Enhancing Mobile Devices through Code Offload

by

Eduardo Cuervo

Department of Computer Science
Duke University

Date: ____________________________
Approved:

____
Landon P. Cox, Supervisor

____
Bruce M. Maggs

____
Jeffrey S. Chase

____
Romit R. Choudhury

____
Alec Wolman

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012

ABSTRACT

Enhancing Mobile Devices through Code Offload

by

Eduardo Cuervo

Department of Computer Science
Duke University

Date: ____________________
Approved:

____
Landon P. Cox, Supervisor

____
Bruce M. Maggs

____
Jeffrey S. Chase

____
Romit R. Choudhury

____
Alec Wolman

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012

Abstract

Advances in mobile hardware and operating systems have made mobile a first-class development platform. Activities such as web browsing, casual game play, media playback, and document reading are now as common on mobile devices as on full-sized desktop systems. However, developers are still constrained by the inherent resource limitations of mobile devices. Unlike desktop systems, mobile devices must sacrifice performance to accomodate smaller form factors and battery-backed operation. Opportunistic offloading of computation from a mobile device to remote server infrastructure (i.e., ”code offload”) offers a promising way to overcome these constraints and to expand the set of applications (i.e., ”apps”) that can run on devices.

Deciding to offload requires a careful consideration of the costs and benefits of a range of possible program partitions. This cost-benefit analysis depends on external factors, such as network conditions and the resources availability, as well as internal app properties, such as component dependencies, data representations, and code complexity. Thus, benefiting from offload requires some assistance from developers, but requiring developers to adopt arcane or unnatural programming models will hinder adoption of regardless of the potential benefits.

In this dissertation I characterize two frameworks that reduce the amount of developer effort required to improve the performance of mobile apps through code offload. The first, *MAUI*, is designed for computationally intensive general-purpose apps such as speech and facial recognition. The second, *Kahawai*, is designed for graphics-intensive apps like fast-

action video games.

MAUI continuously monitors the device, network, and app, and uses its measurements to compute an energy-efficient program partition. MAUI reduces the burden on developers by taking advantage of core features of the managed code environments common to mobile platforms: code portability, serialization, reflection, and type safety. These features allows MAUI to automatically instrument and potentially offload methods that the developer has tagged as suitable for offload. MAUI is particularly effective on applications composed by operations whose computational cost is large compared to the transfer cost of their input parameters and their output results.

Kahawai is designed for graphics-intensive apps such as console-style games and takes advantage of two features of today's mobile gaming platforms: capable mobile GPUs and reusable game engines. Even though today's mobile devices cannot duplicate the sophisticated graphical detail provided by gaming consoles and high-end desktop GPUs, devices have seen rapid improvements in their GPU processing capabilities. Kahawai leverages a device's GPU to provide *collaborative rendering*. Collaborative rendering relies on a mobile GPU to generate low-fidelity output, which when combined with server-side GPU output allows a mobile device to display a high-fidelity result. The benefits of collaborative rendering are substantial: mobile clients can experience high-quality graphical output using relatively little bandwidth. Fortunately, because most modern games are built on top of reusable game engines, developers only have to identify the sources of non-determinism in the game logic to take advantage collaborative rendering.

Together, MAUI and Kahawai demonstrate that code offload can provide substantial benefits for mobile apps without overburdening app developers.

To my father and my grandmother who would have loved to read this, to the rest of my family for their unconditional support, and to my friends for being there for me all this time.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

Glossary of terms

The terminology used throughout this document follows that given in [4]. Terms not found in the referred document are listed in the following glossary

Property Specific	Property Specific Offload is the set of techniques presented in this dissertation. It allows two large classes of applications to be offloaded easily, only requiring the developers to identify a small set of properties about their applications.
Code offload	Opportunistic offloading of computation from a mobile device to remote server infrastructure.
Delta	An incremental change in the application state. In MAUI it represents the difference in the state before and after performing an offload operation. In Kahawai it represents the pixel-wise difference of the raster output of two render operations performed at different video quality settings.
Entity	In Kahawai, an entity refers to the 3D representation of a logical object in the game. For example an enemy, a door, a box or a wall. As well as light sources and cameras.
Game Engine	The primary form of code reuse in game development. Game engines typically separate game content from the core functionality of the game execution.
High detail config	In Kahawai, a high detail configuration represents an optimal video configuration that represents the best quality output possible for a given game.
H.264/AVC	Also known as MPEG-4 AVC and informally called H.264, it is the state of the art standard for video compression.

idTech 4	idTech 4 is the fourth iteration of id Software's well known game engine. It is also known as the Doom 3 engine, after the game for which it was originally built.
I-Frame	In video compression, an I-Frame is a frame that is encoded without using information coming from any other frame.
Kahawai	Kahawai is an offload system for graphic intensive applications such as games and 3D visualization applications.
Low detail config	In Kahawai, a low detail configuration is a video configuration used at the client. Usually presenting lower quality output than the high detail configuration, but presenting a low enough complexity for the mobile device to handle.
MAUI	MAUI stands for Mobile Assistance Using Infrastructure. MAUI is an offload system for general purpose computationally-intensive applications.
P-Frame	A predictive frame. P-Frames are produced using reference frames (I-frames or other P-frames) as a source, providing the necessary information necessary to reconstruct a frame using its source reference frames.
Remoteable	In MAUI, a remoteable method is considered suitable for offload because it does not make use of sensors, presents output to the user, or results in external side-effects.
Scene	In Kahawai, a scene is the set of graphical entities formed by objects, light sources and cameras and its particular attributes defined by the video settings.
Scene composition	In Kahawai, scene composition refers to the arrangement of entities and their abstract basic properties such as position, basic shape, and average color.
Transitive closure	When serializing, the transitive closure of a given object, is the set of all objects that it depends upon.

Acknowledgements

By writing this dissertation I finish a 6 year challenging and enjoyable journey. While working my way through it, I was both influenced and supported by wonderful people, without whom it would have been impossible to overcome the many challenges faced.

First, I would like to thank Landon Cox, my advisor, for having faith in me and supporting me all along this journey, keeping me on track and fighting together through all adversities. Through him I met a lot of wonderful people that also influenced me deeply. We shared both the moments of joy and frustration that come with working through a dissertation, and made it to the end.

I am also very thankful to my committee members, Romit Choudhury, Jeff Chase, Bruce Maggs, and Alec Wolman. I consider myself lucky for having such a strong and well balanced committee. I had the privilege to meet and discuss my research with them, always receiving very valuable feedback, closely related to their area of expertise.

From my committee I am particularly grateful to Alec, who from the first day I had the pleasure to meet him at MSR, became more than an internship mentor. Alec followed my career closely and took interest in my development. Alec influenced my research direction, my oral and written skills and was always there to provide advice, and of course, to share a couple of tequila shots.

I also thank my labmates: Amre Shakimov, Ryan Scudellari, Bi Wu, Dongtao Liu, Linda Deng and specially Peter Gilbert. They are the people that I can best relate to in this journey, as we traveled most of it together. We were a great group and had fun working together. Peter, my xenbr0, deserves a special mention. We started and finished at about the same time. He has not only been an incredible coding partner, but a true friend. It has been a pleasure to share an office with him for the last five years, to share success and failure, just the way we shared lunch and coffee. His valuable advice kept me on track many times, and his influence on me has been tremendous.

I would also like to thank the great people at Microsoft Research. I will never forget the MAUI team in 2009: Alec, Stefan Saroiu, Ranveer Chandra, Victor Bahl, Aruna Balasubramanian and Dae-ki Cho. Working with them still remains as one of the dearest memories of my grad school years. All of them were, and are still great influences to my work, my career , always supporting me through advice, recommendation letters, and of course the obligated tequila session whenever we meet.

Aside from the MAUI people, there is more great people at MSR that helped me get through both my internships: James Mickens, who was super fun to be with. He hosted me for a few nights in a mindblowing experience full of metal and awesomeness, and was the kind reviewer of my research statement. I would also like to mention Dinei Florencio, who helped me get started with video compression; and my fellow interns: Shravan Rayanchu, Cindy Rubio, Ruy Cervantes, Ang Li, Patrick Colp, Xuan Bao and Xin for all the fun times we had in the Northwest.

I would also like to thank the nice people and interns from Nokia: Zhigang Liu, Paul James, Sonia Kim, Tao Jin and Bo Xing. Working at Nokia was a very fun experience that I remember fondly.

I would like to make a special mention to my former mentors Barbaro Ferro, Lorrie Cranor and Greg Kesden, it is thanks to them that I decided to get into grad school. I learned so much from them and feel lucky to have had them as mentors.

I am also thankful to my friends in Durham, that helped me stay happy during this 6 years. I thank everyone I met and shared a good time with. My friends in the good and the bad, from the beginning until the end. The Computer Scientists: Hero, the best party partner; Jeff, “El Jefe” Martin, Mike “The daily troll” Sirivianos; my ad-hoc mentor Justin Manweiler and his wife Jane; my sumo opponent Brittany Fasy and her fiancé Dave; Nedyalko “California Baby” Borisov; Souvik Sen and Rolando Estrada. My fellow Dukies: Liz and Firas, always awesome; Laura and Miles, with memories of those wine and cheese gatherings; Maciek “Macieeek” Mazurowski and his wife Kelly; Andreas “Vive Andreaas” Gondikas; Kate Kucharzyk; Genevieve Lipp; my dance partner Halley Brantley; and Stephen Rosenzweig, the beer expert.

I would like to thank the latin people at Duke “pa las que sea”, specially Pablo Gainza and Wendy Gonzalez for keeping me from starving during deadlines, Luis Tobon for helping me with wiring in those critical moments; Paula for being my dance teacher; Melina Altamirano, always knowing what to say; Sandra Ley; Marco Fernandez; Felipe “Pipe Howser” Quiroz; German “Chingon” Forero; Samuel Ramirez; Andrea Carneiro; Andrea Velasquez; and Valerie Tornini for being such great friends.

Although not physically present during the process, my lifelong friends in Mexico, they were always supportive and provided me with a lot of fun and courage during breaks to give me the energy to overtake the challenges ahead. They paved the road that took me here. From elementary school to college, I don’t have enough words to make justice to these guys, but I am forever grateful. My great friends Victor Hernandez; David Teran, Gonzalo Arredondo, Pablo Fernandez, Gabriel Macchia, Carlos Campos, Rodrigo Mazin, Jose Manuel Valledor, Raul Lucero, Roberto (cyberdemon) Grimaldi, Mercedes Senties, Elisa Guadiana and Paty Macchia.

At last, and definitely not least, I would like to thank my beloved family. Those that are still here as well as those that are not anymore. Thanks for being so supportive, and proud of me. Thanks for being models of what is good and right, for providing a shelter when things didn’t go well, and to celebrate when they did. Thanks to my mother Hilda, my father Eduardo, my grandmother Victoria, my brother Luis, my aunt Lisette and my uncle Luis. Their unconditional support and example was essential to finish this dissertation.

1

Introduction

Over the last few years, mobile application development has increased dramatically as a result of improvements in mobile hardware and operating systems. However, application developers are still constrained by the resource limitations of mobile devices. Unlike full-sized desktop computers, mobile devices aggressively sacrifice performance in favor of smaller sizes, lower costs, and power efficiency. Furthermore, the rate at which the hardware components in mobile devices draw power is increasing faster than the amount of power a battery of a given size can store. As a consequence, battery capacity has become the most important limitation when developing resource-intensive applications. Despite this limitation, mobile devices are still capable of performing common tasks like web browsing, media playing, and document reading. One way to overcome mobile device limitations is to augment a mobile device's resources by opportunistically offloading computation to remote servers.

The key challenge that arises when using code offload is reasoning about tradeoffs between energy, network bandwidth, and computational fidelity. This tradeoff changes constantly with dynamic factors such as the type of application, the device's specifications, available network characteristics and user preferences. Therefore, offloading decisions must be informed by the current context in order to produce the best results.

Furthermore, adapting existing applications to take advantage of code offload or, *cyber-foraging*, is not a simple task. Balan first proposed RapidRe[4] as a solution to this problem. RapidRe provides a quick and effective process for retargeting existing applications for code offload. Unfortunately, RapidRe still requires substantial involvement from the developer and restricts itself to *interactive* applications.

However, over the last five years, several trends warrant revisiting some of the assumptions underlying solutions like RapidRe. First, two of modern mobile platforms, Android[35] and Windows Phone[65] use managed languages and runtime environments such as the Dalvik Virtual Machine[34] and the .NET Common Language Runtime[84]. Second, games that require sophisticated graphics have become an important class of applications for mobile devices. Finally, wireless broadband technologies like LTE provide mobile bandwidth that is better than or on par with home cable and DSL connections. However, wireless broadband users are commonly restricted by usage quotas of several gigabytes per month.

Due to the popularity of smartphones and tablets, developers have written hundreds of thousands of mobile applications (i.e., "apps"). Thus, it is critical to make retargeting applications as simple and transparent as possible. Fortunately, managed environments' language and runtime support make fine-grained offload much easier.

The rising importance of mobile games has other implications. Unlike traditional operation-based, interactive applications, modern games are *continuous* in nature. These apps display and continuously update highly detailed graphics that are computationally expensive to generate, even when there is no user input. Offloading these operations requires large amounts of bandwidth, and mobile broadband can support a thin client model of code offload, such as the one used by the commercial service OnLive[75]. However, because of monthly usage limits set on mobile bandwidth, offload must not consume an excessive amount of data.

In the face of these trends, approaches to code offload must simplify the task of re-targeting of both operation-based applications and continuous, graphics-intensive applications. Ideally, developers would only have to identify a small number of key application properties to the offload engine. The engine could then use this input to boost an application's performance, while reducing the amount of power and bandwidth it consumes.

1.1 Property Specific Offload

In this dissertation, I propose a set of techniques for offloading code from a mobile device to remote server infrastructure called *Property Specific Offload*. These techniques allow the developer to take advantage of code offload by identifying a small number of properties of their applications. Propery specific offload is most appropriate for computationally intensive operations and real-time interactive graphics. The goal of my dissertation is to demonstrate that it is possible to characterize the amount of developer effort needed to benefit from offload for these two important classes of mobile applications.

1.2 This thesis

My thesis statement can be stated as follows:

It is possible for mobile developers to obtain significant benefits from code offload by identifying a small number of properties of their applications.

1.3 Road map for the dissertation

This dissertation validates the thesis statement via the design, implementation, and experimental evaluation of two systems: MAUI and Kahawai. The rest of this dissertation is organized in five chapters as follows:

Chapter 2 describes the characteristics of the code offload environment and the applications this thesis targets. In particular, it defines the classes of computationally intensive and graphics intensive applications. It then elaborates on their distinguishing features, as well as the specific properties that developers must identify to maximize the benefits of code offload.

Chapter 3 describes MAUI. MAUI is an energy-efficient, general-purpose code offload system based on the combination of extensive device, network and application profiling and an integer linear problem solving facility for dynamically computing optimal program partitions. MAUI is best suited for operations that consume and produce small amounts of information compared to their computational requirements.

Chapter 4 describes Kahawai, a system that provides offloading graphics-intensive applications such as games, CAD and medical imaging applications. Kahawai is better suited for continuous real-time graphics intensive applications whose output is deterministic and changes gradually over time.

Finally, **Chapter 5** presents related work and **Chapter 6** concludes the dissertation with a summary of the key contributions, future research directions, and closing remarks.

This dissertation also includes one appendix, **Appendix A**, that describes the serialization mechanisms used in MAUI. The MAUI prototype was built using two different serialization schemes: the first using XML Serialization and incremental updates, and a second that uses binary serialization. The appendix first describes the advantages and shortcomings of XML serialization while motivating the use of incremental updates (deltas). Next, it explains the preferred approach, binary serialization, and explains it in the context of the MAUI implementation, in particular the MAUI proxy component.

2

Property Specific Offload: Applications and their properties

In this chapter, I describe the mobile code offload environment envisioned in this dissertation. I describe the characteristics of the devices, the infrastructure, and the classes of applications this dissertation focuses on. When describing the applications, I also describe the important properties that distinguish each class, and the set of properties that developers have to identify in order for their applications to take advantage of code offload.

2.1 The Code Offload Scenario

Code offload is the opportunistic offload of computation from a mobile device to remote server infrastructure. Thus, offload decisions must be informed by information about the mobile device, the computation, and the remote server infrastructure. This section describes all three in detail, as well as their role within this dissertation.

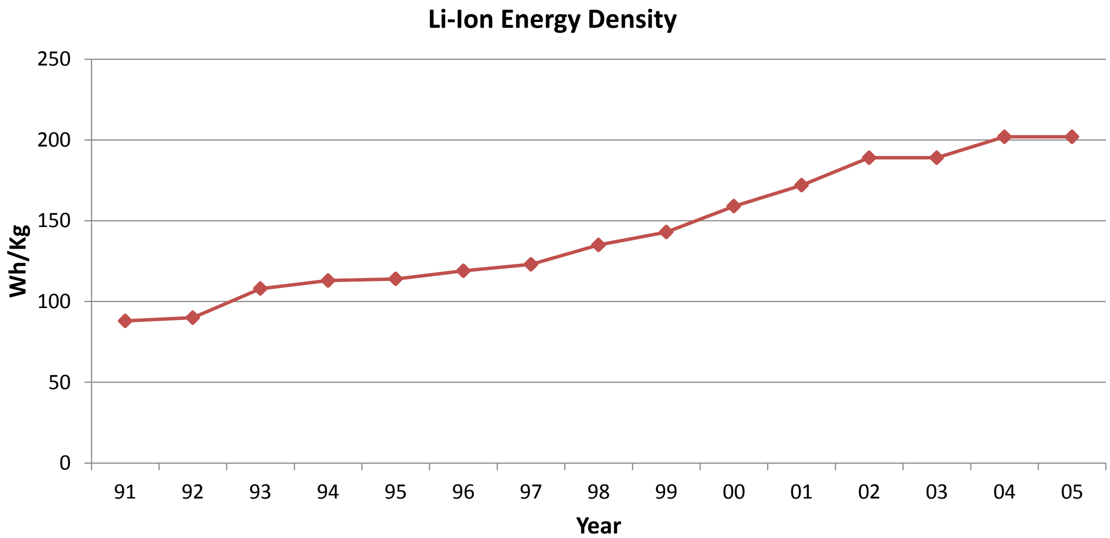

FIGURE 2.1: **Li-ion battery energy density from 1991 to 2005**

2.2 The Mobile Device

Since the arrival of the iPhone in 2007, the use and popularity of mobile devices has exploded. Modern palm sized mobile devices now feature high resolution displays, dedicated graphics processors, multicore CPUs, many gigabytes of storage, and gigabytes of memory. These devices capabilities are increasing at an accelerated rate, allowing sophisticated operations to be performed by pocket-sized devices. However, despite these advances, there are two limitations that prevent a modern device from performing computational complex operations as well as a full-sized desktop computer. The first is battery capacity, which limits what a mobile device can do on a single charge. This limitation prevents intensive tasks from running for long periods in order to conserve power. This problem is unlikely to improve through advances in battery technology alone, since as shown in [79] and [4] (Figure 2.1), battery density has roughly improved by a factor of 2 during the period spanning 1991 to 2005. This problem is worsened by the consumers' demand for lighter and smaller devices, coupled with the increasing energy demands of high resolution screens, GPS units, discrete graphics, and more sophisticated CPUs.

The second limitation is manufacturers' need to sacrifice performance in favor of smaller, cheaper, and more power-efficient devices. This means that mobile devices are consistently slower and more resource constrained than their full sized counterparts. As a consequence, despite technological advancements in mobile, a full sized desktop computer or server is likely to perform complex computations more quickly than its mobile counterpart.

These resource limited devices come in different sizes and form factors. Nevertheless, all of these devices are resource constrained, and as such they are the object of study of this dissertation. Specifically, mobile devices considered in this dissertation include, but are not limited to, laptops, netbooks, tablets, PDAs, and smartphones.

2.3 The Offloaded Computation

Hundreds of thousands of mobile applications have been built for Apple's App Store and Google's Android Market. These applications include social networking clients, media consumption and entertainment clients, productivity suites, communication clients, shopping apps, and games. These applications are designed with mobile devices' resource constraints in mind, which limits their potential.

Augmented reality illustrates how resource constraints limit the kind of apps that developers build. Augmented reality services aim to enhance the live video seen through a device's camera with extra information about the content of the feed. In existing augmented reality apps, this extra information usually pertains to *fixed* landmarks and businesses, since an app can infer what the camera is pointed at using a device's GPS, gyroscope, and accelerometer sensors. However, an app would ideally also use the source video itself to identify dynamic objects (e.g., people). Unfortunately, processing live video is too computationally intensive for a mobile device. Real-time speech recognition and translation represent another application that cannot be performed using only mobile hardware.

As a result, most modern mobile platforms use a simple thin-client model to offload computationally intensive apps. Under a thin client model, a developer statically partitions a program so that the computationally intensive portions execute on a remote server. The primary problem with this approach is that the app is unavailable when disconnected from the network. Furthermore, statically partitioned apps cannot adapt dynamically to newer devices that may be able to handle greater portions of the task locally. For example, applications such as face recognition, handwriting recognition, image manipulation, and activity recognition may run better on a remote server, but could run locally when a device is disconnected.

Games are also constrained by the limitations of mobile hardware. Modern full-featured videogames display sophisticated graphics that require power-hungry hardware to provide the best user experience. The high-end Graphics Processing Units (GPUs) found in desktop systems have both high energy and cooling requirements. For example, the Nvidia GeForce GTX 690 [74] draws 300W and can heat up to 98 degrees Celsius. This would be wildly impractical for a mobile device that would typically consume two orders of magnitude less power and be unable to provide cooling for such high temperatures. Instead, mobile GPUs are a compromise between performance and power/thermal efficiency, resulting in graphics that are not as immersive as those of a modern desktop GPU.

A primary goal of this dissertation is to characterize the classes of applications that can be efficiently offloaded using only a small number of developer-provided application properties. First, I will describe the classes of applications and their distinctive characteristics. Next, I will describe the properties that developers have to identify to the offload engine. I will borrow terminology found in Balan's Application Taxonomy[4] to name application classes. Listed below are the two classes of applications addressed by this dissertation: computationally-intensive applications and graphically-intensive applications.

2.3.1 Computationally-intensive applications

This class of applications includes those with the following features: 1) They are *computationally intensive*, requiring significant CPU, memory, storage, or energy resources; 2) They use an operation model, taking input from the user or sensors, operating on the input, and then storing or presenting the results; 3) The required input and output of the operations, are small compared to the effort needed to perform the operation. Specifically, the operation's input and output data require less time and energy to be transferred to and from a remote server than the time and energy required to perform the operation locally on the device. As a consequence, applications that perform computationally simple operations but handle small amounts of input and output data are well suited for code offload.

To maximize the benefits for this class of applications, the developer must first be able to distinguish it from the other supported class (i.e., graphically-intensive, continuous applications). Once an application has been identified as a member of this class, developers must identify occurrences of the following properties of their code in order to allow our offloading system, MAUI (Chapter 3), to dynamically partition their application.

- **Use of sensors, input, or output devices:** By following an operation model, this dissertation treats the start of the operation (reading user input or sensors) and its end (displaying or storing results) as not offloadable. Therefore, the developer to identify these events. In MAUI the beginning and end of an operation are identified at the method granularity by not annotating them with a "Remoteable" attribute.

