INTEGRATED VECTOR INSTRUCTION TRANSLATOR AND OFFLOADING FRAMEWORK FOR MOBILE CLOUD COMPUTING

JUNAID SHUJA

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

INTEGRATED VECTOR INSTRUCTION TRANSLATOR AND OFFLOADING FRAMEWORK FOR MOBILE CLOUD COMPUTING

JUNAID SHUJA

THESIS SUBMITTED IN FULFILMENT
OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND
INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

UNIVERSITI MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: Junaid Shuja (I.C./Passport No.: [redacted])

Registration/Matrix No.: WHA130039

Name of Degree: Doctor of Philosophy

Title of Thesis: Integrated Vector Instruction Translator and Offloading Framework for Mobile Cloud Computing

Field of Study: Distributed Systems

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date

Subscribed and solemnly declared before,

Witness's Signature Date

Name:
Designation:

ABSTRACT

Mobile Cloud Computing (MCC) facilitates energy efficient operations of mobile devices through computational offload. The mobile devices offload computations to nearby cloud servers while limiting energy consumption in the low-power wait mode. The MCC offloading frameworks are enabled by system virtualization, application virtualization, and native code migration techniques to address the heterogeneous computing architectures. The existing MCC offloading techniques suffer from either computational or communicational overheads leading to higher execution time and energy consumption on the cloud server. This research work addresses the overhead of conventional MCC offloading frameworks while focusing on vectorized applications based on Single Instruction Multiple Data (SIMD). We propose SIMDOM, a framework for SIMD instruction translation and offloading in heterogeneous MCC architectures. The SIMD translator utilizes re-compilation of SIMD instructions of the mobile device (ARM architecture) that are translated to corresponding cloud server instructions (x86 architecture). Based on inputs from the application, network, and mobile device energy profilers, the offloader module decides upon the feasibility of code offload. The SIMD translator is analyzed for its accuracy and translation overhead. The impact of code offload size, application partition, and device sleep time is investigated on the energy and time efficiency of the mobile applications. The lower feasibility bounds for server speed and application partition are derived from the system model. The SIMDOM framework prototype is implemented on a cloudlet and a cloud server. Results show that SIMDOM framework provides 85.66% energy and 3.93% time efficiency compared to MCC-disabled execution. Comparison with state-of-the-art code offloading framework reveals that SIMDOM provides 55.99% energy and 57.30% time efficiency. The SIMDOM framework provides 31.10% higher energy efficiency while translating SIMD instructions as compared to existing MCC offloading frameworks. The improvement in energy and time efficiency increases the usability of MCC offloading frameworks for vectorized applications.

ABSTRAK

Perkomputeran Awan Mudah-alih (MCC) memudahkan operasi cekap tenaga peranti yang bergerak melalui pengiraan penurunan. Pengiraan peranti bergerak penurunan kepada pelayan pelayan awan yang berdekatan sementara membataskan penggunaan tenaga dalam mod penantian yang berkuasa rendah. Rangka kerja penurunan MCC dibolehkan oleh system virtualisasi, aplikasi virtualisasi, dan teknik penghijrahan kod asli untuk menangani arkitektur pengkomputeran yang pelbagai. Teknik penurunan MCC yang sedia ada mengalami masalah samada dalam pengiraan atau komunikasi berlebihan yang membawa kepada masa perlaksanaan yang tinggi dan penggunaan kuasa pada pelayan awan. Kerja penyelidikan ini mengemukakan rangka kerja penurunan MCC konvensional lebihan ketika menumpukan pada aplikasi divektorkan berdasarkan Data Beberapa Arahan Dalam Satu (SIMD). Kami mencadangkan SIMDOM, satu rangka kerja bagi terjemahan arahan SIMD dan penurunan dalam seni bina MCC yang pelbagai. Penterjemah SIMD menggunakan penyusunan semula arahan-arahan SIMD dalam peranti bergerak (seni bina ARM) yang diterjemahkan untuk menjawab arahan-arahan pelayan awan (seni bina x86). Berdasarkan input dari aplikasi, rangkaian, dan tenaga peranti bergerak, modul penurunan memutuskan di atas kemungkinan kod yang menurunkan. Penterjemah SIMD dianalisa untuk ketepatan dan berlehihan terjemahan. Kesan daripada kod saiz code penurunan, menurunkan saiz, sekatan aplikasi, dan waktu tidur peranti disiasat pada kecekapan tenaga dan masa aplikasi mudah alih. Kemungkinan yang rendah sempadan dan untuk kelajuan pelayan dan sekatan aplikasi timbul dari model sistem. Keputusan menunjukkan ranga kerja SIMDOM menyediakan tenaga 85.66% dan 3.93% masa efisien dibandingkan dengan perlaksanaan MCC yang tidak digunakan. Perbandingan dengna kod penurunan rangka kerja menemukan SIMDOM menyediakan 55.99% tenaga dan 57.30% masa yang efisien. Rangka kerja SIMDOM menyediakan 31.10% tenaga efisien yang lebih tinggi sementara menterjemahkan arahan SIMD jika dibandingkan dengan rangka kerja MCC yang sedia ada. Penambahbaikan dalam tenaga dan masa yang efisien menambahkan kepenggunaan penurunan rangka kerja MCC untuk aplikasi yang divektorkan.

ACKNOWLEDGEMENTS

ALLAH Almighty provides the courage, knowledge, and resources to every human being in this world. I am thankful to ALLAH Almighty for blessing me in every form of human quality such that I have reached this point of life and completed my Ph.D. thesis.

I am thankful to my supervisors, family, and friends who have supported and encouraged me through difficult times of life. I am highly thankful to my supervisor, Prof. Dr. Abdullah Gani, who took me under his guidance and provided me his support for the completion of my Ph.D. After the early struggles in my Ph.D., I knew that only one supervisor could guide me through this difficult phase of life. I am also thankful to Dr. Anjum Naveed, who provided the technical guidance and ideas that form the crux of this thesis. I was lost in ideas of mobile virtualization when Dr. Anjum advised me to work on SIMD instructions and ARM emulation for my research.

I would also like to thank my fellow lab mates from whom I have consulted on various technical issues. In particular, I would like to thank Abdullah, Wasim, and Dr. Ejaz. My family has been a source of inspiration for me. I dearly want my grandmother, mother, daughter, and wife to share the prosperous and happy days of my life with me. Through all the difficult times, I have imagined care-free days of enjoying my time with my family as an inspiration for hard work. I am also thankful for their support and prayers. My college friends have been the bond of my life. They provide me hope for future meaningful conversations and adventurous outings. I am also thankful to the FSKTM support staff at WISMA RND and University for lending their support and resources.

Last but not the least, I am thankful to the BrightSpark Program for financially supporting me throughout my Ph.D. tenure. Their strict KPI's were a blessing in disguise that enabled me to publish multiple Tier-1 articles as the first author.

To my Family,
especially Aqsa

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

CHAPTER 1: INTRODUCTION

The advances in mobile devices and wireless technology have led to a paradigm shift from static computing to mobile computing. The smart mobile devices provide users with reasonable computational power within the constraints of size and battery. Smartphone users can perform tasks and execute applications on the modern mobile devices that were previously only feasible on desktop and server systems. The mobile devices still remain resource-scarce compared to server devices. The mobile devices have several communication interfaces that allow them to offload or migrate large computations to nearby cloud servers. The task offloading saves the mobile device energy and provides efficient execution time. The extension of cloud resources to the mobile device for task offloading is known as Mobile Cloud Computing (MCC).

MCC facilitates resource-scarce mobile devices to utilize resource-rich cloud server. The mobile user provides the input to the application and the rest of computations are performed on the cloud server. Meanwhile, the mobile device waits in low-power energy saving state. Mobile applications are not directly executable on the cloud servers due to the heterogeneity of computing architectures. This research work addresses the challenge of computational offload in heterogeneous MCC computing architectures.

This chapter introduces the foundations of the research work carried out in the thesis. The background of the primary research domain, MCC, is provided. Key motivations in undertaking the research problem of the thesis are described. The research problem is emphasized in the form of statements of the problem. Research aim and objectives are highlighted in the domain of MCC. Moreover, the research methodology employed to solve the research problem is presented.

The structure of the rest of the chapter is as follows. Section 1.1 provides the background knowledge of the field of research. Section 1.2 explains the motivations for inspiring the research presented in this thesis. In Section 1.3, the statement of research problem is elaborated. Section 1.4 presents the research aim and objectives. In Section 1.5, the research methodology followed to solve the research problem is defined. At last, Section 1.6 presents the layout of the rest of the thesis.

1.1 Background

Smartphones and mobile devices have emerged as a new computational platform over the last decade known as Mobile Computing (MC). The paradigm of MC has been possible due to rapid increase in the processing power and wireless access technology for mobile devices. Modern handheld and mobile devices, known as smartphones due to their capabilities, have equivalent processing power as some of the desktop systems. However, modern smartphones still lag behind in performance due to battery and size constraints when compared to server devices (Satyanarayanan et al., 2015).

The performance of the mobile devices can be enhanced with Mobile Cloud Computing (MCC). MCC is an operational integration of mobile and cloud computing technologies. Cloud computing offers vast resources of computation and storage that can be augmented with resource constrained smartphones (Kumar & Lu, 2010). In MCC, the majority of tasks are offloaded from the smartphone to the cloud. The cloud then performs the required computations and sends back the result to the smartphone over the communication interfaces. Mobile and cloud server hardware architectures are heterogeneous. Intel x86 based processors dominate the server market while ARM Instruction Set Architecture (ISA) based processors power 90% of smartphones (Shuja, Gani, Bilal, et al., 2016). Therefore, code offloaded from mobile device to the cloud server requires Dynamic Binary Translation (DBT) due to the heterogeneity of underlying hardware architectures. The DBT of an application translates its code from one hardware architecture to another. However, significant overhead is incurred during the process of DBT (Hsu, Hong, Hsu, Liu, & Wu, 2015). Other than DBT, system and application virtualization are also employed as MCC offload enabling techniques for heterogeneous hardware architectures (Ahmed, Gani, Sookhak, Ab Hamid, & Xia, 2015).

A number of compute-intensive applications, such as augmented reality, natural language translation, object, voice recognition, and multimedia-based software are dependent on MCC due to resource scarcity of smartphones. Popular multimedia based smartphone applications are; **(a)** cloud storage (store retrieve files, photos, videos e.g., Instagram, Facebook) and **(b)** audio video streaming e.g., YouTube, SoundCloud (Satyanarayanan et al., 2015).

Multimedia based applications rely on Single Instruction Multiple Data (SIMD) commonly known as vector instructions. SIMD instructions are a hardware capability that allows execution of the same operation on multiple data points simultaneously. For example, the change in brightness of a picture requires retrieving of N pixels and ADD/SUB operation on the N pixels. The utilization of SIMD instructions significantly boosts the performance of applications. Up to 25-50% of the code of multimedia based applications can be SIMD instructions (Mitra, Johnston, Rendell, McCreath, & Zhou, 2013). SIMD instructions and intrinsics vary from one ISA to another. Applications programmed with SIMD instructions are not portable and executable on heterogeneous ISAs. Therefore, SIMD instruction translation techniques are required to enable vectorized multimedia applications to offload between heterogeneous MCC architectures (Manilov, Franke, Magrath, & Andrieu, 2015).

