, wait for the arrival of linear memory asynchronously transmitted through network, and copy the linear memory to the dummy memory. *Others* is the rest of

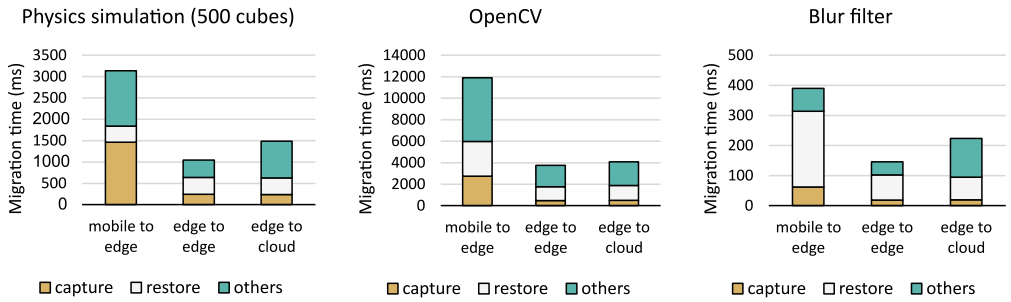


Figure 9: Migration time of a mobile worker. *Capture* and *restore* indicate the time to capture and restore the worker state, respectively. *Others* includes the time to send data through network and inter-process (or inter-thread) communication time between worker and MWW manager.

the migration time, which includes the time to send the snapshot and the wasm file through the network and the time spent for inter-process (or inter-thread) communication between a worker and a MWW manager. Table 8 shows the size of the snapshot, the linear memory, and the wasm file of each app. Those data are automatically compressed by web platforms before transmitted through a web socket.

The migration time of a web worker heavily depends on libraries used in the worker. *Opencv* showed a long migration time (3.8∼11.9 sec), because it uses a huge library (opencv.js) containing many wasm functions that implement OpenCV interfaces, thus spends a long time to transmit and instantiate the wasm file. Table 8 shows that *opencv* uses a much larger wasm file (5.9 MB) than other apps (892 KB for *physics* and 34 KB for *filter*). Also, opencv.js maintains a large linear memory (128 MB), so the *restore* time of *opencv* was quite long (1.3∼3.2 sec) due to the transmission time of the linear memory (even though the linear memory was compressed before transmitted). *Physics* uses a relatively small library and compact linear memory (16 MB), so it showed shorter migration time (1.0∼3.1 sec) than *opencv*. *Filter* uses a tiny library for image processing, so the migration time was much shorter (146∼390 ms).

In all apps, the migration time of *mobile-to-edge* was longer than that of *edge-to-edge* and *edge-to-cloud*. This is because the mobile device takes a longer capture time than servers. The capture time can be reduced by keeping the state of the worker as

App	Data	Size
physics	snapshot	2.7 MB
	linear memory	16 MB
	wasm file	892 KB
opencv	snapshot	4.6 MB
	linear memory	128 MB
	wasm file	5.9 MB
filter	snapshot	77 KB
	linear memory	16 MB
	wasm file	34 KB

Table 2: The size of snapshot, linear memory, and wasm files of test apps.

little as possible. For example, the libraries used in *physics* and *opencv* include many unused wasm functions, which significantly increase the number of JavaScript objects to wrap those functions. Trimming those unnecessary wasm functions will reduce the capture time. Another reason for the long migration time of *mobile-to-edge* is the low network speed between the mobile device and edge server. I anticipate the low network speed can be resolved in the near future, as new 5G internet infrastructure becomes popular.

An interesting observation in Figure 18 is the pattern of *restore* time in each app. In *physics* app, *restore* time was almost the same in all configurations (377∼393 ms), and I found that it was mostly the time to instantiate wasm functions. In *opencv* app, *restore* time of mobile-to-edge (3.2 sec) was much longer than that of *edge-to-edge* (1.3 sec) and edge-to-cloud (1.4 sec). This is because *mobile-to-edge* network is relatively slow, so the time to transmit the linear memory determined the restore time. Network speed was much faster in *edge-to-edge* and *edge-to-cloud*, so wasm instantiation dominated the restore time in those configurations. In *filter* app, the wasm file was very small (34 KB), i.e., the wasm functions were quickly instantiated, so the restore time was determined by linear memory transmission time in all configurations. These results indicate that no matter how fast the worker state (JavaScript objects, wasm functions, built-in objects) is restored, the web worker migration is finished after the linear memory

Data	VM Synthesis
Migration time (mobile to edge)	18.2 sec
Migration time (edge to edge)	7.9 sec
Migration time (edge to cloud)	7.7 sec
VM overlay size	23 MB

Table 3: Overhead of base system migration in VM synthesis

is restored. Linear memory can grow dynamically according to application logics, so it is important to keep the linear memory footprint low for fast web worker migration.

If I migrated the web worker code and its dependency encapsulated within a VM, I would have to migrate the *base system* (e.g., Node.js) as well as the web worker state. To observe the overhead of base system migration, I conducted an experiment that measures the migration time of a VM encapsulating Node.js and a very simple JavaScript program that just increments a number for every second; the migration of app state would be negligible, so the migration time can be seen as the time to migrate the base system (Node.js). I experimented with an open source project that implements *VM synthesis* [25] where a client transmits *VM overlay*, which encapsulates Node.js and the program, to the server where an Ubuntu base VM image was pre-installed, so that the server can synthesize a VM instance. Table 9 summarizes the migration time and the size of VM overlay. The VM migration took a significant migration time (7.7$\sim$18.2 sec) even when migrating a very simple JavaScript code, which indicates that VM migration is much heavier than web worker migration, thus inefficient for offloading web app computations.

2.4.3 Application Execution Performance

In this section I investigate the performance impact of offloading web workers on app execution. I implemented each test app in two different versions: one that executes compute-intensive jobs using JavaScript, and another using wasm. I migrated a worker from the client to the edge server and measured the execution performance of each app

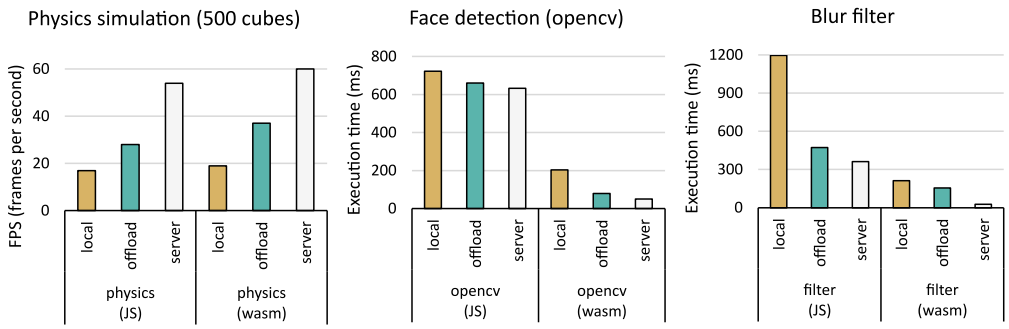


Figure 10: Execution performance of test apps. For *physics* app, I measured fps (higher is better). For *opencv* and filter apps, I measured execution time, the time between when the main thread sends a request to the worker and receives the result (lower is better).

(*offload*); I measured *fps* for *physics*, and execution time for *opencv* and *filter*. I also measured the execution performance when the app is executed solely on the mobile device (*local*) and on the server (*server*).

Figure 26 shows the result. In *physics* app, *offload* showed 1.6 times more fps in the JavaScript-version app and 1.9 times more fps in the wasm-version app, compared to local execution. Especially, the wasm-version app achieved 37 fps, which exceeds 30 fps, the minimum fps required for smooth motion. However, despite the significant improvement from *offload*, there is still a large performance gap between *offload* and *server*. This is not the problem of the proposed system but of the offloading logics of the app. I only offloaded the execution of physics calculation, so the computations for rendering objects were still performed at the client with mobile GPU. This is due to the limitation of the current web standard, which supports GPU-assisted rendering (based on WebGL) only at the main thread. If WebGL is supported in the worker thread in the future, e.g., using off-screen canvas, it will be possible to offload GPU rendering as well, and as such the offloading performance will be improved.

In *face* app, the execution time was dramatically reduced by using wasm instead of JavaScript. The speedup from offloading was only 1.1x in the JavaScript-version app, because the JavaScript execution of the offloaded computation (Haar cascade classifier) was very slow even in the server. Wasm code is executed much faster in the server, so

the wasm-version app showed much higher speedup (2.6x). I also measured the time to perform the same computation (Haar cascade classifier) using the native version of OpenCV, to compare the performance of wasm with that of native codes. The best execution time of native code was 10 ms, which was executed by code compiled with a third party parallel library (pthread) to automatically parallelize loop execution. When I compiled opencv without the parallelization library, the execution time was 35 ms. Wasm's execution time was 51 ms, still slower than both versions of native codes, but reaching the same order of magnitude as native execution time. Wasm is actively developed by major browser vendors, so I anticipate the performance gap will soon be reduced, especially by more parallelization (multi-threading or SIMD).

In *filter* app, the JavaScript-version app achieved a significant speedup (2.4x) by offloading, but the execution time of *offload* (493 ms) was even longer than the local execution time of the wasm-version app (212 ms), which presents the performance benefit of wasm. The speedup of the wasm-version app (1.4x) was not so high, because the performance improvement from offloading was limited due to the long transmission time to send input/output images. The wasm execution time in the server was very short (26 ms), which means that the execution performance of wasm was not the bottleneck when offloading computations of the *filter* app, thus showing the feasibility of replacing native codes with wasm.

Chapter 3. IONN: Incremental Offloading of Neural Network Computations

3.1 Motivation: Overhead of Deploying DNN Model

In this section, I describe a motivating example where the overhead of uploading a DNN model obstructs the use of decentralized cloud servers (throughout this dissertation, I will refer to the decentralized cloud servers as *edge servers*).

Scenario: A man with poor eyesight wears smart glasses (without powerful GPU) and rides the subway. In the crowded subway station, he can get help from his smart glasses to identify objects around him. Fortunately, edge servers are deployed over the station (like Wi-Fi Hotspots), so the smart glasses can use them to accelerate the object recognition service by offloading complex DNN computations to a nearby server.

The above scenario is a typical case of *mobile cognitive assistance* [26]. The cognitive assistance on the smart glasses can help the user by whispering the name of objects seen on the camera. For this, it will perform image recognition on the video frames by using DNNs [51] [70]. I performed a quick experiment to check the feasibility of using edge servers for this scenario, based on realistic hardware and network conditions.

The client device is an embedded board Odroid XU4 [74] with an ARM big.LITTLE CPU (2.0GHz/1.5GHz 4 cores) and 2GB memory. The edge server has an x86 CPU (3.6GHz 4 cores), GTX 1080 Ti GPU, and 32GB memory. I assumed that the client is connected to Wi-Fi with a strong signal, whose bandwidth is measured to be about 80 Mbps. I experimented with AlexNet [51], a representative DNN for image recognition.

Figure 11 shows the result. Local execution on the smart glasses takes 1.3 seconds to handle one DNN query to recognize an image. Although the CPU on the client board is competitive (the same one used in Samsung Galaxy S5 smartphone), 1.3 seconds per query seems to be barely usable, especially when the smart glasses must recognize

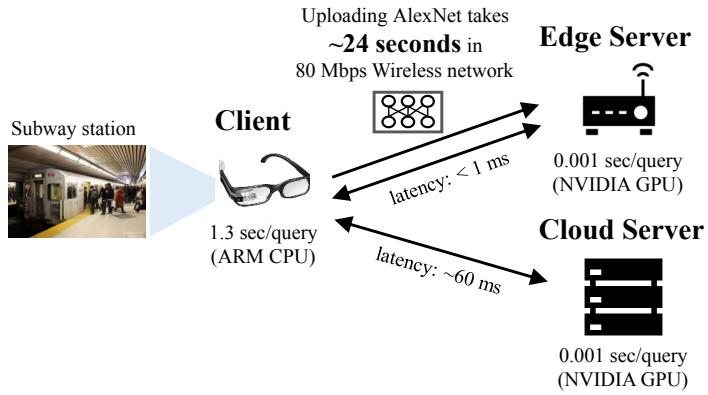


Figure 11: Example scenario of using remote servers to offload DNN computation for image recognition.

several images per second.

If I employ the edge server for offloading DNN queries, one query will take about ∼1 ms for execution, which would make a real-time service. However, the DNN model should be available at the edge server in advance to make the edge server ready to execute the queries.

A popular technique to use a random edge server is *VM (virtual machine)-based provisioning*, where a mobile client uploads a service program and its execution environment, encapsulated with VM, to the edge server (or the edge server can download them from the cloud), so that the server can run the service program [76]; some recent studies have proposed using a lightweight container technology instead of VM [58] [67]. If I use these techniques for the purpose of DNN offloading, I would need to upload a VM (or a container) image that includes a DNN model, a DNN framework, and other libraries from the client to the edge server. However, today's commercial DNN framework, such as caffe [44], tensorflow [15], or pytorch [69], requires a substantial space (more than 3 GB)[1], so it is not realistic to upload such an image on demand at runtime. Rather, it is more reasonable for a VM (or a container) image for the DNN framework to be pre-installed at the edge server in advance, so the client uploads only

[1]I measured the size of a docker image for each DNN framework (GPU-version) from dockerhub, which contains all libraries to run the framework as well as the framework itself.

the client's DNN model to the edge server on demand.

To check the overhead of uploading a DNN model, I measured the time to transmit the DNN model through wireless network. It takes about 24 seconds to upload the AlexNet model, meaning that the smart glasses should execute the queries locally for 24 seconds before using the edge server, thus no improvement in the meantime. Of course, worse network conditions would further increase the uploading time.