- **External side-effects:** We consider external side-effects, those effects of executing an operation that changes state external to the mobile device and the infrastructure server. External side-effects are undesirable because they make it hard to guarantee correct execution of an offloaded operation after a disconnection has happened. Therefore, to provide correct execution semantics, operations that may generate external side-effects cannot be offloaded. By identifying this property in the code of the application, the developer can let our system know that a given operation is not suitable for offload, while still allowing the rest of the application to be considered for fine-grained offload.

- **Multiple computation threads:** Concurrent threads of execution create data dependencies that are difficult to manage in an offload environment. This is specially true when only a portion of the threads is offloaded to the infrastructure server, while the rest of the program executes on the mobile device.

  Fortunately, our model does not exclude all forms of concurrency. For example, applications may have many client-only threads to handle the user interface, but may have only one offloadable thread with data dependencies on the client threads. In our prototype, the developer only needs to identify threads that are not suitable for offload (i.e., client-only threads and threads with data dependencies), so that only the most resource intensive thread executes remotely.

  Examples of this class include voice recognition, face recognition, activity recognition, casual games, photo enhancing, and anti-virus.

2.3.2 Graphically-intensive applications

This class of applications includes those that rely on real-time graphics that are updated continuously, often without user's input. These applications include sophisticated 3D games, medical imaging, CAD, and interactive 3D visualizations. There are already applications of this type designed for mobile phones, but as discussed in the previous section they are carefully designed to not exceed the graphical capabilities of the mobile phone. These applications offer significant opportunities for improvement by ignoring devices' limitations and taking advantage of code offload.

Previously, graphically-intensive applications were considered impractical to offload, both because of their continuous nature and their typically large outputs. The output of each sequence of graphics operations is an image, which is produced continuously between 30 and 60 times per second. As a result, offloading can cause large data transfers between the infrastructure server and the client. This is not feasible within the tight restrictions of latency and mobile bandwidth usage limits on mobile devices. Existing thin client solutions partially address this problem by encoding the graphics output as compressed video[75], which drastically reduces its size. However, high-resolution video bandwidth requirements are still impractical for most mobile environments.

Fortunately, graphically-intensive applications generally presents the following patterns:

1. The output changes gradually over time. In other words, it presents *temporal locality*.

2. Graphically-intensive applications include *user configurable video settings* that provide different levels of quality, or *fidelities*. The resulting output at different fidelities often has a high degree of similarity.

3. Their output consists of *discrete samples*. In a graphics application, graphics operations are processed at a high frequency. This is done to provide the illusion of movement by feeding enough samples of an animated scene to the user's eyes. This frequency can often be reduced, resulting in consistent samples at different frequencies. In other words, if the sampling frequency is decreased from 60 Hz to 6 Hz, the new sample 1 would be equal to the original sample 10, the new sample 2 would be equal to the original sample 20, and so on.

4. By eliminating simple sources of non-determinism, such as calls to the system timer, and keeping input synchronized, two different copies of a graphics application can be made to produce the same output without requiring low-level memory or application state synchronization.

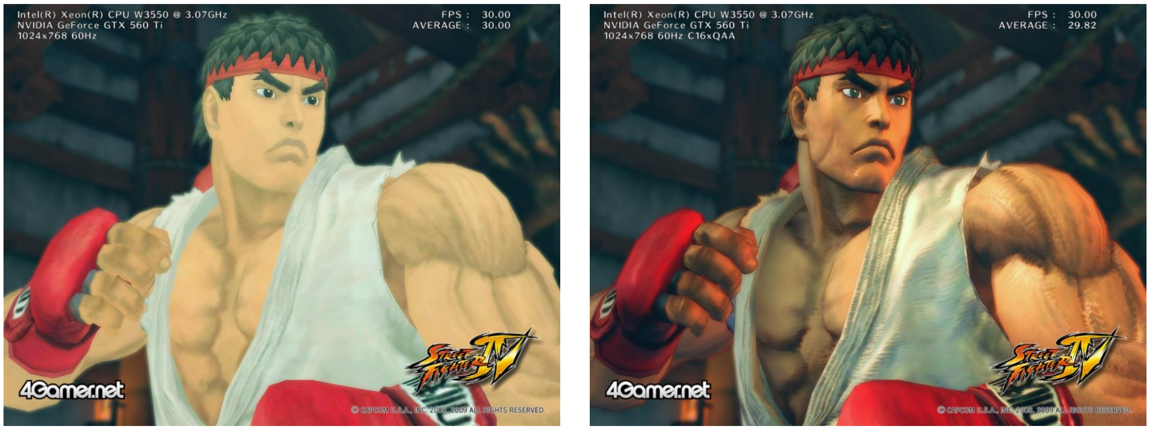

FIGURE 2.2: **Example of two different video quality settings: Consistent scenes taken from the game Street Fighter IV[11]**

As a result of these four patterns, it is possible to execute two collaborative copies of the same application. A resource constrained client can produce a subset of the desired output samples, while an over-provisioned server supplements it with the missing samples by working at the desired frequency. In a similar way, a constrained client can work under lower video quality settings and receive information about the missing detail from an over provisioned server that runs both a low video quality version and a high video quality version of the application.

In order to understand how to ensure the correct execution of offloading for graphics intensive applications, I built Kahawai (Chapter 4). Once an application is identified as a graphics intensive application, the developer must make sure that the application has the following properties:

- **Deterministic behavior:** The developer should identify sources of non-determinism and annotate them, so that the underlying runtime can ensure that non-deterministic sources are replayed appropriately in all concurrent versions of the application.

- **Consistent scene composition across video quality settings:** As previously mentioned, it is common for graphics applications like games to provide similar output

FIGURE 2.3: **Example of two different video quality settings: Non-consistent scenes taken from the game Lost Planet II[12]**

at different video quality settings. This is, however, not always the case, and it is up to the developer to identify whether her application exhibits this property so that the runtime has enough flexibility to decide which offload mechanism to use.

We consider the output of two executions of a graphics applications running with two different video quality settings to be similar if their *scene composition* is consistent. We define the composition of a scene as the set of logical entities in the scene. Such entities include, but are not limited to, physical objects (e.g., characters, walls, trees, lamps, doors, etc.), light sources, and cameras. For the composition of two scenes to be consistent they should have exactly the same entities, all of them sharing attributes such as position, rotation, average color, and basic shape. However entities in a higher quality scene can have a higher level of detail. For example, the high detail scene entities may consist of a larger number of polygons, use finer textures, or have a more accurate shading model. Figures 2.2 and 2.3 show a consistent and a non-consistent pair of scenes, respectively. Figure 2.3 shows two snapshots of the same scene using different video quality settings. Observe that the high quality scene adds extra vegetation to the scene, resulting in a different scene composition. Scenes presenting such behavior are not suitable for the Delta Encoding technique

FIGURE 2.4: **Example of two different video quality settings: A low video quality scene with higher entropy than its higher quality counterpart**

explained in Chapter 4.

- **Lower overall entropy in lower quality scenes:** It is common for higher detail versions of a scene to use better lighting models, more complex textures, higher-count polygon objects, and more sophisticated filters. As a result, their output often shows a higher number of high frequency values, i.e., higher entropy. However, there are situations where game developers try to make up for the lack of a sophisticated features by using higher entropy elements in lower detail scenes. Figure 2.3 shows two scenes, where the high detail frame (right) uses an advanced feature, tesselation, to give the water a more realistic shape and motion. The low detail frame (left) instead uses a complex static texture to make up for the missing advanced element. This adjustment to the low detail scene makes its entropy higher than the high detail scene's and makes them different enough to render techniques such as Delta Encoding (Chapter 4) ineffective. For this reason, it is up to the developer to identify whether the application produces low detail scenes with an overall lower entity.

2.4 The Infrastructure

The final element in code offload is the infrastructure. The infrastructure is one or more well provisioned desktop or server computers. These computers can be owned by the mobile user and available within the user's house or office[85], or be owned by a third-party service provider. For example, coffee stores, airports, or bookstores can provide their own offload infrastructure as an extra value to their customers. Similarly, wireless carriers can deploy offload infrastructure near their towers as an incentive to potential customers. Finally, offload infrastructure can also be located in the cloud and accessed for a fee paid by the mobile user.

In general, we assume that the server infrastructure is powerful enough to serve all the needs of its mobile clients. This dissertation does not address issues of server scalability, service discovery, or security. Work on these issues is orthogonal to the techniques described in this dissertation.

3

Making Smartphones Last Longer with Code Offload

In this chapter we present MAUI, a system that enables fine-grained energy-aware offload of mobile code to the infrastructure. MAUI uses the benefits of a managed code environment to offer the best of both worlds: it supports fine-grained code offload to maximize energy savings with minimal burden on the programmer. MAUI decides at run-time which methods should be remotely executed, driven by an optimization engine that achieves the best energy savings possible under the mobile device's current connectivity constrains.

3.1 Introduction

One of the biggest obstacles for future growth of smartphones is battery technology. As processors are getting faster, screens are getting sharper, and devices are equipped with more sensors, a smartphone's ability to consume energy far outpaces the battery's ability to provide it. Unfortunately, technology trends for batteries indicate that these limitations are here to stay and that energy will remain the primary bottleneck for handheld mobile devices [81]. Given the tremendous size of the mobile handset market, solving the energy impediment has quickly become the mobile industry's foremost challenge [41].

One popular technique to reduce the energy needs of mobile devices is *remote execution*: applications can take advantage of the resource-rich infrastructure by delegating code execution to remote servers. For the last two decades, there have been many attempts to make mobile devices use remote execution to improve performance and energy consumption. Most of these attempts took one of the following two approaches to remote execution. The first approach is to rely on programmers – to specify how to partition a program, what state needs to be remoted, and how to adapt the program partitioning scheme to the changing network conditions [26, 27, 3, 7]. This approach leads to large energy savings because it is fine-grained – applications can remote only the sub-parts that benefit from remote execution. For example, an application that both decodes and plays video would remote only the decoder, which is the CPU-intensive part, without remoting any of the screen-intensive parts. The second approach is to use full process [77] or full VM migration [16, 85] in which individual applications (or entire OS's in the case of VMs) can migrate to the infrastructure. This approach reduces the burden on programmers because applications do not need to be modified to take advantage of remote execution; instead, all their code and program state is automatically sent to the remote infrastructure.

We present MAUI, an architecture that combines the benefits of these two approaches: it maximizes the potential for energy savings through fine-grained code offload while minimizing the changes required to applications. MAUI achieves these benefits by using several properties of today's managed code environments (we use the Microsoft .NET Common Language Runtime (CLR) [84] for MAUI, although Java would offer the same properties). First, MAUI uses *code portability* to create two versions of a smartphone application, one of which runs locally on the smartphone and the other runs remotely in the infrastructure. Managed code enables MAUI to ignore the differences in the instruction set architecture between today's mobile devices (which typically have ARM-based CPUs) and servers (which typically have x86 CPUs). Second, MAUI uses *programming reflection* combined with *type safety* to automatically identify the remoteable methods and extract only the program state needed by those methods. Third, MAUI profiles each method of an application and uses *serialization* to determine its network shipping costs (i.e., the size of its state). MAUI combines the network and CPU costs with measurements of the wireless connectivity, such as its bandwidth and latency to construct a linear programming formulation of the code offload problem. The solution to this problem dictates how to partition the application at runtime to maximize energy savings under the current networking conditions. Because serialization can be done at runtime, MAUI's application profiling is run continuously to provide up-to-date estimates of each method's costs. This continuous profiling makes MAUI's program partitioning scheme highly dynamic. The combination of code portability, serialization, reflection, and type safety allows MAUI to offer dynamic fine-grained code offload with minimal burden on the programmer.

In implementing MAUI, we discovered a number of unforeseen challenges to implementing program partitioning for mobile applications. One such challenge is that using power-save mode (PSM) when transferring state remotely can *hurt* the overall energy consumption of the application when the latency to the server is low. Moreover, PSM mode helps save energy but only when latencies approach the sleep interval (today's hardware uses 100 ms). Another unforeseen challenge is related to using profiling to estimate the energy savings of code offload. On one hand, profiling the state transfer overhead of a method each time it is called can provide the freshest estimate; on the other hand, the cost of this profiling is not negligible and it can impact the overall application's performance. Throughout our presentation of the MAUI architecture, we provide in-depth descriptions of the lower-level implementation challenges discovered in an effort to guide future implementations of remote execution to avoid them.

Our evaluation shows that MAUI offers significant energy and performance benefits to mobile applications. With MAUI, we partitioned four smartphone applications that have different latency and computational requirements. Our results show that (1) MAUI improves the energy consumption of resource-intensive applications, such as face recognition, by almost one order of magnitude (i.e., a factor of eight); (2) MAUI allows latency-sensitive applications, such as games, to more than double their screen refresh rates; and (3) MAUI enables the development of applications that bypass the limitations of today's smartphone environments; we developed a real-time voice-based Spanish-to-English translator even though today's smartphones do not provide enough RAM to run our advanced speech recognition engine.

The remainder of this paper presents the design, implementation, and evaluation of MAUI. We start by raising four questions that investigate the benefits of remote execution given today's mobile landscape (Section 3.2). Next, we present a high-level description of MAUI's architecture (Section 3.3). The next three sections describe MAUI's program partitioning mechanism (Section 3.4), its ability to profile mobile applications and the wireless environment (Section 3.5), and its dynamic optimization framework (Section 3.6). The paper's final section evaluate MAUI's performance (Section 3.7). A summary of the work related to remote execution for mobile applications can be found in (Chapter 5), and the conclussions to this chapter in (Section 3.8).

3.2 The Need for Remote Execution in Today's Mobile Landscape

This section's goal is to examine remote execution through the lens of today's mobile landscape. Today's mobile users expect their smartphones to run sophisticated applications, they have almost ubiquitous access to 3G connectivity, and their computing applications are moving to the cloud. How do these trends affect the remote execution problem? The remainder of this section examines these issues by raising three questions:

1. MAUI's goal is to mitigate the energy problem for mobile handhelds, the mobile industry's foremost challenge. *How severe is the energy problem on today's handheld devices?*

2. Mobile handheld devices are increasingly equipped with wireless wide-area network interfaces, such as 3G. Unlike Wi-Fi, 3G offers ubiquitous connectivity. However, 3G connections are known to suffer from very long latencies and slow data transfers [42]. For code offload, this poor performance may lead to increased energy consumption. *How energy efficient is using 3G for code offload?*

3. With cloud computing, industry is building massive infrastructure to offer highly available resources. Offloading mobile code to the cloud is an attractive possibility for systems like MAUI. However, typical round-trip times (RTTs) to cloud servers are on the order of tens of milliseconds. An alternative for MAUI is to use *nearby* servers [85] (with RTTs less than 10ms), such as a server co-located with a Wi-Fi access point or a user's personal desktop in the home. *How sensitive is the energy consumption of code offload to the RTT to the remote server?*

3.2.1 How Severe is the Energy Problem on Today's Mobile Handheld Devices?

The two main contributors to the energy problem for today's mobile devices are: 1) limited battery capacity, and 2) an increasing demand from users for energy-hungry applications.

To illustrate the severity of the battery limitations of today's handheld devices, we performed the following experiment. We created a synthetic resource-intensive application, and measured how long it takes to completely drain the battery of a new smartphone (the HTC Fuze with a 1340 maH battery, released in November 2008). Our synthetic application performs a large bulk-data transfer over the Wi-Fi interface, consumes the entire CPU, and keeps the display backlight on. Although it is synthetic, this application is not unrealistic: a streaming video decoder would also heavily utilize the CPU and the Wi-Fi network with the screen on. When running this application, we found that the fully charged battery drained after only one hour and twenty minutes.

In today's mobile handheld market, user demand is increasing for three categories of resource intensive applications. First, video games are very popular (for example, many of the top applications on Apple's AppStore are video games) and they have large energy

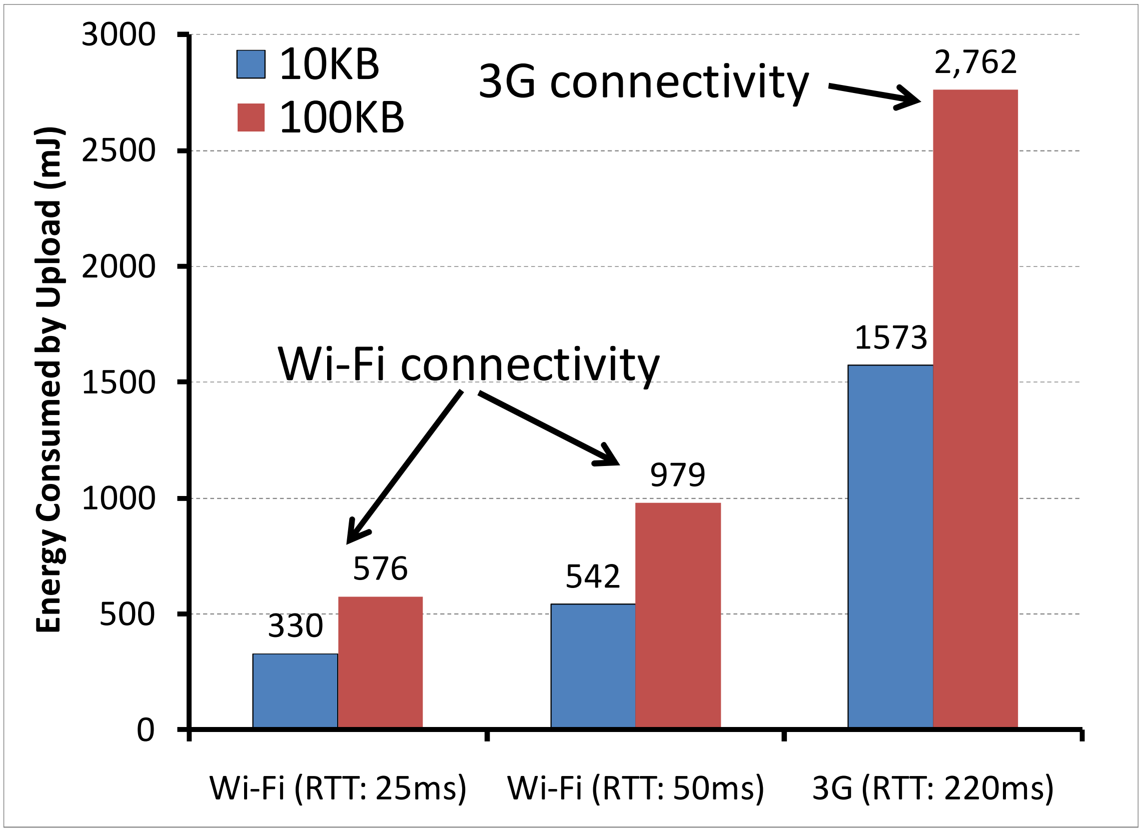

FIGURE 3.1: **The Energy Consumption of Wi-Fi Connectivity vs. 3G Connectivity**

demands. Second, mobile users are increasingly watching streaming video. For example, one third of iPhone users accessed YouTube [59] and the number of mobile uploads to YouTube increased 400% during the first six days after iPhone 3GS's release [52]. Finally, mobile devices are increasingly equipped with new sensors that produce continuous streams of data about the user's environment. New applications that rely on continuous processing of sensor data are emerging, such as car navigation systems, pedometers, and location-based social networking. Today, developers restrict their applications by making judicious use of sensors due to energy concerns (e.g., obtaining a GPS reading is expensive).

The trends in battery technology make it unlikely that the energy problem will disappear in the future. Although the chemical composition of batteries has changed significantly over the years, these changes have had less impact on battery capacity than on other aspects of battery technology, such as recharging and "memory effects". While recent research has suggested using newer chemicals [78], these changes are unlikely to significantly improve battery lifetime. Another proposed technology is the use of "fuel cells", yet even the most optimistic predictions of their performance do not suggest that energy concerns are likely to disappear [69].

3.2.2 How Energy Efficient is 3G for Code Offload?

Handheld devices have two alternatives for code offload: use either 3G or Wi-Fi. The primary benefit of 3G over Wi-Fi is the near-ubiquitous coverage it provides. However, recent studies have shown that round-trip times over 3G are lengthy and bandwidth is limited. RTTs are consistently on the order of hundreds of milliseconds and they even reach seconds [42]. Such long latencies may make code offload slow and expensive. For example, sending just a few tens of kilobytes to the cloud requires multiple RTTs, and thus the transfer time may take on the order of seconds.

We performed a simple experiment to contrast the energy consumption of 3G versus Wi-Fi. We setup the HTC Fuze smartphone do to two small code uploads (10 KB and 100 KB) to a remote server using either 3G or Wi-Fi. The server uses a driver that adds a controlled amount of queuing delay to the network path between the smartphone and the server. We then evaluated two scenarios: (1) the smartphone using Wi-Fi to reach the server (and adding an additional 25 ms or 50 ms of queuing delay); and (2) the smartphone using 3G (with a measured RTT of 220 ms). Figure 3.1 shows the energy consumed by the smartphone during the two uploads. Using 3G, the smartphone consumes three times as much energy as it does using Wi-Fi with a 50 ms RTT, and five times the energy of Wi-Fi with a 25 ms RTT. This implies that our HTC Fuze phone, with a 1340 mAH battery, will last less than 2 hours if the phone repeatedly downloads a 100 KB file over a 3G connection.

Such drastic discrepancies in energy consumption made us reconsider using 3G for MAUI. Despite the coverage benefits, 3G has poor performance and high energy consumption. For a system aimed at saving energy (like MAUI), the cost of using 3G today is almost prohibitive. In contrast, Wi-Fi has much better performance and lower energy consumption. MAUI can use either Wi-Fi or 3G for code offload.

3.2.3 How Sensitive is the Energy Consumption of Code Offload to the RTT to the Server?

As the previous experiment showed, the longer the RTT to the remote server, the higher the energy consumed by the code offload. To investigate this latency versus energy tradeoff in more-depth, we performed a series of experiments where we offloaded code to a remote server via a link with increasing RTTs from 10 up to 150 ms. We used two configurations for Wi-Fi: with power-save mode (PSM) enabled, and without. With PSM, the smartphone's Wi-Fi radio wakes up only when it has data to transmit and once every 100 ms when it checks whether the access point has any incoming data.

Figure 3.2 shows the energy consumed when offloading 10 KB and 100 KB of data with Wi-Fi using PSM. We also measured the results when offloading pieces of code with different sizes (20 KB, 50 KB, 500 KB, and 1 MB) and using Wi-Fi with PSM disabled, but we omit showing these results to avoid cluttering the graph since the results show the same trends. As Figure 3.2 shows, the energy consumption of code offload grows almost linearly with the RTT. In fact, for offloading 10 KB of code, the energy consumption almost doubles when increasing the RTT from 10 ms to 25 ms only, and it doubles again when the RTT reaches 100 ms.