1.2 Motivation

MCC is a rapidly growing technology in terms of both commercial applications and research work. Over the past five years, mobile cloud computing has grown exponentially from its inception to current vast research and application development industry. It is pre-

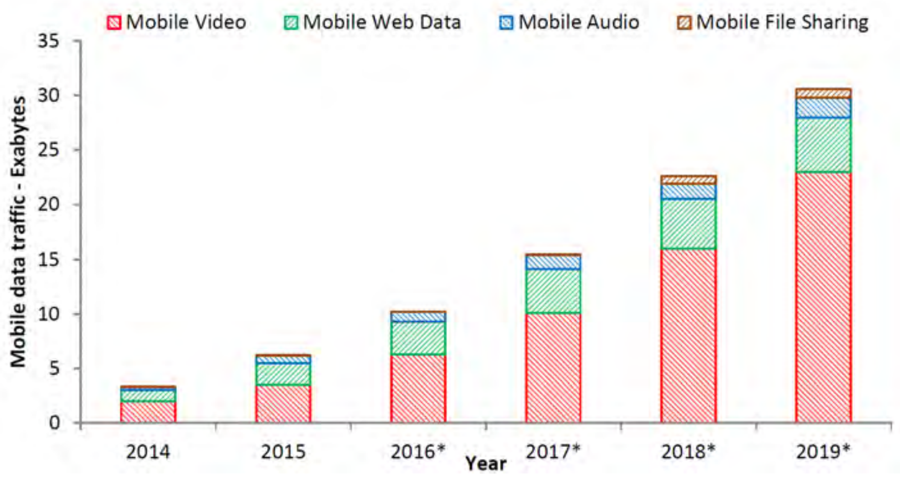


Figure 1.1: Distribution of Mobile Data Traffic Among Applications From 2014 To 2019

dicted that the mobile cloud market will grow to over $46.90 billion by 2019. Similarly, mobile data traffic grew 74% globally in 2015. It is predicted that out of total mobile data traffic, the cloud-based traffic will increase from 81% in 2014 to 90% in 2019. It is estimated that 75% of total data accessed through the mobile networks by 2020 will be of multimedia applications (Statista, 2015; Cisco, 2015). To limit the energy consumption of increasing multimedia content on mobile devices, computational offloading techniques are unavoidable. Offloaded mobile applications execute seamlessly over the cloud servers while providing energy and execution time efficiency to the mobile users. Figure 1.1 shows the increasing multimedia content share in mobile traffic that motivates the development of computational offloading frameworks for multimedia applications.

Specialized hardware known as vector processors and corresponding instruction sets are devised to adapt to the computational demands of multimedia applications. The vector processors increase the energy consumption of mobile devices due to their data and compute intensity. Therefore, vector instructions are offloaded to the cloud server while limiting local execution on energy-constrained mobile devices. The offloading requires that vector instructions are cross-platform compatible. In reality, vector instructions are platform-dependent and vary from one ISA to another. Hence, a vector instruction based application programmed for one architecture is often not executable on another (Fu, Wu, Liu, Hong, & Hsu, 2015). The challenge of offloading and seamless execution of vectorized smartphone applications on cloud servers for energy efficiency drives and motivates this research work.

1.3 Statement of The Problem

Multiple techniques enable offloading of computations between heterogeneous MCC architectures. These techniques are **(a)** system virtualization, **(b)** application virtualization, and **(c)** process or native code migration (Shuja, Gani, Naveed, Ahmed, & Hsu, 2016). System virtualization addresses the heterogeneity of MCC architectures through abstraction and migration of a full virtual machine (VM) instance. The network overhead of VM migrations is very high for mobile devices. Due to high network latency, system virtualization based MCC offloading is not deemed feasible for most of the mobile networks (Satyanarayanan, 2015). On the contrary, application virtualization has high computational overhead. Application virtualization techniques, such as Java-based Dalvik VM of Android leads to low performing code as compared to native code. The Dalvik code is interpreted to bytecode that is platform-independent and can be executed on any Java-based VM. The intermediate bytecode translations of Dalvik VM and their platform-specific compilations lead to computational overhead. Hence, application virtualization is also not favorable for compute-intensive vectorized applications (Oh, Kim, Choi, & Moon, 2012).

Compiled or pre-compiled native code can be migrated from the smartphone to the cloud server. The heterogeneity of the architectures demands Dynamic Binary Translation (DBT) of the compiled code at the cloud server. The DBT process can slow the execution of the code by large factors due to the overhead of instruction translation (Nimmakayala, 2015). Moreover, DBT techniques often translate vector instructions to scalar instructions, hence, losing the advantage of vectorization. On the contrary, native pre-compiled code based MCC offloading techniques require re-compilation and translation of mobile-based libraries to server-based libraries. In the case of vectorized multimedia applications, it is necessary to translate and map the SIMD instruction library of the mobile device to the cloud server (Shuja, Gani, Naveed, et al., 2016).

Most of the current state-of-the-art MCC offloading frameworks are dependent on system and application virtualization while addressing the requirements of heterogeneous computing architectures (Ahmed, Gani, Khan, Buyya, & Khan, 2015). Less frequently, compiled code migration is utilized to offload computations from the mobile device to the cloud server (G. Lee et al., 2015). Vectorized applications require specialized techniques for cross-platform execution of SIMD instructions. An efficient MCC offloading framework that translates and maps guest vector instructions to corresponding target vector instructions such that the vectorized applications retain their performance needs to be developed. Based on the aforementioned discussion, we can state that the problem of efficient SIMD translations in an MCC offloading framework has not been the focus of research. The research gap discussed above leads to the problem statement of this thesis.

MCC technologies empower applications to execute efficiently over cloud servers. Applications rich with SIMD instructions increase the performance of smartphones. Generic MCC frameworks do not address the heterogeneity of computing architectures for efficient translation of SIMD instructions. MCC offloading frameworks lead to higher computational overhead and inefficient vector-to-scalar translations of SIMD instructions. In MCC offloading frameworks, vector instructions that can be executed in one cycle are translated to scalar instructions that take several cycles. This leads to increased execution time and energy consumption on the cloud server as the applications require more instruction cycles when offloaded to the cloud. As a result, the overall performance of vectorized multimedia applications degrades in MCC offloading frameworks.

1.4 Statement of Objectives

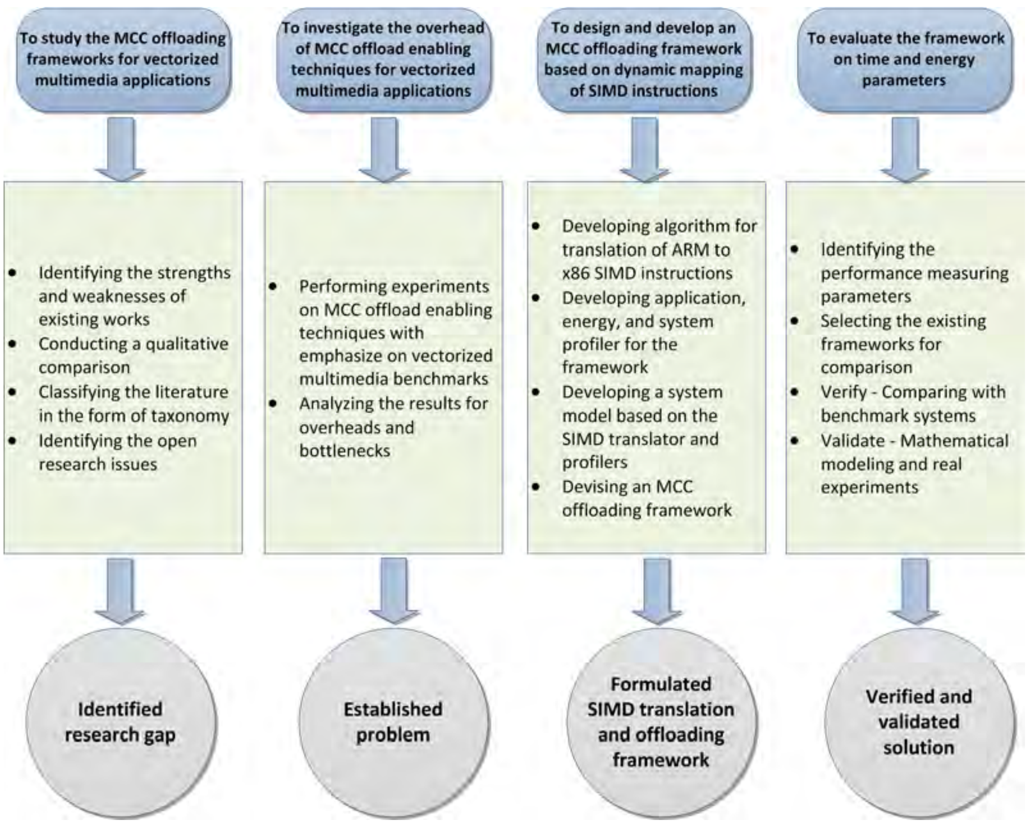
In this thesis, the problem of inefficient SIMD instruction translations in MCC offloading frameworks is addressed. The aim of our research is to enhance energy and time efficiency of pre-compiled code based MCC offloading frameworks through efficient translation of offloaded SIMD instructions. The objectives of this research are as follows.

- To study the MCC offloading frameworks from the perspective of offload enabling techniques to gain insights to the performance limitations of current state-of-the-art solutions.
- To investigate the overhead of MCC offload enabling techniques to reveal inefficiency in SIMD instruction translations.
- To design and develop an MCC offloading framework based on dynamic mapping of SIMD instructions that supports heterogeneity of computing architectures while providing energy and time efficiency to mobile devices.
- To evaluate the proposed framework for energy and time efficiency and compare it with the state-of-the-art MCC code offloading frameworks.

1.5 Research Methodology

The research carried out in this thesis can be divided into four main phases according to the four objectives defined in Section 1.4. Figure 1.2 illustrates the proposed research methodology along with the details of the research objectives corresponding to each phase of research.