If I used a central cloud server with the same hardware where the user's DNN model is installed in advance, I would have obtained the same DNN execution time, yet with a longer network latency. For example, if I access a cloud server in the local region (East Asia) [20], the network latency would be about 60 ms, compared to 1 ms of the edge server due to multi-hop transmission. Also, it is known that the multi-hop transmission to distant cloud datacenters causes high jitters, which may hurt the real-time user experience [76].

Although edge servers are attractive alternatives for running DNN queries, the experimental result indicates that users should wait quite a while to use an edge server due to the time to upload a DNN model. Especially, a highly-mobile user, who can leave the service area of an edge server shortly, will suffer heavily from the problem; if the client moves to another location before it completes uploading its DNN, the client will waste its battery for network transmission but never use the edge server. To solve this issue, I propose IONN, which allows the client to offload partial DNN execution to the server while the DNN model is being uploaded.

3.2 Background

Before explaining IONN, I briefly review a DNN and its variant, Convolutional Neural Network (CNN), typically used for image processing. I also describe some previous approaches to offloading DNN computations to remote servers.

3.2.1 Deep Neural Network

Deep neural network (DNN) can be viewed as a directed graph whose nodes are *layers*. Each layer in DNN performs its operation on the input matrices and passes the output matrices to the next layer (in other words, each layer is *executed*). Some layers just perform the same operations with fixed parameters, but the others contain *trainable* parameters. The trainable parameters are iteratively updated according to learning algorithms using training data (*training*). After trained, the *DNN model* can be deployed as a file and used to infer outputs for new input data (*inference*). DNN frameworks, such as *caffe* [44], can load a pre-trained DNN from the model file and perform inference for new data by executing the DNN. In this dissertation, I focus on offloading computations for inference, because training requires much more resources than inference, hence typically performed on powerful cloud datacenters.

A CNN is a DNN that includes convolution layers, widely used to classify an image into one of pre-determined classes. The image classification in the CNN commonly proceeds as follows. When an image is given to the CNN, the CNN extracts *features* from the image using convolution (*conv*) layers and pooling (*pool*) layers. The *conv/pool* layers can be placed in series [51] or in parallel [88] [31]. Using the features, a fully-connected (*fc*) layer calculates the scores of each output class, and a *softmax* layer normalizes the scores. The normalized scores are interpreted as the possibilities of each output class where the input image belongs. There are many other types of layers (e.g., about 50 types of layers are currently implemented in *caffe* [44]), but explaining all of them is beyond the scope of this dissertation.

3.2.2 Offloading of DNN Computations

Many cloud providers are offering machine learning (ML) services [66] [4] [20], which perform computation-intensive ML algorithms (including DNN) on behalf of clients. They often provide an application programming interface (API) to app developers so that the developers can implement ML applications using the API. Typically, the API

allows a user to make a request (query) for DNN computation by simply sending an input matrix to the service provider's clouds where DNN models are pre-installed. The server in the clouds executes the corresponding DNN model in response to the query and sends the result back to the client. Unfortunately, this centralized, cloud-only approach is not appropriate for the scenario of the *generic* use of edge servers since pre-installing DNN models at the edge servers is not straightforward.

Recent studies have proposed to execute DNN using both the client and the server [47] [30]. *NeuroSurgeon* is the latest work on the collaborative DNN execution using a DNN partitioning scheme [47]. NeuroSurgeon creates a prediction model for DNN, which estimates the execution time and the energy consumption for each layer, by performing regression analysis using the DNN execution profiles. Using the prediction model and the runtime information, NeuroSurgeon dynamically partitions a DNN into the front part and the rear part. The client executes the front part and sends its output matrices to the server. The server runs the rear part with the delivered matrices and sends the new output matrices back to the client. To decide the partitioning point, NeuroSurgeon estimates the expected query execution time for every possible partitioning point and finds the best one. Their experiments show that collaborative DNN execution between the client and the server improves the performance, compared to the server-only approach.

Although collaborative DNN execution in NeuroSurgeon was effective, it is still based on the cloud servers where the DNN model is pre-installed, thus not well suited for the edge computing scenario; it does not upload the DNN model nor its partitioning algorithm considers the uploading overhead. However, collaborative execution gives a useful insight for the DNN edge computing. That is, I can partition the DNN model and upload each partition incrementally, so that the client and the server can execute the partitions collaboratively, even before the whole model is uploaded. Starting from this insight, I designed the incremental offloading of DNN execution with a new, more elaborate and flexible partitioning algorithm.

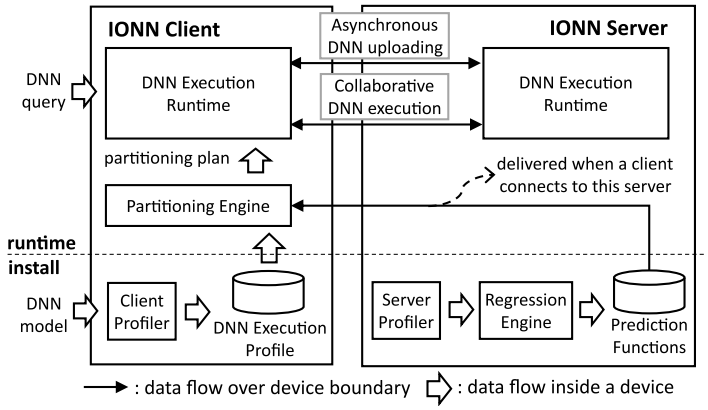


Figure 12: Overall architecture of IONN.

3.3 IONN For DNN Edge Computing

In this section, I introduce IONN, an offloading system using edge servers for DNN computations. Figure 19 illustrates the overall architecture of IONN, working in two phases. In the *install* phase, IONN collects the execution profiles of DNN layers. In the *runtime* phase, IONN creates an uploading plan that determines the DNN partitions and their uploading order, using the profile information collected in the *install* phase and the dynamic network status. According to the uploading plan, the client asynchronously uploads the DNN partitions to the server using a background thread. When a new DNN query is raised, IONN will execute it collaboratively by the client and by the server, even before the uploading of the partitions completes. I explain both phases more in detail below.

Install Phase (Client) - Whenever a DNN application is installed on the mobile device, *IONN Client* runs the DNN models used in the application and records the execution time of each DNN layer in a file called *DNN execution profile* (lower left of Figure 19). The DNN execution profile will be used by the partitioning engine at runtime to create an uploading plan.

Install Phase (Server) - An edge server cannot know which DNN models to execute in the future, so it is infeasible for the server to collect the DNN execution

profiles as the client does. Instead, when installing *IONN Server* on the edge server, I create prediction functions for DNN layers, which can estimate the time for the server to execute a DNN layer according to the type and the parameters of the layer. The prediction functions will be shipped to the client at runtime when the client enters its service area and used for the client to partition a DNN. To create the prediction functions, *IONN Server* performs linear regression on the execution data of DNN layers gathered by running diverse DNN models with different layer parameters (lower right of Figure 19), as NeuroSurgeon does [47]. I used regression functions listed in [47] to estimate the prediction functions. For the layers not mentioned in [47], I performed linear regression using the input size as the model variable.

Runtime phase - Runtime phase starts when a mobile client enters the service area of an edge server. When a client establishes a connection with an edge server, the edge server transmits its prediction functions to the client. Since the size of the prediction functions is small (hundreds of bytes for 11 types of layers in the prototype), the network overhead for sending them is negligible. After the client receives the prediction functions, the *partitioning engine* in the client creates an uploading plan using the graph-based partitioning algorithm (explained in the next section). *DNN execution Runtime* uploads the DNN partitions to the server according to the plan and performs collaborative DNN execution in response to DNN queries. Figure 20 depicts how *DNN Execution Runtime* works in more detail. Since *DNN Execution Runtime* uploads DNN partitions and executes DNN queries concurrently, I need two threads: one for uploading and the other for execution.

The *uploading thread* in the client starts to run as soon as the partitioning engine creates an uploading plan, which is a list of DNN partitions, each of which is composed of DNN layers. First, the uploading thread sends the first DNN partition in the list to the server. The server builds a DNN model with the delivered DNN partition and sends an acknowledgement message (ACK) for the partition back to the client. Then, the uploading thread in the client sends the next partition to the server. This uploading

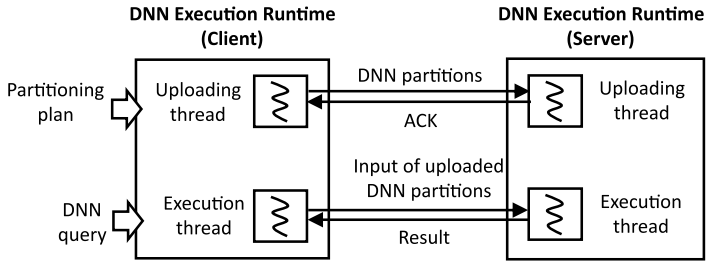


Figure 13: Asynchronous DNN uploading and collaborative DNN execution in DNN Execution Runtime.

process repeats until the last DNN partition is uploaded to the server.

The *execution thread* executes a DNN query in accordance with the current status of DNN partitions uploaded so far. The client is aware of which partitions have been uploaded to the server by checking if the ACK of each DNN partition arrived. I refer to the partitions currently uploaded to the server as *uploaded* partitions, as opposed to the *local* partitions. When a DNN query is raised, the execution thread executes the *local* partitions until just before the *uploaded* partitions and sends the result (i.e., the input matrices of the *uploaded* partitions) to the server, along with the indices of the DNN layers in the *uploaded* partitions. The server executes DNN layers whose indices are the ones delivered from the client and sends the result (i.e., the output matrices of the *uploaded* partitions) back to the client. The client and the server continue to execute the DNN partitions in this way, until the execution reaches the output layer.

3.4 DNN Partitioning

In this section, I explain how the *partitioning engine* creates an uploading plan. The partitioning algorithm tries to upload those DNN layers, needed to be at the server to achieve the best expected query performance, as early as possible. However, I do not upload those layers all at once, but one partition at a time, so that if a query is raised during uploading, the uploaded layers so far will be executed at the server for collaborative execution. For this, the algorithm partitions the DNN layers considering

both the performance benefit and the uploading overhead of each layer, so that the computation-intensive layers will be uploaded earlier. To derive such an uploading plan, I build a graph-based DNN execution model, named *NN execution graph*, and create DNN partitions by iteratively finding the fastest execution path on the graph.

3.4.1 Neural Network (NN) Execution Graph

NN execution graph expresses the collaborative execution paths by the client and the server for a DNN query at the layer level. Figure 21 illustrates an NN execution graph created from a DNN composed of three layers. Each layer is converted into three nodes (layer A: 1, 2, 3, layer B: 4, 5, 6, layer C: 7, 8, 9). Nodes in the left side (0, 1, 4, 7, 10) belong to the client, and nodes in the right side (2, 3, 5, 6, 8, 9) belong to the server. Edges between the client nodes (e.g., 1→4) indicate local execution, and edges between the server nodes in the same layer (e.g., 2→3) indicate server-side execution plus the uploading of the layer. Edges between a client node and a server node (e.g., 1→2 or 3→4) represent the transmissions of input or output matrices over the network. Each edge is added with a weight to depict the corresponding overhead. Some edges have zero weight (e.g., 0→1 or 3→5) since no computation or transmission overhead is involved. The client sets up the edge weights as follows. The client can get its local layer execution time from the DNN execution profile and estimate the server's layer execution time using prediction functions. Also, the data and layer transmission time can be calculated by dividing the size of transmitted data by the current network speed.

I can express the execution path of a DNN query as the path on the graph. Let us assume a DNN query is raised with an input data (node 0→1). If layer *A* is executed at the client, execution flow will directly go to layer *B* (node 1→4). Or if the client offloads the execution of the layer *A* to the server, then execution flow will go to node 2, and then 3. If the next layer (layer *B*) is also executed at the server, execution flow will go from node 3 to node 5. Or, if the layer *B* is executed at the client, execution flow will go to node 4. In this way, I can express the execution path of a DNN query in the graph,

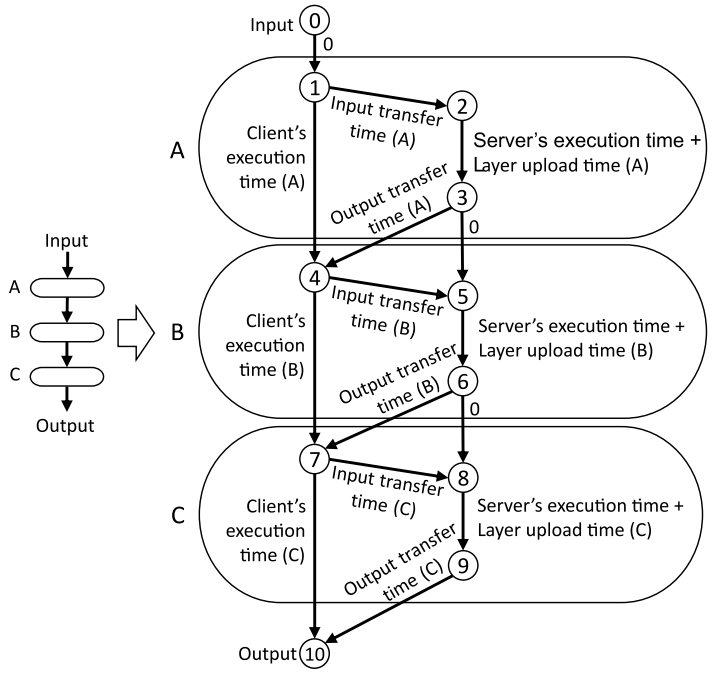


Figure 14: Example of NN execution graph whose edge weights indicate the execution time of each execution step.

and a path from an input (node 0) to an output (node 10) indicates the execution path to run the whole DNN layers. For example, a path 0-1-4-5-6-7-10 in Figure 21 means the client executes the layer *A*, offloads the layer *B*, and continues to execute the layer *C*.