Two important consequences arise from these results. First, cloud providers should strive to minimize the latency to the cloud for mobile users. Shorter RTTs can lead to significant energy savings. Second, the benefits of remote execution are most impressive when the remote server is *nearby* (RTT of 10 ms), such as on the same LAN as the Wi-Fi

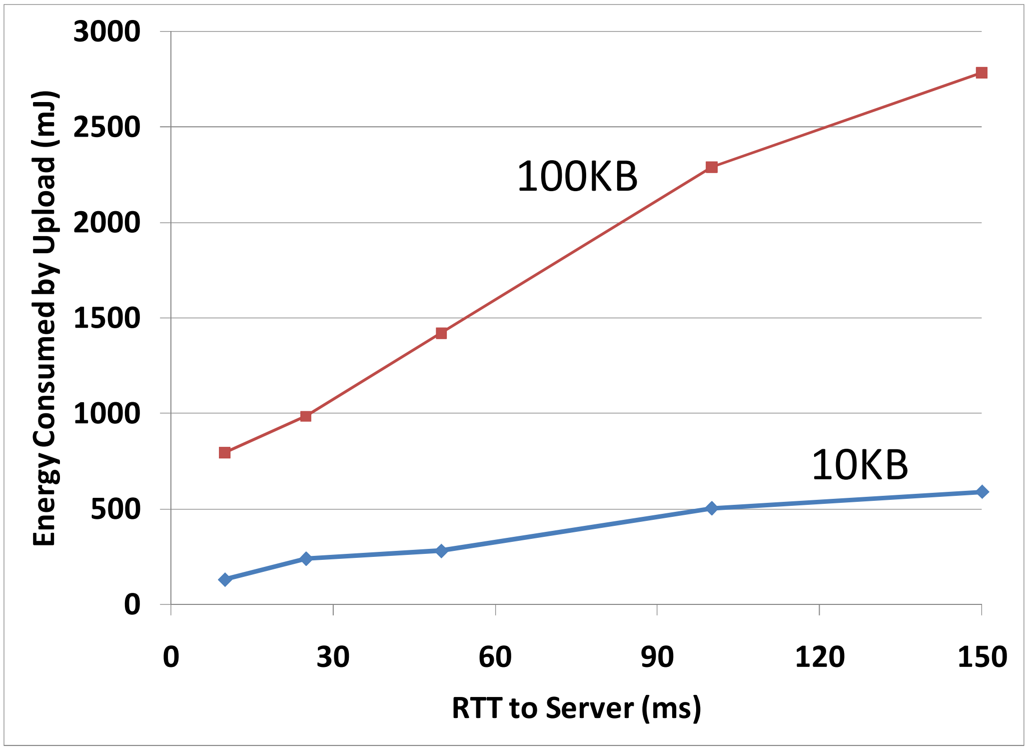

FIGURE 3.2: **The Energy Consumed by Offloading 10 KB and 100 KB of Code to the Cloud for Various Round-Trip Times to the Server.**

access point, rather than in the cloud (RTT $\geqslant$ 25 ms). There are many scenarios where it is straightforward to deploy remote execution servers near smartphone users, such as in an enterprise to help increase the battery lifetimes for employees' smartphones, and in private homes to help home smartphone users. Others also pointed out the huge energy benefits of offloading code to nearby servers [85].

3.3 MAUI System Architecture

MAUI's goal is to maximize the benefits of code offload for today's smartphone devices. In this section, we present a high-level overview of MAUI's components on a mobile device and in the infrastructure in order to understand how they all integrate into one platform for developing mobile applications.

MAUI provides a programming environment where developers annotate which methods of an application can be offloaded for remote execution. Each time a method is invoked and a remote server is available, MAUI uses its optimization framework to decide whether the method should be offloaded. Once an offloaded method terminates, MAUI gathers profiling information that is used to better predict whether future invocations should be offloaded. If a disconnect occurs, MAUI resumes running the method on the local smartphone; in this case, the application's energy consumption only incurs a small penalty – the cost of shipping the control and program state to the server.

MAUI instruments each method of an application to determine the cost of offloading it, such as the amount of state that needs to be transferred for remote execution, and the benefit of offloading it, such as the number of CPU cycles saved due to the offload. Additionally, MAUI continuously measures the network connectivity to the infrastructure, estimating its bandwidth and latency. All these variables are used to formulate an optimization problem whose solution dictates which methods should be offloaded to the infrastructure and which should continue to execute locally on the smartphone. Because mobile users may move in and out of a MAUI server's range relatively rapidly, the optimization problem is re-solved periodically to enable MAUI to adapt to changes in the networking environment.

Figure 3.3 provides a high-level overview of the MAUI architecture. On the smartphone, the MAUI runtime consists of three components: 1) an interface to the decision engine (to save energy, the solver actually runs on the MAUI server); 2) a proxy, which handles control and data transfer for offloaded methods; and 3) a profiler, which instruments the program and collects measurements of the program's energy and data transfer requirements. On the server side, MAUI provides four components: the profiler and the server-side proxy which perform similar roles to their client-side counterparts; the decision engine which periodically solves the linear program; and the MAUI coordinator, which handles authentication and resource allocation for incoming requests to instantiate

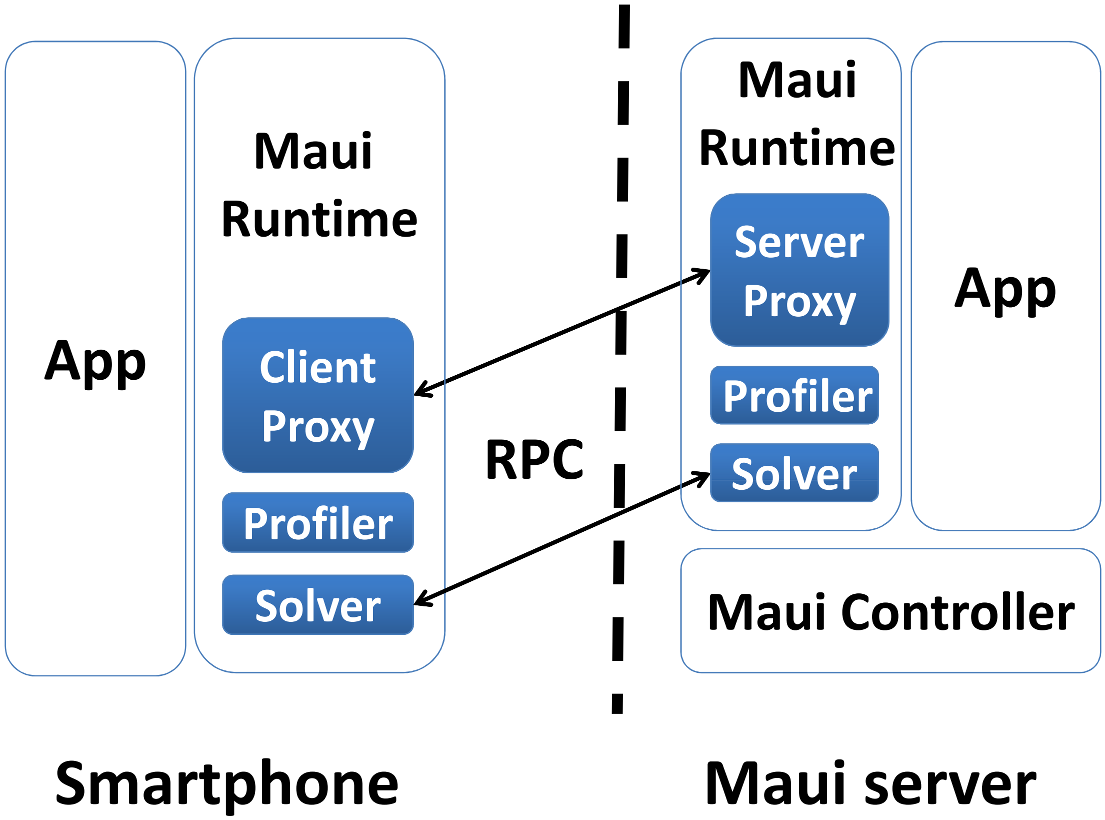

FIGURE 3.3: **High-level view of MAUI's architecture.**

a partitioned application. In Sections 3.4, 3.5, and 3.6, we describe in detail MAUI's mechanisms for program partitioning, the MAUI profiler, and the MAUI solver, respectively.

3.4 Program Partitioning

MAUI enables developers to produce an initial partitioning of their applications with minimal effort. The developer simply annotates as *remoteable* those methods and/or classes that the MAUI runtime should *consider* offloading to a MAUI server. Certain types of code should not be marked remoteable: 1) code that implements the application's user interface; 2) code that interacts with I/O devices where such interaction only makes sense on the mobile device; and 3) code that interacts with any external component that would be affected by re-execution (MAUI's failure handling is discussed in detail in Section 3.4.2).

Examples of the second category include code that reads an accelerometer, or code that determines the location of smartphone by reading from the GPS device. An example of the third category is code that uses a network connection to perform an e-commerce transaction (such as purchasing an item from an online store).

Our intent for the remoteable annotations is that developers do not need to guess about whether or not a particular method makes sense to offload (from the perspective of energy and performance). Instead, they should mark as remoteable all code that meets the criteria specific above, and it is up to the MAUI solver (described in Section 3.6) to determine at runtime whether it makes sense from a performance and energy standpoint to execute it remotely.

We considered and then discarded an alternative approach: require the programmer to annotate those methods and/or classes that can only run on the smartphone as *local*, and then consider all the remaining code as remoteable. Although this approach may lead to fewer annotations, we decided against this approach because it favors performance over program correctness. If the programmer makes a mistake, such as forgetting to label a method as *local*, then the program may no longer behave correctly should MAUI decide to offload that method. Instead, with our approach such mistakes only affect the program's performance and not its correctness.

3.4.1 Partitioning .NET Applications

To support partitioning an application across multiple machines, MAUI needs to address the following challenges:

1. Because today's smartphones typically use a different instruction set architecture (ARM) than desktop and server machines (x86), MAUI needs the ability to execute the same program on different CPU architectures, preferably without access to the program source code.

2. MAUI must be able to automatically identify which methods are marked *remoteable* and which are not.

3. MAUI must be able to automatically identify and migrate the necessary program state from a running program on one machine to another.

4. MAUI must be able to dynamically select whether to run a method locally or remotely based on the current environment.

5. MAUI must be able to detect and tolerate failures without affecting the original program semantics.

In the rest of this section we discuss how our MAUI prototype handles these challenges.

Executing the Same Code on Different CPU Architectures

MAUI is currently designed only to support applications written for the Microsoft .NET Common Language Runtime (CLR) [84]. While the CLR supports a variety of programming languages, our MAUI applications are all written in C#[1]. All CLR applications are compiled to the CIL intermediate language, regardless of what source language they are written in. An "executable" for the CLR contains CIL instructions, and the CIL is dynamically compiled at execution time. Thus, by leveraging the CLR's capabilities, MAUI obtains independence from differences in the instruction set architecture between today's smartphones (which typically have ARM-based CPUs) and today's desktop and server machines (which typically have x86 CPUs).

To begin running an application, the MAUI runtime must ensure that the MAUI server has copies of the application executables. The MAUI runtime supports two options: 1) the MAUI server can obtain copies of the program executables directly from the smartphone (which incurs a delay and consumes energy), or 2) the MAUI runtime on the smartphone can send signatures of the executables to the MAUI server which then downloads the actual executables from a cloud service.

[1] Some include native dll's with C# wrappers.

Extracting Remoteable Methods Using Reflection

To identify remoteable methods in a language-independent manner, we use the custom attribute feature of the .NET CLR. Attributes are meta-data that annotate specific code elements such as methods or classes, and they are included in the compiled .NET CLR executables. The application developer modifies the application's source code by adding the "[Remoteable]" attribute to each method that is safe to execute remotely. The MAUI runtime uses the .NET Reflection API [53, 84] to automatically identify which methods the developer has marked as suitable for remote execution, simply by searching through the executable for those methods tagged with the "[Remoteable]" attribute.

Identifying the State Needed for Remote Execution Using Type-Safety and Reflection

At compile time, MAUI generates a wrapper for each method marked as remoteable. This wrapper follows the original method's type signature, but with two changes: it adds one additional input parameter, and one additional return value. Figure 3.4 shows a snippet of the video game application's original interface and the MAUI wrapper interface. The additional input parameter is used to transfer the needed application state from the smartphone to the MAUI server, and the additional return value is needed to transfer the application's state back from the server to the smartphone.

Performing application state transfer leverages the type-safe nature of the .NET runtime. Type safety allows us to traverse the in-memory data structures used by the program, and to only send over the network data which is potentially referenced by the method being offloaded. To determine which state needs to be serialized beyond the explicit method parameters, we currently take the conservative approach of serializing all the current object's member variables, including not only simple types, but also nested complex object types. We also serialize the state of any static classes and any public static member variables.

To perform the serialization, we considered two approaches. The first one uses .NET built-in support for XML-based serialization, making extensive use of the .NET reflection API. Because the size of XML-based serialized state is potentially large compared to the in-memory representation, we also optimize the overhead of state transfer by only shipping incremental deltas of the application state rather than the entire state. The second approach relies on Protobuf.net [36], an implementation of Google's Protocol Buffers[33]. Our final prototype uses the second approach, which is faster and results in more compact representations. A more detailed explanation and comparison of both approaches can be found in Appendix A.

Performing Code Offload

At compile time, MAUI generates two proxies, one that runs on the smartphone and one that runs on the MAUI server. The role of these proxies is to implement the decisions made by the MAUI solver (described in Section 3.6). The solver decides, based on input from the MAUI profiler, whether the method in question should be executed locally or remotely, and the proxies handle both control and data transfer based on that decision. The granularity of state and control transfer in MAUI is at the method level; we do not support executing only portions of a method remotely. For calls which transfer control from the local smartphone to the remote server, the local proxy performs state serialization before the call and then deserialization of the returned application state after the call. When a remoteable method which is currently executing on the MAUI server invokes a method which is not remoteable, the server-side proxy performs the necessary serialization and transfers control back to the smartphone. The MAUI runtime currently has only limited support for multi-threaded applications – we only offload methods from a single thread at any given point in time.

```
//original interface
public interface IEnemy {
  [Remoteable] bool SelectEnemy(int x, int y);
  [Remoteable] void ShowHistory();
  void UpdateGUI();
}

//remote service interface
public interface IEnemyService {
  MAUIMessage<AppState, bool> SelectEnemy (AppState state, int x, int y);
  MAUIMessage<AppState, MauiVoid> ShowHistory(AppState state);
}
```

FIGURE 3.4: **Local and remote interfaces.**

3.4.2 Handling Failures

MAUI detects failures using a simple timeout mechanism: when the smartphone loses contact with the server while that server is executing a remote method, MAUI returns control back to the local proxy. At this point, the proxy can either re-invoke the method locally, or it can attempt to find an alternate MAUI server and then re-invoke the method on the new MAUI server. Because program state is only transferred at the start and end of methods, re-executing a portion of the method will not affect the program's correctness. However, programmers should not label methods as remoteable if those methods interact with any external component that would be affected by re-execution (e.g., using a network connection to perform an e-commerce transaction).

3.4.3 Additional Program Modifications Can Bring Performance Benefits

One of the key benefits of MAUI's approach to program partitioning is the extremely low barrier to entry for developers. Once the program is working with MAUI, the programmer may be interested in further optimizing the program's performance and/or energy consumption. From our experience, we find that some restructuring of the video game application has a significant impact on its performance and energy characteristics. The information produced by the MAUI profiler as input to the MAUI solver can also benefit the application developer in terms of understanding the energy and state transfer characteristics of the program.

In the video game application, the program logic that is best offloaded determines the attack strategy for the enemies. In the unmodified application, the method that implements this logic is called once per enemy during each frame refresh. If there are 60 enemies, this leads to 60 remote calls per frame. We modified the game to perform all the enemy update operations in a single call, which leads to a significant improvement in latency (i.e., the frame rate), and a reduction in energy consumption. We found that our legacy face recognition application and the chess game did not require any code restructuring to enable partitioning, and we wrote the voice translator application from scratch and therefore we deliberately separated out the logic that implements the graphical UI and I/O on the smartphone (e.g., the translator uses the microphone and the speaker) from the bulk of the application that implements the translation functionality.

3.5 MAUI Profiler

At runtime, before each method is invoked, MAUI determines whether the method invocation should run locally or remotely. Offload decisions depend on three factors: 1) the smartphone device's energy consumption characteristics; 2) the program characteristics, such as the running time and resource needs of individual methods; and 3) the network characteristics of the wireless environment, such as the bandwidth, latency, and packet loss. The MAUI profiler measures the device characteristics at initialization time, and it *continuously* monitors the program and network characteristics because these can often change and a stale measurement may force MAUI to make the wrong decision on whether a method should be offloaded. The current implementation of the MAUI profiler does not

FIGURE 3.5: **The hardware power meter used for energy measurements.**

explicitly incorporate the CPU load on the MAUI server. In the rest of the section, we provide an in-depth description of MAUI's techniques for device, program, and networking profiling.

3.5.1 Device Profiling

Today's smartphones do not offer a way to obtain fine-grained energy measurements of an application. Instead, they just offer a simple "how much battery is left" API that is very coarse-grained and often unreliable. Instead, to measure a device's energy consumption, we attach a hardware power meter [67] to the smartphone's battery (see Figure 3.5). This power meter can provide fine-grained energy measurements; it samples the current drawn from the battery with a frequency of 5000 Hz.

We use the power meter to build a simple energy profile of the smartphone. For the CPU, our approach is inspired by JouleMeter [50]. We constructed a synthetic benchmark that uses both integer and floating point arithmetic. We ran our synthetic benchmark multiple times, varying the run-time of the benchmark and instrumenting it to record the number of CPU cycles it required for each run. We then used the collected samples of CPU utilization and the corresponding smartphone energy consumption to build a simple linear model, using least-squares linear regression. This model lets us predict the energy consumption of a method as a function of the number of CPU cycles it requires to execute. We validated our model by comparing its prediction of energy consumption with the actual measurements produced by hardware power monitor for the applications we have built using MAUI. We found that the median error produced by the model is less than 6%, while the mean error is less than 8% and the standard deviation is 4.6%. Finally, we also used the energy meter to characterize the energy consumption of using the smartphone's Wi-Fi and 3G wireless radios.

3.5.2 Program Profiling

The MAUI profiler instruments each method to measure its state transfer requirements, its runtime duration, and the number of CPU cycles required for its execution. The amount of state needed to be sent to the MAUI server to execute the method includes the size of all data potentially referenced by the method, as well as the amount of state required to be returned to the smartphone once the method is completed (Section 3.4.1 describes in detail how MAUI identifies which program state needs to be transferred).

MAUI uses the method's duration and CPU cycles to estimate the energy consumed by running the method on the smartphone. Characterizing an application's energy consumption is very challenging for two reasons. First, applications are not deterministic: each subsequent invocation of a method can take a different code path leading to a different running duration and energy profile than the previous invocation. While more sophisti-

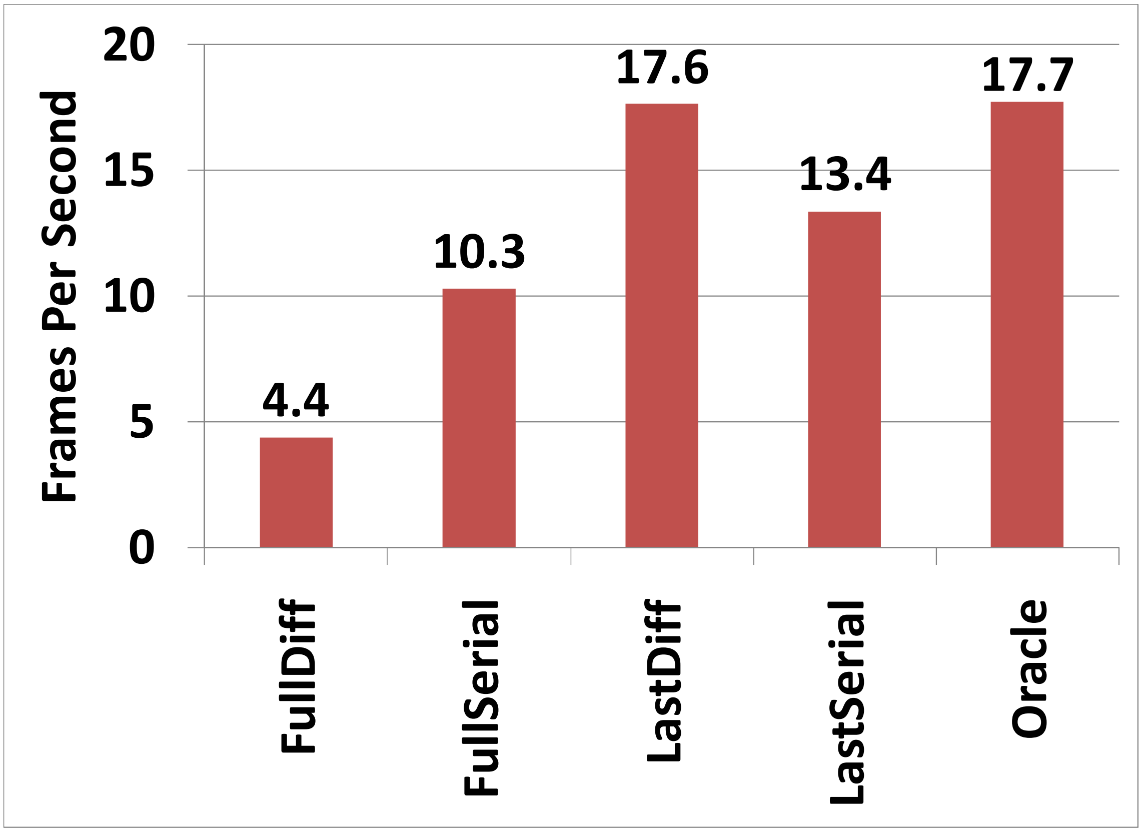

FIGURE 3.6: **Performance overhead of MAUI profiling.**

cated program instrumentation can help mitigate this problem, such techniques may be prohibitively expensive. Instead, MAUI uses past invocations of a method as a predictor of future invocations; we found this assumption to work well for the phone applications we used to experiment with MAUI. Second, today's smartphones scale the CPU's voltage dynamically to save energy (i.e., dynamic voltage scaling) without informing the software running on the phone. While the dynamic voltage scaling mechanisms could drastically change the energy profile of an application, our experiments validating the CPU energy model discussed earlier did not show this to be a significant problem. Once the application terminates and the user does not interact with the phone, the smartphone often lowers the CPU's voltage to save energy.

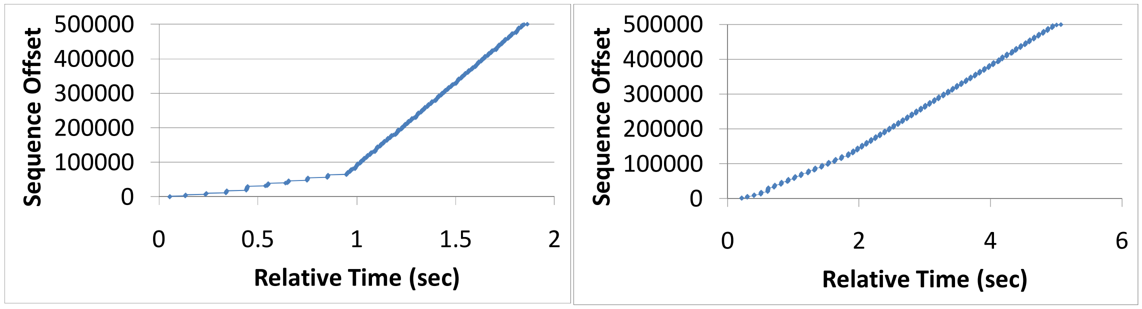

FIGURE 3.7: **The dynamics of a 500 KB TCP transfer with PSM disabled over a two links, one with a 25 ms RTT and one with 75 ms RTT.**

Profiling Overhead

As mentioned in Section 3.4.1, when using the XML Serializer, the MAUI proxies ship incremental deltas rather than the full application state to reduce the network and energy overhead of state transfer. To implement this, the local MAUI proxy keeps track of what state it has previously sent to the remote proxy, and just before each remote invocation, the proxy calculates the delta between the local state and the remote state. If the size of the delta is smaller than the full state, it sends the delta instead. This optimization, which seems straightforward, turns out to introduce unintended complexity for the MAUI runtime.