The state-of-the-art MCC offloading frameworks with an emphasize on native code offloading are reviewed in the first phase. The MCC offloading frameworks are classified based on the offload enabling techniques for heterogeneous computing architectures. Further, the MCC offload enabling techniques are classified as, **(a)** system virtualization, **(b)**



To study the MCC offloading frameworks for vectorized multimedia applications
To investigate the overhead of MCC offload enabling techniques for vectorized multimedia applications
To design and develop an MCC offloading framework based on dynamic mapping of SIMD instructions
To evaluate the framework on time and energy parameters
• Identifying the strengths and weaknesses of existing works
• Conducting a qualitative comparison
• Classifying the literature in the form of taxonomy
• Identifying the open research issues
• Performing experiments on MCC offload enabling techniques with emphasize on vectorized multimedia benchmarks
• Analyzing the results for overheads and bottlenecks
• Developing algorithm for translation of ARM to x86 SIMD instructions
• Developing application, energy, and system profiler for the framework
• Developing a system model based on the SIMD translator and profilers
• Devising an MCC offloading framework
• Identifying the performance measuring parameters
• Selecting the existing frameworks for comparison
• Verify - Comparing with benchmark systems
• Validate - Mathematical modeling and real experiments
Identified research gap
Established problem
Formulated SIMD translation and offloading framework
Verified and validated solution

Figure 1.2: Proposed Research Methodology

application virtualization, and **(c)** native code migration. The qualitative assessment of the MCC offload enabling techniques sheds light on the computational overhead of application virtualization based solutions and communicational overheads of system virtualization based solutions. Therefore, native code offloading frameworks and the optimization of DBT process are further reviewed. In particular, the translation and porting techniques for SIMD instructions are debated. Through a comprehensive literature review, the research issue of native code offloading for SIMD instruction based applications in MCC is identified.

The second phase of this research involves the investigation and performance evaluation of MCC offload enabling techniques, namely, system virtualization, application virtualization, and DBT for compiled code migration. Multimedia benchmarks are utilized in the performance evaluation to investigate the overheads of the MCC offload enabling techniques. The evaluation is exercised to reveal the application execution time and performance of the MCC offload enabling techniques.

The third phase of this research work proposes SIMD instruction translation and offloading framework, SIMDOM, that enables execution of applications on heterogeneous cloud and mobile architectures. Based on the SIMD translator algorithm, a native code based MCC offloading framework and corresponding system model are formulated. The basic objective of the SIMDOM framework is to reduce the energy consumption of the mobile device while cyber-foraging computations to nearby cloud servers. To reduce the computational and communicational overhead of offloading, SIMDOM framework focuses on native code offloading and addresses the heterogeneity of the guest and host architectures through re-compilation and vector-to-vector instruction translations.

The SIMD translator based SIMDOM offloading framework is evaluated in the last phase of our research. The basic implementation of the SIMD translator is concerned with translating and mapping ARM SIMD instructions to the corresponding x86 SIMD instructions. SIMD to scalar translations are avoided as they degrade the overall performance of the system. A mathematical model is proposed to derive the energy and execution time efficiency of the framework. The mathematical model is validated by experimental results. Moreover, the effectiveness of the framework for SIMD instruction translation and offloading is verified based on the energy, execution time, and performance parameters. Furthermore, the framework performance is compared with state-of-the-art code offloading and translation frameworks in MCC.

1.6 Thesis Layout

This research entitled, "Integrated Vector Instruction Translator and Offloading Framework for Mobile Cloud Computing" comprises of an extensive study. Therefore, the thesis is divided into chapters for reader understandability. The thesis layout is expressed in Table 1.1.

Table 1.1: Thesis Layout

Chapter	Why	How
Introduction	**(a)** To emphasize the motivation for research **(b)** To state the problem and objectives	**(a)** By stating the rational for undertaking the research **(b)** By formally writing the statement of problem and statement of objectives **(c)** Present the thesis organization
Literature review	**(a)** To classify and investigate the strengths and weaknesses of the state-of-the-art literature **(b)** To identify the open issues	**(a)** By critical analysis of the existing frameworks **(b)** By formulation of the taxonomy and comparison based on the taxonomy
Problem analysis	**(a)** To investigate the severity of the overhead of existing MCC offload enabling techniques **(b)** To identify the impact of SIMD instructions and their translations on application performance	**(a)** By performance analysis of MCC offload enabling techniques on multimedia benchmarks **(b)** By analyzing performance of vector and SIMD-based benchmarks
SIMDOM framework	**(a)** To provide details of SIMDOM framework, algorithms, and system model	**(a)** By providing pseudo code of the SIMD translator and offloading frameworks
Evaluation	**(a)** To discuss the framework evaluation parameters and their analysis **(b)** To detail the data collection methodology	**(a)** By explaining the methods and tools utilized in data collection **(b)** By reporting and analyzing collected data
Results and discussion	**(a)** To highlight the effectiveness of the proposed solution by analyzing the experimental results **(b)** To verify and validate the experimental results	**(a)** By discussing the insights gained from the experimental results **(b)** By comparing the SIMDOM framework with state-of-the-art native code offloading and translation frameworks (Qemu) **(c)** By comparing experimental results with the mathematical model
Conclusion	**(a)** To summarize the findings of the research work and stress the significance of the proposed solution **(b)** To discuss the limitations of the research work and propose future directions of the research	**(a)** By re-examination of the research objectives

Chapter 2 presents the literature review for MCC offloading frameworks and offload enabling techniques. We emphasize on native code offloading techniques that provide lower offloading overhead. Moreover, techniques for cross-platform translation of SIMD instructions in heterogeneous MCC architectures are focused. Qualitative comparison and critical analysis in the aforementioned research directions based on the parameters derived from the corresponding taxonomies is provided. The research issues highlighted by the literature review reveal the need for a framework for cross-platform execution of

SIMD instructions in heterogeneous MCC architectures.

Chapter 3 reports the performance evaluation of MCC offload enabling techniques. Vectorized benchmarks are utilized in most of the experiments to determine the overhead of MCC offload enabling techniques. The computational overhead of system virtualization (VirtualBox), application virtualization (Dalvik), and DBT (Qemu) is analyzed to gather insights to the performance limitations of the MCC offload enabling techniques. The analysis shows that the existing MCC offloading techniques have either high computational or high communicational overhead. Moreover, the performance gain for SIMD instruction based native applications and their cross-platform translations are analyzed.

Chapter 4 describes the SIMDOM framework for translation and offloading of SIMD instruction based vectorized applications in MCC. Each module of the SIMDOM framework is described in detail with appropriate examples. Pseudo codes of the SIMD translator and offloading frameworks are also described. Furthermore, a system model for the SIMDOM framework is formulated in terms of energy optimizations.

Chapter 5 reports on the evaluation methodology for the SIMDOM framework. The experimental setup is described with accompanying devices and vector application benchmarks. The data collection methodology regarding experimental and mathematical model parameters is described. Moreover, SIMD translator is analyzed for accuracy while application profiler is analyzed for overhead in terms of execution time. Furthermore, case studies are provided to derive the communicational overhead of system and application virtualization based MCC offloading techniques.

Chapter 6 presents the results of experimental evaluation of the SIMDOM framework to prove its significance and efficiency. The experimental evaluation is based on two primary parameters, i.e., energy and execution time. The SIMDOM framework is compared with the native code offloading and translation framework of Qemu. Moreover, the SIMDOM framework is analyzed on varying values of system parameters, such as device sleep time and benchmark size. Furthermore, the system model of SIMDOM framework is validated with experimental data.

Chapter 7 concludes this research work by re-visiting the research objectives. The chapter summarizes the contributions of this research work, highlights its significance, and lists its limitations. Moreover, future research directions are provided.

CHAPTER 2: OFFLOADING FRAMEWORKS IN MCC

This chapter presents a literature review on the native code offloading frameworks in MCC targeting vectorized multimedia applications. The purpose of this chapter is to detail the literature work related to our problem domain and to identify the potential research issues in the field of MCC native code offloading frameworks. The primary research issue identified through the literature review is that the cross-platform translation of native code in current MCC frameworks is not efficient. The taxonomies are devised with reference to the MCC offloading frameworks, native code cross-platform translation, and SIMD instruction porting techniques. Qualitative comparison of the state-of-the-art research works is detailed in each section. The chapter also provides the basic knowledge of the technical elements found in the thesis, such as the MCC architecture, computational offloading, DBT techniques, and SIMD instructions.

In Section 2.1 we describe the architecture of a generic MCC framework, the process of computational offload, and the MCC offload enabling techniques. Rest of the sections are divided into three subsections, i.e., taxonomy, review, and comparison of the state-of-the-art research works. Section 2.2 provides a survey of the MCC offload enabling techniques from the perspective of system virtualization, application virtualization, and native code migration. We focus on native code migration and survey the techniques of DBT for cross-platform code migration between heterogeneous ISAs in Section 2.3. We also discuss the DBT optimization techniques in this regard. In Section 2.4, we focus on DBT and porting of SIMD instructions that are commonly found in multimedia applications. We list the studies that focus on porting of SIMD instructions across heterogeneous ISAs. Section 2.5 lists the identified research issues in current MCC offloading, DBT optimization, and SIMD instructions translation techniques. Section 2.6 provides the concluding remarks.

2.1 MCC Architecture and Computational Offload

The MCC paradigms functions in two major directions. Firstly, cloud computing technologies can enhance mobile features with cloud augmented applications. The major examples of such applications are cloud enabled email services (Gmail), social media applications (Facebook), and messaging applications (WhatsApp). Secondly, resource intensive applications can utilize offloading techniques to migrate application instances to a cloud server. The computational offloading is also termed is cyber-foraging (Balan, Gergle, Satyanarayanan, & Herbsleb, 2007). The major focus of this study is the computational offloading based MCC techniques that offload vectorized applications from resource-scarce mobile devices to resource-rich cloud servers.

Over the past decade, cloud computing technologies have gained immense popularity due to the underlying virtualization and pay-as-you-go model. The cloud services are hosted in large-scale data centers that house thousands of processing and storage devices with high energy requirements (Shuja, Bilal, et al., 2016). On the contrary, the primary goal of modern mobile devices is to provide end-users with interactive features within the constraints of limited battery, computation power, and limited network accessibility. However, smartphone applications, such as voice recognition, augmented reality, and personal health monitoring are pushing the boundary of computational power and testing the long-term battery operations (Shuja, Gani, Naveed, et al., 2016).

Modern mobile devices are multicore, with gigabytes of memory and gigahertz of computational power. Still, smartphones are and will always remain resource-constrained as compared to desktop systems and cloud servers. For example, the latest server (Xeon E5) is ten times more powerful than the latest smartphone device (Samsung S5) (Satyanarayanan, 2015). While the absolute ability of mobile devices will increase over the years, their relative ability compared to cloud servers will remain low.