The sum of edge weights on a path from the input to the output indicates the estimated query execution time of the execution path plus the time to upload the DNN layers executed at the server. For instance, the sum of edge weights on the path 0-1-4-5-6-7-10 in Figure 21 is the sum of the time to execute layer *A* and *C* at the client, the time to execute layer *B* at the server, the time to transmit the feature data, and the time to upload the layer *B*. So, if I compute the shortest path on the NN execution graph, the path will tell which layers should be initially uploaded to the server to minimize query execution time.

3.4.2 Partitioning Algorithm

The DNN partitioning problem is to decide which layers to include in the uploading partitions, and in what order to upload them, to minimize the query execution time. Unfortunately, it is impossible to find an optimal solution unless I fully know the future occurrence of queries; I need to know how soon the next query will occur to decide an optimal amount of DNN partitions to upload now (e.g., if the next query comes late, I would better upload a large partition, but if it comes soon, I would better upload a small partition quickly). Since it is hard to predict the client's future query pattern, I propose a heuristic algorithm that can work irrespective of the pattern, based on two rules. First, I prefer uploading DNN layers whose performance benefit is high and whose uploading overhead is low. This will make the server quickly build a partial DNN with high expected speedup, thus improving the query performance rather early. Second, I do not send unnecessary DNN layers, those that do not result in any performance increase, to reduce the cost associated in offloading them.

The proposed algorithm is based on the shortest path in the NN execution graph. As mentioned above, the shortest path on the NN execution graph represents the fastest

execution path for a DNN query in the *initial state* (i.e., no DNN layers are uploaded). On the other hand, if I set the layer upload time of all DNN layers in the graph to zero, the graph will represent the situation where the whole DNN model is uploaded to the server, which is the *optimal state* for offloading DNN execution. If I compute the shortest path of such a graph, it will be the execution path with the best query performance I can achieve with collaborative execution; it is not necessarily a path that executes all layers at the server (i.e., offloading the entire DNN) since some layers might better be executed at the client due to the high data transmission overhead. To eventually reach the best performance, I create an uploading plan by iteratively computing the shortest path in the NN execution graph, while changing the edge weights of the graph from the *initial state* to the *optimal state*.

Algorithm 1 DNN Partitioning Algorithm

Input: DNN model description, DNN execution profile, prediction functions, network speed, K (positive number less than 1)
Output: Uploading plan (a list of DNN partitions)

```
1:  procedure PARTITIONING
2:      partitions ← [ ] ;
3:      n ← 0;
4:      Create NN execution graph using input parameters;
5:      while K^n ≥ 0.01 do        ▷ Until layer upload time becomes ≈ 0
6:          Search for the shortest path in the NN execution graph;
7:          Create a DNN partition and add it to partitions;
8:          Update the edge weights of the NN execution graph by multiplying K
            to layer upload time;
9:          n ← n + 1;
10:     return partitions
```

Algorithm 1 describes the DNN partitioning algorithm that computes an uploading plan, a list of DNN partitions named *partitions*, which is an empty list initially (line 2). I first build an NN execution graph using the DNN model description, the DNN execution profile, the prediction functions, and the current network speed (line 4). Next, I search for the shortest path from the input to the output in the NN execution graph (line 6). I identify those layers whose server-side nodes are included in the shortest

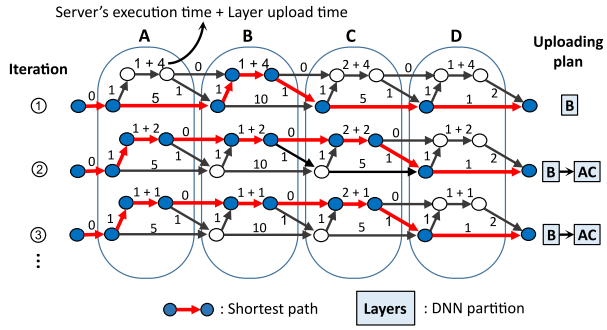


Figure 15: Illustration of the DNN partitioning algorithm.

path, create a DNN partition composed of those layers, and add it to *partitions* (line 7). Since the shortest path is computed based on the edge weights including both the layer execution time and the layer upload time, the DNN partition would include DNN layers with high expected speedup but short upload time, satisfying the first rule. I can use a shortest path algorithm for DAG using topological sorting, whose time complexity is O(n) (n : the number of layers) [19]. Next, I reduce the layer upload time by multiplying K (positive number less than 1) (line 8) and search for the new shortest path in the updated graph (line 6). The new shortest path is likely to include more server-side nodes than the previous one due to the reduction of the server-side edge weights. I create the next DNN partition with the layers whose server-side nodes are newly included in the new shortest path. I repeat this process until the edge weights for the layer upload time become almost zero (line 5), which is nearly the *optimal state* for offloading. Hence, the query performance becomes the best performance after the last partition is uploaded, satisfying the second rule. The output of the algorithm is a list of DNN partitions, and the client will upload the partitions in the list from the first to the last.

Figure 15 illustrates the partitioning algorithm with an example DNN composed of four layers (A∼D). I first build an NN execution graph for the initial state as explained in section 3.4.1. In the first iteration, I search for the shortest path from an input to an output in the graph (depicted in red arrows) and create a DNN partition [B] since only the layer B is in the server-side. Next, I multiply K (0.5 in this example) to the layer

upload time and search for the new shortest path in the next iteration. I create the second DNN partition [*A,C*], since the server-side nodes of *A* and *C* are newly included in the shortest path. In the third iteration, the shortest path is the same as before although the layer upload time is reduced by half, thus no DNN partition is newly created. In fact, layer *D* will never be uploaded, because the shortest path will not include the server-side nodes of the layer *D* even when the layer upload time reaches zero. This means the layer *D* would better be executed at the client for the best query performance. So, the algorithm generates an uploading plan, *partitions*=[[*B*], [*A,C*]]. The uploading thread will upload the layer *B* first, then the layers *A* and *C*. In this way, I can quickly upload a computation-intensive layer (*B*) and ultimately achieve the best query performance by uploading three layers (*A, B, C*) as needed. I could also save the client's energy consumption by not uploading the layer *D*.

I can adjust the granularity of DNN partitions by changing the value of *K*. If the value of *K* is small, the weight for the layer upload time will decrease sharply in each iteration. This will let the partitioning algorithm finish within a small number of iterations, therefore making a few, large DNN partitions. On the other hand, a large *K* will lead to many iterations and create many, small DNN partitions. I evaluate the impact of *K* values in Section 3.5.

Note that the algorithm assumes that edge servers have plenty of network/computing resources, so contention for shared edge server resources between multiple clients is negligible. The partitioning algorithm with multiple clients under limited edge server resources is left for future work.

3.4.3 Handling DNNs with Multiple Paths

There is an issue in the partitioning algorithm to handle DNNs with multiple paths. Figure 22 (a) illustrates the problematic situation where the algorithm does not work. The example NN has a layer whose output is delivered to three layers, and the outputs of the three layers are concatenated and given as the input of the next layer (left in

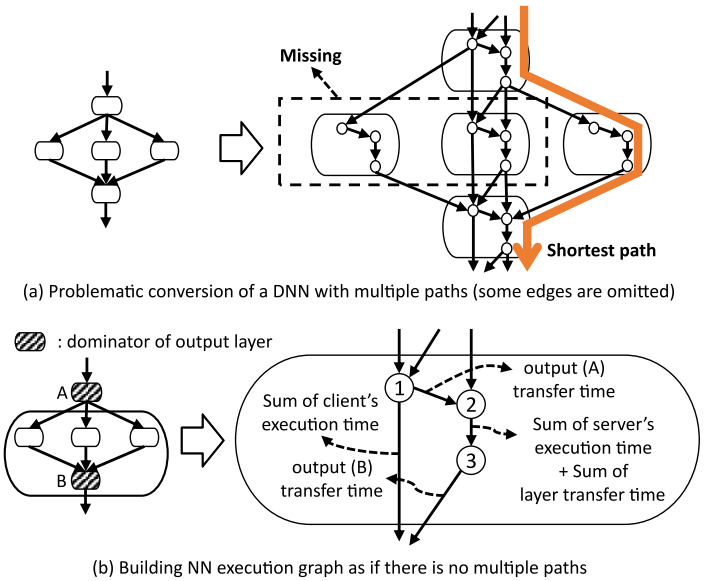


Figure 16: (a): Conversion of a DNN with multiple paths. (b): Building NN execution graph as if the DNN does not have multiple paths.

Figure 22 (a)). If I convert the layers one by one as I did in Figure 21, then the NN execution graph will be created as the right of Figure 22 (a). In this graph, the shortest path from an input to an output will include just one layer among the three layers in the middle, missing the execution of the rest two layers. This will derive a wrong uploading plan based on incomplete execution path.

To solve this problem, I build NN execution graph as if the original NN does not have multiple paths, as shown in Figure 22 (b). First, I find *dominators* of the output layer, which are layers that must be included in the path from the input to the output [1]. Next, I build NN execution graph as if layers in between two neighboring dominators (layers between A and B) and the latter dominator (B) are just one layer. The edge weights of the combined layers are shown in the right side of Figure 22 (b). Since there are no paths bypassing the dominators (due to the definition of dominators), the path from the input to the output will not miss any layer execution.

Name	Size (MB)	Number of layers	Reference
AlexNet	233	24	[51]
Inception	129	312	[89]
ResNet	98	245	[31]
GoogleNet	27	152	[88]
MobileNet	16	110	[34]

Table 4: DNNs For Evaluation

3.5 Evaluation

In this section, I evaluate IONN in terms of the query performance and the mobile device's energy consumption.

3.5.1 Experimental Environment

I implemented IONN on *caffe* DNN framework [44] using a network library *boost.asio*. I used the same client device and the edge server as those used in the experiment in Section 3.1. The client was connected to lab Wi-Fi which has a bandwidth of about 80 Mbps. The server was connected to the internet with Ethernet. I made a cognitive assistance scenario similar to the example in Section 3.1. I assumed the client repeatedly raises a DNN query for image classification after pre-processing an incoming video frame for 0.5 second, i.e., pre-processing (0.5 sec) $\rightarrow$ DNN execution $\rightarrow$ pre-processing (0.5 sec) $\rightarrow$ DNN execution $\rightarrow$... I experimented with five CNNs ranging from a small DNN (MobileNet) used in mobile devices to larger DNNs. Table 7 shows the size of DNN models and the number of layers after each DNN model is loaded on the *caffe* framework. I compared IONN with the *local* execution, and the *all-at-once* execution where the client uploads the entire DNN model and then offloads DNN queries to the server; DNN queries raised during uploading are executed at the client.

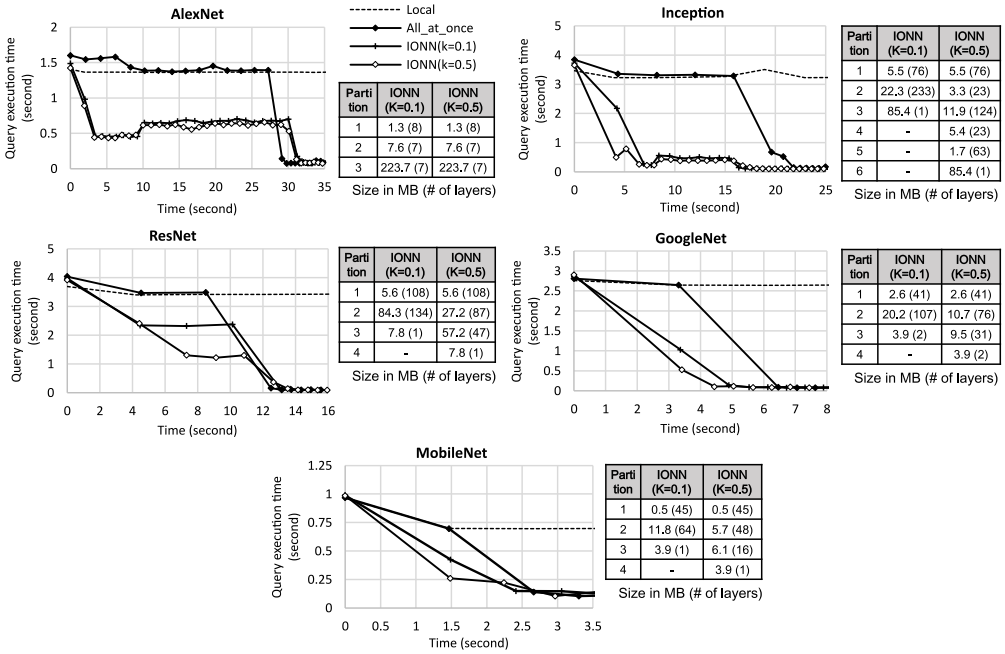


Figure 17: Execution time of DNN queries and the size of each DNN partition in the benchmark DNNs.

3.5.2 DNN Query Performance

Figure 23 shows the execution time of DNN queries in three cases: *IONN (K=0.1)*, *IONN (K=0.5)*, *all_at_once*, compared to *local*. The X value of each data point is the time when a DNN query is raised. Y value is the time spent to execute the DNN query. It should be noted that the number of executed queries (the number of data points) differs for each case; more queries are executed if the query execution time is shorter. *All_at_once* starts to offload the DNN execution only after the whole DNN is uploaded, so its query performance is low (same as *local*) until the uploading is over. On the other hand, *IONN* (both *K=0.5* and *K=0.1*) offloads partial DNN execution before the uploading is over, so the query performance is much better while uploading the DNN model. Also, I observed the query execution time of both *IONN*s rapidly decreases after a few queries and eventually reaches the minimal, which is the same execution time of *all_at_once* when the uploading is over. This implies that *IONN* can quickly offload

Name	All_at_once	IONN (K=0.1)	IONN (K=0.5)
AlexNet	28.7	30.3	30.7
Inception	16.4	16.7	16.8
ResNet	12.1	12.6	13.0
GoogleNet	3.9	3.9	4.4
MobileNet	2.3	2.5	2.8

Table 5: Uploading Completion Time (second)

computation-intensive layers and eventually achieve the best performance.