The MAUI profiler observes the program behavior over time as it executes, and the MAUI solver uses past program behavior as a predictor of how the application will behave in the future. The current MAUI profiler does not persist the profiling data across multiple runs of the same program. When a method A calls method B, the MAUI profiler measures the size of state that would need to be transferred over the network to enable B to run remotely, and this measurement is performed regardless of whether B is actually remoted. The performance overhead of taking this measurement is simply the cost of using the XML serializer to transform the state into XML format, and then measuring the size of the buffer. The more times the profiler observes A calling B, the better the estimate it obtains of future behavior.

The introduction of deltas changes this picture. When A calls B, the size of the state that needs to be transferred is now not only a function of the program's previous behavior, but also a function of the MAUI runtime behavior. In other words, MAUI has perturbed the very thing that it is measuring about the program. If MAUI has offloaded a method recently the delta will likely be small, whereas if MAUI has not offloaded in a long time then the delta is likely to be substantially larger.

Another unintended consequence of calculating deltas is the performance impact it has on interactive applications. To characterize this impact, we performed the following experiment. We used five different strategies for profiling an interactive application (a video game) and we measured the video game's performance (in frames per second) when offloading code to a nearby server. These strategies are:

1. *FullDiff* – the profiler uses delta calculation, serialization on every call to a remoteable method.

2. *FullSerial* – the profiler uses serialization only on each call to a remoteable method.

3. *LastDiff* – the profiler uses delta calculation, serialization on the remoteable method's *first* call, but then re-uses this estimate without recomputing it on subsequent calls to the method. However, whenever MAUI chooses to offload the method, MAUI updates these estimates.

4. *LastSerial* – the profiler uses serialization on the remoteable method's first call similar to LastDiff.

5. *Oracle* – the profiler knows in advance exactly how much state is transfered for each remoteable method without any computation.

Figure 3.6 shows the effect of MAUI's profiling on the performance of the video game under each of these five scenarios. The additional overhead of calculating deltas reduces the game's frame rate from 10 fps to 4 fps, and the two heuristics, *LastDiff* and *LastSerial*, both provide a significant benefit in terms of interactive performance of the game. *LastDiff* provides more benefit than *LastSerial* because it enables the decision engine to offload certain methods that are not offloaded with *LastSerial*. These results show that our simple heuristic can balance the trade-off between the cost of performing profiling frequently and the freshness of its estimates of application's behavior.

3.5.3 Network Profiling

Initially, we believed that profiling the wireless networking environment would be relatively straightforward. We planned to use Wi-Fi's power-save mode (PSM) to maximize the energy savings when offloading code, and we planned to use network measurement tools to estimate the wireless link's round-trip time, bandwidth [82], and packet loss [86]. Instead, we discovered that both design choices were either wrong or unnecessarily complicated.

First, we discovered that Wi-Fi's power-save mode (PSM) can hurt the energy consumption of code offloading. With PSM, the smartphone's Wi-Fi radio is put to sleep, waking up just before the access point (AP) sends a beacon (every 100ms) to check if the AP has any incoming data destined for the phone. If there is no data sent by the MAUI server available at the AP, then the radio goes back to sleep for another 100ms. Because MAUI uses TCP, PSM's interactions with TCP have some surprising effects in terms of time to complete a transfer, and in terms of energy consumption.

We characterized these interactions by performing a suite of experiments transferring different amounts of data to and from a MAUI server varying the connection's RTT, both with PSM enabled and with it disabled. To illustrate our findings we select two specific examples of a 500 KB TCP transfer over a link with a 25 ms RTT and one with 75 ms RTT with PSM enabled.

Figure 3.7 shows the dynamics of the TCP transfer by plotting the TCP's byte sequence offsets over time. With an RTT of 25 ms, TCP experiences the PSM effects during slow-start, when the TCP sender has a small window size. The TCP sender finishes sending all the segments and then goes to sleep. It wakes up 100 ms later, receives the TCP ACKs and again sends a window worth of segments. However, after about one second, the window has become large enough that the TCP ACKs arrive back at the sender before it has run out of segments to send. This prevents the sender from going into sleep mode; at this point, the TCP transfer behaves similar to one without PSM enabled. Effectively, the transfer experiences an RTT of 100 ms (due to PSM) during the first second and an RTT of 25 ms during the last second. A similar effect occurs when the RTT is 75 ms. However, because the RTT is much closer to 100 ms (the PSM sleep interval), the discrepancy between the first part of the transfer and the second is not so large.

This PSM behavior raises an interesting trade-off. On one hand, with PSM, the sender saves energy by sleeping whenever it has no packets ready to send. On the other hand, PSM increases the duration of a transfer. The device consumes more energy overall when the transfer lasts longer. This trade-off has important consequences for the use of PSM with a system like MAUI. For example, for short RTTs (lower than 75 ms in our experiments), disabling PSM *saves energy* because the transfers are much shorter, and the device consumes less energy overall during the transfer duration. However, when RTTs approach 100 ms, enabling PSM saves energy because the wireless sender is sleeping while waiting for ACKs from the receiver. The same pattern occurs with RTTs higher than 100 ms – during slow start, the RTTs are effectively rounded up to 200 ms.

Unfortunately, we had to leave PSM disabled in our current implementation of MAUI, because Windows Mobile does not expose an API that allows regular applications to enable or disable PSM, we can only turn it on and off through the systems settings.

Second, we found that it was unnecessary to use specialized tools to individually measure the wireless link's round-trip time, bandwidth, and packet loss. Instead, our experiments showed that the following very simple technique works well: send 10 KB of data over TCP to the MAUI server and measure the transfer duration to obtain an average throughput. We use 10 KB because this is representative of typical transfers made to a

MAUI server. This simple approach allows MAUI's network profiler to take into account both the latency and bandwidth characteristics of the network. Each time MAUI offloads a method, the profiler uses this opportunity to obtain a more recent estimate of the network characteristics; these recent estimates are then averaged (we use a sliding window) with the current estimate. Finally, if no transfers are performed for one minute, MAUI conducts another measurement by sending 10 KB to the server to obtain a fresh estimate.

3.6 MAUI Solver

The MAUI solver uses data collected by the MAUI profiler as input to a global optimization problem that determines which remoteable methods should execute locally and which should execute remotely. The solver's goal is to find a program partitioning strategy that minimizes the smartphone's energy consumption, subject to latency constraints.

Deciding where to execute each method is challenging because it requires a global view of the program's behavior. Figure 3.8 shows a simplified version of the call graph for a face recognition application. Each vertex represents a method and its computational and energy costs, and each edge represents the size of the method's state and the energy consumed to transfer this state remotely. For each individual method, its remote execution is more expensive than its local execution, and yet remote execution can save energy if *FindMatch()*, *InitializeFaceRecognizer()*, and *DetectAndExtractFaces()* are *all* remoted. Thus, the MAUI solver's decisions must be globally optimal (i.e., across the entire program) rather than locally optimal (i.e., relative to a single method invocation).

We now describe our formulation of the global optimization problem. At a high level, the execution behavior of the program is modeled as an annotated call graph, and both the graph and all the annotations are provided by the MAUI profiler as inputs to the solver (Section 3.5 describes how these input parameters are estimated). We use a linear program solver to find the optimal partitioning strategy that minimizes the energy consumed by the

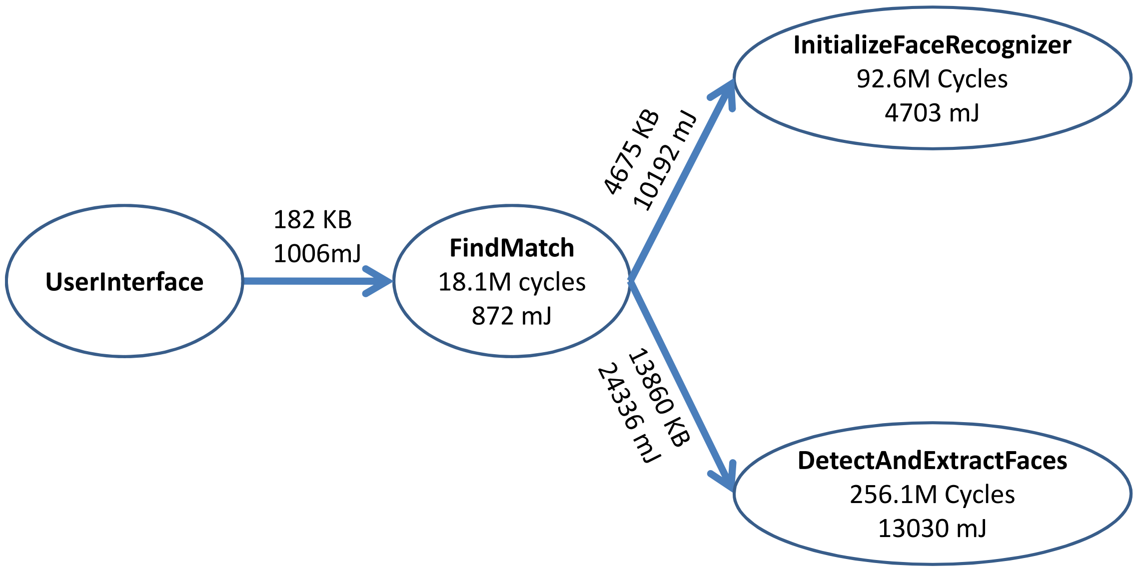

FIGURE 3.8: **The call graph of a face recognition application.**

smartphone subject to a set of latency constraints. To ease the burden on the programmer, we provide a default latency constraint: the total execution latency L must not exceed 5% more than the latency incurred if all the methods in the program are executed locally. There are some applications where absolute latency constraints are important to maintain the application's quality of service. To handle such cases, we optionally allow the application developer to specify additional latency constraints. As the application runs, we re-run the solver periodically for two reasons: to adapt to changing environmental conditions, and also to learn from the historical behavior of the program. The MAUI solver is invoked asynchronously from the mobile device to avoid affecting the interactive performance of the application.

In more detail, we start with the application's call graph $G = (V, E)$. The call graph represents the call stack as the program executes. Each vertex $v \in V$ represents a method in the call stack, and each edge $e = (u, v)$ represents an invocation of method v from method u. We annotate each vertex $v \in V$ with the energy it takes to execute the method locally E_v^l, the time it takes to execute the method locally, T_v^l, and the time it takes to execute the method remotely, T_v^r. We annotate each edge $e = (u, v)$ with the time it takes to transfer the necessary program state $B_{u,v}$ when u calls v, and the energy cost of transferring that state $C_{u,v}$. Each vertex v is also annotated with the parameter r_v that indicates if the method is marked remoteable. If a method is called from within a loop, all these costs are scaled up by a factor that corresponds to the profiler's estimate of the number of loop iterations.

Formally, MAUI solves the following 0-1 integer linear programming (ILP) problem shown below. The solver solves for variable I_v. I_v is the indicator variable: $I_v = 0$ if method v is executed locally and is 1 if executed remotely.

$$\text{maximize} \sum_{v \in V} I_v \times E_v^l - \sum_{(u,v) \in E} |I_u - I_v| \times C_{u,v}$$

$$\text{such that:} \sum_{v \in V} ((1 - I_v) \times T_v^l) + (I_v \times T_v^r))$$

$$+ \sum_{(u,v) \in E} (|I_u - I_v| \times B_{u,v}) \leqslant L$$

$$\text{and} \qquad I_v \leqslant r_v,\ \forall v \in V$$

The first term in the objective function is the total energy saved by executing methods remotely – the savings are essentially the energy cost if the method had been executed locally. The second term in the objective function captures the energy cost of data transfer to execute a method remotely. Note that data transfer incurs an energy cost only if the two methods u and v are not both executed in the same location. The first constraint stipulates that the total time to execute the program be within L. The second constraint stipulates that only methods marked remoteable can be executed remotely.

3.7 Evaluation

In this section, we evaluate MAUI's ability to improve the energy consumption and performance of smartphone applications, using a combination of macrobenchmarks (in Sec-

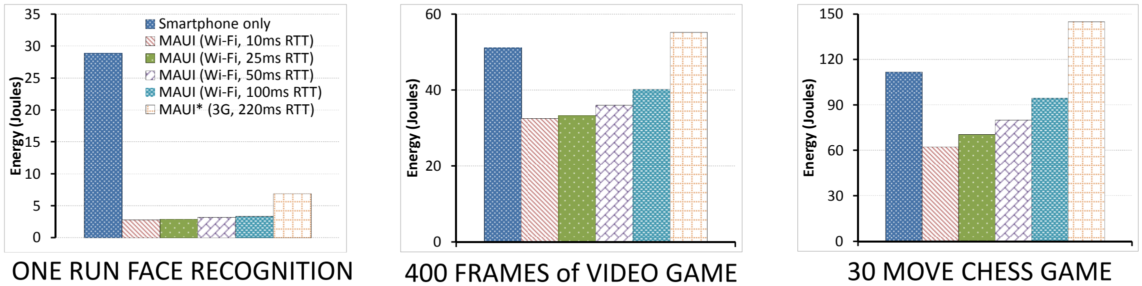

FIGURE 3.9: **A comparison of MAUI's energy consumption.**

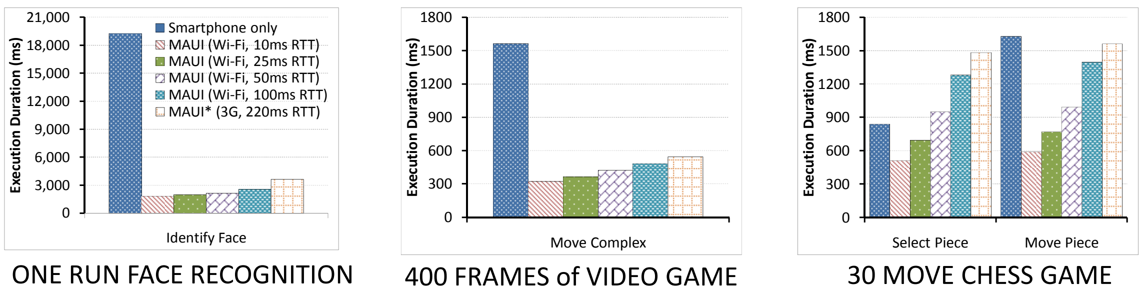

FIGURE 3.10: **A comparison of MAUI's performance benefits.**

tion 3.7.2) and microbenchmarks (in Section 3.7.3).

3.7.1 Methodology

We used an HTC Fuze smartphone running Windows Mobile 6.5 with the .NET Compact Framework v3.5. For the MAUI server, we used a regular dual-core desktop with a 3 GHZ CPU and 4 GB of RAM running Windows 7 with the .NET Framework v3.5. Note that the .NET Compact Framework running on the smartphone is more restrictive than the one running on the MAUI server. The server is equipped with an NDIS intermediate driver that inserts packet queuing delays to control the RTT of the path between the smartphone and the server (without adding any delay, the path between the smartphone and the server has an RTT of 10 ms in our setup). We measure energy on the phone using a hardware power meter [67] attached to the smartphone's battery. This power meter samples the current being drawn from the battery at 5000 Hz.

We evaluate MAUI's benefits on four applications. Three of these applications were already pre-built and running on Windows Mobile phones: a face-recognition application, a highly-interactive video game, and a chess game. Although these applications are relatively simple, they each contain on the order of 10 remoteable methods. We developed the fourth application from scratch – a real-time voice-based language translator (Spanish to English). The language translator must use remote execution to run on the smartphone because its memory resource requirements exceed those offered locally by the smartphone.

3.7.2 Macrobenchmarks

We now look at three macrobenchmarks that characterize MAUI's ability to reduce energy, improve performance, and bypass the resource limitations of smartphones.

1. MAUI's primary goal is to reduce the energy consumption of mobile applications. *How much does MAUI reduce energy consumption of mobile applications?*

2. In addition to saving energy, MAUI can also improve the performance of mobile applications. This is especially important for interactive applications, such as games. *How much does MAUI improve the performance of mobile applications?*

3. MAUI also allows developers to build applications that cannot currently be supported on mobile devices because of their resource requirements. *Can MAUI run resource-intensive applications?*

How Much Energy Does MAUI Save For Mobile Applications?

Figure 3.9 presents a comparison of the energy consumption of three applications (the face-recognition application, the video game, and the chess game) when executing in six different scenarios. In the first scenario, the applications are running standalone on the smartphone. In the next four scenarios, we use MAUI to offload code to a server over a link with different RTT values (10 ms, 25 ms, 50 ms, and 100 ms). In the final scenario, MAUI offloads code over a 3G interface. In the case of the video game and chess, code offload over 3G consumes more energy than local execution, and thus MAUI's optimizer refuses to perform the offload. However, we modified MAUI to force the code offload when running over 3G to measure its energy consumption. We refer to this modification as MAUI* in the Figure. Note that we did not use the voice translation application because it cannot run standalone on the smartphone.

As seen in Figure 3.9, the face recognition application can achieve drastic energy savings when using MAUI. When the server is nearby, the energy savings reach one of order of magnitude; as the latency to the server is increasing, MAUI saves less energy. In fact, the energy consumed when offloading code over 3G is 2.5 times higher than offloading code to a nearby server. The energy savings for both video and chess are less drastic but they remain significant; when offloading to a nearby server, MAUI saves 27% energy for the video game and 45% for chess.

How Much Does MAUI Improve the Performance of Mobile Applications?

We ran the applications in the same six scenarios but instead measured performance rather than energy consumption and we present the results in Figure 3.10. The performance results are also impressive; offloading the code to a nearby server reduces the latency of performing face recognition by one order of magnitude from 19 seconds down to less than 2 seconds. At these performance levels, face recognition becomes an interactive application.

For the video game, MAUI offloads one method (called "MoveComplex"), whereas for chess, MAUI offloads two methods: "Select Piece" and "Move Piece". The results shown are averaged over 20 executions. For the video game, MAUI also provides substantial performance improvements: the nearby MAUI server reduces latency by a factor of 4.8; this latency reduction effectively double the game's refresh rate from 6 frames per second to 13. For chess, offloading using MAUI when the RTT is higher than 25 ms can hurt

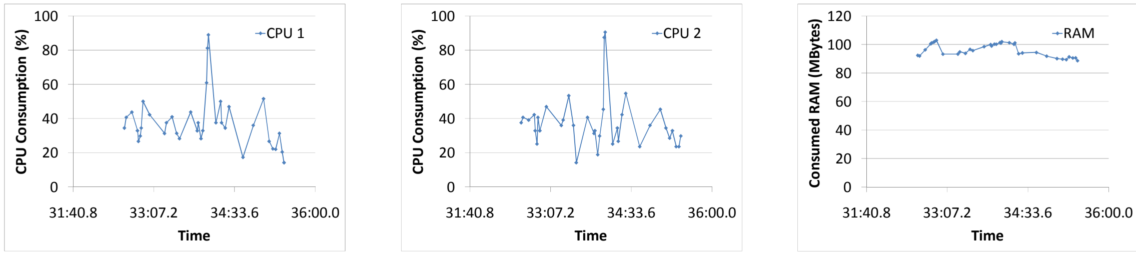

FIGURE 3.11: **Profiling the Speech Recognition Engine on the MAUI Server.**

performance; for example, the "SelectPiece" method takes longer to execute when the RTT is 50 ms. The worst case for performance is offload over 3G: the 'Select Piece" method for chess incurs an 77% performance overhead and the "Fire Bullet" method for the video game incurs a 54% overhead.

Can MAUI Run Resource-Intensive Applications?

Figure 3.11 shows the memory consumption and CPU utilization over time of our translator application when running on a PC with a 2 GHz Intel Core2 CPU and 2 GB of RAM. The peak memory consumption of the speech recognizer is approximately 110 MB, which is higher than the maximum memory limit for a single process on a typical smartphone (32 MB for our HTC Fuze or 48 MB for an iPhone). MAUI enabled us to bypass the smartphone's limits on an application's memory footprint.

3.7.3 Microbenchmarks

We now look at four microbenchmarks that characterize the behavior of both the MAUI solver and the MAUI proxy.

1. MAUI uses an optimization framework to decide what code to offload to a remote server. Upon invoking a function whose code can be offloaded, MAUI solves a 0-1 integer linear problem to decide where the function should execute. *What is the performance overhead of MAUI's optimizer?*

2. Deciding where to execute each method of a program is challenging because it requires a global view of the program's behavior. As Section 3.6 describes (and Figure 3.8 shows), the decision of whether to offload a method depends on more than just its own computational cost and its state. *Is the MAUI solver effective at identifying offload opportunities based on a global view of the program?*

3. The MAUI proxy keeps track of what state it has already sent to the remote server, and it ships incremental deltas to the remote server rather than the entire state for those cases where the deltas are smaller. *How effective are incremental deltas at reducing MAUI's data transfer overhead?*

4. The MAUI solver receives new information from the profiler on a periodic basis as the program executes. *Does the solver adapt to changing network conditions and to changes in the computational costs of individual methods?*

What is the Overhead of MAUI's Solver?

To measure the overhead of MAUI's optimizer, we instrumented the optimizer to record how long it takes to solve each instance of the integer linear programming (ILP) problem. We found that the solver takes 18 ms on average to solve the call graph of the chess application and 46 ms for the video game. Our results show that the ILP problems arising from MAUI's program partitioning needs can be solved quickly.

Does MAUI Require a Global View of the Program to Identify Offload Opportunities?

To answer this question, we constructed a much simpler solver as an alternative to MAUI. This naïve solver considers each method separately; a method is offloaded only if its remote execution (i.e., the cost of transferring its state) consumes less energy than its local execution. We used both the MAUI solver and the naïve solver to solve the call graph shown in Figure 3.8, which is a simplified version of the call-graph for the face recognition application. Figure 3.12 shows that the MAUI solver consumed two orders of magni-

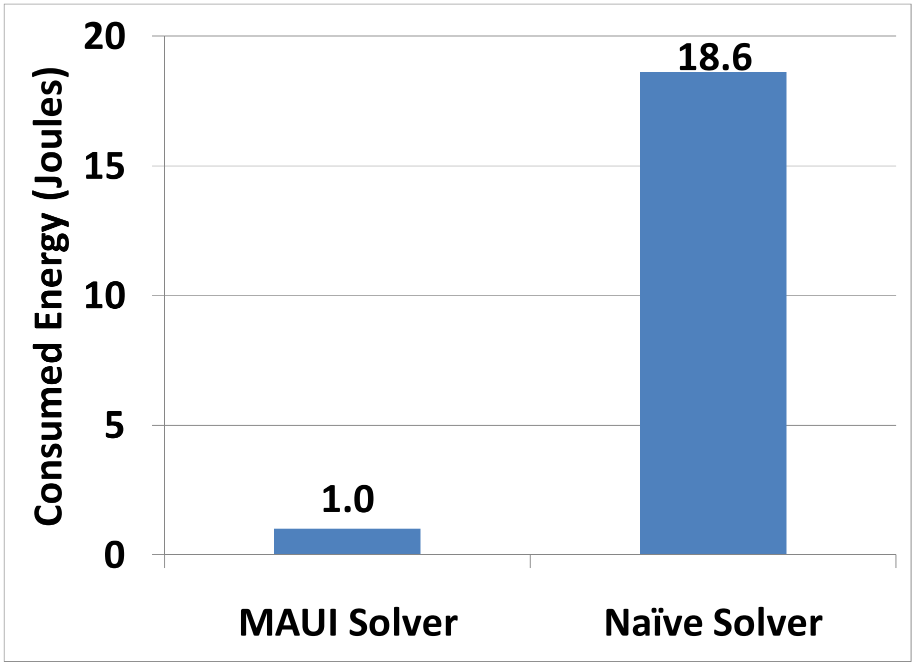

FIGURE 3.12: **The energy consumed by the face recognition application for different code offload solvers.**

tude less energy than the naïve solver. By taking a local view of each method, the naïve solver decided to execute all methods locally, whereas MAUI chose to offload parts of the application.