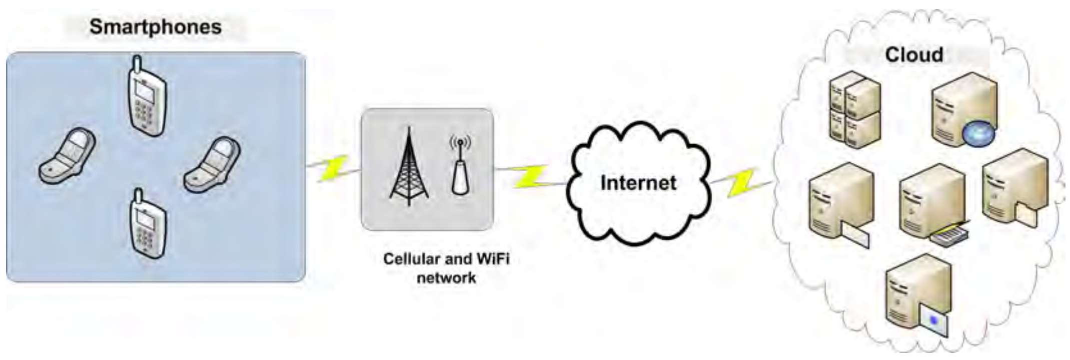


Figure 2.1: Generic Architecture of MCC Offloading

Cloud Data Centers (CDC) are an ideal candidate for augmentation with mobile devices. CDCs comprise of thousands of server, storage, and network devices interconnected with each other to provide pay-as-you-go business model to end users (Rehman, Liew, Wah, Shuja, & Daghighi, 2015). Keeping in view the resource scarcity of mobile devices, the paradigm of Mobile Cloud Computing (MCC) was established with the amalgam of Mobile Computing (MC) and Cloud Computing (CC) (Kumar & Lu, 2010). Precisely, MCC can be defined as an integration of cloud computing technology with mobile devices to make the mobile devices resource-full in terms of computational power, storage, and energy (Khan, Othman, Madani, & Khan, 2014). The generic architecture of MCC is depicted in Figure 2.1.

The main objective of the MCC offloading techniques is to enable mobile devices to operate for longer periods while saving energy utilized in compute-intensive tasks. Along with this primary objective, computational offloading can also enable time efficiency if the network delay is minimal. To reduce the network latency of MCC operations, cloudlets have been proposed. Cloudlet based servers lie in the user proximity to provide low response time and low network latency for devices. However, cloudlets are not as resource-rich as cloud based servers (Fernando, Loke, & Rahayu, 2013; Khan et al., 2014).

Computation offloading is a process that enables mobile devices to migrate compute-

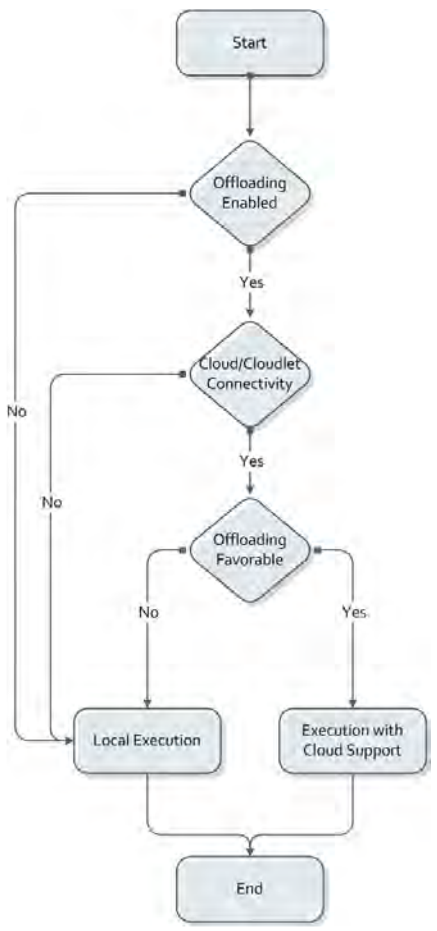


Figure 2.2: Process Diagram of Computational Offload

intensive tasks to nearby cloudlet or cloud server. The voice command enabled applications in smartphones are the most common example of computational offload. The voice is pre-processed on the smartphone and sent to the cloud server to convert voice data into text. One of the earliest examples of cloud augmented computational offloading application in smartphones was Apple's voice recognition application Siri (Flinn, 2012). The computational offload can be in the form of process state, compiled or pre-compiled code, complete application, or an Operating System (OS) instance (Virtual Machine) (Shuja, Gani, Ahmad, et al., 2016). The process diagram of the computational offload is presented in Figure 2.2.

In the simplest scenario, if smartphone has offloading enabled, it checks for cloud connectivity. If offloading is not enabled, or cloud connectivity is absent, the process executes locally. Based on the offloading process parameters, the feasibility of offloading is analyzed in the offloading scenario. If the cloud execution is feasible, such that it saves smartphone energy without hindering the real-time response of the application, then the process is offloaded to the cloud. The feasibility of the computational offload is determined by four basic queries, i.e., what to offload, when to offload, where to offload, and how to offload (Flores et al., 2015; Othman, Khan, Abid, Madani, et al., 2015).

2.1.1 MCC Offload Enabling Techniques

A number of possible techniques enable offloading of data, application, or the complete mobile workspace to the cloud over the internet. An overview of MCC offload enabling techniques is provided in the subsections below.

2.1.1.1 System Virtualization

System virtualization enables a Virtual Machine (VM) to reside and migrate between multiple heterogeneous physical hosts, such as a smartphone and a cloud server. System virtualization technology is the backbone of cloud services. Cloud servers are virtualized to abstract the underlying resources for the purpose of sharing among multiple clients. A virtualization solution comprises of three major components: **(a)** a hardware device to be virtualized, **(b)** a hypervisor or Virtual Machine Monitor (VMM), and **(c)** guest OS or VM that resides over the virtualized hardware. Resource consolidation, energy efficiency, and fault tolerance are the major use cases of virtualization in server space devices (Shuja, Gani, Naveed, et al., 2016; Mustafa, Nazir, Hayat, Madani, et al., 2015).

Smartphones and mobile devices are not suitable for system virtualization due to large performance overhead. Mobile devices are resource constrained with limited processing power, memory, and battery. Hosting an additional OS on a resource-constrained mobile device is challenging as it imposes serious performance overheads and decreases the real-time responsiveness of the devices. Moreover, ARM is the dominant mobile architecture. System virtualization solutions for ARM devices rely heavily on paravirtualization techniques and trap-and-emulate procedures to share mobile components among guest OSs. The paravirtual techniques result in large overhead, thereby, compromising the real-time capability of mobile devices by burdening the already constrained resources (Shuja, Gani, Bilal, et al., 2016). Therefore, smartphones are often not virtualized at the system level. Moreover, smartphone OSs, such as Android and Windows Mobile do not have support for system virtualization. Instead, smartphones enable application virtualization in their ecosystem (Shuja, Gani, Naveed, et al., 2016; Shuja, Gani, & Madani, 2014).

2.1.1.2 Application Virtualization

Applications written in platform-independent languages, such as Java, facilitate remote execution of tasks over heterogeneous platforms. An application executing in an application VM can be offloaded to a similar virtual instance over the cloud (Chun, Ihm, Maniatis, Naik, & Patti, 2011). Modern mobile devices are often equipped with application virtualization solutions, such as Java based Dalvik Runtime in Android OS and .Net runtime environment in Windows Mobile. Due to this reason, most of MCC offloading techniques are enabled by application virtualization. Android OS has captured more than 80% of smartphone market share (Robinson & Weir, 2015; Ahmed, Gani, Khan, et al., 2015). Dalvik is an application VM in Android OS that executes Java-based applications.

Applications written in Java are compiled to bytecode by Dalvik and can be executed on any Java Virtual Machine (JVM). There are many disadvantages of application virtualization approach to computational offloading in MCC. Firstly, applications executing in the application VM suffer from computational overhead. For instance, the overhead of a Java application can be twice as high as native C application (Sartor, Lorenzon, & Beck, 2015). Native C applications are favored for compute-intensive tasks (Yadav & Bhadoria, 2015; Shuja, Gani, Ahmad, et al., 2016). Secondly, application virtualization puts a restriction on application development language and execution environment.

2.1.1.3 Native Code Migration

Process state or native pre-compiled code can be migrated from ARM based mobile devices to Intel based cloud servers. As compared to system and application virtualization, the amount of data to be offloaded is very less in process migration. Process code executing on a mobile device cannot be run as it is over a server due to the heterogeneity of computing architectures (Shuja, Bilal, et al., 2016). Instruction Set Architecture (ISA) emulation or Dynamic Binary Translation (DBT) is required while offloading code between heterogeneous processors. Emulated mobile instances are widely used to support code offloading in MCC. The most commonly used ARM ISA emulators are Qemu (Bellard, 2005) and gem5 (Binkert et al., 2011). ISA emulation is inherently slow due to instruction translation overhead. Existing emulators have been largely developed with the objective to test mobile applications on server architectures. Therefore, the performance of emulators is not considered in the development process. The performance of an ARM emulated system can be 10X slower than the physical system (Hong et al., 2014). Hence, code migration between heterogeneous processors is often not deemed feasible.

2.1.2 Cloud Augmentation for MCC Offloading Frameworks

System virtualization based MCC frameworks offload a VM image from the smartphone to the cloud server. The cloud provides Infrastructure-as-a-Service (IaaS) model where clients can access processing, storage, and network resources. The main advantage of this approach is that the cloud servers are generally virtualized and provide a standard set of tools for task execution (Shuja, Bilal, et al., 2016). Application virtualization based MCC frameworks migrate an application from the smartphone to the cloud server. Application virtualization, such as that exercised in Dalvik VM, allows offloading of Java compiled bytecode that can be executed in any JVM over the cloud. Execution environ-

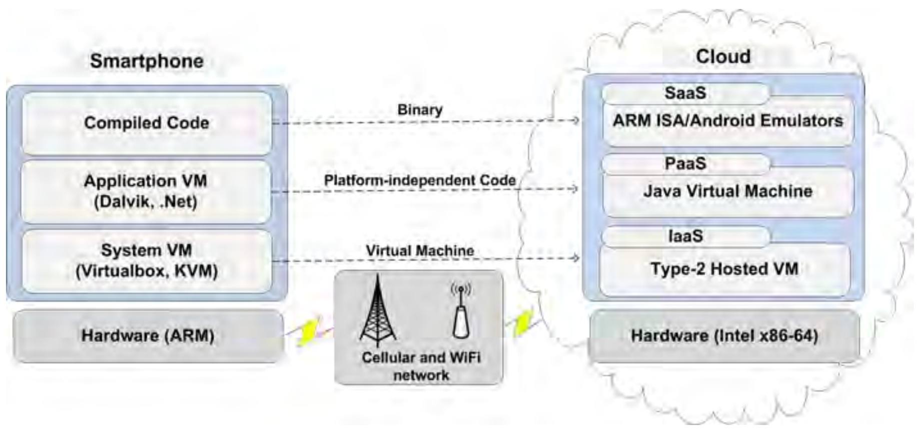


Figure 2.3: MCC Offload Enabling Techniques

ments such as the JVM are provided in the form of Platform-as-a-Service (PaaS) by the cloud providers (Chun et al., 2011; Othman et al., 2015). Cloud computing paradigm also provides Software-as-a-Service (SaaS) model that enables compiled code of a mobile device to be emulated on a cloud server. Compiled code has the least communication overhead. However, mobile and cloud server hardware profiles are heterogeneous. Therefore, code compiled for a mobile device requires emulation over the cloud server, which is an inherently slow process (Dall & Nieh, 2014). An illustration of MCC offload enabling techniques within the generic MCC framework is depicted in Figure 2.3.