Figure 23 also shows the impact of the K value on the granularity of DNN partitions (i.e., number/size of DNN partitions) and the query performance. As expected, *IONN (K=0.5)* created more partitions than *IONN (K=0.1)* (except for AlexNet), so its average size of the partition was smaller. Since smaller DNN partitions can be uploaded to the server more quickly, the query performance of *IONN (K=0.5)* will improve earlier than *IONN (K=0.1)*. This is the reason why *K=0.5* performed better than *K=0.1* in the 3rd and 4th queries of ResNet; the large second partition of *K=0.1* was still being uploaded, while the small second partition of *K=0.5* had been already uploaded. A similar result can be observed in the 2nd query of Inception, GoogleNet, and MobileNet.

Another expected impact of K value is network overhead to handle multiple DNN partitions. The larger K value will create more, smaller partitions, and the total time to upload a DNN model will increase due to the handling of more ACK messages. Table 8 shows the impact of the network overhead on the time to upload the whole DNN layers. As expected, the uploading completion time of each DNN model is longer when the number of DNN partitions is larger (*All_at_once* $<$ *IONN (K=0.1)* $\leq$ *IONN (K=0.5)*). But the difference of the uploading completion time is small, meaning that the overhead of uploading multiple partitions is insignificant.

I observed that the query execution time in *IONN* sometimes increases during the uploading of the DNN layers, as at 10$\sim$30 second in AlexNet and 8$\sim$16 second in Inception. This phenomenon appears to be due to the transmission of the last DNN

partition, which is much larger than other partitions, interfering severely with the transmission of feature data. Nonetheless, the performance benefit of offloading is even higher than the interference overhead, so the overall query execution time of *IONN* is much shorter than that of *all_at_once*.

3.5.3 Accuracy of Prediction Functions

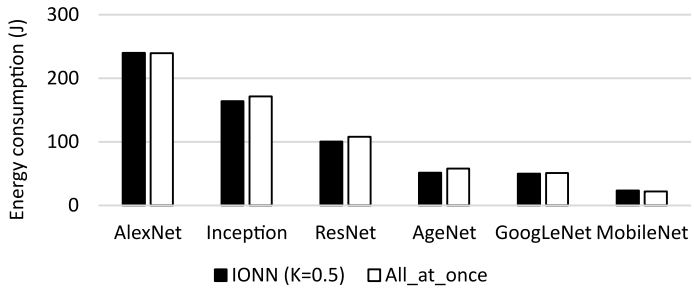
The uploading plan is created using prediction functions for the server's layer execution time, so the accuracy of the prediction functions will affect the preciseness of the edge weights in the NN execution graph, thus the final uploading plan. I evaluated the accuracy of the prediction functions as follows. I compared the uploading plan generated from the *predicted* layer execution time, with the uploading plan generated from the *real* layer execution time gathered by recording the time on the server. The uploading plans from both configurations were the same; the number of partitions and the layers included in each partition were the same. This means that the prediction functions are good enough for IONN to generate an accurate uploading plan under real hardware and network conditions.

For a statistical analysis, I calculated the coefficient of determination $(R^2)^2$ and the root mean square error (RMSE) of the prediction functions after the regression. Table 9 shows the result. The R^2 values of all layers, except the conv layer, are close to 1, which means that the prediction functions are suitable for estimating the layer execution time. Although the prediction function for the conv layer has low R^2 value (0.428), the final uploading plan generated by using the prediction function was the same as the one using the real data. This is because the server's execution time is much smaller than the client's execution time, so the error of the predicted server's execution time has little effect on the final uploading plan. The RMSE in Table 9 backs up the statement. The RMSE of the conv layer is 0.025 ms, which is negligible compared to the client's conv

[2]The coefficient of determination is the proportion of variation in the dependent variable which is explained by the independent variables. This value can be used to measure the accuracy of a prediction model. R^2 can take 0 as minimum (bad model), and 1 as maximum (good model).

Layer Type	R^2	RMSE (ms)	Layer Type	R^2	RMSE (ms)
Conv	0.428	0.025	FC	0.997	1.291
ReLU	0.999	0.001	Softmax	1.000	0.256
Pooling	0.853	0.002	BatchNorm	0.953	0.004
LRN	1.000	0.009	Scale	0.953	0.002
Concat	1.000	0.018	Eltwise	0.991	0.002

Table 6: R^2 and RMSE of Prediction Functions



Executed queries	AlexNet	Inception	ResNet	AgeNet	GoogLeNet	MobileNet
IONN	31	26	6	11	7	3
All_at_once	20	9	5	8	3	3

Figure 18: Execution time of DNN queries and the size of each DNN partition in the benchmark DNNs.

layer execution time (about tens of milliseconds), so the final uploading plan would be hardly affected by the error of the predicted server's execution time.

3.5.4 Energy Consumption

I measured the energy consumption of the client board using SmartPower2 [74] until the client finishes uploading its DNN model to the server (e.g., 0∼30.5 seconds for *IONN* in AlexNet in Figure 23). Figure 18 shows the result. In all benchmarks, *IONN* and *all_at_once* consumed a similar amount of energy (overall, *IONN* consumed slightly less energy than *all_at_once*, but the difference is insignificant). This implies the overhead of incremental offloading (e.g., ACK messages, a longer uploading time) is not burdensome for mobile devices. Figure 18 also shows the number of queries executed during the uploading. *IONN* executed more queries than *all_at_once* in all

benchmarks except MobileNet, showing higher throughput. These results imply that *IONN* improves query performance without increasing energy consumption compared to *all_at_once*.

Chapter 4. PerDNN: Offloading DNN Computations to Pervasive Edge Servers

4.1 Motivation: Cold Start Issue

To offload DNN execution from a client to a remote server, the client's DNN model has to be present in the server. Commercial edge computing solutions of major cloud providers (*Amazon GreenGrass, Microsoft Azure IoT Edge, Google Cloud IoT Edge*) let users save their DNN models in the cloud and make an edge server download the model before executing it. However, since modern DNN models for complex tasks typically consist of numerous weights, users should wait for a quite long time to download full DNN models. For example, *Inception 21k* [90], a popular DNN model for image classification with size of 128 MB, is completely downloaded in $\sim$18 seconds under the global average download speed ($\sim$57.91 Mbps [85]), which is hardly tolerable for today's mobile users. The high overhead of model deployment makes it difficult to offload DNN execution to public edge servers on demand, especially when the user moves frequently.

IONN [43] mitigates the waiting time for deploying a DNN model by *partitioning* a DNN model between a client and a server and by *incrementally uploading* the server-side layers from the client to the server (IONN works for an environment where a model is saved in the client and uploaded to the edge server for offloading [43]). IONN partially offloads the execution of uploaded layers before the full model is transmitted, so DNN execution performance is gradually improved as more layers are uploaded to the server. This deployment strategy improves DNN execution performance during uploading of DNN layers, but initial performance remains low when a client just starts uploading DNN layers, as the client has to execute most of DNN layers by itself.

Fig. 19 demonstrates the performance drop when a client starts offloading DNN computations to a new edge server. I measured the execution times of 40 consecutively

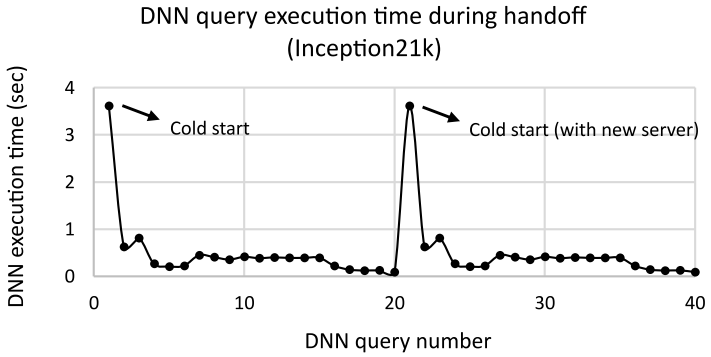


Figure 19: DNN execution time while a user moves from one edge server to another.

generated DNN queries on *Inception21k* [90]. Each query was raised 0.5 second after the previous query is executed. The client device was an embedded board (ODROID XU-4 [74]), and the server was a desktop PC equipped with a high-end GPU (Titan Xp). For the first query, the client executed all DNN layers at the local device as no layers had been uploaded to the edge server yet, so the execution time was quite high. DNN execution time decreased as more layers were uploaded, but soared at the 21st query, where the client changed its offloading server. Although DNN execution time decreased again rapidly due to incremental offloading, the spike of execution time at the start of offloading would harm users' mobile experience, e.g., frame drops in video analytics whenever a user moves from one edge server to another. Mobile users who frequently change their target edge servers would be especially vulnerable to the fluctuation.

4.2 Proposed Offloading System: PerDNN

In this section, I present the design and implementation of PerDNN, which reduces cold starts by predicting the movement of mobile users and proactively migrating DNN layers to the next edge server to visit. I first describe our edge server environment and the overall architecture of PerDNN. Next, I delve into how I partition a DNN model and how I predict the next edge server.

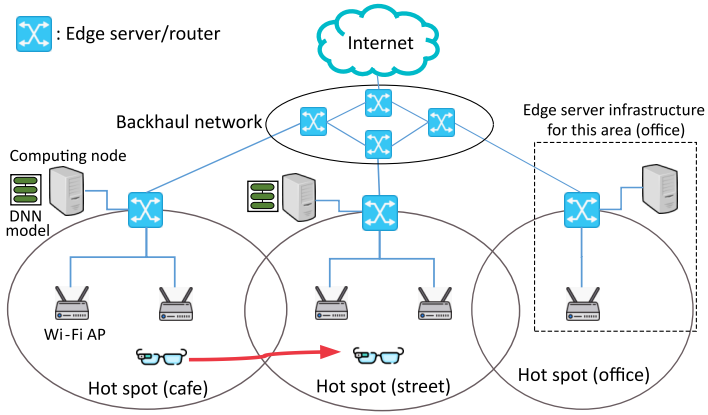


Figure 20: Edge server environment based on Wi-Fi APs.

4.2.1 Edge Server Environment

Fig. 20 illustrates the edge server environment. I considered a general wireless local area network (WLAN) where mobile users connect to nearby Wi-Fi APs in public places, e.g., cafe, street, or office. Next, I envisioned edge server infrastructure integrated with WLAN, where computing nodes, equipped with accelerators such as GPUs, are located near the hotspots. The edge server of each hotspot routes data between the users and the nearby computing node, so hotspot users can access the computing node with low latency. Edge servers are inter-connected through *backhaul network*, which has been used for file caching [5] [79] or service migration [57]. In this dissertation, the backhaul network is used to transmit DNN layers between edge servers.

A hot spot client (smart glasses in Fig. 20) can offload DNN execution to the computing node after deploying its DNN model to the node. I assumed the model is uploaded from the client to the node (like IONN [43]), but it is also possible to make the node download the model from the cloud. When the client moves to another hotspot, it can either (1) offload DNN execution to the new computing node in the current hotspot, as previously or (2) keep connection with the previous server and route input/output data through backhaul networks. In this dissertation, I focus on the first case, because routing leads to the sub-optimal offloading with increased latency and

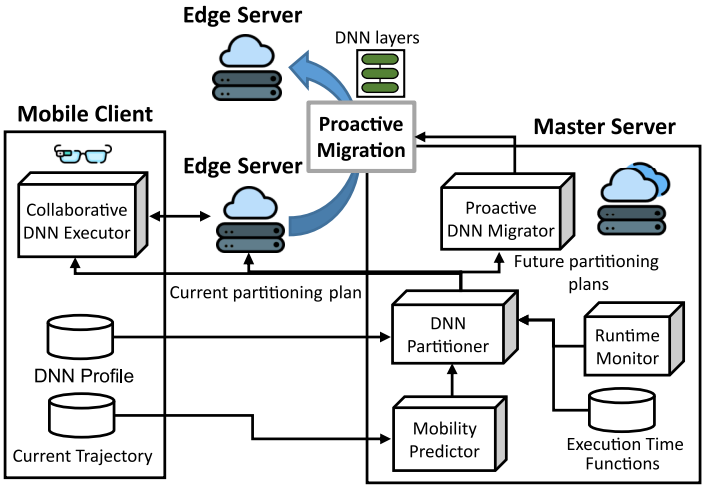


Figure 21: Overall architecture of proposed system.

constantly consumes backhaul traffics while the client offloads computations. It should be noted that despite these drawbacks, routing can be useful in some senses (e.g., load balancing [11]), but I leave the possibilities as the future work.

4.2.2 Overall Architecture

Fig. 21 shows the overall architecture of PerDNN. When a mobile client first connects to an edge server, it sends its *DNN profile* and *current trajectory* to the master server. *DNN profile* includes the types and hyperparameters of DNN layers, which are used to partition a DNN model. *DNN profile* does not contain the weights of layers (the heaviest part of a DNN model), so it can be quickly uploaded to the master server. *Current trajectory* is the recent locations of the client during a certain time period, gathered from GPS or Wi-Fi positioning system. The client periodically transmits its current trajectory to the master server, so the master server can keep track of the latest trajectory of the client.

Using the *DNN profile* and the *current trajectory* of the client, the master server controls the following actions: 1) collaborative DNN execution and 2) proactive migration of DNN layers to the next visited edge server.