How Effective are Incremental Deltas at
Reducing MAUI's Data Transfer Overhead?

We instrumented MAUI to measure the amount of data exchanged with a remote server when offloading code both with and without using the incremental deltas optimization. Figure 3.13 shows the benefits of transferring deltas rather than the entire uncompressed state, for the video game for each subsequent execution of the offloaded method. After the first two iterations, when the method is still being initialized, MAUI benefits from using incremental deltas: this optimization reduces the amount of state transferred by a factor of two from 23 KB to 12 KB.

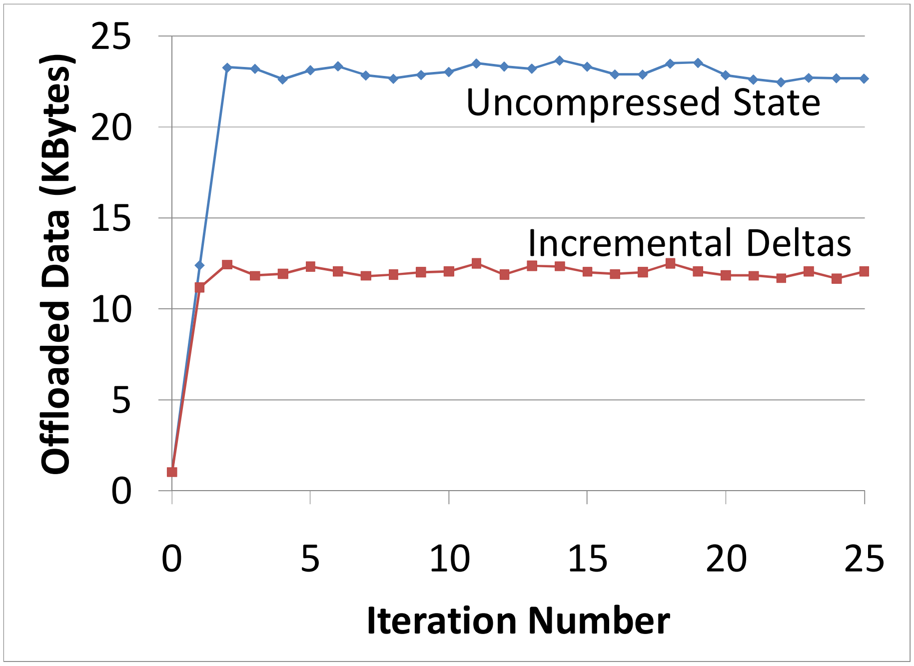

FIGURE 3.13: **The Performance of the Incremental Deltas to and from the Remote Server.**

Does the MAUI Solver Adapt to Changing Network Conditions and CPU Costs?

To answer this question, we examine the decisions that the MAUI solver makes for the modified version of the video game[2]. As we mentioned in Section 3.4.3, we restructured the video game to improve its offload behavior. At the same time, we extended the game's functionality to include physics modeling, using an off-the-shelf physics engine [80]. In particular, we modified the missiles fired by the enemies to act as homing missiles, which means that we increase the chance that the missiles will hit the player. Because the physics calculations are CPU intensive, when the number of active missiles becomes larger, the CPU cost of *HandleMissles()* grows significantly.

The modified game's structure is as follows: there is a top-level method called *DoLevel()* which performs the vast majority of the video game's computation that can be offloaded,

[2] All the results we presented earlier were for the unmodified version of the video game.

because it does not draw directly to the screen. *DoLevel()* in turn invokes *HandleEnemies(), HandleMissiles(),* and *HandleBonuses()*. The size of the game state that needs to be transferred in order to offload *DoLevel()* is just over 11 KB. When MAUI decides to offload *DoLevel()*, it will also offload the *HandleEnemies(), HandleMissiles(), HandleBonuses()* methods because they do not require any additional state to be transferred. Of the three *Handle* methods, only *HandleMissiles()* performs a significant amount of computation because of the physics modelling, and only when there are a moderate number of active missiles. The size of the missile state that needs to transferred in order to offload only *HandleMissiles()* is small: each additional active missile adds approximately 60 bytes of state transfer overhead.

To demonstrate how the solver adapts to changing network conditions and CPU costs, we examine the solver's behavior in two specific scenarios: 1) shortly after the game begins, when there are no active missiles; and 2) after the game has been running for some time and the enemies can fire up to five active missiles. When no missiles are active, MAUI will offload *DoLevel()* only when the round-trip latency to the MAUI server is less than 10 ms. When the latency is greater than 10 ms, no methods are offloaded. After the game progresses and the enemies have five active missiles, MAUI will offload *DoLevel()* when the latency is less than 30 ms, and when the latency to the MAUI server is between 30 ms and 60 ms, it will only offload *HandleMissiles()* but not *DoLevel()*. This is because the state transfer required to offload *HandleMissiles()* is much smaller than that required to offload *DoLevel()*. Finally, when the latency exceeds 60 ms, the solver decides not to offload anything. This example demonstrates that the MAUI solver can make dynamic decisions to save energy that incorporate changes in the network conditions as well as changes in the CPU consumption of individual methods over time.

3.8 Conclusions

In this paper we proposed MAUI, a system that enables fine-grained energy-aware offload of mobile code to the infrastructure. MAUI uses the benefits of managed code to reduce the burden on programmers to deal with program partitioning while maximizing the energy benefits of offloading code. This paper presented how MAUI partitions programs, how it profiled them, and how it formulated and solved program partitioning as a 0-1 integer linear programming problem. Throughout our presentation of MAUI architecture, we also showed several examples of low-level challenges discovered during our implementation. Our results showed that MAUI's energy savings and performance are impressive (up to one order of magnitude for one of our applications).

4

High-Quality Mobile Gaming Using GPU Offload

In this chapter we describe Kahawai[1] – a system that enables mobile devices, such as tablets and smartphones, to perform high-end GPU rendering by offloading a portion of the GPU computation to server-side infrastructure. In contrast with previous thin-client approaches that require the server-side GPU to render the entire content, Kahawai uses collaborative rendering: the output of the mobile GPU and the server-side GPU are combined into the displayed output.

4.1 Introduction

With the advent of consumer mobile devices equipped with high resolution touchscreens and powerful CPUs and GPUs, gaming has become one of the most popular activities on smartphones and tablets. Recent studies estimate that 46% of mobile users play games, and out of the total time spent using their devices, users play games 49% of the time [29]. As device screens grow larger and screen resolutions increase, finer graphical detail and advanced graphical effects are becoming more important for mobile applications, espe-

[1] Kăhăwăĭ is the Hawaiian word for stream.

cially games.

To provide richer visual experiences, mobile devices have seen rapid improvements in their GPU processing capabilities, but today's devices cannot duplicate the sophisticated graphical detail provided by gaming consoles and high-end desktop GPUs. The primary reason for this performance gap is power consumption. A high-end desktop GPU may consume 500 Watts of power, whereas a high-end mobile GPU will consume less than 10 Watts. Furthermore, mobile GPUs will likely always lag behind their desktop contemporaries. The battery capacity of mobile devices is limited and growing slowly, and high power consumption requires sophisticated and bulky thermal dissipation mechanisms that are incompatible with mobile form factors.

Prior research efforts sought to close the performance and energy gaps between mobile devices and server-side infrastructure through remote execution and code offload [18, 15, 6, 25]. However, this work has focused almost exclusively on general purpose workloads running on the mobile device's CPU. Although there has been previous work on remote rendering [58, 91], we are unaware of any prior systems that support GPU offloading from mobile devices to server infrastructure.

Thin-client architectures are another relevant and widely explored area of previous work [55, 89], and commercial systems such as OnLive [75], Gaikai [31], and Ubitus [90] have recently taken a thin-client approach to mobile gaming. Here, all game execution and rendering are performed on a cloud server with a powerful CPU and GPU. The mobile device forwards the user's input to a server, and receives the game's audio-visual output encoded as compressed video. Unfortunately, thin-client gaming has two primary drawbacks. First, transmitting game content that meets gamers' expectations with regards to screen resolution, frame rate, and video quality results in high bandwidth requirements. Second, thin-client approaches cannot support offline gaming since the mobile device must remain connected to the rendering server at all times.

In this paper, we present *Kahawai* a GPU offload system that overcomes the drawbacks of thin-client gaming. The main technique used by Kahawai is *collaborative rendering*. Collaborative rendering relies on a mobile GPU to generate low-fidelity output, which when combined with server-side GPU output allows a mobile device to display a high-fidelity result.

The key insight behind collaborative rendering is that while fine-grained details are prohibitively expensive for mobile GPUs to render at an acceptable frame rate, these details represent a small portion of the total information encoded within a stream. Thus, collaborative rendering relies on the mobile GPU to render low-fidelity output containing most of a stream's content, and relies on the server infrastructure to fill in the details.

Collaborative rendering in Kahawai addresses thin-client gaming's main shortcomings. First, when the mobile device is connected to a server, Kahawai provides high-quality gaming at significant network bandwidth savings relative to a thin-client. Second, disconnected Kahawai clients can play games offline, albeit with reduced visual quality. To enable collaborative rendering, Kahawai requires tight synchronization between an instance of the game executing on the mobile device and another instance of the same game executing on the server. To support this, we integrated partial deterministic replay into a popular open-source game engine, the idTech 4 game engine used by many commercial games.

We develop two separate collaborative rendering techniques as part of Kahawai, with each targeting a different aspect of the game stream's fidelity (i.e., per-frame detail and frame rate). In our first technique, *delta encoding*, the mobile device produces reduced-fidelity output by generating every frame at a lower level of detail. The server-side GPU concurrently renders two versions of the game output: a high-fidelity, finely detailed version, and a low-fidelity version that matches the mobile device's output. The server uses these two outputs to calculate delta frames representing the visual differences between the high-fidelity and low-fidelity frames. The server then sends a compressed video stream of delta frames to the client, and the mobile device decompresses the stream and applies the deltas to the frames that it rendered locally.

In our second technique, *client-side I-frame rendering*, the mobile device produces reduced-fidelity output by rendering highly detailed frames at a lower rate. The server renders the missing frames at a higher rate, and sends the missing frames as compressed video. The mobile device decodes the video, combines the frames in the correct order, and displays the results. Both of our techniques are compatible with the hardware-accelerated H.264 video decoders built in to many of today's mobile devices.

Our Kahawai prototype is integrated into the idTech 4 game engine, and demonstrates the benefits of collaborative rendering using a commercial game, Doom 3, built on this engine. We show that our delta encoding technique, when compared with a thin client using H.264, provides much better visual quality when bandwidth is low (less than 0.5 Mbps). Even more impressive, we show that a thin client requires four times as much bandwidth as client-side I-frame rendering to achieve comparable visual quality.

4.2 Background

Before presenting the design and implementation of Kahawai, we first provide a brief introduction to four topics needed to understand the design space and the techniques we develop in Kahawai. These topics are: cloud gaming; game structure and performance; a primer on video compression; and image quality comparisons.

4.2.1 Cloud Gaming

Cloud gaming is a thin-client model for playing high-end games. In this model, the game executables and content are installed on cloud server machines with high-end GPUs, and the entire execution of the game is performed on a server. The client relays user input to the server over the network, and the server sends back a compressed audio/video stream of the game's output. This places minimal requirements on the client machine – it only needs to efficiently decode the compressed video.

In contrast with a purely local execution, network latency is a key challenge for cloud gaming. End-to-end latency represents the time from when the user performs an input to when the effect of that input becomes visible on the screen. Latency requirements vary depending on the nature of the game, but studies of fast-action games have shown that users can tolerate end-to-end latencies on the order of 100 to 200 milliseconds [94, 56].

Fortunately, modern desktop GPUs can produce game output quite quickly (response times are typically on the order of 15 to 30 ms), which leaves a budget of somewhere between 70 and 185 ms for network overhead. Cloud gaming services such as OnLive [75] distribute their datacenter infrastructure across multiple geographic locations to meet these latency requirements [57].

Beyond latency, cloud gaming also has significant bandwidth requirements, particularly for mobile clients. For modern games, these requirements may be even larger than that of video streaming services, because gamers expect frame rates to be near 60 frames per second (FPS) and screen resolutions can even go beyond HD resolution of 1920 x 1080. In practice, services like Onlive offer frame rates of 60 FPS at a resolution of 1280 x 1024. OnLive currently lists their recommended network requirements as “5+ Mbps”.

4.2.2 Game Structure and Performance

Many modern games are built around a game engine. A game engine provides the primary form of code reuse in game development – using a game engine greatly simplifies the task of rapid game development. Game engines typically separate game content such as artwork, levels, characters, and weapons, from the core functionality of the game execution such as rendering, physics, sound, and input. Two of the most popular game engines for fast-action games are the Unreal engine [23] and the idTech engine [46]. We have implemented Kahawai as a set of modifications to version 4 of idTech, which was released as open-source. Although our changes require access to the engine source code, this does not limit the generality of our approach. When a game developer decides to adopt a particular game engine, they typically obtain a license that includes access to source code.

To implement Kahawai's delta encoding, we need to produce both high-fidelity and low-fidelity versions of the game output. Thankfully, desktop PC games commonly provide a large number of game settings that can be used to control both the performance and the visual quality of the rendered graphics. Game designers expose these settings to players so that those with a more powerful GPU can experience a higher-quality version of the game, while those with less powerful GPUs can play at an acceptable frame rate. Game settings can be used to control a wide variety of graphical details within the game, including lighting, shadows, fog, texture compression, bump mapping, anti-aliasing, anisotropic filtering, and even the complexity of the 3D models given to the rendering pipeline.

The standard APIs used to program modern GPUs are OpenGL [76] and Direct3D [63]. These frameworks describe how a given 3D scene should be rendered, in a way that abstracts away the hardware details of how different GPUs handle the rendering. Hardware manufacturers implement support for these high-level APIs in a way that best exploits the characteristics of their hardware. Each implementation is provided as a binary device driver from the GPU manufacturer [8].

Given the exact same scene description, rendering APIs do *not* guarantee pixel-exact outputs for different GPUs. In fact, we observe pixel-level differences even for two different GPUs produced by the same manufacturer and using the same device driver. Thankfully, these differences are usually imperceptible to the human eye and have a negligible effect on the quality of a combined image. However, even subtly divergent outputs can impact Kahawai's efficiency, as we explain and quantify in the following sections.

To provide the appearance of consistent and smooth motion, most common real-time rendering applications such as games attempt to render between 30 and 60 frames per second. In practice it is hard to define the exact point where a frame rate is high enough to avoid the perception of flicker. However, even at high illumination perceived quality plateaus at around 60 Hz [40].

4.2.3 Video Compression Primer

Cloud gaming and video streaming services rely heavily on modern video compression techniques to reduce the amount of bandwidth required. Kahawai likewise uses video compression to reduce network usage. In this section, we provide a brief primer on the key concepts of video compression used in building Kahawai. A more detailed description of these concepts can be found in [83].

Video compression takes advantage of two common characteristics of real-world videos: strong temporal and spatial locality. Temporal locality refers to similarities among contiguous frames within the same video. For example, scenes of an object in motion will preserve many of the pixels found in previous frames with small changes in their position. Similarly, gradual changes in color or illumination preserve most of the information from the previous frame, with small changes. Spatial locality refers to similarities among contiguous sets of pixels within the same frame. Examples of spatial locality are gradual texture variations within a scene, and shading of uniformly colored objects or backgrounds. Both forms of locality create opportunities for very high compression ratios.

Video encoders use the temporal and spatial locality characteristics of video through a prediction model. This model reduces redundancy by predicting future frames within a sequence based on information found in previously coded frames (temporal prediction), or by predicting future image samples within a frame using previously coded samples (spatial prediction). The prediction is then subtracted from the actual data, resulting in a set of residuals that shrink as the prediction becomes more accurate.

The resulting residuals undergo lossy compression through a process known as *quantization*. This process offers a tradeoff at encoding time between size and quality. In general, scenes presenting strong locality result in better predictions and therefore smaller video representations for a given quantization parameter.

Compressed video is composed of three types of frames: Intra-coded I-frames, Predicted P-frames, and Bi-directional Predicted B-frames. An I-frame is coded without referencing other frames. P-frames are coded using a temporal prediction from one (or more) reference frames that precede it. Finally, B-frames are a special type of P-frame. These frames are coded with temporal prediction using reference frames displayed either before or after it. While an I-frame resembles a typical lossy compressed image, a P-frame is formed by the set of compressed residuals and motion vectors required to recreate the source frame. The sequence of residuals forming each frame are entropy coded [87] and augmented with metadata that will assist the decoder in determining the encoding settings.

4.2.4 Comparing Image Quality

The collaborative rendering techniques we develop in this paper modify the pixel content of the game's output. For any technique that makes such modifications, we need to objectively evaluate the impact of those changes on visual quality. Fortunately, the designers of lossy video compression techniques face the same challenge. A number of previous metrics, such as peak signal-to-noise ratio (PSNR) and mean-squared error (MSE), have been designed to measure visual quality. However, such metrics have proven to be inconsistent with the human eye's perception. Two recent metrics take into account how the human eye perceives differences in image quality – these are structural similarity (SSIM) [92] and perceptual evaluation of video quality (PEVQ) [47]. In this paper, we use SSIM to evaluate, on a frame-by-frame basis, the impact on image quality of our transformations. SSIM scores of greater than 0.9 indicate a strong level of similarity between two images (an SSIM score of 1 indicates no differences).

4.3 Kahawai Architecture

Our primary goal in developing Kahawai is to faithfully reproduce the user experience of having a high-end GPU on a mobile device, while avoiding the drawbacks of a pure thin-client model. In this section, we first provide a brief overview of some of the design alternatives we considered and our reasons for discarding those alternatives. We then provide a high-level view of the two collaborative rendering techniques we have developed for Kahawai, and we summarize the requirements of each.

4.3.1 Design Alternatives

We initially considered a fine-grained approach to offloaded rendering, similar to a number of previous offload systems for CPU-intensive workloads [6, 15, 18]. Our goal was to find ways of dividing up responsibility for rendering different objects in a given scene, and to split the work of rendering those objects across the mobile GPU and the server GPU. However, we quickly ran into two roadblocks with this approach.

The first roadblock arises from the structure of the GPU rendering pipeline combined with the workload characteristics of high-end games. At a high-level, the graphics pipeline consists of a set of stages early in the pipeline that manipulate the scene as a set of polygons, and a set of stages later in the pipeline (i.e., after rasterization) that manipulate the scene as a set of pixels or fragments. One problem with fine-grained partitioning is that if one splits up the objects in a scene and then executes the entire graphics pipeline on both the mobile and server GPUs, there is no obvious way to merge the two rasterized results back into a single scene.

Furthermore, there does not appear to be a good way to partition the pipeline itself, i.e., to assign responsibility for specific pipeline stages to a server GPU and others to a mobile GPU. This is because of the workload characteristics: high-end games use a large number of polygons in each scene, which means that splitting the pipeline at any stage before rasterization would be prohibitively expensive in terms of network bandwidth between the client and server. After rasterization, splitting the pipeline would require a transmission larger than the entire frame buffer (because fragments include additional information such as depth), and, as a result, would cost as much bandwidth as the thin-client model.

The second roadblock to the fine-grained approach is due to software engineering. The

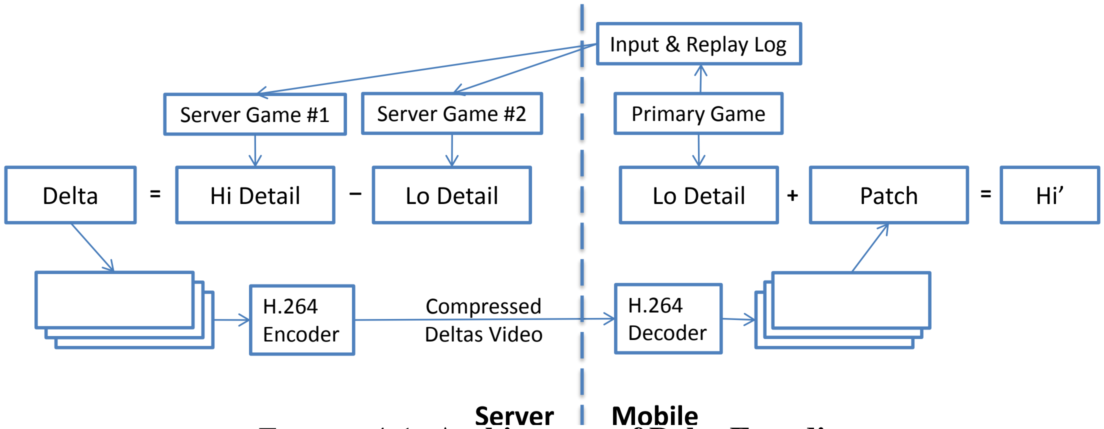

FIGURE 4.1: **Architecture of Delta Encoding.**

device drivers that implement the OpenGL or Direct3D graphics pipeline for a vendor's GPU hardware tend to be binary drivers with no visibility into their internal data structures or the GPU hardware implementation of the graphics pipeline. The internal graphics pipelines for different hardware manufacturers are also very different, further complicating the task of reconciling GPU state across machines. As a result, it appears to be very difficult to experiment with fine-grained partitioning using realistic game workloads.

As a result of these roadblocks, we accepted that the graphics pipelines on both the mobile GPU and the server GPU must execute without modification, and developed an alternative approach called *collaborative rendering*. The key observations behind collaborative rendering are that (1) unmodified graphics pipelines can generate outputs of varying fidelity, and (2) the low-fidelity output of mobile GPUs capture most of the information contained in a more powerful GPU's high-fidelity output. Thus, under collaborative rendering, a client generates low-fidelity output using its mobile GPU, relies on the server to compute and efficiently deliver the missing details over the network, and combines those details with its low-fidelity output to display a high-fidelity result.

4.3.2 Kahawai Overview

We now present the two approaches to collaborative rendering that we have developed for Kahawai. While both techniques are instances of collaborative rendering, they reduce the fidelity of a mobile GPU's output along different dimensions: under delta-encoding the mobile GPU renders low-detail frames at a high rate, while under client-side I-Frame rendering the mobile GPU renders high-detail frames at a low rate.

Figure 4.1 shows the architecture for delta-encoding. The basic idea is to utilize game settings to render two versions of the game output: a high-detail and a low-detail version. We exploit the graphical similarities between these two versions to compute a highly compressed video of the differences. To accomplish this, we execute one copy of the game on the mobile device and use the mobile GPU to render the low-detail output. The server must run two instances of the game, one for each level of detail. To calculate delta frames, the server computes a pixel-by-pixel visual difference between its high-detail and low-detail frames called a delta frame.

We use *partial deterministic replay*, described in Section 4.5, to control concurrent instances of the game running on the server. The server sends a compressed H.264 video composed of delta frames, and the mobile device decompresses that video and applies each frame in the video as a patch to each low-detail frame it renders with its mobile GPU. The end result is an image stream that nearly duplicates the high-detail output rendered on the server.