The energy consumption of resources is also a critical factor in cloud computing. Software, hardware, and renewable energy based techniques are applied to CDC facilities for energy efficient and sustainable operations (Shuja, Bilal, et al., 2016; Shuja, Gani, Shamshirband, Ahmad, & Bilal, 2016). Software based cloud energy efficiency techniques employ resource scheduling algorithms to match workload demand and energy consumption. On the contrary, hardware based techniques exploit hardware power states and circuit properties to achieve energy efficiency (Shuja, Bilal, et al., 2016). CDCs also exploit renewable energy resources for sustainable operations. All CDC resources, such as servers, network devices, power distribution, and cooling systems are considered in software and hardware based energy efficiency techniques. However, the basic objective of MCC paradigm is to save mobile device energy irrespective of cloud energy consumption. The notion behind this objective is the difference in the cloud and mobile resources. Mobile devices are battery operated with limited power charge opportunities contrary to the continuous power input for cloud resources. Moreover, the cloud resources are near infinite as compared to mobile device resources (Satyanarayanan, 2015). Therefore, MCC offloading frameworks generally do not consider the energy consumption of cloud resources.

2.2 MCC Offloading Frameworks

In the below subsections, we will discuss the MCC offloading frameworks categorized on the basis of offload enabling techniques. First, we provide a taxonomy of the MCC offloading frameworks based on the identified qualitative parameters. Further, we provide a review of state-of-the-art MCC offloading frameworks. Lastly, we present a comprehensive comparative analysis of the MCC offloading frameworks based on the identified qualitative parameters.

2.2.1 Taxonomy of MCC Offloading Frameworks

In this subsection, we provide a comprehensive taxonomy of MCC offloading frameworks. The taxonomy of MCC offloading frameworks based on the qualitative parameters is presented in Figure 2.4.

MCC Offload Enabling Technique: System virtualization, application virtualization, and native code migration enable the MCC offloading frameworks. Each of the MCC offload enabling technique leads to varying computational and communicational overhead. System virtualization based MCC offloading is enabled by virtualization solutions, such as Xen and VMware. Application virtualization is facilitated by the Dalvik VM in Android OS and .Net CLR in Windows Mobile. Native code migration is enabled by cross-platform ISA emulators, such as Qemu and gem5 (Shuja, Gani, Naveed, et al.,

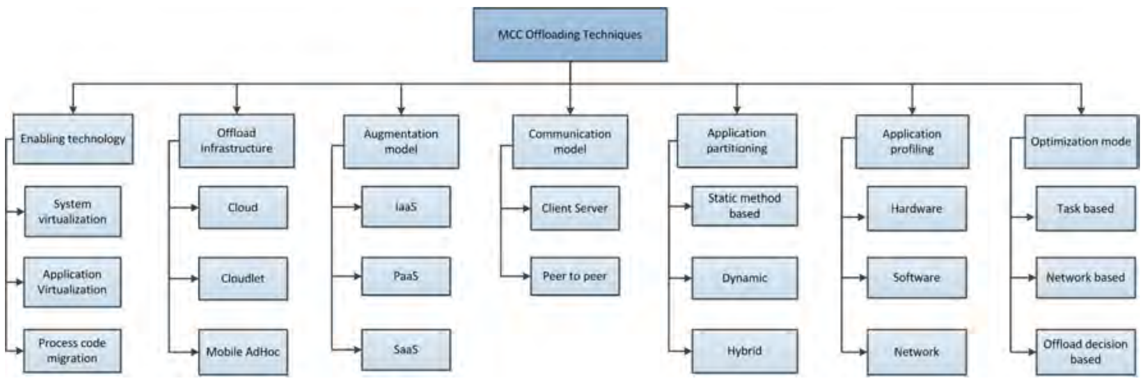


Figure 2.4: Taxonomy of MCC Offloading Frameworks

2016; ur Rehman, Sun, Wah, Iqbal, & Jayaraman, 2016).

Offload Infrastructure: The offload infrastructure facilitating MCC frameworks can be a cloud, cloudlet, or mobile ad-hoc. Task offload to cloud server faces the challenge of proximity with the mobile clients. This challenge is addressed by cloudlet based infrastructure that does not possess the near infinite resources of CDCs. However, the cloudlet is a proximate and less capable version of a resource-rich CDC that can limit the network latency between mobile and cloud servers. Mobile ad-hoc infrastructure is formed by proximate collaborative mobile devices to facilitate each other in compute-intensive task executions (Yaqoob et al., 2016; Whaiduzzaman, Naveed, & Gani, 2016).

Augmentation Model: The mobile client can augment to the CDC through three service models, i.e., IaaS, PaaS, and SaaS. System virtualization enabled offloading frameworks augment to the cloud through (IaaS) model. Application virtualization based MCC offloading frameworks augment through PaaS services such as JVM while native code migration based MCC offloading frameworks utilize SaaS services through cross-platform emulators (Ahmed, Gani, Sookhak, et al., 2015).

Communication Model: The mobile and cloud entities in the MCC framework can communicate in a client server model or peer-to-peer model. Client-server model is generally used for the resource-scarce mobile device and resource-rich cloud server. On the contrary, in the case of mobile ad-hoc cloud, the mobile devices communicate in a peer-to-peer model where each mobile device can utilize the resources of another mobile device in a collaborative network (Yousafzai, Chang, Gani, & Noor, 2016).

Application Partitioning: The candidate application for the cloud offload can be partitioned into a local execution part and remote execution part through a static method based annotations, dynamic, or hybrid analysis. The static method annotates the resource or compute-intensive part of the application as a remoteable entity. The dynamic analysis based application partitioning profiles the application, network and cloud resources to make an optimal decision regarding remoteable part of the application. However, the overhead of finding the remoteable code based on dynamic analysis is high. The hybrid partitioning utilizes static annotations to minimize the overhead of finding the optimal partitioning of the application (Niu, Song, & Atiquzzaman, 2014; Shaukat, Ahmed, Anwar, & Xia, 2016).

Profiler: Three types of profilers can be used while deciding upon the feasibility of application offloading. These are network profiler, hardware (CPU) profiler, and software (application) profiler. Most of the frameworks only utilize the application traces to lower the complexity of offloading module (Ahmad, Gani, Hamid, Xia, & Shiraz, 2015).

Optimization Model: The MCC offloading frameworks apply optimization techniques to lower the communicational or computational cost of the process. These optimization techniques can be categorized as, **(a)** task based optimization techniques that optimize the offloaded task, **(b)** network optimization techniques that either compress the candidate code or find proximate cloud resources, and **(c)** offload decision optimization techniques (Shuja, Gani, Ahmad, et al., 2016).

2.2.2 Review of MCC Offloading Frameworks

In this section, we will review the MCC offloading frameworks from the perspective of the offload enabling techniques. We will review state-of-the-art MCC frameworks enabled by system virtualization, application virtualization, and native code migration.

2.2.2.1 System Virtualization based MCC Frameworks

System virtualization based MCC frameworks rely on hardware virtualization technologies, such as VMware on both mobile client and cloud server. A VM is migrated from the mobile device to the server that executes the VM with the help of the VMM. There are two techniques to transport a VM infrastructure from the mobile device to the cloud server; **(a)** VM migrations and **(b)** VM synthesis. If a decision is made to migrate the VM from the mobile to the cloud, the VM at the mobile client is stopped and its state is saved in terms of CPU, memory, and disk content and context. The state of the VM along with its disk contents are transferred page by page to the server. However, the size of a VM instance can be in GBs, which is often too large for bandwidth-scarce mobile networks. The second technique for VM augmentation on a cloud server is dynamic VM synthesis. In VM synthesis, the basic building blocks of the VM at the mobile client are sent to the cloud server that converts them into fully functional VM (Satyanarayanan et al., 2015; Shuja, Gani, Shamshirband, et al., 2016).

ThinkAir is a framework based on VM migration for task offloading in MCC (Kosta, Aucinas, Hui, Mortier, & Zhang, 2012). ThinkAir is based on the assumption that the mobile network bandwidths will grow over the years to allow low round trip times for VM migrations. The framework of ThinkAir consists of the Application Programming Interfaces (APIs) for mobile applications, compiler support for mobile and cloud server, mobile application execution controller, and execution flow controller on the server side. The programmer API works on annotated code marked for remote execution. The compiler provides support for generation of code suitable for x86 ISA based cloud servers. The execution controller makes decision of local or remote execution based on network parameters, historical application execution times, energy consumptions, and the cost of resources on the cloud servers. The execution flow controller on the server side further consists of three modules, namely, client handler to manage client offloading requests, cloud infrastructure in the form of customized VM (Android x86 port on VirtualBox), and automatic parallelizer that parallelizes the offloaded task execution on multiple cloud instances. The hardware profiler stores the state of hardware interfaces with parameters, such as CPU utilization and Wi-Fi power state. The software profiler records a number of parameters related to the software execution, such as number of instruction executed and overall execution time. The network profiler estimates parameters, such as the perceived network bandwidth. Moreover, the framework also incorporates an energy estimation model based on PowerTutor (L. Zhang et al., 2010). Experimental results show that both energy and time can be saved by cloud augmented remote execution of mobile applications.

The earliest proposal on system virtualization based cloud augmentation and **VM synthesis** was put forward by (Satyanarayanan, Bahl, Caceres, & Davies, 2009). To overcome the bandwidth constraints of mobile networks, the work proposes VM-overlay approach. A VM-overlay consisting of configuration parameters is migrated from the mobile to the cloudlet, which already possesses the base-VM. The base-VM is minimal configured VM which is able to construct the full VM from the VM overlay with all the CPU state, memory, and disk configurations. The cloudlet executes the VM until task completion and returns VM residue to the mobile device. The performance of this approach is dependent on the bandwidth to the cloudlet and cloudlet resources. The work provided proof-of-concept in the form of a prototype, Kimberly. The results show that for most of the VMs tested, the VM synthesis overhead is between one to two minutes. Most of the VM synthesis time is spent on decompressing the VM overlay.

2.2.2.2 Application Virtualization based MCC Frameworks

Most of MCC frameworks utilize the application virtualization support in Windows Mobile and Android OS for seamless application execution. Application virtualization allows intermediate or bytecode to be migrated between JVMs running on heterogeneous hardware platforms.

MAUI was the first framework for MCC offloading based on application virtualization enabled by Microsoft .NET CLR (Cuervo et al., 2010). CLR applications are compiled to the Common Intermediate Language (CIL). The CIL is compiled at execution time for the guest hardware ISA. MAUI framework is designed to offload application methods to the remote server which results in energy optimization. MAUI framework is based on managed codes for code portability for heterogeneous mobile and server architectures. Furthermore, to identify and migrate remoteable application instances to the cloud, MAUI utilizes program reflections. MAUI also utilizes serialization to determine the cost of the network for the process offload. The CPU cost, network cost, and the network state (bandwidth, latency, etc.) are used as input to linear programming problem for code offload decision module. MAUI Framework consists of three basic modules, namely, a client/server proxy for communication, a profiler, and a solver. Profiler instruments device and programs to gather the execution, energy, and data transfer requirements of applications. The major part of the solver is run on the servers while getting input from the mobile device module. The solver makes the decision of offloading based on inputs from the profiler module. The candidate code for offloading is statically annotated. Experiments show that MAUI is able to slash the energy consumption of resource-intensive application by a factor of eight.