Collaborative DNN execution

DNN partitioner in the master server creates a partitioning plan for a client, which specifies the execution location of each DNN layer and how to upload DNN layers. DNN partitioning is further explained in Section 3.C. The partitioning plan is delivered to the client, and the client incrementally uploads DNN layers to the edge server according to the plan. When a DNN query is raised, the client executes layers one by one until reaching the uploaded layer, and sends the input of the uploaded layer to the edge server. The edge server executes the uploaded layers and returns the result to the client. The client continues to execute the rest of the layers, if any. Since the partitioning plan is used for this current DNN execution, I call the plan *current partitioning plan*.

Proactive DNN migration

While the client and the edge server collaboratively execute a DNN model, the master server predicts the next location of the client using the client's current trajectory. Then, it finds edge servers around the predicted location by finding nearby hotspots in the Wi-Fi database such as WiGLE [94] under the assumption that the master server has the mapping between edge servers and Wi-Fi hotspots. The master server then creates partitioning plans for the edge servers near the predicted location; these plans are derived for future execution, so I call them *future partitioning plans*. The server-side layers of the future partitioning plans are transmitted from the current edge server to the corresponding edge servers. If the current edge server does not have all of the server-side layers, it sends layers as many as possible, so the client can upload only the rest of the layers when it visits the predicted server. After receiving DNN layers, edge servers keep the layers for a certain duration (*TTL*: *Time To Live*) and discard them after *TTL*. *TTL* is reset when another server attempts to send the DNN layers of the same client to the server, so the layers already existing at the server are not sent again, thus avoiding duplicate transmissions.

PerDNN periodically repeats the above processes to maintain the best execution

performance and to reduce the overhead of deploying DNN layers. When a client visits an edge server, PerDNN creates the *current partitioning plan*, as mentioned in Section 4.2.2; it should be noted that the plan is based on the current runtime states (server workloads and network conditions), so it might be different from the future partitioning plan made previously for proactive migration to this server. If the server already received all (or parts) of the server-side layers from other servers, the client can immediately start offloading the execution of those layers. If there are some missing server-side layers, the client will upload the remaining layers to complete the current partitioning plan. If the server does not have any server-side layer of the current partitioning plan, the client incrementally uploads the server-side layers from scratch. Interestingly, I found that migrating only a tiny fraction of the server-side layers can improve the performance substantially, which I can exploit to reduce the network traffic (see Section 4.3.1).

4.2.3 GPU-aware DNN Partitioning

The objective of DNN partitioning is to create a partitioning plan, which determines the execution location of DNN layers to minimize execution latency. Previous studies on DNN partitioning [43] [35] have already introduced algorithms to find the best partitioning plan with minimum execution time based upon the estimated execution time of each DNN layer. So far, however, no previous study has considered the congestion of GPU, which is critical to DNN execution performance, when partitioning a DNN model, thus having difficulty when multiple clients simultaneously offload DNN execution to an edge server equipped with a shared GPU. PerDNN uses GPU information when estimating layer execution time, so it can derive a more accurate partitioning plan than previous approaches in case of multi-client offloading.

However, estimating layer execution time is extremely challenging, because it is affected by numerous factors. When a server concurrently executes DNN inferences of multiple clients, the execution time is influenced by interference in shared GPU or system resources such as streaming multiprocessors, GPU memory, and PCIe bus.

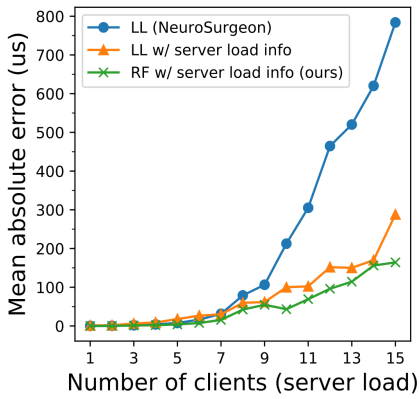
Various GPU sharing schemes (temporal sharing [92], spatial sharing [103], or hybrid of them) make it more difficult to predict the execution time. Building an estimation model considering all these factors is extremely complicated, not feasible in general. Therefore, I take a practical approach that does not require any prior knowledge of hardware details or GPU scheduling policies.

PerDNN predicts layer execution time considering GPU statistics as well as layer hyperparameters. I assumed edge servers can measure GPU statistics[1] including kernel/memory utilization, GPU memory usage, and GPU temperature without significant overhead. So, before deriving a partitioning plan, the master server pings to an edge server to obtain the current server workload and then estimates layer execution time using the *execution time estimator* of the server based on the current server workload.

The *execution time estimator* of each edge server is trained offline using the dataset generated by profiling the execution time of a DNN layer while concurrently running DNN inferences issued by multiple clients. During the profiling, server workload is changed by adjusting the number of clients; I extended *perf_client* in NVIDIA TensorRT inference server [63] to control the concurrency level and to measure the execution time. At the same time, the server records its GPU statistics whenever receiving a DNN request from a client. Using the dataset, the edge server trains random forest models for each layer type (conv, fc, etc) that predict the execution time given the server workload and the layer hyperparameters.

Fig. 22 shows the mean absolute error (MAE) of the estimation of the execution time for a convolution layer. For comparison, I plotted the MAE of the estimation model of the latest approach (*NeuroSurgeon* [47]), which only uses layer hyperparameters to train linear/logarithmic regression models (represented as *LL*); different models were trained for each server load ($\approx$ number of clients), as described in their dissertation. The result shows that MAE of *LL* surges as the number of clients increases. The MAE

[1] Kernel/memory utilization and GPU temperature were measured with nvml, Nvidia management library. Kernel/memory utilization means the percentage of time spent for kernel execution or memory operation over the past sample period (between 1 second and 1/6 second)



Feature	Importance (%)
# of clients	35.3
GPU Kernel util	24.2
Memory op util	11.5
GPU temperature	6.9
input z	5.2
kernel size	5
input y	3.8
input x	3.6
GPU memory usage	2.4
filter size	2.0
stride	0.1

Figure 22: Mean absolute errors of execution time estimation with different server workloads (conv layer).

of a single layer is insignificant (at most ∼800 us), but modern DNN models often use hundreds of layers, so the aggregate error of the entire model would be substantial.

To test if the GPU information improves the estimation, I trained the same *LL* models but with GPU statistics as well as layer hyperparameters (represented as *LL w/ server load info*). *LL w/ server load info* showed much lower MAE than *LL* when many clients send requests, which implies that GPU information is useful for estimating layer execution time under heavy workloads. My method (*RF w/ server load info*) showed even less MAE than *LL w/ server load info*, suggesting that a random forest is better than linear/logarithmic models to learn the non-linear relationship between execution time and workload features (# of clients, kernel/memory utilization, GPU temperature). The right side of Fig. 22 shows the *importance* of each feature in the random forests [78]. It indicates that the workload features were more important than layer hyperparameters when estimating layer execution time.

Given the estimated layer execution time, the optimal partitioning plan leading to the minimal execution time can be found with an algorithm proposed in IONN [43], which models DNN execution as a directed graph and finds the shortest execution path from the input to the output. By applying the algorithm to each edge server visible by the client, I can find the best edge server and the corresponding execution locations of

the DNN layers, which is the current partitioning plan. To derive future partitioning plans for proactive migration, I apply the above algorithm to all edge servers within a certain distance (r: 50 m or 100 m in the evaluation) from the predicted location.

After the execution location of each layer is determined, I need to decide the order of uploading server-side layers. My strategy is to send layers with high offloading benefits first [80]. I defined the *efficiency* of layers as the latency reduced by offloading the layers divided by the size of the layers. I create the *partitions* of the server-side layers, which are all possible successive layers in the server-side layers, and calculate the *efficiency* of each partition. Then, I decide to upload a partition with the highest *efficiency* first and update the *efficiency* of the remaining partitions. The same process is repeated until all server-side layers are uploaded. I use the same algorithm for proactive migration as well, to determine the order of sending layers from the current edge server to the target edge server.

4.2.4 Mobility Prediction

The purpose of mobility prediction is to predict the location of a mobile user to determine the next visited edge servers. Since PerDNN continuously makes predictions in real time, its prediction mechanism has to be lightweight. Also, the prediction should capture the point where the client enters the service area of the next server, so it needs to be short-ranged. To meet these requirements, PerDNN predicts the user's next location based on the user's *recent trajectory*, which can be easily collected in modern mobile devices and highly correlated with the user's next movement [45]. I assumed a client periodically collects its location (x, y coordinates) every time interval t and sends n most recent locations to the master server. The master server predicts the next location of the client after the interval t.

To achieve high prediction accuracy and system efficiency, the values of n and t have to be carefully determined. Trajectory length (n) directly affects the prediction accuracy, e.g., if n is too small, the prediction will be inaccurate due to the lack of

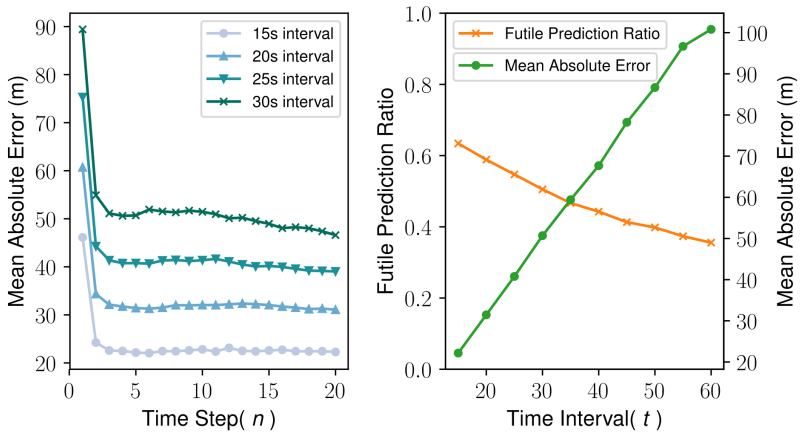


Figure 23: Left: Prediction errors with different time steps. Right: Effects of time intervals on futile predictions and prediction errors.

previous location information. Time interval (t) affects the number of *futile predictions* as well as prediction accuracy; *futile predictions* are predictions made while the client stays in the same edge server, which do not contribute to any performance improvement but waste the mobile device energy and the backhaul traffics. If I make predictions too frequently (i.e., t is small), the master server will make many futile predictions before a client leaves the server. If I increase t to reduce futile predictions, however, the prediction error will increase, because the master server will predict the client's location of a too distant future.

To determine the values of n and t, I investigated their impacts on mobility prediction using the international-scale open source dataset named *Geolife* [104]. *Geolife* contains trajectories of mobile users with short measurement intervals (1∼5 seconds), so I could make datasets with different time intervals by sampling the trajectory data in a different rate. I trained *singular vector regression* (SVR) models that predict the mobility of the users in the datasets and tested the performance of the predictors.

The left side of Fig. 23 shows prediction errors (MAE) of the mobility predictors while changing the length of trajectories (n) used for prediction. For all time intervals (15, 20, 25, 30 seconds), the prediction error dropped when n is two, which implies

that the locations of the recent two time steps are crucial to predict the location of the next time step; this finding is consistent with that of Song et al. [83], which evaluated various location predictors with extensive Wi-Fi mobility data. The prediction errors slowly decreased (30s interval) or remained the same (the others) after five steps, so I set *n* as five in this dissertation.

The right side of Fig. 23 shows the effects of time interval (*t*) on prediction errors (MAE) and the ratio of futile predictions over all predictions. For the experiment, I divided the region of the dataset into a hexagonal grid and assumed an edge server is allocated in each cell with radius of 50 m (see Section 4.3.2 for more details about the environment). The result shows that the larger *t* reduces the futile prediction ratio but at the same time increases the prediction error. To determine the proper value of *t*, I devised a scoring metric based on the *benefit* and the *cost* of proactive migration, which have following characteristics.

$$benefit \propto a \times (t - f) \tag{4.1}$$

$$cost \propto t \tag{4.2}$$

where *t* is the number of total predictions, f is the number of futile predictions, and *a* is the prediction accuracy when the predicted location is inside the service range of the next edge server. I calculated the benefit-to-cost ratios for different time intervals ranging from 15 seconds to 60 seconds, and selected the best *t* with the maximum benefit-to-cost ratio. The best *t* was 20 seconds in the settings with Geolife dataset.

After *t* and *n* are determined, the next step is to build a mobility prediction model, which predicts the next value (x, y coordinates) in the trajectory given historical trajectory information. There exist a ton of previous studies on trajectory-based mobility prediction [83] [61] [45] [100]. I engineered three previous prediction algorithms (Markov, SVR, RNN) to suit the edge server environment, and used the one with high prediction accuracy and fast performance (linear SVR). Comparison of each algorithm

is reported in section 4.3.2. Detailed implementation of each algorithm is explained below.

Markov: We implemented a Markov model based on several previous studies on mobility prediction [61] [100] [83]. The client's location (x, y coordinates) was discretized by mapping to the identifier of the closest edge server; edge servers are assumed to be distributed in a hexagonal grid (see Section 4.2.1 for more details). We created a variable-order Markov model, implemented as a prediction suffix tree [73], from the history of trajectories, based on the frequency of a sequence. Given a new user's recent trajectory, we search the longest matching pattern in the suffix tree. We multiply a ($0 < a \leq 1$) to the length of the longest matching pattern, and use the sampled subsequence with that length to get prediction [40]. The subsequence ratio (a) was set to 0.7, which achieved the best accuracy in our datasets.

Support Vector Regression (SVR): We made an SVR model [82] which takes the array of x, y coordinates, the recent trajectory of a client, as an input and outputs the x, y coordinates of the client's next location. The x, y coordinates were normalized to standard scores before fed into the SVR model for training and testing. We compared SVR models with different kernel functions (linear, polynomial, rbf) using scikit-learn v0.20.3 and chose the best one (linear SVR) with the highest accuracy. The model parameters (epsilon, tolerance) were empirically determined.