Figure 4.2, shows the architecture for our second technique, client-side I-frame rendering. For this technique, the client generates a high-detail version of the game output, but at a significantly reduced frame rate. We use the server to also generate a high-detail version of the game output, but the server GPU is fast enough to produce this output at a higher rate. The server generates an H.264 compressed video of the game output, but then discards the I-frames from that video before sending it over the network. The mobile client

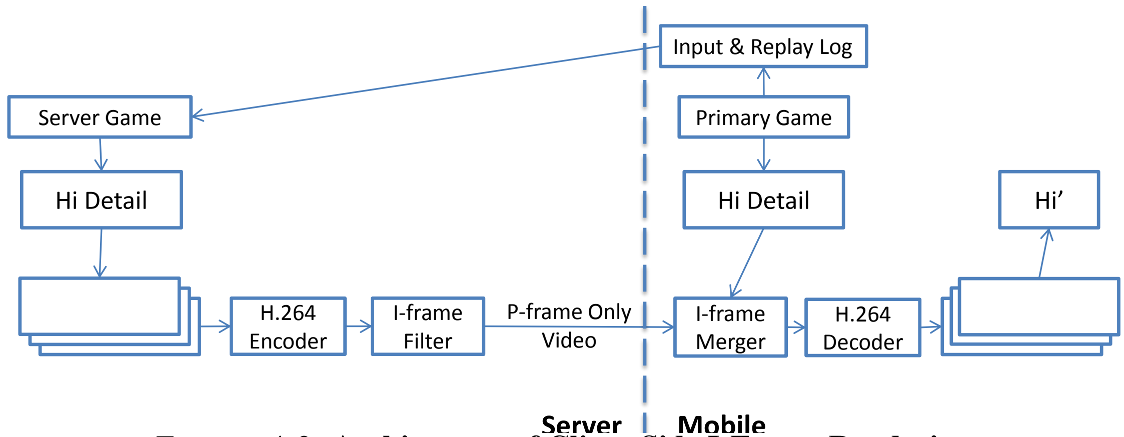

FIGURE 4.2: **Architecture of Client-Side I-Frame Rendering.**

receives the compressed video containing only P-frames, and uses the frames rendered by its mobile GPU to merge the missing I-frames back into the H.264 video stream. The mobile device then uses the H.264 decoder to display the final output. The end result, as with delta encoding, is graphical output that nearly duplicates the high-detail, full frame-rate version rendered by the server GPU. Although we think it is possible to combine our two techniques, we have not done so and leave it for future work.

4.3.3 Requirements

We now explain the requirements for using each of these techniques. Both techniques require that the compressed video sent over the network must be compatible with the H.264 standard. H.264 is the standard video encoding for mobile devices, and almost all mobile devices support hardware-accelerated H.264 decoding. Hardware acceleration improves both the performance and energy-efficiency of decoding, and it allows the main CPU to focus on executing the game logic.

Both techniques also require frame-by-frame synchronization. Specifically, because we run multiple instances of the game, each rendered object must be in the same position within a given frame in all executions. Low-detail versions may have missing objects found in a high-detail version, but the positions of objects common to both versions must match exactly. We describe in Section 4.5 how our implementation supports this requirement.

Our delta-encoding technique places two specific requirements on the game itself (as opposed to the game engine). First, the game must provide configuration settings that allows the mobile device to render a low-detail version of the game output at an acceptable frame rate. We view 30 FPS as the minimum acceptable frame rate, and 60 FPS as the maximum. Second, the H.264 video composed of delta frames must be smaller than an H.264 video of the high-detail output. The size of the difference represents delta-encoding's bandwidth savings over standard thin-client gaming.

Client-side I-frame rendering also places a requirement on games. The mobile device must be able to render high-detail frames fast enough, but without impacting the game's responsiveness. Consider, for example, a mobile device that can generate 4 FPS of high-detail frames. Even if the server GPU is fast enough to generate the missing 56 FPS, the latency of generating each of those 4 frames on the client will be about 250 ms per frame. As mentioned earlier, latencies above 200 ms negatively impact users' experience of most fast-action games. We estimate that a mobile device must generate at least 6 FPS of high-detail output for client-side I-frame rendering to be beneficial.

4.4 Kahawai Design

In this section, we describe the design details of Kahawai's two collaborative rendering techniques and characterize when they are most effective.

4.4.1 Delta Encoding

Our delta encoding technique relies on adjusting a game's settings to produce both high-detail and low-detail versions of the game's graphics output. The high-detail version should contain all of the game's visual effects at their maximum level of detail. The low-detail version should enable only those settings that allow the mobile GPU to reach an acceptable frame rate, somewhere between a 30 and 60 FPS.

In Section 4.3, we outlined the basic steps involved in delta encoding. We now detail the process of encoding delta frames using H.264 compression, with a focus on the impact of H.264's lossy compression.

The primary benefit of delta encoding is that the server should have to send less information to the mobile client compared to the thin-client approach of encoding a high-detail version of the game output using H.264. In the common case, the bandwidth savings of delta encoding will be significant, although there are circumstances in which delta encoding will generate more traffic than a thin-client approach.

H.264 is a lossy compression scheme specifically designed to provide high compression ratios for video. At first glance, it is not obvious that H.264 would be effective at compressing a video composed of delta frames. However, we find that delta frames exhibit strong temporal locality in much the same way that the original video frames have temporal locality. As long as this holds, motion compensation can achieve high compression ratios.

Lossy compression is another key aspect of H.264. H.264 is designed so that lost information is difficult for the human eye to perceive. The information loss is a result of the transformation and quantization process in the encoder, where individual pixel values that define each frame's video signal are rounded to improve the compression ratios. These losses introduce two challenges when transmitting a video of delta frames: (1) quantization of the delta values introduces visible visual artifacts when the delta is patched into low-detail frames on the mobile device, and (2) the encoded delta does not provide a pixel-exact representation of the difference between a high-detail and low-detail frame.

The challenge of visual artifacts is due to the extended value range that deltas encode. For each pixel, the delta represents the difference between the high-detail value and the low-detail value. The difference can be positive or negative, requiring an extra bit to encode the sign information. Without quantization, one could use modular arithmetic as a simple workaround. Consider an example: if the maximum channel value is 255, high = 10, and low = 253, then delta = -243, which means one could also represent the delta as +12.

However, quantization on the deltas introduces a problem. Consider another example: suppose high = 2 and low = 253, so that the delta = 4 with modular arithmetic. During quantization, the delta is rounded from 4 to 2. When applying the patch, low (253) + delta (2) = 255 which means that high is now 255. With quantization of the delta, we have converted a very dark pixel value into a very bright pixel value, which creates a visible artifact.

We use a simple solution to correct this problem. Before the invoking the H.264 encoder we apply the transformation in Equation 4.1.

$$\Delta = \frac{Hi - Lo}{2} + 127 \tag{4.1}$$

This has the effect of dropping the least significant bit in favor of the sign bit. After H.264 decode, we apply Equation 4.2 before we apply the patch.

$$Hi = Min(2(\Delta - 127) + Lo, 255) \tag{4.2}$$

Note that this solution is not new: Levoy's work suggested the same transformation [58]. One consequence of this solution is the loss of one bit of information in the deltas. However, this information loss is small and comparable to the information loss that comes from quantization noise in H.264.

The quantization mechanisms in H.264 were designed for regular video sequences and not sequences of deltas. Despite the similarities between delta and regular video frames, deltas could benefit from a custom quantization process specifically tailored to their signal characteristics. For delta frames, most of the pixel values encoded using Equation 4.1 will fall around 127 representing small differences between the two sources. Because H.264

allows for custom scaling quantization matrices, this could be used to offset the loss of information caused by dropping the least significant bit.

The second challenge with delta encoding is that the encoded delta does not completely represent the difference between the high detail and low detail frames, due to the lossy compression. At a fixed bitrate, the information loss is inversely proportional to the similarity of the high and low detail frames. A delta between two relatively similar frames will result in a smaller distribution of delta values, and will therefore suffer from less quantization noise than a delta from dissimilar frames. In our experiments, when the difference between the high and low detail versions scores less than .8 on the SSIM [92] similarity metric, our delta encoding technique can result in either higher bandwidth or worse quality than an H.264 encoding of the high-detail output.

4.4.2 Client-Side I-Frame Rendering

Our second collaborative rendering technique is client-side I-frame rendering. For this technique, the client uses its mobile GPU to render high-detail frames at a low rate, and the server renders high-detail frames at full frame rate. The server sends a compressed video with only P-frames, and the mobile device inserts its rendered frames into the compressed video as I-frames.

As was the case with delta encoding, H.264's quantization has an effect on client-side I-frame rendering. As the level of quantization increases, an encoded frame becomes less similar to its original source. As a result, P-frames based on compressed reference frames have to encode more information to make up for this loss. These P-frames must convey both the changes of the current frame and make up for any losses from previously compressed frames.

If the encoder is configured to output a stream of a given size, then frames will be encoded using heavier compression with greater information loss. Information loss will accumulate across frames until another I-frame arrives in the sequence. This loss-accumulation phenomenon is similar to temporal error propagation [14], in which frames dropped from a streaming video affect contiguous frames in the sequence. Because of this phenomenon, having I-frames appear more frequently in the sequence leads to higher-quality P-frames under the same level of quantization. Unfortunately, P-frames have a much better compression rate than I-frames, and increasing the frequency of I-frames increases the overall size of the encoded video stream.

Client-side I-frame rendering relies on a client's mobile GPU to render high-detail frames at a low rate. The server is responsible for supplementing the missing frames needed to create a smooth scene (i.e., between 30 and 60 FPS). The server renders all high-detail frames in the sequence, and encodes the frames that the client will also render as I-frames. However, before sending the final stream over the network, each client-rendered frame is removed. As a result, the server's encoder can generate high-quality P-frames using less information. This has two effects: (1) the size of the video transfer is reduced by a factor proportional to the number of frames rendered in the client, and (2) the size of each transferred P-frame is reduced.

A final potential benefit of client-rendered I-frames is that they are perfect reference frames since they will not be quantized. This allows P-frames referring to client-rendered I-frames to be encoded using fewer bits, leading to better image quality for a given bitrate. However, periodically switching between perfect I-frames and quantized P-frames can introduce undesirable visual effects. Similarity metrics do not capture these effects, but the human eye can be sensitive to subtle changes in quality, resulting in a degraded visual experience. There are two simple ways to address this problem. The first is to even out I-frame quality by quantizing client-rendered frames. This would slightly reduce overall video quality, but improve user experience in some cases. The other approach is to switch at irregular intervals to make the switching effect less apparent.

Finally, it is also possible to achieve a desired frame rate at the expense of a moderate reduction in quality by decreasing the game settings used to render the high-detail version.

Doing so would allow the mobile device to render I-frames at a higher rate. If this is not enough to meet the delay and bandwidth constraints of the game, then Kahawai considers the game to be unsuitable for client-side I-Frame rendering and would be better handled by an alternative technique.

4.5 Implementation

To evaluate our collaborative rendering techniques, we built a prototype on top of the open source version of the idTech 4 Engine, also known as the Doom 3 engine (named after the first game built on it) [45, 46]. The idTech 4 engine is implemented in C++ and uses OpenGL for graphics. Besides Doom 3, other commercial games have been built on the idTech 4 engine, including Quake 4, Wolfenstein, Prey, and Brink. In this section, we describe how we addressed the two key implementation challenges: (1) supporting partial deterministic replay, and (2) implementing the codec support for our two collaborative rendering techniques.

Previous work has suggested using deterministic replay as part of a computational offload system [25]. One serious problem with deterministic replay is the high overhead (especially on a multi-processor) – slowing down the game could lead to a poor user experience. In this work, we develop support for *partial* deterministic replay within the idTech 4 engine. We do not need to perfectly replay the entire behavior of the game, we simply need to ensure that the frame-buffer output behavior of the game is deterministic for a given set of inputs and a given configuration of the game settings. In fact, full deterministic replay is not compatible with our delta encoding technique, because we need to replay the game with different internal settings (which leads to different OpenGL calls).

4.5.1 Deterministic Graphics Output

All interactions with the OpenGL graphics subsystem in the idTech engine are performed from a single thread, which we call the render thread. At a high-level, the main loop for the render thread looks like:

```
while (1) {
    Frame();          // update the game state
    UpdateScreen();  // make calls to OpenGL, then
                     // glFinish() and SwapBuffer()
}
```

The call to *Frame* is used to update all of the relevant game state, and the call to *UpdateScreen* is used to render those updates onto the screen. In particular, for objects that are in motion, the game engine uses calls to *GetSystemTime()* inside *Frame* to determine how much time has elapsed since the last loop iteration. This elapsed time is used to determine how far to move those objects within the scene.

Our goal for partial deterministic replay is to ensure that, for a particular loop iteration rendering frame N, the sequence of OpenGL calls (and their parameters) that occurs within UpdateScreen for frame N are identical across the different executions of the game. To verify that we have succeeded at implementing partial replay correctly, we run the game multiple times using a specific input log and a specific configuration of the game settings. We then use glReadBuffer to grab the rendered output of each frame for each of the different executions of the game, and we confirm that there are no pixel differences.

With the idTech engine, it turns out that there are three sources of non-determinism that can affect the visual output of the rendering thread. The first source is system time: we intercept all calls to GetSystemTime from the render thread, and we ensure that the replayed versions see the deterministic behavior. The second source is the pseudo-random number generator. Interestingly, we did not need to intercept these calls because of the unusual configuration of the random number generator. Simply replaying the keyboard and mouse input correctly ensures that random number generation is deterministic. The last source of non-determinism comes from music: the rendering thread reads directly from the sound buffer so that the music being played affects the illumination of the scene. To work around this problem, we set a configuration flag to tell the engine to use system time instead of the sound buffer.

Finally, our current Kahawai prototype cannot handle mid-game disconnections or reconnections with the server, but this is not a fundamental limitation of our approach. Adding these features should be straightforward, since the client can simply fall back onto displaying low-detail frames at a high rate upon disconnection or transition into either collaborative rendering technique once reconnected.

4.5.2 Codec Support

To implement both collaborative rendering techniques, we need to intercept and capture the GPU output on a frame-by-frame basis. To build this component, we used techniques similar to those found in game video recording software, such as Fraps [30]. We add hooks to graphics API calls associated with a frame being rendered to the screen (such as GLSwapBuffer), and we dump the contents of the framebuffer to memory.

Once we have captured frames from the GPU or received frames over the network, we send them to our collaborative video codec. For delta encoding, two simultaneous instances on the server intercept frames, and share the frames through memory mapped files. A delta is computed between the two frames, transformed as described in Section 4.4.1, and then encoded using x264 [95]. On the mobile side, we use ffmpeg [24] to decode, apply the correction as described in Section 4.4.1 and then apply patches to captured frames from the mobile GPU.

For client-side I-frame rendering, the trick is to configure x264 to generate I-frames at predictable intervals. We then filter the encoded video to discard the I-frames, while sending the compressed P-frames along with placeholders for the missing I-frames. On the mobile side, we use frame capture to extract the I-frames and insert them into the video stream, and send the video to ffmpeg for decoding.

4.6 Evaluation

In this section, we characterize the effect of Kahawai's collaborative rendering techniques on bandwidth and visual quality when compared to a thin-client approach.

Table 4.1: x264 Encoder Settings.

deltas, *thin-client*, and *iframes*	
Defaultpreset	medium/zerolatency
i_threads	9
b_sliced_threads	1
i_width	width
i_height	height
i_fps_den	1
i_keyint_max	60
b_intra_refresh	0
rc.i_rc_method	X264_RC_CRF
rc.f_rf_constant	20,25,30,35
rc.f_rf_constant_max	25,30,35,40
b_repeat_headers	1
b_annexb	1
iframes only	
i_scenecut_threshold	0
b_intra_refresh	0
param.i_keyint_max	_iFps
i_type (for I-frames)	X264_TYPE_IDR
i_qpplus1 (for I-frames)	1
i_type (for P-frames)	X264_TYPE_PK
i_qpplus1 (for P-frames)	X264_QP_AUTO

4.6.1 Methodology

Our Kahawai server was a custom gaming machine with a six-core 3.2Ghz Intel Core i7-3930K, 16GB of RAM and two Nvidia GTX 580 GPUs in SLI. Our mobile client was a five-year old Dell XPS M1330 with a 2.1Ghz Core 2 Duo and a Nvidia 8400M GS GPU. We chose this client because its specifications are comparable to a late-2012 mobile device [20]. This machine also has the earliest NVidia GPU to support full hardware H.264 video decoding and the latest "Verde" driver.

It is important to note that while there is no technical reason why our Kahawai client could not take advantage of H.264 hardware decoding, our current implementation does not. As a result, our current client implementation supports a maximum frame rate of 48 FPS under collaborative rendering. This frame rate is well within the range of 30 to 60 FPS considered acceptable by game players, but achieving a full 60 FPS will require hardware decoding.

For our experiments, we used our Kahawai prototype, based on the idTech 4 engine, and the game Doom 3, running under Detours binary interception [43]. Doom 3 input was generated using idTech 4's demo recording; we used the well known "demo1" script of a player moving through a dimly lit environment (*demo1*), a script of a player moving slowly through a brightly lit and subtly textured environment (*light room*), and a battle agains the final boss featuring varied lighting conditions and a complex composition (*cyberdemon*). We chose these scenes to expose Kahawai to a range of graphical content. Each scene contained 500 frames with 1024 x 768 resolution. For our iframe experiments, the client generated 12 I-frames per second.

For our Doom 3 experiments, we compared three approaches to mobile gaming: delta-encoding (*deltas*), client-side I-frame rendering (*iframes*), and H.264 streaming (*thin-client*). As mentioned before, our Kahawai client prototype currently supports a maximum frame rate of 48 FPS. As a result, the recorded scenes take slightly longer to be replayed but no frames are skipped. For our experiments, we computed the bitrate used as if the client was rendering 60 FPS. We estimated the replay time as the total number of frames divided by 60, and we estimated the bitrate as the total size of the video feed divided by the estimated time.

We configured the idTech 4 engine using two graphic configurations. For our high-detail settings, we used the "Ultra" quality preset, while for our low quality settings we used a modified version of the "Low" preset with bump mapping disabled. H.264 video encoding can be configured using a wide range of parameters, but we chose a single typical configuration for our experiments, which remained constant throughout. Table 4.1 summarizes our settings for the x264 encoder.

4.6.2 Doom 3 Experiments

Given this setup, we sought answers to following questions:

1. For a target encoding quality, how do Kahawai's collaborative-rendering techniques' bandwidth requirements compare to a thin-client approach?

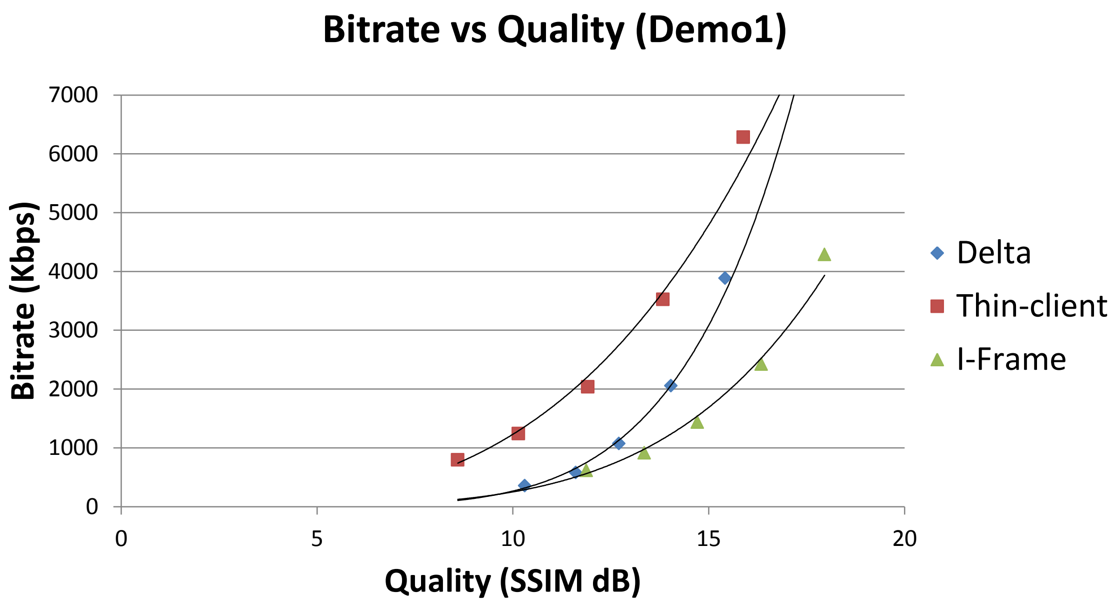

FIGURE 4.3: **Bitrate vs quality for the built-in demo1 in Doom 3.**

2. For a target encoding quality, how do Kahawai's collaborative-rendering techniques' visual quality compare to a thin-client approach?

3. In Delta encoding, how does the client rendering quality affects bitrate?

4. In Client side I-Frame encoding, how does the client I-frame rate affects bitrate?

To answer these questions, we played our two scenes (dark and bright) using three approaches (deltas, iframes, and thin-client) while varying the H.264 constant ratefactor (CRF). Specifying a particular CRF value sets the target quality for a video stream, which the encoder tries to achieve at a minimum bitrate (i.e., the required bandwidth). Lower CRF values correlate with higher target quality, and higher CRF values correlate with lower target quality.

The encoder reduces the required bandwidth to achieve a target quality by opportunistically dropping information from fast-moving frames. The idea is that when frames change quickly, losing information will be less noticeable to the human eye. The higher the CRF, the more aggressively the encoder will drop information. Thus, CRF will impact both the measured quality of each frame (i.e., SSIM) and the size of the video stream (i.e, required bandwidth).

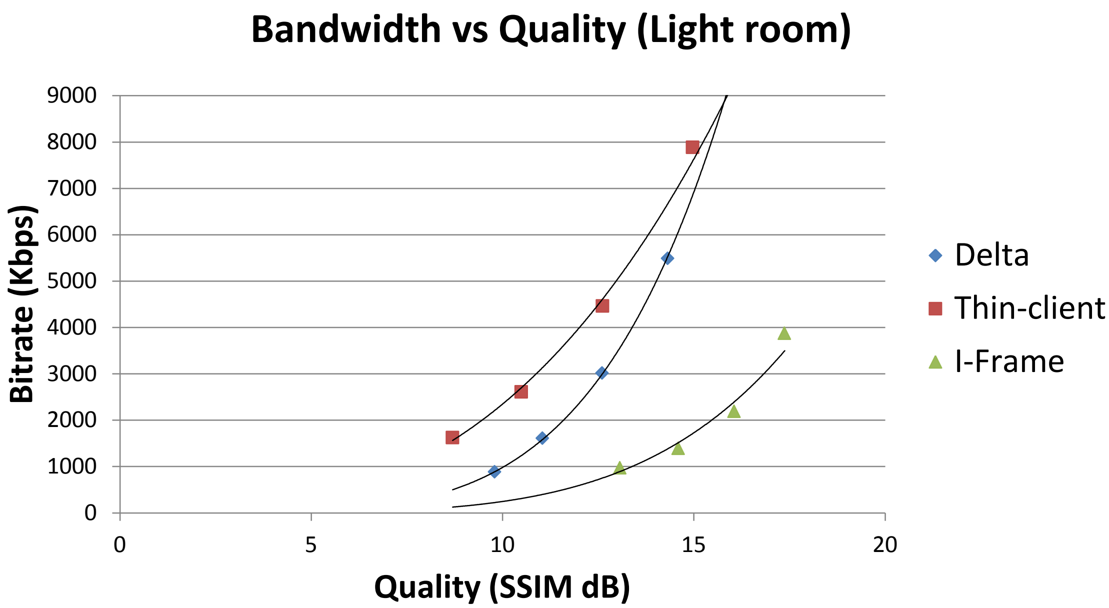

FIGURE 4.4: **Bitrate vs quality for a bright scene in Doom 3.**

The default CRF value for x264 is 23. For our experiments, we used CRF values of 20 (highest target quality), 35 (middle target quality), and 35 (lowest target quality). During our experiments, we used SSIM to measure the actual quality of each frame by comparing the frame the client displayed to the lossless original generated by the server. Since a SSIM score of 1 indicates a perfect match, and a score of .9 indicates strong similarity, and everything below .9 suggests a low quality match, we used SSIM with the deciBel[1] logarithmic scale. This scale narrows our area of interest to values between 10 (SSIM of .9) and 20 (SSIM of .99). We also measured the bandwidth between the client and server.