CloneCloud is an MCC offloading framework based on Dalvik enabled application virtualization (Chun et al., 2011). CloneCloud is implemented as a flexible application partitioning and execution framework that migrates part of an application seamlessly to improve application execution time. The CloneCloud application migration has three components, namely, a smartphone based migrator thread that manages the migration and re-integration of migrating code, a node manager that manages communication between mobile and servers, and a partition database. The application partitioning module combines static analyzer and application dynamic profiler to fulfill execution objectives while ensuring application correctness constraints. The static analyzer identifies legal partitioning choices and set of exit and re-integration points for the application according to the constraints. The dynamic profiler collects the data required as input for the cost model of computational offload under different execution settings. The data is collected from the application execution trace. The framework examines the execution profile of each invocation based on its execution time. Furthermore, optimizer solver is utilized to contemplate the application methods that should be migrated based on the application constraints determined by the static analyzer. The Dalvik VM was modified to integrate the migration capability. Experiments showed up to 20X performance enhancement and energy savings for migrated applications.

Koukoumidis et al. (Koukoumidis, Lymberopoulos, Strauss, Liu, & Burger, 2011) propose a **pocket cloudlet** architecture that takes advantage of the unused non-volatile memory capacity of the proximate smartphones. As a result, the latency and energy wasted in approaching the distant cloud services are reduced. The pocket cloudlet framework leverages both the community and personal access models to increase the hit rate of user queries. In this manner, the pocket cloudlet decreases the overall service latency and energy consumption of smartphone devices. As the storage capabilities of existing smart devices are high, either partial or full cloud services can be replicated locally. In this manner, the mobile device is transformed into a pocket cloudlet. Pocket cloudlet provides three main features for the smartphones. Firstly, it lowers the latency to access cloud services and lowers burden on the cellular network while caching the cloud service on mobile devices. Secondly, as the user interactions for the services take place only on the mobile cloud, service personalization is possible. Thirdly, user privacy is preserved as user information is not stored on third-party cloud servers.

2.2.2.3 Native Code Migration based MCC Frameworks

Most of the research work done in code offloading MCC frameworks is either based on system virtualization or application virtualization. If a native code is migrated from the mobile device to the cloud server, it requires DBT or recompilation techniques for execution. Researchers so far have ignored the challenge of efficient DBT for MCC offloading frameworks and mostly focused on virtualization enabled code offloading.

Lee et al. (G. Lee et al., 2015) proposed an **architecture-aware native code offloading framework** for mobile devices. To support heterogeneous mobile and cloud ISAs, the native offloader relies on LLVM front-end compiler for intermediate representation (IR) (Zhao, Nagarakatte, Martin, & Zdancewic, 2012). The IR binaries are further compiled for native hardware at runtime by LLVM back-end compilers. Due to the machine independence of IR codes, seamless application migration is enabled. The native offloader allocates a unified virtual address space between the mobile device and the server to efficiently share memory objects without high overhead memory translations. The IR binaries are target compiled in four steps, namely, target selection, target ISA specific optimizations, memory unification code generation, and application partitioning. In target selection step, the framework ignores machine dependent tasks and selects tasks that can profit from offloading through static analysis. For memory unification, the compiler allocates a Unified Virtual Address (UVA) for mobile and server devices and updates its content from the mobile device. The compiler also takes care of variable memory sizes (e.g.,

Table 2.1: Comparison of MCC Offloading Frameworks

Framework	Enabling technique	Cloud infrastructure	Augment. model	Comm. model	Application partitioning	Profiling	Optimization model
ThinkAir (Kosta et al., 2012)	System Virtualization	Cloud	IaaS	Client server	Static	Network, application traces	Task optimization
VM-overlay cloudlet (Satyanarayanan et al., 2009)	System Virtualization	Cloudlet	IaaS	Client server	Static	NA	Network optimization
MAUI (Cuervo et al., 2010)	Application Virtualization	Cloud	PaaS	Client server	Dynamic	CPU, network traces	Task optimization
CloneCloud (Chun et al., 2011)	Application Virtualization	Cloud	PaaS	Client server	Hybrid	Application traces	Offload decision optimization
Pocket Cloudlets (Koukoumidis et al., 2011)	Application Virtualization	Mobile Adhoc	PaaS	Peer to peer	Dynamic	Application traces	Network optimization
Architecture-aware native offloading (G. Lee et al., 2015)	Native code migration	Cloud	SaaS	Client server	Dynamic	NA	Task optimization

32 bit, 64 bit) and endianness at this step with translation codes. The native offloader divides IR binaries for native and target architectures and inserts data communication codes to application sections that require migration during the application partitioning. In the optimization step, the target-specific function pointer and I/O management techniques are applied. The framework also adopts copy-on-demand to migrate live application configurations from mobile to the server.

2.2.3 Comparison of MCC Frameworks

We identified several parameters for qualitative analysis of MCC offloading techniques from the taxonomy. Table 2.1 lists the MCC offloading frameworks and their comparative analysis based on the identified qualitative parameters.

The MCC offloading frameworks generally utilize cloud infrastructure as the resource-rich offload entity (Cuervo et al., 2010; Chun et al., 2011). However, VM-overlay based framework (Satyanarayanan et al., 2009) utilizes cloudlet infrastructure to lower the overhead of VM migration between mobile clients and cloud servers while pocket cloudlet utilizes mobile ad-hoc infrastructure (Koukoumidis et al., 2011). The cloud augmentation model depends on the offload enabling technique. System virtualization, application virtualization, and process code migration based MCC offloading frameworks exploit IaaS, PaaS, and SaaS service models respectively.

Communication between the mobile device and cloud server can follow a client-server or a peer to peer model. Most of the MCC frameworks are based on a client-server model where a mobile device offloads tasks to a cloud server for further execution (G. Lee et al., 2015; Chun et al., 2011). However, pocket cloudlet utilizes a peer-to-peer model where multiple mobile devices harness each others memory capabilities to store and retrieve data that is otherwise accessed from cloud servers (Koukoumidis et al., 2011). The cloudlet and peer-to-peer communication models result in lower resource proximity and latency. Static application partitioning through annotation has low overhead. However, such methods are not scalable to a broader range of mobile applications. On the contrary, dynamic application partitioning frameworks, such as MAUI (Cuervo et al., 2010) can identify dynamically the optimal part of the application for cloud server execution through rigorous application profiling. However, such approach leads to complexity in MCC framework design which burdens the mobile client.

Resource-intensive profiling activities also lead to complexity in the MCC framework and adds latency to the offload decision process (Koukoumidis et al., 2011). MCC offloading frameworks, such as MAUI (Cuervo et al., 2010) and ThinkAir (Kosta et al., 2012) utilize multiple profilers for the offloading decision process. Most of the frameworks only utilize the application traces to lower the complexity of the offloading module. MAUI (Cuervo et al., 2010) and ThinkAir (Kosta et al., 2012) apply task parallelization on the cloud to achieve faster execution and optimization. On the contrary, VM-overlay (Satyanarayanan et al., 2009) and pocket cloudlets (Koukoumidis et al., 2011) apply network optimizations in the form of offload compression and proximate node calculation to optimize the offloaded task.

Most of the MCC offloading frameworks are enabled by either system or application virtualization. There are several implications to virtualization based approaches in

MCC offloading (Shuja, Gani, Naveed, et al., 2016; Y.-J. Kim et al., 2012; G. Lee et al., 2015). Firstly, system virtualization is only feasible in high bandwidth networks. Secondly, system virtualization requires that mobile OS be migrated from traditional Android, Windows Mobile, and iOS based OSs to virtualization based OSs, such as Linux. Hence, the mobile applications market reachable to smartphone users is not available in such scenario. Thirdly, application virtualization has performance overhead as compared to native applications. This performance overhead is due to intermediate interpretation techniques applied to platform independent application development frameworks, such as Java. The overhead of Java based applications can be twice as high as that of native C language applications (Shuja, Gani, Naveed, et al., 2016). Fourthly, application virtualization restricts the programming language domain for application developers. Fifthly, a study shows that up to 50% of mobile application code can be native. Therefore, application virtualization based schemes nullify the performance of half of the total application code that is written in native C (G. Lee et al., 2015).

Based on the aforementioned implications of virtualization based MCC offloading techniques it can be concluded that native code migration has much lower communicational and computational overhead. However, native code migration based MCC offloading frameworks have received lesser attention from research community. In the further subsections of this chapter, we focus on the ARM based cross-platform native code migration and emulation techniques and highlight related issues.

2.3 ARM Emulation Techniques

The native code migrated between heterogeneous ISAs requires DBT or emulation. Therefore, DBT is a necessary criterion for a complete MCC framework that consists of heterogeneous smartphone devices (ARM) and cloud server devices (x86). There are two commonly utilized open-source DBT or cross-platform ISA emulators for computer architectures; Qemu (Bellard, 2005) and gem5 (Binkert et al., 2011). A DBT system comprises of three main components; emulation engine, translator, and code cache. The translator fetches the guest binary code and translates it into host binary while placing the translated code in the code cache. The emulation engine controls the DBT by fetching the translated code from the cache and emulating it (Michel, Fournel, et al., 2011; X. Zhang, Guo, Chen, Chen, & Hu, 2015). In this section, we present applications of ARM emulations techniques, a taxonomy of ARM emulation techniques, review the ARM emulation tools, and compare ARM emulation techniques based on identified qualitative parameters.

2.3.1 Applications of ARM Emulation Techniques

ARM emulation techniques are adopted in multiple fields of research. The basic objective of DBT and emulation tools is to translate the application binary compiled for one ISA to another ISA. However, the ARM emulation techniques serve other purposes in research, such as computer architecture analysis and malware analysis. ARM emulation techniques are utilized to simulate computer architectures. Computer architectures that are expensive or not available in the market are simulated and analyzed with the help of ISA emulators. Moreover, before launching an architecture in the consumer market, extensive early design analysis is investigated on ISA simulators. Cache, memory, bus, and processor configurations are tested for performance before the manufacturing process. ARM ISA emulators are used in cases where actual hardware is not available for research purposes (Abadal, Martínez, Solé-Pareta, Alarcón, & Cabellos-Aparicio, 2016). Real hardware is expense and testing for malware analysis can lead to high costs. Therefore, simulated hardware ISA is used in malware analysis and security related research (Petsas, Voyatzis, Athanasopoulos, Polychronakis, & Ioannidis, 2014).