Recurrent Neural Network (RNN): We made an RNN model using a long short-term memory (LSTM) cell [33], which has been widely used for mobility prediction [84] [2] [45]. The client's trajectory was transformed to a sequence of x, y coordinates normalized to standard scores. An LSTM cell reads an input sequence and produces a *latent vector* with size of 16$\sim$32 (depending on a dataset). The latent vector is delivered to an fc layer with no activation function which outputs the x, y coordinates of the predicted location. We used MAE as a loss function and the Adam optimizer [48] with learning rate of 0.001. Hyperparameters (number of LSTM cells, latent vector size, learning rate, and optimizers) were empirically determined by using grid search.

Table 7: DNN models used for evaluation.

Name	# of Layers	Size (MB)	Description
MobileNet	110	16	MobileNet v1. Image classification among 1k classes [34]
Inception	312	128	Inception. Image classification among 21k classes [90]
ResNet	245	98	ResNet-50. Image classification among 1k classes [31]

4.3 Evaluation

In this section, I evaluate PerDNN in two setups. First, I examined the performance improvement that a single client can gain with proactive migration. Next, I conducted large-scale simulation where a number of mobile users offload DNN computations to public edge servers in the smart city.

The test application was a *cognitive assistance*, which continuously performs DNN inference to recognize objects around a visually-impaired person [26]. I assumed that a mobile client constantly generates a DNN inference query 0.5 seconds after the previous query is executed. I used three widely used DNN models for experiment. *MobileNet* [34] is a tiny DNN model designed to run on resource-constrained devices. *Inception* [90] and *ResNet* [31] are larger DNN models for a more complex task and higher accuracy. Table 7 shows the details of each model.

The client board was an ODROID XU4 [74], equipped with ARM big.LITTLE CPU (2.0/1.5 GHz 4 cores) with 2 GB memory. I used an x86 desktop PC equipped with a quad-core CPU (i7-7700), Titan Xp GPU, and 32 GB memory as an edge server. The client was connected to lab Wi-Fi (35$\sim$50 Mbps) provided by the router directly connected to the edge server. Mobility predictor was implemented with a machine learning library named *scikit-learn* [78], and the rest of the system (DNN executor and DNN partitioner) was implemented based on a popular DNN framework named *caffe* [44].

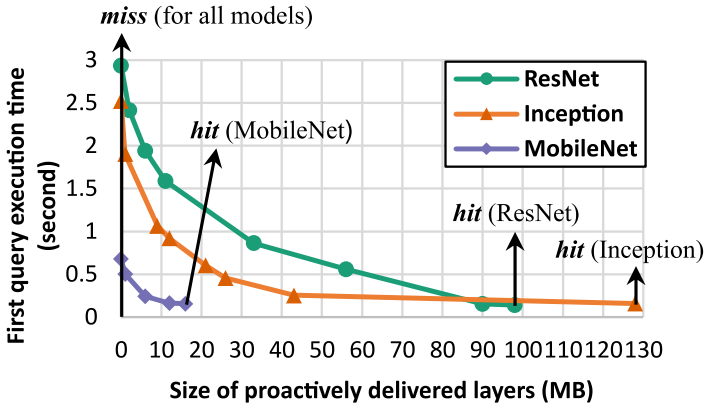


Figure 24: First query execution time with proactive migration.

4.3.1 Performance Gain of Single Client

To evaluate the performance improvement from proactive migration, I measured the time to execute the first DNN query that a client raises after connecting to a new edge server. All server-side DNN layers might have already been migrated at the new server (*hit*), or none of them were migrated (*miss*). I also measured for the cases when the layers are migrated partially, according to the amount of layers migrated.

Fig. 24 shows the result. For all DNN models, the first query execution time rapidly decreases as the server has more migrated layers. The sharp decline of execution time results from my partitioning algorithm, which sends layers with high *efficiency* first (Section 4.2.3). Especially, *Inception* showed a remarkable speedup (2.5x) in the first query execution when only 8% of the total model (∼10 MB) was sent to the server in advance. The dramatic speedup of *Inception* is due to its structure, where compute-intensive convolution layers (having high *efficiency* for efficiency-based uploading order mentioned in Section 4.2.3) are concentrated in the front part, so the execution performance was quickly improved by sending those layers first; other models have relatively lower-efficiency, evenly-distributed layers. These results indicate that migrating only a small fraction of a model can lead to significant performance improvement, which I can exploit to reduce the network traffic (see Section 4.3.2).

Table 8: Number of DNN queries executed during uploading of a DNN model (throughput).

Data	MobileNet	Inception	ResNet
Uploading time (sec)	3.7	29.3	22.4
Executed queries of *miss* (IONN [43])	4	33	14
Executed queries of *hit* (PerDNN)	5	44	34

Next, I observed throughput gains achievable by proactive migration. I measured the number of executed queries while a client uploads all DNN layers to an edge server, i.e., throughput during uploading a DNN model. A *hit* case indicates that the server received all DNN layers of a client from another server in advance, so the client immediately offloads DNN execution; this represents the best performance of PerDNN. A *miss* case indicates that the edge server did not receive any DNN layer, so the client incrementally uploads its DNN layers from scratch, which represents the baseline (IONN [43]). Table 8 shows the throughput of each case. The throughput increase was small (4→5) in *MobileNet* due to its short uploading time, but for large DNN models, proactive migration significantly improved the throughput (*Inception* (33→44) and *ResNet* (14→34)).

4.3.2 Large-Scale Simulation

Simulation Environment

I envisioned a public edge server environment where edge servers are pervasively distributed, so mobile users can access at least one edge server in any place. I set up such an environment based on two real-world mobility datasets listed below.

KAIST: KAIST is daily GPS tracks of students, collected every 30 seconds on campus [71]. The dataset contains outliers moving very far, so I only used the data points in the rectangular area (1.5 km x 2 km) including the campus site.

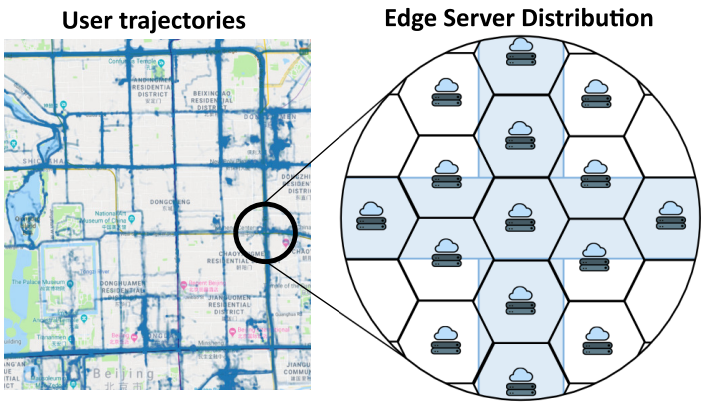


Figure 25: User trajectories and edge server distribution. The blue part indicates where the user trajectories have passed.

Geolife: Geolife is a GPS trajectory dataset of 182 users, collected every 1∼5 seconds in multiple countries [104]. I only used the data inside the rectangular area (7.2 km x 5.6 km) encompassing the circular railway of subway line 2 in Beijing, China; the ranges of latitudes and longitudes are 39.900341∼39.950932 and 116.353370∼116.437765.

Fig. 25 visualizes the user trajectories in the Geolife dataset. The blue points indicate data points of user trajectories. I divided the region into a hexagonal grid where each cell has the radius of 50 m, which is the service range of a typical Wi-Fi AP [95]. I allocated an edge server to a cell which had been visited by any user, so all users in the dataset can offload computations to the server in the current cell. The wireless network speed between the client and the edge server was set to 50 Mbps for download and 35 Mbps for upload, the average speed of lab Wi-Fi (5 GHz) in various crowdedness. Edge servers were distributed in the KAIST dataset in the same way.

To simulate realistic user movements, I played back user trajectories in the test sets of our datasets, which were not used for training mobility predictors; the number of mobile users was 31 for KAIST and 138 for Geolife. The location of each user was updated every time interval. All clients used the same DNN model for each simulation run; the DNN model of each client was not shared at the edge server, because in a real scenario the model could be personalized and is likely to be different, thus by default

Table 9: Accuracy of edge server prediction (%). The number inside parenthesis indicates mean absolute error (m).

Dataset	**Markov**		**SVR**		**RNN**	
	top-1	top-2	top-1	top-2	top-1	top-2
KAIST	4.6	44.4	8.1 (12.9)	54.1 (12.9)	9.2 (12.4)	54.6 (12.4)
Geolife	15.0	32.0	38.1 (31.4)	59.6 (31.4)	36.9 (32.1)	58.1 (32.1)

not sharable across different clients. Time to perform DNN partitioning and mobility prediction was ignored, since it is negligible compared to DNN execution time. As I could not prepare hundreds of real edge servers, I profiled the execution time of DNN layers at the server and at the client in advance using *caffe* [44], and performed the simulation using those execution profiles.

Accuracy of Mobility Prediction Algorithms

We compared the accuracy of the mobility prediction algorithms (Markov, SVR, RNN) explained in section 3.4. When calculating the accuracy, we only counted *non-futile* predictions, i.e., predictions made just before when a client moves to another server, because futile predictions are useless for proactive migration. For Markov, the top-n accuracy means that the prediction is correct when the client visits one of n highest probable servers. For SVR and RNN, the prediction is correct when the client visits one of n closest servers from the predicted location. We considered top-2 accuracy as well as top-1, because most of top-1 result was the current server, i.e., the client was predicted to stay in the same edge server. Top-2 results always include a server other than the current one, thus more suitable for evaluating the edge server prediction for proactive migration.

Table 9 summarizes the result. Markov shows definitely lower top-1 and top-2 accuracy than SVR and RNN in both datasets. This is because Markov predictor loses the exact location information of clients when mapping from x, y coordinates to a

discrete edge server identifier. SVR and RNN showed similar accuracy and MAE in both datasets. We tested various RNN models, but the best configuration for minimum MAE required just a single LSTM cell (with an output dimension of 32 for KAIST and 16 for Geolife), which is much simpler than RNNs for difficult tasks such as speech recognition [87]. The result implies that location prediction with a short-term trajectory does not require such a complex RNN model, or our training data was not sufficient enough for training. Since linear SVR showed an accuracy similar to RNN and was faster than RNN in terms of both training and testing, we decided to use it in the simulation.

Simulation Result

To evaluate the impact of proactive migration on DNN execution performance, I measured the total number of executed DNN queries and hit ratios (the ratio of *hit* cases over the sum of *hit* and *miss* cases) while all clients traverse their trajectories. For baseline (IONN [43]), the clients incrementally upload DNN layers from scratch whenever connecting to a different edge server (hit ratio = 0%). PerDNN predicts the next location of a client and proactively migrates DNN layers to all edge servers within a certain distance (r meters) from the predicted location; edge servers keep those layers for five time intervals (TTL=5). *Optimal* is when all DNN layers are always available in all edge servers, so DNN queries are always executed in full speed (hit ratio = 100 %). I only measured the number of queries executed for a time interval right after a client connects to a new edge server, i.e., whenever a cold start occurs, which are our optimization targets.

Fig. 26 shows the result. The hit ratio of the system was 37% (r=50) and 90% (r=100) in KAIST and 43% (r=50) and 70% (r=100) in Geolife; the hit ratio is proportional to the increase of query counts from the *baseline*. Larger r means edge servers further away from the predicted location can receive DNN layers, thus increasing the hit ratio. The result of KAIST with r=100 is highly promising, because it means that I can remove

90% of cold starts. In Geolife, the hit ratio was relatively low even with r=100 (70%), because users in Geolife moved much faster than them in KAIST, hindering accurate user location prediction; the average speed of users in KAIST and Geolife was ∼0.5 m/s and ∼3.9 m/s, respectively. Since Geolife dataset was collected from different modes of transportation, I anticipate that the hit ratio of Geolife can be improved with advanced prediction techniques such as transportation mode inference [84].

MobileNet showed a small increase in query counts, because it is a tiny model that can be quickly uploaded even in the wireless network; the difference between the number of executed queries of *baseline* and that of *optimal*, i.e., *optimizable queries*, was small (∼3.5k). *Inception* and *ResNet* have much more *optimizable queries* than *MobileNet* (∼40k and ∼64k, respectively), such that the increase in executed queries was clearly visible. This indicates that PerDNN is more effective for large DNN models with high deployment overheads.

Backhaul Traffics

The cost of proactive migration is the backhaul traffics for migrating DNN layers. I measured the backhaul traffics of each edge server for each time interval in two directions: *uplink traffic*, the sum of all data traffics sent from the server within the time interval, and *downlink traffic*, the sum of all data traffics coming into the server within the time interval. When clients were using *Inception*, the peak uplink/downlink traffic of the most crowded server was 616/205 Mbps in KAIST and 667/359 Mbps in Geolife; at that time, the server was transmitting DNN models simultaneously to 13 clients and 18 clients, respectively. That amount of traffics is beyond the capacity of typical wireless broadband, so a wired network system such as fiber-optic cables would be needed for connecting the crowded server with nearby servers. Fortunately, I found that a relatively small number of servers require such a high traffic (60∼70% of the servers needed less than 100 Mbps uplink/donwnload traffics), so it would be possible to build the PerDNN system on the hybrid network where most edge servers

are connected via wireless broadband and some crowded ones are connected via wired links.

Fractional Migration

Although I found that the number of crowded servers are relatively small in the previous section, setting up wired connections for those servers would require significant costs. Based on the observation in Section 4.3.1 (DNN execution performance can be highly improved even if I proactively migrate a fraction of a DNN model), I explored the possibility to reduce the peak backhaul traffics of crowded servers by migrating only a fraction of a DNN model, instead of the whole model.