Figures 4.3, 4.4 and 4.5 show our bitrate and SSIM results for each technique in the demo1, light room and cyberdemon scenes, respectively. Each line represents a different technique, and each point on the line represents a different CRF value. The top-right most point for each line represents the average SSIM and Bitrate for CRF=20, the following point represents average SSIM and Bitrate for CRF=25, the second to last to the left represents CRF=30 and the left-bottom most point represents average SSIM and Bitrate for CRF=35. It is important to notice that using a higher CRF does not mean that any stream encoded with it will have a lower bitrate, it just means that the compression factor is higher. For example, in figure 4.4, the bitrate of the I-Frame stream at CRF=20 is lower

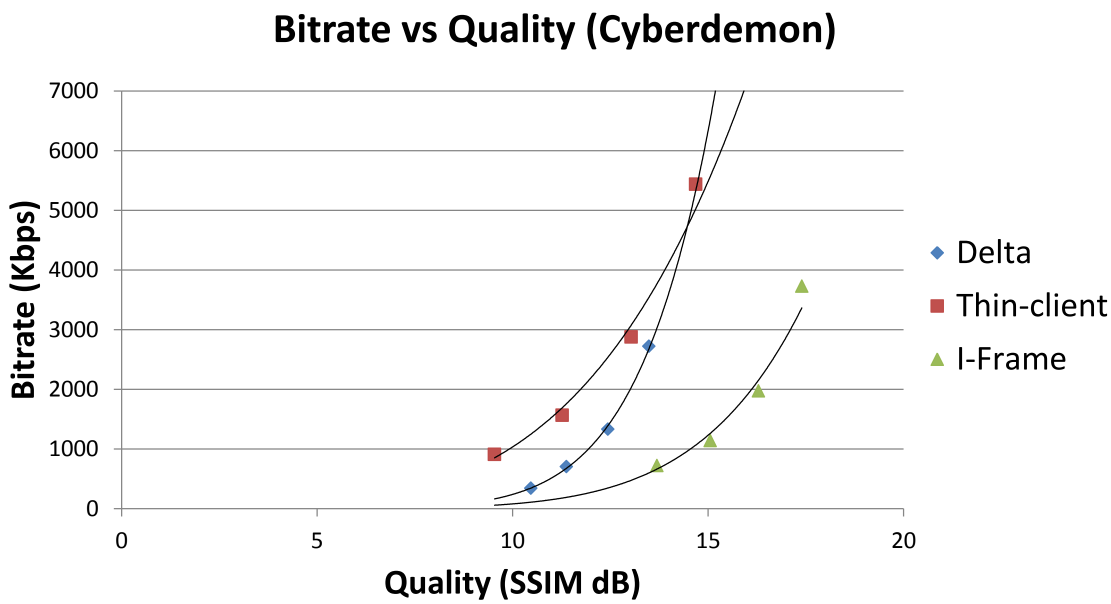

FIGURE 4.5: **Bitrate vs quality for the final boss scene in Doom 3.**

than the bitrate for the thin-client at CRF=25.

Our results are impressive, especially for client-side I-frame rendering. For the demo1 scene, client-side I-frame rendering can achieve an average SSIM of .954 using only 900 Kbps (CRF=35) and an average SSIM of .984 using only 4.1 Mbps. In comparison, the thin client approach needed over 3 Mbps to achieve a similar .96 SSIM score. Furthermore, thin-client was not able to reach an SSIM of over .974 even when using over 6 Mbps. Similarly, the cyberdemon scene shows how the client-side I-frame rendering can achieve similar quality to the thin-client approach using only one fourth of the bandwidth (CRF=30 and 35 for I-frame, CRF=20 for thin-client). Finally, the light room approach behaved similar to the other two scenes, but shows even higher savings due to the strong temporal locality present in the slow moving scene. Delta encoding does not fair as well as client-side I-frame rendering, but provides much better quality than the thin-client approach when bandwidth is constrained, and allows clients that are not able to render full detail I-frames fast enough to have access to high quality gaming.

We also wanted to show the impact of the client's output fidelity in the resulting bitrate. In the case of delta encoding, the client's fidelity is measured as how much detail can the client include in its ”‘low”' quality setting. Figure 4.6 shows a comparison using delta

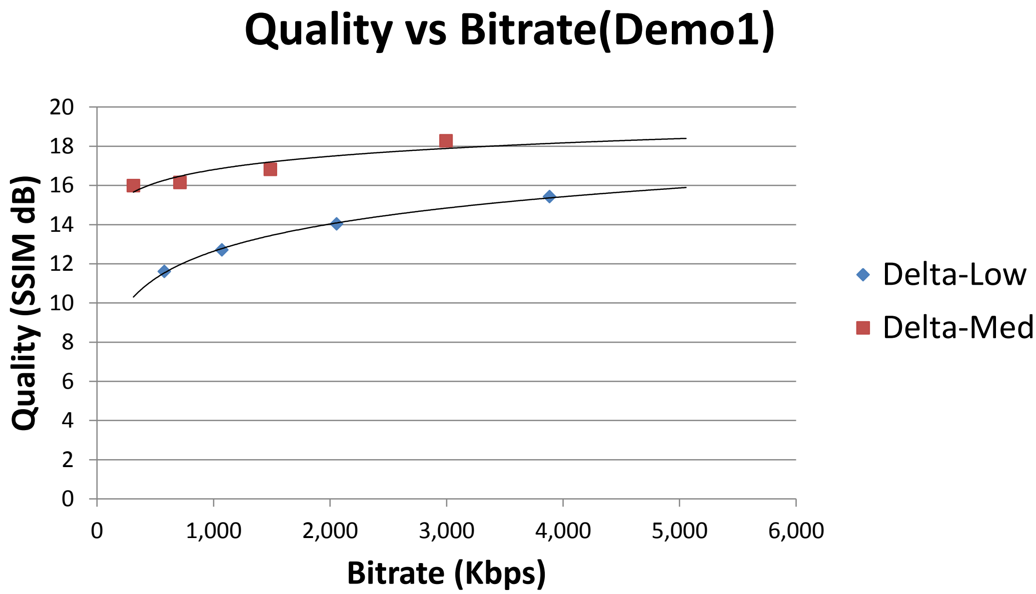

FIGURE 4.6: **Quality versus Bitrate for the demo1 scene in Doom 3 for different image quality fidelities.**

encoding on the demo1 scene. One line shows the client rendering the game using our modified low settings *Delta-Low*, while the other shows the client rendering the game using the built-in medium setting. It is clear from the results, that generating a delta from two scenes that are more similar to each other will result in more compressible deltas. In this scenario, delta encoding when using medium level settings can achieve an average SSIM of .985 using only 3.8 Mbps. Even better, it can achieve an average SSIM of .975 with as little as 400 Kbps.

Similarly, increasing the client's temporal fidelity (the number of I-frames rendered per second), when using client-side I-frame rendering also reduces the required bitrate. Figure 4.7 shows a comparison using client-side I-frame rendering on the demo1 scene. One line shows the client rendering 12 I-frames per second, while the other shows the client rendering only 6 I-frames per second. Although not as noticeable for the lower bitrates, the difference becomes more noticeable as the desired quality increases.

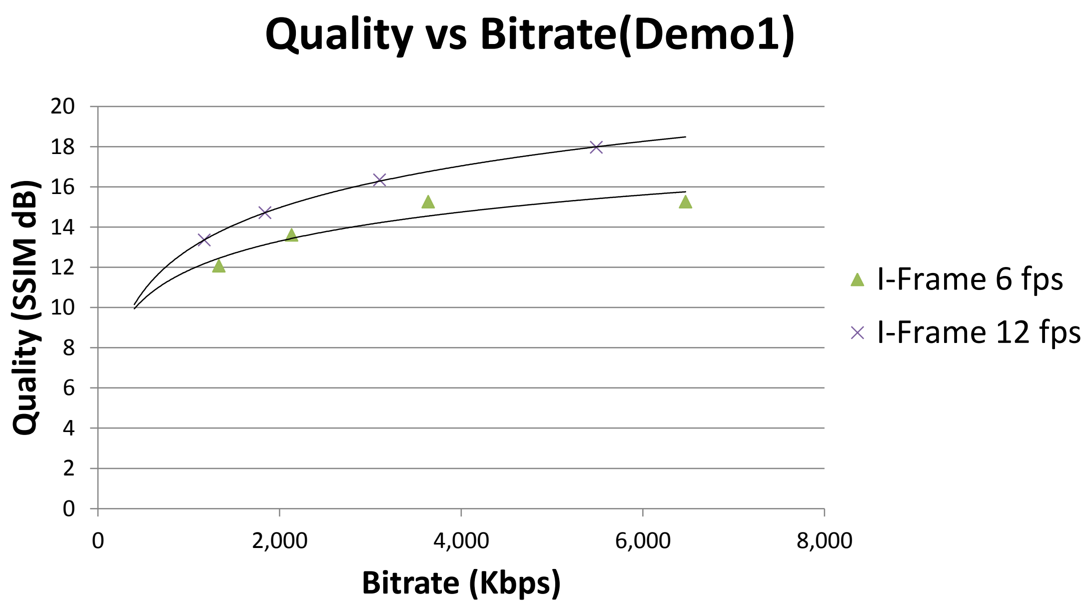

FIGURE 4.7: **Quality versus Bitrate for the demo1 scene in Doom 3 for different temporal fidelities.**

4.7 Conclusions

This paper has presented Kahawai, an approach to GPU offload for mobile games that relies on collaborative rendering. Under collaborative rendering, a mobile device renders low-fidelity graphics output, and combines it with information sent by a server to create a high-fidelity result. We have developed two collaborative rendering techniques through Kahawai.

Under the first technique, called delta encoding, a mobile device generates low-fidelity output by computing low-detail frames at a high frame rate. A remote server computes low-fidelity output that is identical to the client's as well as a stream of highly detailed frames at the same rate. The server then computes the per-frame difference between its low and high fidelity frames and sends a compressed stream of differences to the client. The client can then patch its low-fidelity frames to produce a high-fidelity result.

Under the second, technique, called client-side I-frame rendering, a mobile device generates low-fidelity output by computing highly detailed frames at a low rate. The server computes highly detailed frames at a high rate, but removes the client-rendered frames from the transferred stream. The client can then reinsert its locally-rendered frames to create a high-fidelity result.

Experiments with our Kahawai prototype demonstrate the potential for collaborative rendering. We show that our delta encoding technique, when compared with a thin client using H.264, provides much better visual quality when bandwidth is low (less than 0.5 Mbps). Even more impressive, we show that a thin client requires over four times as much bandwidth as client-side I-frame rendering to achieve high visual quality.

5

Related work

In this chapter, I present the related work that is relevant to this dissertation. The work presented in this dissertation spans mobile computing, video compression and computer graphics. Over the last 20 years there have been many important efforts involved in allowing weak devices to offload computation to a stronger entity. In particular, this dissertation follows after the principles presented in the work of Balan [4]. This dissertation and the thesis statement behind it extend on previous work by observing key properties of mobile applications, and the potential gains in performance and effort reduction obtained by correctly identifying them. Furthermore it extends the scope of applications that can be efficiently offloaded to include a larger range of popular mobile applications, including continuous applications such as graphically-intensive videogames. This dissertation observes important properties of applications such as deterministic behavior, temporal locality, output consistency across different fidelities, and operation based computation. This work is not the first one to address any of those specific attributes. However, to the best of my knowledge, it is the first one that exploits the identification of these properties to drastically reduce the effort required to offload applications and at the same time obtain significant performance benefits when offloading.

Section 5.2 compares MAUI to the most relevant related work in code offload of computationally intensive applications. I contrast MAUI's fine-grained approach achieved through the usage of managed languages; its extensive use of profiling and an integer linear problem solver; and the high priority it gives to energy consumption to previous offload strategies.

In Section 5.2 I discuss in detail the work relevant to Kahawai. Kahawai uses two techniques Delta Encoding and Client Side I-Frame rendering. There are previous attempts in the computer graphics literature to use similar scenes rendered at different quality levels to assist remote rendering. I show that in addition to the similarity between scenes, the temporal locality present in encoded deltas as well as its high resiliency to quantization make them a very suitable mechanism to offload graphics computation in weaker devices. The second approach, Client Side I-Frame rendering takes advantage of a reduction in temporal error propagation to achieve better overall compression. Although to the best of my knowledge, I am the first one to use this property for collaborative rendering and offloading, I address previous work that attempts to reduce temporal error propagation under other circumstances.

5.1 General purpose code offload

5.1.1 Related work: MAUI

Over the last two decades, there has been much work on supporting remote execution for mobile applications. Most of this previous work used remote execution to increase the performance and improve the availability of of resources, such as faster CPU's and more RAM, for mobile applications. Some previous efforts sought to balance the thirst for performance with energy conservation, because these goals can sometimes be contradictory [27]. In contrast, MAUI's primary goal is using remote execution to save energy. With MAUI, code is offloaded to a remote server only if MAUI predicts that remote execution ends up saving energy. Despite the difference in goals, MAUI borrows many techniques and ideas from these previous efforts.

5.1.2 Program Partitioning

One common approach for remote execution is to rely on programmers to modify the program to handle partitioning, state migration, and adaptation to changes in network conditions. In Spectra [26, 27], programmers provide execution plans on how to partition an application given different *fidelities*, which is an application-specific measure of quality of service. At runtime, Spectra monitors the connectivity to a remote server and chooses the execution plan that maximizes a user-provided utility function. Chroma [3, 7] builds on ideas from Spectra while making an effort to reduce the burden on the programmer. Chroma allows programmers to specify "tactics" (strategies for how a program can utilize infrastructure resources) using a declarative language. While Spectra requires users to provide the utility function to the system, Chroma relies on an external system called Prism [5] to automatically construct these utility functions by tracking users. Both Spectra and Chroma borrow ideas from Odyssey [72], an initial system that investigated operating system support for applications adapting their fidelity to changes in network bandwidth, CPU load, and battery conditions. Another common previous approach, used by Protium [96], manually partitions applications into local viewers that execute on a mobile device, and remote servers that execute the application logic. Although MAUI's architecture borrows ideas from all these systems, MAUI reduces the burden on programmers by automating many of the steps needed for program partitioning.

There are many earlier efforts on lightweight approaches to code migration, including systems for mobile objects, such as Emerald [49], Network Objects [10], Obliq [13], Rover [48], and Agent Tcl [38]. The focus of these systems is primarily on enabling code and data to easily move between nodes in a distributed system. To the best of our knowledge, none of these systems attempted to hide distribution from the programmer, nor did they focus on automating migration to optimize for energy consumption.

Previous work also investigated the use of automatic program partitioning [44, 54, 70, 71]. Coign [44] provides coarse-grain automatic partitioning of DCOM applications into client and server components without source-code modification. Kremer et al. [54] propose using static analysis to select tasks for remote execution to save energy. In [70] the authors propose statically partitioning a C-like program into a collection of nodelevel nesC programs that run on sensor motes. Hydra [93] provides support for offloading computation to specialized processors such as GPUs, NICs, and disk controllers, and it uses an ILP to decide what code to offload. Wishbone [71] uses a profile-based approach to partition applications, specified as a data-flow graph of operators, between sensor nodes and servers. MAUI retrofits many of these ideas to today's mobile landscape, where mobile devices and remote servers use different instruction set architectures (unlike [44]) and where mobile applications differ from the data-collection applications that typically run on sensor networks.

Another approach is to build replication into mobile applications and to use distributed protocols to synchronize the application's replicas. In such systems, the mobile user can run the application using any of the available replicas, whether they are local or remote. Data staging [28] and fluid replication [51] propose opportunistic use of "surrogates" (nearby untrusted and unmanaged public machines) as staging servers for the applications' replicas. Slingshot [88] extends this earlier work by adding the capability of dynamically instantiating replicas of "stateful" applications. All these systems also rely on the programmer to partition the applications and to build in support for replication.

Finally, of all the extensive previous work on program partitioning and remote execution, the system that is closest to MAUI is the OLIE system [39]. OLIE performs dynamic partitioning of Java applications at runtime, with little burden on the programmer. The OLIE runtime monitors network conditions and profiles the program behavior. However, one of the principal differences is the focus of OLIE's dynamic offloading engine on overcoming the memory resource constraints of mobile devices, in constrast with MAUI's focus on reducing energy consumption.

5.1.3 Process and VM Migration

Another approach to remote execution is providing operating system support for process migration, as in systems such as Sprite [21] and Amoeba [68]. More recently, Zap [77] enabled process migration using OS support for checkpoint and restart. Recent work on live migration of virtual machines [17] enables moving an entire OS and all its running applications, and the CloneCloud system [16] and Cloudlets [85] suggest applying this technique to mobile device environments. All these approaches drastically reduce the burden on the programmer, which is also one of MAUI's goals. However, MAUI's focus on energy savings made us choose a design that is more aggressive and exploits more opportunities to offload code. This includes the ability to offload portions of a single application; we did not want to restrict MAUI's code migration to the granularity of a whole process or an entire OS.

5.2 Graphics processing offload

Adjusting data fidelity to match the quality of a mobile client's connectivity is a well-studied technique in mobile computing. For example, in Odyssey [73] depending on the network conditions, a mobile client can specify the fidelity at which a server should send it data items. Collaborative rendering also relies on low-fidelity data representations to reduce network usage, but combines locally-computed low-fidelity data with information from a server to give mobile clients access to high-fidelity data.

Overcoming the resource constraints of mobile devices by partitioning programs between a device and more powerful server infrastructure is also a well-studied area. Chroma [6] allows programmers to specify program partitions and conditions under which to adopt those partitions. CloneCloud [15] and MAUI [18] leverage features of managed-language runtimes to automatically partition a program depending on runtime prediction of the partition's energy and performance benefits. Unfortunately, the each of these systems are designed for general-purpose CPU workloads and are inappropriate for applications like fast-action games that heavily rely on a GPU.

Thin-client computing is an extreme form of code offload in which all of a program's computation runs on a remote server [89, 55]. This approach has been embraced by companies like OnLive [75], Gaikai [31], and Ubitus [90] as a way to deliver high-fidelity gaming experiences on mobile devices. Collaborative rendering has two advantages over thin-client gaming: (1) it offers substantial bandwidth savings, and (2) it allows mobile clients to provide low-fidelity game play when a device is poorly connected or completely disconnected.

Delta-encoding's use of synchronized, duplicate processes is similar to techniques developed for primary/backup fault tolerance[2] and deterministic replay [22, 25]. However, the work that most closely resembles delta-encoding is that of Levoy[58]. In Polygon-assisted compression of images, Levoy proposed selectively disabling detail and effects that are complex to render in clients and sending them as compressed deltas. Delta encoding takes a similar approach, and we have borrowed some of this work's calculations. The primary difference between our work and Levoy's is our efficient use of H.264. In pariticular, delta-encoding is well suited to H.264 because of the temporal locality among delta frames exhibited by fast-action games.

Mann's work on Selective Pixel Transmission[60] also tries to reduce network communication by extrapolating frames through locally rendered, textureless models and frequent reference frame deltas. A high frequency of the reference frame deltas is necessary to limit the temporal error propagation caused by extrapolating frames from the basic model. Furthermore, in order to obtain a better balance between bandwidth and image quality, only a subset of the pixels in the delta are transmitted.

This approach resembles some of the techniques used in client-side I-frame rendering.

However, instead of extrapolating frames, we constantly transfer progressive differences between the reference frames (P-frames). We use the deltas to make up for limitations in the mobile GPU instead of limiting the amount of error. The high frequency of I-frames in our approach is used to reduce the size of the P-frames rather than to sync the images in the server and client.

6

Conclusion

6.1 Contributions

This dissertation makes contributions in three major areas. The first area is conceptual – it consists of the novel ideas generated by this work. The second area is a set of artifact system that we developed to validate this dissertation. The final area of contribution is the experimental evaluation of the systems that validate the feasibility of this dissertation's thesis.

6.1.1 Conceptual contributions

This dissertation makes the six following conceptual contributions:

- I showed that by identifying a small set of properties about mobile applications, developers can enable offload and reap the most benefits out of it.
- I identified two sets of classes suitable for offload, and the key properties they present in relation to offload.
- I recognized that an application energy consumption can be accurately estimated by measuring periodically the network, application and device. Furthermore this

estimation can lead to an optimal application partition.

- I showed that using fine-grained language assisted offloading is appropriate for current mobile applications, but also significantly easier for developers and more effective at obtaining optimal application partitions than coarse grained offload.

- I showed that mobile devices despite being resource limited are still able to perform a significant part of the computation required by the user, even when using complex graphics.

- I proposed a model to describe graphic fidelities in mobile devices, in detail space (graphics settings) and time space (framerate) and match them with an appropriate fidelity supplementing technique. Delta encoding for detail fidelity and client side I-frame rendering for time fidelity.

6.1.2 Artifacts

In the course of this dissertation, I have developed two major artifacts to validate the thesis: MAUI, and Kahawai. I built prototypes of these systems to confirm the feasibility of systems' implementations and their usability.

- I built MAUI, a general purpose code offload facility for mobile phones running Windows Mobile 6.5. MAUI is composed of different smaller components including a profiler engine, a stub compiler, and an energy optimization solver. Along with MAUI, four enhanced applications were built: A simple chess game, a casual arcade game, a face recognizer and a real time speech translator.

- I built a prototype of Kahawai, a GPU offload system that allows both time and detail fidelity supplements through delta encoding and client I-frame rendering. Kahawai supports games built using iD software's idTech 4 engine, and features both server and client runtime components. Kahawai is composed by a Delta Encoder, a Client

I-frame encoder, a patch decoder and a video stream mixer for decoding server P-Frames. Kahawai also includes a specification of how a deterministic engine should work and a set of modifications enforcing that specification applied to the idTech 4 engine.

6.1.3 Evaluation results

The evaluation results in this dissertation present the following insight: A very diverse set of applications, that includes machine learning based applications, rule checking applications, casual games and complex games can be efficiently offloaded with minimal developer effort as long as the developer is able to identify a small set of important application properties (using MAUI). The evaluation has two different sections: the energy savings and performance benefits for those applications identified as computationally intensive, and a second part showing the performance benefits and bandwidth savings of those applications identified as graphically intensive (using Kahawai).

The evaluation of MAUI shows that with minimal annotations based on identified properties, the developer can retarget existing applications to use code offload obtaining large benefits. This benefits can be as large as an order of magnitude reduction in energy consumption as well as an order of magnitude improvement in performance for an application such as face recognition. MAUI also shows that for relatively simple applications such as a rule-checking chess game, benefits can be reaped as long as the representation of the input and output data required by the operation are very small compared to the computational requirements of that same operation. Also, MAUI shows that using managed environments like Java or .NET CLR, the required developer effort is drastically reduced, largely due to the high level language support that is provided by such environments. Finally, MAUI shows that by using a global perspective of the application and a dynamic solver, it can adapt to the constantly changing nature of the mobile environment, where device specifications change every few months, application usage changes constantly and network conditions are often unpredictable.