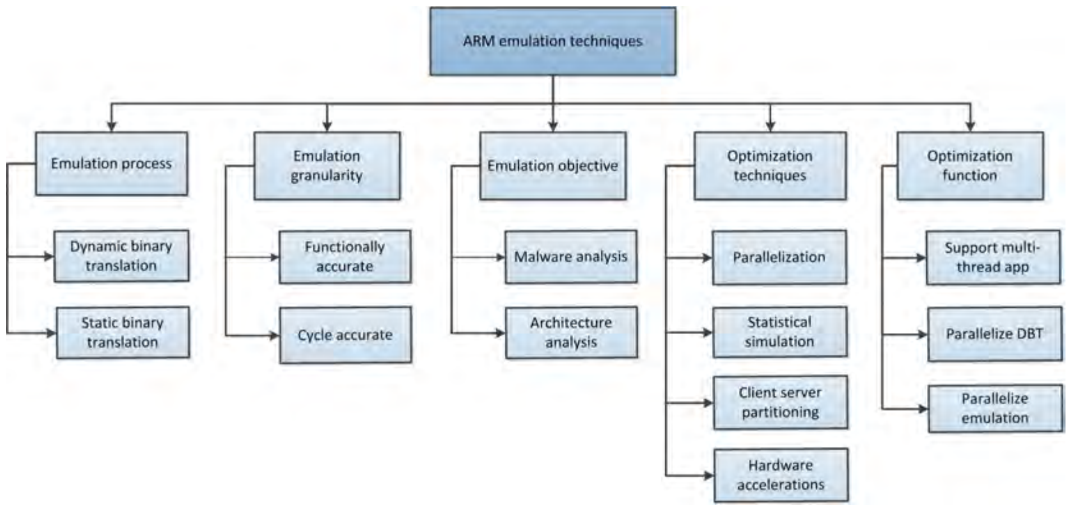


Figure 2.5: Taxonomy of ARM Emulation Techniques

2.3.2 Taxonomy of ARM Emulation Techniques

ARM emulation and emulation optimization techniques have varied parameters, such as objectives towards optimization of performance, emulation granularity, and emulation process. Most of the ARM emulators are open-sourced; hence, optimizations can be applied to existing designs. The taxonomy of ARM emulation techniques is provided in Figure 2.5.

We identified five parameters for classification of ARM emulation techniques. These are listed as follows,

Emulation Process: The cross-platform emulation process is either static or dynamic. SBT of ARM is difficult because it is common for data to be embedded within code sections. If symbolic debug information is not present in the application binary, extensive static analysis is required to distinguish data from code. Moreover, self-modifying code is difficult to identify and manage in SBT systems. On the contrary, DBT has the property to be versatile and adapt to the dynamic nature of the application binary. Therefore, most of the ARM emulation and virtualization solutions are based on DBT (Penneman, Kudinskas, Rawsthorne, De Sutter, & De Bosschere, 2016).

Emulation Granularity: There are basically two kinds of ISA emulators, i.e., functionally accurate emulators, and cycle accurate emulators. Functionally accurate emulators emulate the guest ISA functionality and translate the compiled instructions from one ISA to another. Moreover, functionally accurate ISA emulators emulate what the processor does, not how the instructions are actually executed in the pipeline. On the contrary, cycle accurate ISA emulators emulate the ISA with details up to the instruction cycle level. Cycle accurate emulators are sometimes called simulators. Qemu is a functionally accurate emulator while gem5 is a cycle accurate simulator (Shuja, Gani, Naveed, et al., 2016).

Emulation Objective: The basic objective of DBT and emulation tools is to translate the application binary compiled for one ISA to another ISA. The DBT tools are used for many purposes by the research community: **(a)** computer architecture analysis and **(a)** malware analysis (Petsas et al., 2014).

Optimization Technique: The optimizations techniques applied to enhance the performance of generic emulation process are **(a)** parallelization, **(b)** statistical sampling based execution, **(c)** hardware acceleration, and **(d)** client/server partitioning. Parallel emulation and support for multi-threaded programs leads to faster execution of applications. The parallelization of the emulation process results in significant performance benefits. Statistical sampling based simulation techniques choose points of interest in the application binary to be simulated while statistically measuring the overall execution path. Hardware acceleration requires that the target and guest ISAs be same so that a subset of guest instructions can be executed natively without translation. Client/server based partitioning of DBT process results in light-weight client optimizations and rigorous server side optimizations for frequently recurring code (Q. Guo, Chen, Chen, & Franchettit, 2015).

Optimization Function: The optimization function of the ARM emulation techniques can be **(a)** support for multi-threaded application emulation, **(b)** parallel translation of applications, and **(c)** parallel execution of multiple instances of the emulator (Ding, Chang, Hsu, & Chung, 2011; Hong et al., 2012; Wang et al., 2011).

2.3.3 Review of ARM Emulation Techniques

In the below subsections, we discuss the ARM emulation techniques and tools along with the optimization techniques applied to lower their performance overhead.

2.3.3.1 Qemu

Qemu is a retargetable dynamic code translation based functionally accurate ISA emulator that operates in two basic modes (Bellard, 2005). In full emulation mode, Qemu emulates a complete hardware device including many peripherals in the software. In user mode, Qemu executes a process compiled for one ISA on another ISA. Qemu is based on a front-end that translates the guest code to micro-operations and a back-end that translates the micro-operations to host code. Tiny Code Generator (TCG) transforms target instructions (the processor being emulated) via the TCG frontend to TCG micro-operations (micro-ops or IR) which are further transformed into host instructions (the processor executing QEMU itself) via the TCG backend. Micro-ops are an intermediate level of code translation that is machine independent. Hence, the micro-ops can be translated into any target code leading to the retargetable nature of Qemu. Qemu supports multiple guest ISA (e.g., x86, ARM, PowerPC, and SPARC) and multiple host ISAs (e.g., x86, ARM, PowerPC, SPARC, Alpha and MIPS).

The TCG performs two optimization passes including register liveness analysis and trivial constant expression evaluation when a block of instruction is fetched and translated into micro-operations. The micro-ops are then translated into host code with one-to-one mapping and stored in the translation cache. Qemu does block-chaining, i.e., it does not translate the binary from start to end. In fact, when a translation block is encountered, it is looked up in the translation cache. A block is translated if its translation is not

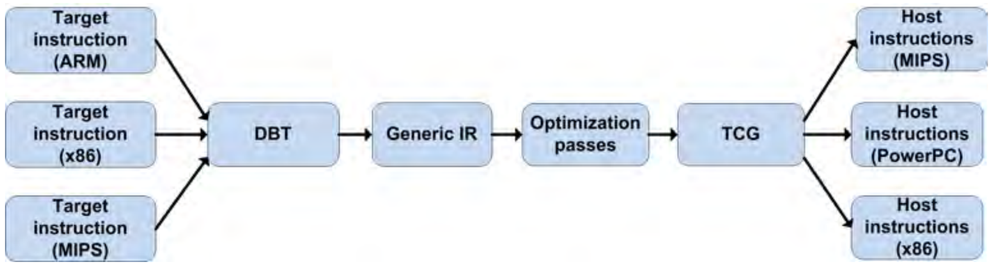


Figure 2.6: High Level Architectural Diagram of Qemu

present in the cache. When a translation block finishes execution, it chains itself to the next block of the execution without going to the main execution loop. Qemu emulates I/O access through signals and pipe functions. Qemu reads the interrupt controller before executing any instruction. Qemu emulates the MMU in the software to handle guest OS page tables. KVM-qemu provides faster emulation speed while utilizing hardware virtualization extensions. KVM virtualization can be used when the CPU supports it and the host architecture is the same as the guest architecture. Typically this means running an x86 guest on an x86 host or ARM guest on ARM host. The KVM module executes most of the instructions natively while interpreting only those instructions that require supervised access. A basic structural diagram of Qemu is provided in Figure 2.6.

2.3.3.2 Gem5

Gem5 (Binkert et al., 2011) simulator emulates multiple ISAs with a modular object-oriented design methodology. Similar to Qemu functionality, gem5 provides both process and full system emulation for ARM. It is based on discrete event simulations allowing for multiple CPU, memory, and device models. Moreover, four CPU models are supported, namely, out-of-order execution, in-order execution, simple atomic execution, and KVM based execution. Gem5 also supports two memory models, namely, classic hierarchical memory model and network connected memory model. The gem5 simulator achieves ISA independence by providing a single C++ base class for all instructions. All hardware components, such as, processor cores, caches, interconnecting devices, are modeled as a

SimObjects. Every SimObject is represented by a Python and a C++ class. The Python class defines the SimObject parameters while the C++ class defines its behavior and state. As gem5 is a cycle accurate simulator, it is used in studies that measure CPU performance, interconnect latencies of processor designs, and DRAM controller scheduling techniques.

ISA emulators and DBT tools are slow in performance than the actual physical devices they simulate. The slow performance of ISA emulators is due to multiple reasons listed as follows,

- Emulation overhead occurs due to the cross-ISA translation of instructions. The optimizations applied to the translated code can induce further performance overhead.
- Existing emulators have been largely developed with the objective to test mobile applications (ARM) on x86 architectures. The performance of emulators is not considered in the development process.
- Most of the optimized applications run several threads for performance optimization. However, such optimizations are lost on single-threaded emulators.
- Special instruction, such as Floating Point (FP) and Single Instruction, Multiple Data (SIMD) are translated to scalar instructions in the emulation process resulting in performance loss.

There have been several optimization works on the performance of ISA emulators, particularly for Qemu. The research works focusing on optimization of cross-platform DBT, particularly ARM to x86, are listed in the below subsections.

2.3.3.3 PQEMU

PQEMU (Ding et al., 2011) emulates one instance of QEMU but parallelizes its internal DBT module. PQEMU achieves minimal overhead in locking and unlocking shared data through the management of the translated code cache sections. Locks are introduced to serialize and synchronize access to shared data. PQEMU explores two cache designs, namely, unified code cache and separate code cache. Guest code that based on parallel execution of threads requires unified code cache, while programs that execute independent code are better suited to separate code cache design. PQEMU assigns each guest thread of a multi-threaded program to a separate emulator and DBT thread. Therefore, PQEMU achieves better performance in emulating multi-threaded programs. Performance optimizations of 3.8X are achieved for parallel execution programs. However, the emulation of the single-thread program remains same as PQEMU does not try to enhance the target guest code in each thread. Moreover, the approach does not consider the host multi-processor architecture for multi-threaded emulation. COREMU (Wang et al., 2011) was proposed based on the observation that current emulation techniques do not exploit multi-processing capabilities of the host hardware.

2.3.3.4 HQEMU

HQEMU (Hong et al., 2012) applies compiler-level optimizations to multi-threaded emulation of QEMU to enhance the performance of both single-thread and multi-thread programs. An enhanced LLVM compiler is utilized for low level code optimization with dynamic binary optimizer (DBO) as a backend for Qemu front-end. The front-end and back-end execute in different threads in HQEMU. Two code caches are maintained for code translated and optimized at different levels, namely, block code cache and a trace cache. The TCG acts as a fast translator that translates guest binary at the granularity of a basic block, and stores translated codes to the block code cache. TCG also keeps the translated guest binary in its IR for further optimizations in the LLVM backend. An emulation module is added to the Qemu design that handles translation, optimization, and execution of the guest program. When the emulation module finds code that has repeated execution or has a pattern for further optimization, it sends it to the LLVM backend along with its IR. The LLVM backend translates the TCG IR to LLVM IR. A set of LLVM optimizations are applied to the LLVM IR for high quality code generation that is stored in the trace cache. The LLVM optimization can lead to significant overhead to the emulation process. However, as LLVM optimizations are executed over a separate thread, the overhead is avoided. Moreover, multiple optimization requests can be serviced by LLVM optimizer running on different worker threads. HQEMU achieves 2.4X performance enhancement for various SPEC benchmarks in ARM to x86 emulation.