I chose the top 5∼7% of the most crowded edge servers (24 servers in KAIST and 86 servers in Geolife) based on the uplink traffics at the peak time and ran a simulation where the chosen servers transmit (or receive) only a fraction of the server-side layers; the rest of the servers transmit (or receive) the whole DNN layers as usual. The migrated layers for the crowded servers were selected based on the highest-efficiency-first rule, same as Section 4.3.1. Fig. 27 shows the result in the *KAIST* dataset. For *Inception*, I could reduce 67% of the peak uplink traffic (616→206 Mbps) by sacrificing only 2% of the executed queries (when transmitting 43 MB of layers instead of the whole layers). For *ResNet*, the peak uplink traffic decreased by 43% (469→268 Mbps) in return for 1% reduction of executed queries when transmitting 56 MB of layers. These results imply that the high backhaul traffics of a few crowded servers can be greatly reduced by fractional migration, with little loss in performance.

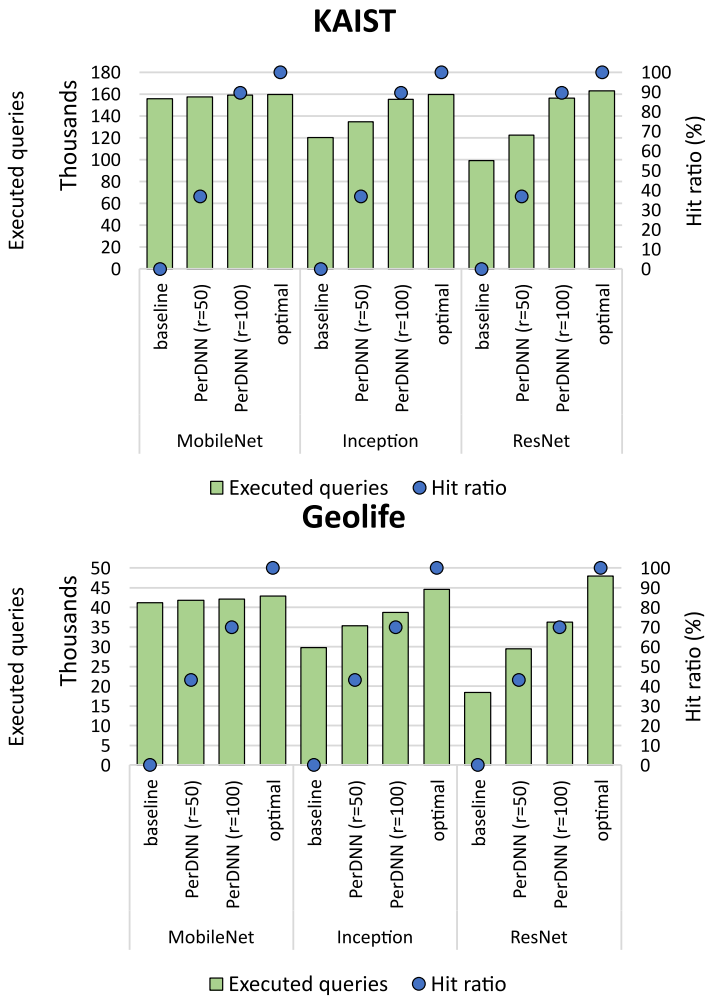


Figure 26: Number of executed queries and hit ratios during simulation.

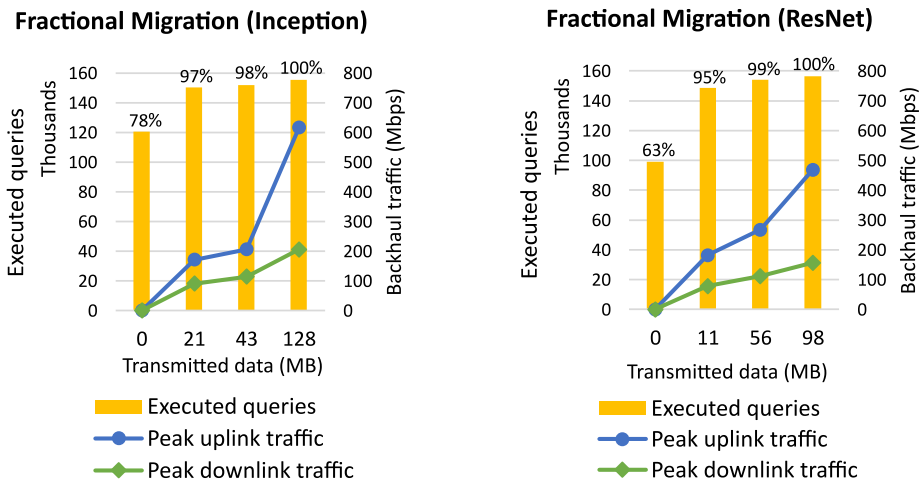


Figure 27: Impact of fractional migration on peak backhaul traffics and execution performance.

Chapter 5. Related Works

Our mobile web worker system stems from a concept of *cyber foraging*, which offloads computations of resource-constrained clients to surrogate machines (cloudlet) [75]. M. Satyanarayanan et al. proposed a technique to deploy a custom service software to any nearby cloudlet by encapsulating the software with VM [77]. Ha et al. improved the performance of VM deployment for rapid provisioning of cloudlet [25]. In the context of edge computing, Ha et al. proposed agile handoff of service by migrating a VM instance between edge servers [27]. Recently, edge computing communities started to focus on using container (such as Docker) to encapsulate the offloaded program rather than VM. Lele Ma et al. proposed a technique to accelerate container migration between edge servers via sharing of container image layers between Docker hosts [57]. The above approaches inevitably involve the overhead of migrating system states such as OS or language runtime. Such overheads slow down the service migration between edge servers, so a more optimized, specialized solution can be used for popular platforms, such as web or machine learning. Our system is one of those solutions specialized for web apps.

MAUI is an early work in partitioning-based offloading, which partitions an application code and offloads the execution of computation-intensive parts to a remote server [12]. Many researchers adopted the partitioning scheme and improved programmability [9], parallelism [49], and stream data processing [97]. We avoided complex issues faced by the previous studies by exploiting the characteristics of web apps. Web workers are intended to run long-running, compute-intensive codes in web apps, so we offload web workers in a regular web app, rather than explicitly partition the app code [12] [9] or use annotations [68]. Also, web workers communicate with the main thread only in a message passing manner, so we need not consider the synchronization of shared data [22] [23]. Finally, once the worker is migrated, the main thread can offload computations by only sending the input data, while some approaches migrate unnecessary app

states whenever offloading computations [12] [9] [41].

Pelin Angin et al. proposed an agent-based offloading system, which offloads autonomous application modules, each of which is packed in a *mobile agent*, to the cloud server [3]. Our mobile worker system can be viewed as a web version of the agent-based offloading. However, this dissertation deals with how to efficiently migrate the state of HTML5 web worker, while Pelin Angin et al. did not concern such issues and relied on an existing Java mobile agent framework named *JADE* [6]. *JADE* heavily depends on programmers, requiring them to design apps based on their application model (agent-behavior) [6]. To port an existing app into a *JADE* app, developers have to rewrite their apps from scratch based on the *JADE* application model. On the other hand, our system automatically migrates web workers in normal web apps, so our system can be easily adapted to existing web apps.

There have been a few researches on computation offloading for web platforms. S. Park et al. proposed an offloading framework for web-centric devices, which offloads the execution of JS functions to a remote server relying on programmers' annotations to designate migrated states [68]. Their work offloads the execution of the main thread of a web app, so user interaction is blocked until the server returns the result. Our system offloads the execution of web workers, hence does not block the main thread execution during offloading. M. Zbierski proposed transparent offloading of web workers by dispatching web workers in the cloud server [102]. However, their work does not support subsequent migration of web workers, meaning that the fallback or handoff of a worker is unavailable. *Imagen* [55] and *ThingsMigrate* [18] support offloading of stateful JS apps without modifying the web platform by instrumenting the app codes to collect the runtime state needed for migration (e.g., closure). The instrumented codes are significantly slower than the original app codes (up to 40% in the micro-benchmark program [18]), so we take another approach that does not require app instrumentation but uses a modified web platform to capture some JS states such as closure [64] [52], obviating performance degradation from running instrumentation codes. Lastly, our

work is different from all of the above studies on the web-based offloading in that they only deal with offloading of JS code, excluding wasm code.

IONN is related to the work of Lei Yang et al. [99], which partitions app execution between a client and a server for reducing the latency of mobile cloud applications under multi-user environments. They model an application as a sequence of modules and determines where to execute each module (client or server) by solving a recursive formula. The shortest path algorithm used in IONN is similar to the recursive algorithm proposed in [99], although IONN treats DNN structure which is more complex than the sequence of modules, requiring handling of multiple paths (section 5.3). Also, [99] does not consider the uploading overhead.

Recently, DNN-focused offloading approaches have been studied. *MCDNN* offloads DNN execution for streaming data in multi-programming environments [30]. MCDNN creates variants of a DNN model and chooses DNN models among them for a given task to satisfy resource/cost constraints with maximal accuracy. MCDNN assumes the same DNN models are pre-installed at the client and the server, which is different from our edge scenario that uploads DNN models at runtime. Also, MCDNN does not focus on DNN partitioning. *NeuroSurgeon* [47] is the first work on DNN partitioning. However, it allows only fixed, two-way partitioning (front layers by the client and rear layers by the server), while IONN can make a more flexible partitioning (e.g., front layers by the client, middle by the server, and rear by the client), depending on the computation power of server/client and the transmission overhead of layers/feature data. As MCDNN, NeuroSurgeon also requires the server to store the DNN model in advance, which is different from IONN.

Machine learning researchers are actively developing techniques that reduce the amount of computation and the size of DNN models to run DNNs on mobile devices. MobileNet [34] decreases the amount of convolution computation by replacing point-wise convolution to depth-wise separable convolution. SqueezeNet [37] is a small DNN ($\sim$4.8 MB) designed to have few model parameters with an accuracy similar to AlexNet

(∼233 MB). Applied with DNN compression techniques [29], the size of SqueezeNet drops to 0.47 MB [37]. If a mobile user wants to offload the execution of such a tiny DNN, the benefit of IONN would be insignificant, because the whole DNN model might be uploaded soon. However, DNNs for complex tasks still have complicated structure and a large model size. For example, SENet [36], one of the winners of ILSVRC 2017, consists of more than 900 layers and has a model size of ∼440 MB. Also, emerging end-to-end DNN architecture, which performs an entire process to solve a cognitive [60] or generative task [13] [93] in a single DNN, might lead to the increase of the DNN model size. It would be difficult to run such a large DNN model on mobile devices. IONN is a feasible solution for offloading large, complex DNNs in mobile applications, because incremental offloading results in high performance benefits even during the uploading stage of DNN models.

To our knowledge, NeuroSurgeon [47] is the first work on the layer-level partitioning of DNN, which finds a partitioning point in a heuristic way. Chuang Hu et al. proposed a partitioning algorithm applicable to DAG-formed DNNs based on the min-cut algorithm [35]. Unlike the above studies, PerDNN uses runtime GPU information for DNN partitioning to cope with congestion in a server. Also, most of the above studies assume DNN models are already saved at the server (either cloud or edge), not focusing on how to deploy DNN models to offloading servers. PerDNN addresses the cold start issue, thus complementing the above studies by helping them to quickly establish optimal offloading status.

Follow-Me cloud migrates cloud-based services between data centers in a federated cloud, to allow mobile users to always be connected with the optimal data center and data anchor [91]. Follow-Me cloud makes a migration decision based on the distance between the client and the data center, while PerDNN migrates DNN layers based on expected execution latency. Also, Follow-Me cloud performs live migration of the whole VM instance, whereas PerDNN allows migrating a fraction of a DNN model to reduce backhaul traffics.

PerDNN's proactive DNN migration resembles the concept of proactive caching (aka femto caching), which reduces data traffics to central clouds by saving popular files at the edge servers in advance [79] [5]. The major difference between proactive caching and PerDNN is that proactive caching saves data at the client far before the client visits the server, but PerDNN transmits DNN layers between edge servers in real time. Also, unlike typical file caching based on user's file popularity, PerDNN selects DNN layers to be offloaded using a partitioning algorithm based on the current runtime states.

Chapter 6. Conclusion

In this thesis, I propose an offloading system that supports seamless offloading of web app computations in the edge cloud environment. The proposed system migrates HTML5 web worker across the client, the edge, and the cloud, and maintains the offloading states while the mobile client changes its target server. Web workers are migrated using *snapshot*, a JavaScript code that restores the runtime state of a web worker when executed. I addressed issues on how to generate a snapshot code that restores not only JavaScript objects but also native data such as built-in objects and *webassembly* functions. Experimental result showed the mobile worker migration is lightweight (a few seconds for non-trivial apps and within 150 ms for a simple image processing app), and offloading of wasm code is much faster than offloading of JavaScript code (up to 8.4x).

In the context of DNN applications, I propose IONN, a novel DNN offloading technique for edge computing. IONN partitions the DNN layers and incrementally uploads the partitions to allow collaborative execution by the client and the edge server even before the entire DNN model is uploaded. Experimental results show that IONN improves both the query performance and the energy consumption during DNN model uploading, compared to a simple all-at-once approach.

I also propose PerDNN to address an issue of performance degradation at cold start, when DNN layers are dynamically deployed to a new edge server. PerDNN tackles the issue with proactive migration of DNN layers based on real-time mobility prediction. Also, I demonstrated the impact of the server's workload on the DNN execution time and proposed a GPU-aware estimation model for layer execution time of edge servers. In the simulation with real world trace datasets and execution profile of real hardware, PerDNN removed up to 90% and 70% of cold starts in KAIST and Geolife, respectively, and achieved 1.6$\sim$2.0x throughput improvement, compared to a baseline with no proactive migration. The backhaul traffics of the system could be

sharply reduced with negligible performance loss, by sending only a small fraction of a DNN model.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles, techniques. *Addison Wesley*, 7(8):9, 1986.