The evaluation of Kahawai presented several interesting results. First, it shows that continuous applications producing massive amounts of output can be efficiently offloaded as long as a lower fidelity of such output can be produced locally by the local device. Second, it shows that by identifying simple properties about the graphics application behavior, different techniques can be applied, maximizing benefits and offload opportunities. Third, it shows that by taking advantage of the reduction in temporal error propagation, Client I-Frame encoding can outperform current state of the art thin client approaches.

Appendix A

MAUI Serialization

Serialization is the process of converting an object to a format that can be stored persistently or transferred over the network, to be used later in the same or another environment. In MAUI, serialization is responsible for persisting the input and output parameters of a method call, as well as all of the application state necessary to execute the method on a MAUI server. This component must hide from the developer the complexities behind identifying the state that has to be transferred, encoding it, and estimating its size.

In this Appendix, I describe and compare the serialization approaches used in MAUI. MAUI is a system written in C#, targetting Windows Mobile 6.5 devices. As such, it is constrained by the limitations of the .NET compact framework[61].

A.1 Design Goals and Constraints

The serialization component was designed to achieve three main goals:

1. **Minimal burden on developers:** One of MAUI's goals is to be easy to use by application developers. The serialization scheme should require little or no changes to the application's data structures, allowing the developer to focus into what methods

are “Remotable”, instead of how these methods are actually offloaded.

2. **Efficient representation:** The serialized representation of application data must be compact. The energy used to transfer serialized data represents a significant portion of the energy consumed when offloading a method.

3. **Performance:** The serialization component must not become a bottleneck in the offloading process. Additional delays incurred during serialization or de-serialization lead to extra offloading latency. Adding latency, not only reduces the performance benefits of code offload, but also eliminates opportunities to turn off the screen and switch the CPU to a lower power mode. These missed opportunities cause extra power consumption.

In addition, the serialization component was designed to satisfy the constraints imposed by the .NET compact framework platform, which does not provide built-in support for binary serialization, as the full .NET framework or Java do. The .NET framework only includes a small subset of the classes found in the full framework, and as such, any serialization mechanism has to be built around those classes only. For this reason I tried two different serialization approaches, one using .NET’s built-in XML serialization and the other using Google’s protocol buffers. Both approaches minimize the burden on developers, but only the second approach provides a fast and efficient representation of the serialized data.

Before introducing each approach, I will first present the basics behind serialization in .NET and Java and why it is possible in these languages to automatically detect all the state reachable within a method. Next, I will present XML based serialization and binary serialization. Finally, I will show the approach used in the final prototype, binary serialization, in context with the MAUI Proxy.

A.2 Background: Serialization in C# and Java

Languages like Java and C# include built-in mechanisms for serialization, and also allow custom serialization mechanisms to be easily developed. Furthermore, these mechanisms can be used by application developers with minimal effort.

These languages are able to offer these mechanisms because they are strongly typed and support reflection. It is because of these two mechanisms that it is possible to know at runtime both a) the exact type of each member variable, and b) the transitive closure of an object. With this knowledge, these languages only need confirmation by the developer, usually with an annotation, of whether it is sensible or not to serialize an object.

Both Java and C# are more strongly typed than languages like C or C++. C and C++ allow arbitrary cast conversions and the use of the void* type. In C# and Java, the closest equivalent to this freedom is through the Object base class, which is monitored by the Java Virtual Machine or the CLR. Casts to incompatible types in Java and C# will result in runtime exceptions, while in C or C++ they are legal.

Similarly, Java and C#, unlike C, support reflection. Reflection is the process by which a program examines and modifies its own structure, behavior and metadata at runtime. Because of reflection, a serialization mechanism can find the name and type of each of the member variables of an object, as well as finding the members of those members until it finds the transitive closure of that object.

A very simple serializer would perform the following steps:

- Obtain all fields, properties and metadata that can be serialized through reflection.
- Fetch the values of each member.
- Persist those values to a file, memory or the network in a serialized format.

Serialization mechanisms can use different formats to persist the serialized data. The most compact representations usually persist objects as binary data, while representations used for Web systems usually prefer human readable text data, resulting in larger representations. While building MAUI I experimented with both classes of serializations, trying to meet the goals outlined in Section A.1.

A.3 XML based serialization

Initially, I built MAUI's serialization component using the only serialization format provided by the compact framework, human readable XML. A couple of built-in serializers in .NET can serialize data in this format, some of them used in .NET RPC technologies like .NET remoting and WCF [66].

When building the first iteration of the serializer I used the XMLSerializer [64], which improves over traditional serializers, by reducing its use of reflection. It does so by generating a serializer class specific to each object on the fly. Unfortunately this class is only able to access public members, placing an additional burden on the developer.

A.3.1 Incremental updates

The XML representation that resulted from using this serializer is many times larger than that obtained from a binary serializer, like the BinaryFormatter, which is not available in the compact framework. In order to obtain smaller representations, I developed an incremental update system. Using this system, the MAUI proxy can keep track of what state has been sent, to instead produce an incremental serial update. I achieved this by integrating the XML diff and patch technique [62] in the MAUI serializer. As it is shown in Chapter 3, Section 3.7.3, using incremental updates disrupts the profiling process and adds overhead. However, the bandwidth savings outweigh both shortcomings.

The XML serializer with incremental updates is able to generate small representations of simple objects with reasonable performance. However the delay imposed by serializing and obtaining a delta is too large for latency sensitive applications, like games. Using the XML serializer prevented games from achieving full framerate (30 frames per second).

A.4 Binary serialization

After finishing the original MAUI prototype, I found a way to implement an efficient, developer friendly binary serialization component for the .NET compact framework. Protobuf-net[36], a powerful serializer that uses Google's Protocol Buffers as its representation format. Protocol buffers is a very efficient representation format, being a couple of orders of magnitude smaller than XML. Protobuf-net is also a very fast serialization implementation, being a couple of orders of magnitude faster than the XmlSerializer[37].

Protobuf-net is a very efficient implementation that avoids expensive operations related to serialization, like excessive reflection. It allows class specific serializers to be generated at compile time. These classes can be generated by the MAUI compiler, when the proxies are being created. Unlike the XMLSerializer, Protobuf-net classes are able to access private members, making it simpler to transform data structures to be used with MAUI.

Unfortunately, there are no binary delta implementations available for the compact framework, so I did not include that feature in the binary serializer. However, for the applications that I tested (arcade game, voice translator, chess game and face recognizer), the state representation was very small, falling in the range of a few KB. Since the binary serializer outperformed the XML serializer in both speed and representation efficiency and met the system requirements, I did not look for a incremental solution. However, it is likely for applications where only small portions of the state are changed, to reap additional network savings by using an incremental binary approach.

A.5 The MAUI proxy and the serializer

The MAUI proxy is the component responsible of handling control and data transfer to remoteable methods. The proxy intercepts calls to remoteable objects, and using information provided by the solver, it redirects the execution to either a local copy of the object or to a MAUI node. In order to allow the application flow to be redirected, the proxy is

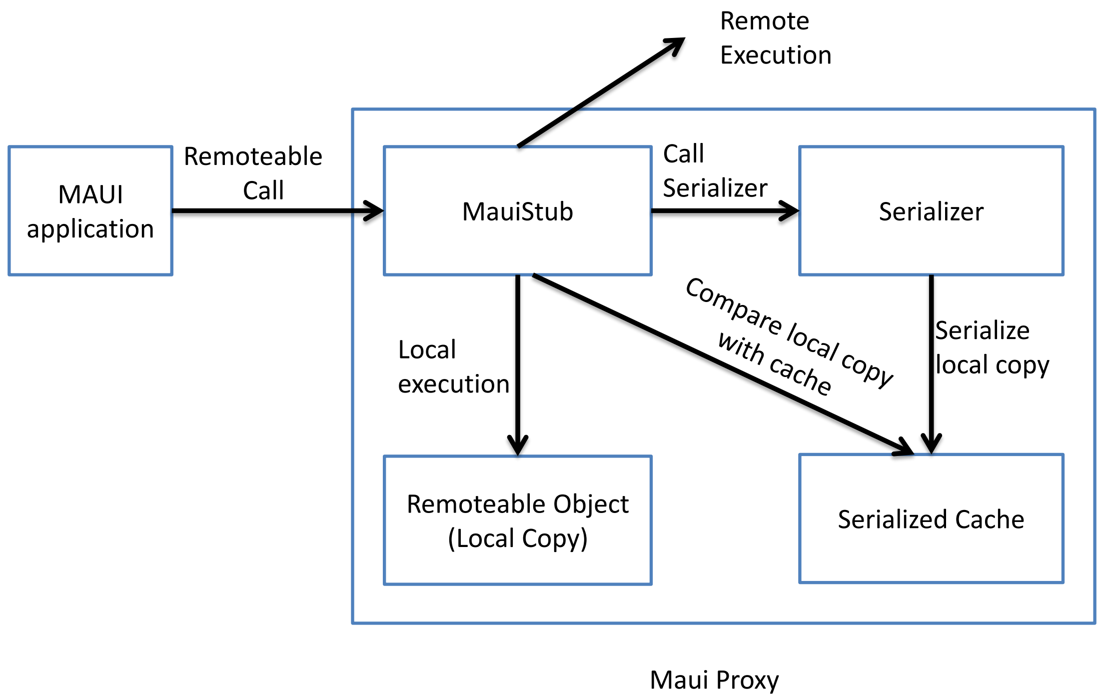

FIGURE A.1: **The interaction between the Maui proxy and the serializer**

responsible of serializing all the state that is needed for the remote method, and any nested calls within it. The state required is formed by the following elements:

1. The input parameters
2. The values of all the member variables in the transitive closure of the remoteable object
3. The values of all the public static member variables in the application

Once the remote call returns, the proxy is responsible for synchronizing the state changes that occured in the server to allow the program to continue. The state required to return the call is the same that was required to send the remote call, but the output values are serialized back instead of the input parameters which are discarded.

Unfortunately, serializing the application state can be a slow process, specially if using text based serialization. For this reason, MAUI keeps a copy of the last serialization as well as a hash of the object at the time of the last serialization. Using this cache, MAUI can avoid serializing the object when it has not changed. This is also true when the object does not change in the server. The MAUI RPC protocol allows for no serialized object to be sent, signaling that the local copy is valid.

This optimization reduces the amount of information that has to be transferred for a given method in a similar way to incremental deltas (Chapter 3, Section 3.7.3). In this case, two different approaches can be taken. The first one averages the transfer sizes when the object changes, giving a value of zero to each invocation when the object did not change, while the second always takes the size of the serialized copy, even if it is not sent. For our current prototype, we use the more conservative second strategy, but there may be applications that could benefit of the first one. This could be particularly true when the size of the state transfer is large, but changes infrequently.

The proxy is also responsible for estimating the amount of information that is sent through the network for each method call. This information is used by the profiler to estimate the energy that would be required to offload a given method. The proxy estimates the size as a moving average of the transfer size at each invocation. Unfortunately, the only way to estimate the network transfer size is to serialize the state. Serialization occurs naturally when the execution is offloaded, but it does not occur when the execution is performed in the local object. Forcing serialization during local execution imposes a performance penalty, and even if binary serialization is fast, performing it on every remoteable call in the program could become very expensive. MAUI instead proceeds as follows:

- If the call is remote, serialize if the object has changed
- If the call is local, only serialize if the object has changed and the size of the state

has not been updated since the last time the solver was called. In our prototype, the solver is called asynchronously once every 8 seconds.

Using this technique, it is possible to notify the solver timely of changes in the transfer size without severely affecting performance. Finally, by using binary serialization along with the proxy in this fashion, it is possible to satisfy the three design goals of the serialization component. The serialization is fast and the representation is compact. The burden on the programmer is minimal, as the annotations required by protobuf-net can be generated by the MAUI compiler for most serializable objects.

Bibliography

[1] *IEEE 100 : the authoritative dictionary of IEEE standards terms*. Standards Information Network, IEEE Press, New York, 2000.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *ICSE '76*, pages 562–570, 1976.

[3] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The Case for Cyber Foraging. In *The 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[4] R. K. Balan. Simplifying Cyber Foraging. Technical report, Carnegie Mellon University, May 2006.

[5] R. K. Balan. *Simplifying Cyber Foraging*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2006.

[6] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb. Simplifying cyber foraging for mobile devices. In *MobiSys '07*, page 272, New York, New York, USA, May 2007. ACM Press.

[7] R. K. Balan, M. Satyanarayanan, S. Park, and T. Okoshi. Tactics-Based Remote Execution for Mobile Computing. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, 2003.

[8] H. Bao. Real-time graphics rendering engine. *Most*, pages 1–6, 2011.

[9] P. Bird. Privacy bird. `http://www.privacybird.org/`.

[10] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *Proc. of the Fourteenth ACM Symposium on Operatings Systems Principles (SOSP)*, 1993.

[11] Capcom. Street fighter iv, 2008. `http://www.streetfighter.com`.

[12] Capcom. Lost planet 2, 2010. `http://www.lostplanet2game.com/`.

[13] L. Cardelli. A Language with Distributed Scope. In *Proc. of the 22nd Symposium on Principles of Programming Languages (POPL)*, 1995.

[14] L. Cheng, A. Bhushan, R. Pajarola, and M. El Zarki. Real-time 3D Graphics Streaming using MPEG-4. In *Proceedings of IEEE/ACM Workshop on Broadband Wireless Services and Applications*, July 2004.

[15] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. *Eurosys 2011*, 2011.

[16] B.-G. Chun and P. Maniatis. Augmented Smartphone Applications Through Clone Cloud Execution. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Monte Verita, Switzerland, May 2009.

[17] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[18] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010.

[19] E. Cuervo, P. Gilbert, B. Wu, and L. P. Cox. CrowdLab: An Architecture for Volunteer Mobile Testbeds. *Proceedings of the 4th International Conference on COMmunication Systems and NETworkS (COMSNETS '11)*, 2011.

[20] G. Danti. Published Kal-El (TEGRA3) performance: is NVIDIA SoC truly faster than a Core2?, 2011. `http://www.ilsistemista.net`.

[21] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, August 1991.

[22] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI '02*, pages 211–224, New York, NY, USA, 2002. ACM.

[23] Epic Games. Unreal game engine. `http://unrealengine.com/`.

[24] Ffmpeg. `http://ffmpeg.org/`.

[25] J. Flinn and Z. M. Mao. Can deterministic replay be an enabling tool for mobile computing? In *HotMobile '11*, pages 84–89, New York, NY, USA, 2011. ACM.

[26] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Schloss Elmau, Germany, May 2001.

[27] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[28] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanaryanan. Data Staging on Untrusted Surrogates. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, March – April 2003.

[29] Flurry. `http://blog.flurry.com/`, Jan 2012.

[30] Fraps. `http://www.fraps.com/`.

[31] Gaikai. Gaikai open cloud gaming platform. `http://www.gaikai.com/`.

[32] P. Gilbert, E. Cuervo, and L. P. Cox. Experimenting in mobile social contexts using JellyNets. *Proceedings of the 10th workshop on Mobile Computing Systems and Applications - HotMobile '09*, 2009.

[33] Google. Protocol Buffers - Google's data interchange format. `http://code.google.com/p/protobuf/`.

[34] Google. Dalvik: Code and documentation from the Android's VM team. `http://code.google.com/p/dalvik/`, 2012.

[35] Google. Google Android OS. `http://www.android.com/`, 2012.

[36] M. Gravell. Protobuf-net - Fast, portable, binary serialization for .NET. `http://code.google.com/p/protobuf-net/`.

[37] M. Gravell. Protobuf-net performance, 2012. `http://code.google.com/p/protobuf-net/wiki/Performance`.

[38] R. S. Gray. Agent Tcl: a flexible and secure mobile-agent system. In *TCLTK'96: Proceedings of the 4th USENIX Tcl/Tk Workshop, 1996*, Monterey, CA, 1996.

[39] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2003.

[40] S. Hecht. Intermittent stimulation by light. *The Journal of general physiology*, 1935:965–977, 1936.

[41] D. Hesse. Sprint's CEO Dan Hesse Chats with Charlie Rose. `http://blueroomsolution.com/showthread.php?t=5689`, 2007.

[42] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing Application Performance Differences on Smartphones. In *Proc. of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, June 2010.

[43] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[44] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[45] IDSoftware. Doom 3, 2004. `http://www.idsoftware.com/games/doom`.

[46] IDSoftware. Doom 3 source, 2011. `https://github.com/TTimo/doom3`.

[47] ITU-T Recommendation J.247. *Objective Perceptual Multimedia Video Quality Measurement in the Presence of a Full Reference*. Aug. 2008.

[48] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, 1995.

[49] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[50] A. Kansal and F. Zhao. Fine-Grained Energy Profiling for Power-Aware Application Design. In *Proceedings of the 1st Workshop on Hot Topics in Measurement & Modeling of Computer Systems (HotMetrics)*, Annapolis, MD, June 2008.

[51] M. Kim, L. Cox, and B. Noble. Safety, Visibility, and Performance in a Wide-Area File System. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[52] J. Kincaid. YouTube Mobile Uploads Up 400% Since iPhone 3GS Launch. `http://www.techcrunch.com/2009/ 06/25/ youtube-mobile-uploads-up-400 -since-iphone-3gs-launch/`, 2009.

[53] P. Kougiouris. Use Reflection to Discover and Assess the Most Common Types in the .NET Framework. `http://msdn.microsoft.com/en-us/magazine/cc188926.aspx#S3`, 2002.

[54] U. Kremer, J. Hicks, and J. M. Rehg. Compiler-Directed Remote Task Execution for Power Management. In *Proceedings of The Workshop on Compilers and Operating Systems for Low Power (COLP)*, Philadelphia, PA, October 2000.

[55] A. Lai and J. Nieh. Limits of wide-area thin-client computing. In *SIGMETRICS '02*, pages 228–239, New York, NY, USA, 2002. ACM.

[56] R. Leadbetter. Console gaming: The lag factor, Sept. 2009. http://www.eurogamer.net.

[57] R. Leadbetter. OnLive Latency: The Reckoning, July 2010. http://www.eurogamer.net/articles /digitalfoundry-onlive-lag-analysis.

[58] M. Levoy. Polygon-assisted JPEG and MPEG compression of synthetic images. *Proceedings of SIGGRAPH '95*, pages 21–28, 1995.

[59] J. Lewin. iPhone Users 30 Times More Likely To Watch YouTube Videos. http:// www.podcastingnews .com/2008/03/19/iphone-users-30-times -watch-youtube-videos/, 2008.

[60] Y. Mann and D. Cohen-Or. Selective Pixel Transmission for Navigating in Remote Virtual Environments. *Computer Graphics Forum*, 16(3), Sept. 1997.

[61] Microsoft. Microsoft .NET Compact Framework. http://msdn.microsoft.com/en-us/netframework.

[62] Microsoft. Xml diff and patch tool, 2002. http://msdn.microsoft.com/en-us/directx.

[63] Microsoft. Directx developer center, 2012. http://msdn.microsoft.com/en-us/directx.

[64] Microsoft. Introducing xml serialization, 2012. http://msdn.microsoft.com/en-us/library/182eeyhh.aspx.

[65] Microsoft. Microsoft Windows Phone. http://www.microsoft.com/ windowsphone, 2012.

[66] Microsoft. Windows communication foundation, 2012. http://msdn.microsoft.com/en-us/netframework/aa663324.aspx.

[67] Monsoon Solutions Inc. Monsoon Power Monitor. http://www.msoon.com/.

[68] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba - A Distributed Operating System for the 1990s. *IEEE Computer*, 23:44–53, 1990.

[69] National Office of Pollution Prevention (Canada). Canadian Consumer Battery Baseline Study - Final Report. Submitted to Environment Canada, http://www.ec. gc.ca/nopp/docs/rpt/battery/en/toc.cfm, 2007.

[70] M. Neubauer and P. Thiemann. From Sequential Programs to Multi-Tier Applications by Program Transformation. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*, Long Beach, CA, January 2005.

[71] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation (NSDI)*, Boston, MA, April 2009.

[72] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. of the ACM Symposium on Operating System Principles (SOSP)*, 1997.

[73] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *SOSP '97*, pages 276–287, 1997.

[74] Nvidia. Nvidia GeForce GTX 690 Specifications, 2012. `http://www.geforce.com/hardware/desktop-gpus/`.

[75] Onlive. Onlive: Cloud gaming. `http://www.onlive.com/`.

[76] OpenGL. OpenGL - The Industry Standard for High Performance Graphics. `http://www.opengl.org/`.

[77] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[78] M. R. Palacin. Recent advances in rechargeable battery materials: a chemists perspective. *Chem Soc Review*, 38:2565–2575, 2009.

[79] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, 2005.

[80] Physics2D.Net. `http://code.google.com/p/physics2d/`.

[81] R. A. Powers. Batteries for low power electronics. *Proceedings of the IEEE*, 83:687–693, April 1995.

[82] R. S. Prasad, M. Murray, C. Dovrolis, and K. Claffy. Bandwidth Estimation: Metrifcs, Measurement Techniques, and Tools. *IEEE Network*, 17(6):27–35, Nov–Dec 2003.

[83] I. Richardson. *The H.264 Advanced Video Compression Standard*. John Wiley & Sons, 2010.

[84] J. Richter. *CLR via C#*. Microsoft Press; 2nd edition, 2006.

[85] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), 2009.

[86] S. Savage. Sting: a TCP-based Network Measurement Tool. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS)*, Boulder, CO, October 1999.

[87] K. Sayood. *Lossless compression handbook*. Academic Press series in communications, networking and multimedia. Academic Press, 2003.

[88] Y.-Y. Su and J. Flinn. Slingshot: Deploying Stateful Services in Wireless Hotspots. In *Proc. of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Seattle, WA, June 2005.

[89] N. Tolia, D. G. Andersen, and M. Satyanarayanan. Quantifying interactive user experience on thin clients. *IEEE Computer*, 39(3), Mar. 2006.

[90] Ubitus. Ubitus inc. `http://www.ubitus.net/`.

[91] VirtualGL. `http://www.virtualgl.org/`.

[92] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[93] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the Fountain of CPUs – On Operating System Support for Programmable Devices. In *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[94] M. West. Measuring responsiveness in video games, July 2008. `http://www.gamasutra.com`.

[95] x264. `http://www.videolan.org/developers/x264.html`.

[96] C. Young and Y. N. Lakshman. Protium, an Infrastructure for Partitioned Applications. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, Schloss Elmau, Germany, May 2001.

Biography

Eduardo Cuervo was born in Mexico City, Mexico on March 7, 1982. He defended his PhD Thesis at Duke University in May 2010. His research interests include mobile and pervasive computing, networks and distributed systems, security and privacy. He received a B.S. in Computer Science from the Monterrey Institute of Technology and Higher Studies, Mexico City in 2004, and a M.S. in Computer Science from Duke University in 2011.

In addition to his dissertation work, Eduardo worked on mobile testbeds, on the CrowdLab project[19][32]. Eduardo also spent a summer at the CyLab Usable Privacy and Security Laboratory in Carnegie Mellon University, working on Privacy Bird[9].

Prior to joining grad school, Eduardo was identified as a force-sensitive human and trained, mastering several abilities including the misterious Hadoken.