[2] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese. Social lstm: Human trajectory prediction in crowded spaces. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 961–971, June 2016. doi: 10.1109/CVPR.2016.110.

[3] P. Angin and B. K. Bhargava. An agent-based optimization framework for mobile-cloud computing. *JoWUA*, 4:1–17, 2013.

[4] A. API. Ibm alchemy api, 2009. URL `https://www.ibm.com/watson/alchemy-api.html`.

[5] E. Bastug, M. Bennis, and M. Debbah. Living on the edge: The role of proactive caching in 5g wireless networks. *IEEE Communications Magazine*, 52(8):82–89, Aug 2014. ISSN 0163-6804. doi: 10.1109/MCOM.2014.6871674.

[6] G. P. A. R. G. Bellifemine, F Caire. Jade: a white paper. 2003.

[7] F. Bonomi, R. Milito, J. Zhu, and S. 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*, pages 13–16. ACM, 2012.

[8] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[9] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[10] C. Cicconetti, M. Conti, and A. Passarella. Low-latency distributed computation offloading for pervasive environments. In *Pervasive Computing and Communications (PerCom), 2019 IEEE International Conference on. IEEE*, 2019.

[11] M. C. Claudio Cicconetti and A. Passarella. low-latency distributed computation offloading for pervasive environments. In *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 262–271, 2019.

[12] E. Cuervo, A. Balasubramanian, D.-k. Cho, 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*, pages 49–62. ACM, 2010.

[13] P. Dou, S. K. Shah, and I. A. Kakadiaris. End-to-end 3d face reconstruction with deep neural networks. *CoRR*, abs/1704.05020, 2017. URL `http://`

arxiv.org/abs/1704.05020.

[14] W. dsp. https://github.com/shamadee/web-dsp, 2016. URL https://github.com/shamadee/web-dsp.

[15] M. A. et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. URL http://arxiv.org/abs/1603.04467.

[16] J. Feng, Y. Li, C. Zhang, F. Sun, F. Meng, A. Guo, and D. Jin. Deepmove: Predicting human mobility with attentional recurrent networks. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1459–1468. International World Wide Web Conferences Steering Committee, 2018.

[17] H. Games. A list of html5/javascript game engines, 2019. URL https://html5gameengine.com/.

[18] J. Gascon-Samson, K. Jung, S. Goyal, A. Rezaiean-Asel, and K. Pattabiraman. Thingsmigrate: Platform-independent migration of stateful javascript iot applications. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[19] M. T. Goodrich. Ics 161: Design and analysis of algorithms lecture notes, 1996. URL https://www.ics.uci.edu/~eppstein/161/960208.html.

[20] Google. Google cloud platform, 2008. URL https://cloud.google.com.

[21] Google. Google stadia, 2019. URL https://stadia.dev/.

[22] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. {COMET}: Code offload by migrating execution transparently. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 93–106, 2012.

[23] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, pages 137–150, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3494-5. doi: 10.1145/2742647.2742649.

[24] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[25] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 153–166. ACM, 2013.

[26] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM,

2014.

[27] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan. You can teach elephants to dance: Agile vm handoff for edge computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 12:1–12:14, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5087-7. doi: 10.1145/3132211.3134453. URL http://doi.acm.org/10.1145/3132211.3134453.

[28] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062363. URL http://doi.acm.org/10.1145/3062341.3062363.

[29] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015. URL http://arxiv.org/abs/1510.00149.

[30] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136. ACM, 2016.

[31] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[32] D. Herrera, H. Chen, and E. Lavoie. Webassembly and javascript challenge : Numerical program performance using modern browser technologies and devices. 2018.

[33] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9 (8):1735–1780, Nov. 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735.

[34] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, 2017.

[35] C. Hu, W. Bao, D. Wang, and F. Liu. Dynamic adaptive dnn surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1423–1431, 2019.

[36] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *CoRR*, abs/1709.01507, 2017. URL http://arxiv.org/abs/1709.01507.

[37] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016. URL http://arxiv.org/abs/

1602.07360.

[38] IEEE. Ieee 1934. openfog reference architecture, 2019. URL https://standards.ieee.org/standard/1934-2018.html.

[39] S. G. Index. Average internet speed of united states, 2019. URL https://www.speedtest.net/global-index/united-states#fixed.

[40] P. Jacquet, W. Szpankowski, and I. Apostol. A universal predictor based on pattern matching. *IEEE Transactions on Information Theory*, 48(6):1462–1472, 2002.

[41] H. Jeong and S. Moon. Offloading of web application computations: A snapshot-based approach. In *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pages 90–97, Oct 2015. doi: 10.1109/EUC.2015.10.

[42] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon. Computation offloading for machine learning web apps in the edge server environment. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1492–1499. IEEE, 2018.

[43] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 401–411, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6011-1. doi: 10.1145/3267809.3267828.

[44] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[45] R. Jiang, X. Song, Z. Fan, T. Xia, Q. Chen, S. Miyazawa, and R. Shibasaki. Deepurbanmomentum: An online deep-learning system for short-term urban mobility prediction. In *AAAI*, 2018.

[46] S. js. https://sugarjs.com/, 2013. URL https://sugarjs.com/.

[47] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 615–629. ACM, 2017.

[48] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[49] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[50] kripken. Ammo.js github repository, 2019. URL https://github.com/kripken/ammo.js/.

[51] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[52] J.-w. Kwon and S.-M. Moon. Web application migration with closure reconstruction. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 133–142, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-4913-0. doi: 10.1145/3038912.3052572.

[53] Y. Kwon, S. Lee, H. Yi, D. Kwon, S. Yang, B.-G. Chun, L. Huang, P. Maniatis, M. Naik, and Y. Paek. Mantis: Automatic performance prediction for smartphone applications. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 297–308, 2013.

[54] H. Liu, F. Eldarrat, H. Alqahtani, A. Reznik, X. de Foy, and Y. Zhang. Mobile edge cloud system: Architectures, challenges, and approaches. *IEEE Systems Journal*, 12(3):2495–2508, Sept 2018. ISSN 1932-8184. doi: 10.1109/JSYST.2017.2654119.

[55] J. T. K. Lo, E. Wohlstadter, and A. Mesbah. Imagen: Runtime migration of browser sessions for javascript web applications. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 815–826, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2035-1. doi: 10.1145/2488388.2488459. URL `http://doi.acm.org/10.1145/2488388.2488459`.

[56] F. Lobillo, Z. Becvar, M. A. Puente, P. Mach, F. L. Presti, F. Gambetti, M. Goldhamer, J. Vidal, A. K. Widiawan, and E. Calvanesse. An architecture for mobile computation offloading on cloud-enabled lte small cells. In *2014 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 1–6, April 2014. doi: 10.1109/WCNCW.2014.6934851.

[57] L. Ma, S. Yi, and Q. Li. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 11:1–11:13, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5087-7. doi: 10.1145/3132211.3134460. URL `http://doi.acm.org/10.1145/3132211.3134460`.

[58] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis. Live service migration in mobile edge clouds. *Wireless Commun.*, 25(1):140–147, Feb. 2018. ISSN 1536-1284. doi: 10.1109/MWC.2017.1700011.

[59] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis. Live service migration in mobile edge clouds. *IEEE Wireless Communications*, 25(1):140–147, February 2018. ISSN 1536-1284. doi: 10.1109/MWC.2017.1700011.

[60] Y. Miao, M. Gowayyed, and F. Metze. EESEN: end-to-end speech recognition

using deep RNN models and wfst-based decoding. *CoRR*, abs/1507.08240, 2015. URL http://arxiv.org/abs/1507.08240.

[61] A. J. Nicholson and B. D. Noble. Breadcrumbs: Forecasting mobile connectivity. In *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*, MobiCom '08, pages 46–57, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-096-8. doi: 10.1145/1409944.1409952.

[62] NVIDIA. Nvidia tegra, 2008. URL https://www.nvidia.com/object/tegra.html.

[63] NVIDIA. Nvidia tensorrt inference server, 2018. URL https://docs.nvidia.com/deeplearning/sdk/tensorrt-inference-server-guide/docs/index.html.

[64] J. Oh, J.-w. Kwon, H. Park, and S.-M. Moon. Migration of web applications with seamless execution. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 173–185, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3450-1. doi: 10.1145/2731186.2731197.

[65] OpenCV. https://docs.opencv.org/3.4/index.html, 2018. URL https://docs.opencv.org/3.4/index.html.

[66] H. Packard. Hp haven, 2013. URL https://www.havenondemand.com.

[67] C. Pahl and B. Lee. Containers and clusters for edge cloud architectures – a technology review. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 379–386, Aug 2015. doi: 10.1109/FiCloud.2015.35.

[68] S. Park, Q. Chen, H. Han, and H. Y. Yeom. Design and evaluation of mobile offloading system for web-centric devices. *Journal of Network and Computer Applications*, 40:105 – 115, 2014. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2013.08.006.

[69] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[70] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[71] I. Rhee, M. Shin, S. Hong, K. Lee, S. Kim, and S. Chong. CRAWDAD dataset ncsu/mobilitymodels (v. 2009-07-23), July 2009.

[72] M. Roberts and J. Chapin. *What is Serverless? (Chapter 4. Limitations of Serverless)*. O'Reilly Media, Incorporated, 2017. URL https://www.oreilly.com/library/view/what-is-serverless/9781491984178/ch04.html.

[73] D. Ron, Y. Singer, and N. Tishby. Learning probabilistic automata with variable memory length. In *Proceedings of the seventh annual conference on Computa-*

tional learning theory, pages 35–46. ACM, 1994.

[74] R. Roy and V. Bommakanti. Odroid xu4 user manual, 2017. URL https://magazine.odroid.com/odroid-xu4.

[75] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8(4):10–17, 2001.

[76] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[77] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4), 2009.

[78] scikit learn. Scikit-learn machine learning library, 2019. URL https://scikit-learn.org.

[79] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire. Femtocaching: Wireless content delivery through distributed caching helpers. *IEEE Transactions on Information Theory*, 59(12):8402–8413, Dec 2013. ISSN 0018-9448. doi: 10.1109/TIT.2013.2281606.

[80] K.-Y. Shin, H.-J. Jeong, and S.-M. Moon. Enhanced partitioning of dnn layers for uploading from mobile devices to edge servers. In *Proceedings of the 3rd international workshop on embedded and mobile deep learning*, pages –, 2019. in press.

[81] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587): 484–489, 2016.

[82] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004.

[83] L. Song, D. Kotz, R. Jain, and X. He. Evaluating location predictors with extensive wi-fi mobility data. In *IEEE INFOCOM 2004*, volume 2, pages 1414–1424 vol.2, March 2004. doi: 10.1109/INFCOM.2004.1357026.

[84] X. Song, H. Kanasugi, and R. Shibasaki. Deeptransport: Prediction and simulation of human mobility and transportation mode at a citywide level. In *IJCAI*, volume 16, pages 2618–2624, 2016.

[85] SPEEDTEST. Speedtest global index, 2019. URL https://www.speedtest.net/global-index.

[86] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press. URL http://dl.acm.org/citation.cfm?id=2969033.2969173.

[87] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with

neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[88] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[89] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567.

[90] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CoRR*, 2015.

[91] T. Taleb and A. Ksentini. Follow me cloud: interworking federated clouds and distributed mobile networks. *IEEE Network*, 27(5):12–19, 2013.

[92] K. Tian, Y. Dong, and D. Cowperthwaite. A full {GPU} virtualization solution with mediated pass-through. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 121–132, 2014.

[93] A. T. Tran, T. Hassner, I. Masi, and G. G. Medioni. Regressing robust and discriminative 3d morphable models with a very deep neural network. *CoRR*, abs/1612.04904, 2016. URL http://arxiv.org/abs/1612.04904.

[94] WiGLE. Wigle wi-fi database, 2018. URL https://wigle.net/index.

[95] Wikipedia. Long range wi-fi, 2018. URL https://en.wikipedia.org/wiki/Long-rangeWi-Fi.

[96] W. workers. Html5 web worker specification, 2018. URL https://html.spec.whatwg.org/multipage/workers.html#workers.

[97] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *SIGMETRICS Perform. Eval. Rev.*, 40(4):23–32, Apr. 2013. ISSN 0163-5999. doi: 10.1145/2479942.2479946. URL http://doi.acm.org/10.1145/2479942.2479946.

[98] L. Yang, J. Cao, S. Tang, D. Han, and N. Suri. Run time application repartitioning in dynamic mobile cloud environments. *IEEE Transactions on Cloud Computing*, 4(3):336–348, 2014.

[99] L. Yang, J. Cao, H. Cheng, and Y. Ji. Multi-user computation partitioning for latency sensitive mobile cloud applications. *IEEE Transactions on Computers*, 64(8):2253–2266, 2015.

[100] L. Yang, J. Cao, S. Tang, D. Han, and N. Suri. Run time application repartitioning in dynamic mobile cloud environments. *IEEE Transactions on Cloud Computing*, 4(3):336–348, July 2016. ISSN 2168-7161. doi: 10.1109/TCC.2014.2358239.

[101] A. Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the*

ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048224.

[102] M. Zbierski and P. Makosiej. Bring the cloud to your mobile: Transparent offloading of html5 web workers. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 198–203, Dec 2014. doi: 10.1109/CloudCom.2014.60.

[103] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang. G-net: Effective {GPU} sharing in {NFV} systems. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 187–200, 2018.

[104] Y. Zheng, X. Xie, and W.-Y. Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data(base) Engineering Bulletin*, June 2010.