2.3.3.5 Trace-driven approach for gem5

Researchers (Butko et al., 2015) proposed a trace-driven approach for fast simulation of multi-core architectures using gem5. The work addresses the slow simulation of gem5 that prohibits its widespread usage in ARM emulation. Abstraction of core execution with traces is applied as a method for fast simulation. The trace-driven approach consists of three steps; trace collection, processing, and simulation. The trace collection phase defines the hardware and software components of the simulated system. Synchronization and core replication techniques are applied to the traces in the trace processing phase. The resultant traces are added to a full system emulator. A trace of memory transactions for one core is captured in full system simulation and augmented with synchronization semantics. The scalar synchronized traces are replicated to vector traces for multiple-core simulation. The vector traces are applied to the gem5 simulator to traffic injectors. In this way, the gem5 simulator only needs to simulate the memory and interconnect systems, hence, lowering the burden on full system simulation. Experimental analysis was performed on gem5 ARM. Results showed a speed of up to 800 times faster than the gem5 full system simulation for some benchmarks. The simulation error of 6% is also reported in this approach.

2.3.3.6 Client/server DBT

Hsu et al. (Hsu et al., 2015) proposed a distributed client/server based DBT solution for ARM emulation optimization. The basic technique adopted by the solution is to divide the DBT into two components. The client performs light-weight DBT and sends requests to the server which performs the full DBT with optimizations. The thin client comprises of all the three components of the emulation engine. However, the translator is light-weight; i.e., it does not perform aggressive code optimizations. In this manner, the thin client does not entirely rely on the server and can execute stand-alone. The optimization manager on the thin client searches for code that requires optimizations and sends it to the optimizer server. The commonly identified code for optimizations is in the form of loops or recursive functions. The server optimizes the DBT and saves it in an optimized code cache. The optimized code is also sent to the optimization manager of the thin client. The optimization manager of the thin client decides between the optimized or non-optimized code executions on the runtime. The TCG of Qemu is utilized as the thin translator and the LLVM is utilized as code optimizer. The results show that the client/server model achieves 17-37% performance improvement over the baseline non-client/server model.

2.3.3.7 Retargetable Static Binary Translation

Static Binary Translation (SBT) allows the execution of aggressive translation optimizations that are not possible in DBT. However, SBT leads to issues such as handling of self modifying code, code discovery, and code location. Researchers (Shen, Hsu, & Yang, 2014) utilized LLVM as a tool for retargetable SBT of ARM binaries. As LLVM is based on IR, the translated ARM binary code can be retargeted to any ISA. Moreover, the LLVM provides for aggressive code optimizations that result in performance improvements. Furthermore, the solution also avoids the code discovery problem that can lead to interpretation. The results show that the LLVM based SBT solution achieves 6X perfor-

Table 2.2: Comparison of ARM Emulation Techniques

ARM emulation technique	**Emulation process**	**Emulation granularity**	**Emulation objective**	**Optimization technique**	**Optimization function**
Qemu (Bellard, 2005)	DBT	Functionally accurate	Malware and architecture analysis	NA	NA
Gem5 (Binkert et al., 2011)	DBT	Cycle accurate	Architecture analysis	NA	NA
PQEMU (Ding et al., 2011)	DBT	Functionally accurate	Malware and architecture analysis	Parallelization	Support multi-threaded application
HQEMU (Hong et al., 2012)	DBT	Functionally accurate	Malware and architecture analysis	Parallelization	Parallel DBT
Trace driven gem5 (Butko et al., 2015)	DBT	Cycle accurate	Architecture analysis	Statistical sampling	NA
Client/server DBT (Shen et al., 2014)	DBT	Functionally accurate	Malware and architecture analysis	Client server	NA
Retargetable SBT (Hsu et al., 2015)	SBT	Functionally accurate	Malware and architecture analysis	NA	NA

mance gain compared to baseline Qemu DBT solution.

2.3.4 Comparison of ARM Emulation Techniques

Table 2.2 lists the comparison parameters and details for ARM emulation techniques.

Most of ARM emulation techniques discussed in above subsections are based on DBT (Bellard, 2005; Ding et al., 2011). The reason behind dominant utilization of DBT over SBT is that DBT techniques are flexible and can manage self-modifying code easily. ARM emulation techniques based on Qemu are functionally accurate. On the contrary, ARM emulation techniques based on gem5 are cycle accurate (Butko et al., 2015). Similarly, the objective of ARM emulation techniques that are based on Qemu is both malware and architectural analysis. gem5 based emulation techniques are not used for malware analysis as the cycle accuracy leads to overhead in the rigorous testing of applications (Shen et al., 2014). Emulation optimization techniques are applied to the basic ARM emulation tools (Qemu and gem5) in order to increase their performance. Parallelization is commonly used to optimize the performance of ARM emulation (Ding et al., 2011; Wang et al., 2011). PQEMU (Ding et al., 2011), COREMU (Wang et al., 2011), and HQEMU (Hong et al., 2012) exploit multi-threaded applications, parallelization of emulation, and parallelization of DBT as an objective function respectively.

Qemu and gem5 are the two candidates for ARM ISA emulation. There are several performance advantages in the selection of Qemu as an ARM emulator. Firstly, Qemu was developed with the design objective of high performance (Quick Emulator). gem5, on the contrary, was developed with the design goal of first order performance accuracy in hardware and software system simulation. Secondly, gem5 is a cycle accurate simulator as compared to functional accurate Qemu. The cycle accuracy of gem5 leads to more simulation overhead as compared to Qemu. A functionally accurate simulator, such as Qemu, focuses on what a processor does and not how it does it. However, a cycle-accurate simulator has to emulate the ISA according to actual hardware and software semantics with accurate timing information. Both Qemu and gem5 are open-source emulators permitting further modification and optimization.

Despite the fact that several optimization techniques have been applied to the DBT of ARM emulation, optimal translation of special instructions, such as FP and SIMD has not been the focus of the aforementioned studies. FP and SIMD instructions are treated as generic scalar instructions in the emulation process resulting in performance loss. Therefore, a cross-platform ARM emulation and translation framework is required that focuses on efficient vector translation of SIMD and FP instructions.

2.4 SIMD Instruction Porting Techniques

In this section, we provide a list of applications of SIMD instructions in mobile and multimedia software. We further provide a taxonomy of SIMD instruction translation and porting techniques followed by a comprehensive review. At last, we compare the SIMD instruction translation techniques based on the identified qualitative parameters from the taxonomy.

2.4.1 Application of SIMD Instructions

The ability to execute multiple instructions in parallel allows a processor to enhance user experience and program quality. Single Instruction, Multiple Data (SIMD) is a type of data level parallelism in which a single instruction leads to computation of multiple data points with multiple outputs. SIMD instructions are also known as vector instructions while instructions other than SIMD are called scalar instructions. The hardware corresponding to the SIMD instructions are known as hardware accelerators. Vector processing uses a single instruction to perform the same operation in parallel on multiple data elements. The constraint is that the data elements should be of the same type and size. In this manner, a 32-bit hardware that normally adds two 32-bit values can perform two parallel 16-bit or four parallel 8-bit operations in the same amount of time. Up to 25% of the code in multimedia based applications can be SIMD instructions (G. Lee et al., 2015). SIMD instructions are able to deliver very compact code for compute-intensive applications. Therefore, the presence of SIMD instruction can significantly increase the performance of applications on mobile devices while achieving efficiency in instruction execution and energy.

A number of algorithms and applications are a candidate for SIMD instructions. Particularly, multimedia based algorithms such as Fast Fourier Transform (FFT), image conversions, and dot multiplications. In such applications, a common operation is to add or subtract the same value from multiple data points, e.g., changing the brightness of a picture. The operation requires that for each pixel, same amount of value is added or subtracted from its red (R), green (G), and blue (B) elements. Two parameters define an SIMD instruction; the size of the registers that store the input and output values and the size of one element in the SIMD. For example, an SIMD instruction can operate on 64-bit registers with configurations of eight 8-bit elements, four 16-bit elements, and two 32-bit

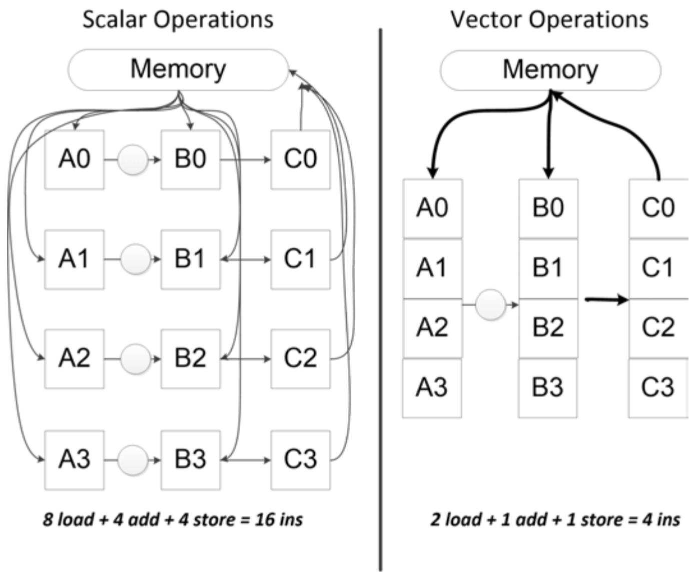


Figure 2.7: SIMD vs Scalar Instructions

elements (FELLOWS, 2014; Jiang et al., 2005; Maleki, Gao, Garzaran, Wong, & Padua, 2011). Chip makers provide support of SIMD instructions at the hardware level. Examples of SIMD ISA are MMX and SSE for Intel and NEON for ARM (Limited, 2009). An ISA that supports SIMD instructions has separate registers and instruction pipeline for this purpose. Such ISA configuration is known as a co-processor. Modern Graphics Processing Units (GPUs) and array processors are often hardware implementations for SIMD (Mitra et al., 2013). The difference between SIMD and scalar instructions is depicted in Figure 2.7.

There are multiple techniques to generate SIMD instructions for the instruction pipeline. These are listed as follows.

- SIMD instructions can be hand-written in the assembly code. Although the performance of such hand-written code can be high, it is less readable, requires high programming skills, and can lead to conflicts in the instruction pipeline due to bypassing of the compiler technology.

- Second method of generating SIMD instructions in assembled code is to use compiler optimizations (e.g., O3, O2) and auto-vectorization options. Auto-vectorization options ask the compiler to look for code that can be translated into SIMD instructions. Therefore, this method solely depends on the capability of the compiler to generate SIMD instructions.

- Intrinsics are built-in functions that are specially handled by the compiler for efficient execution of some instructions. Intrinsic functions can be used in C code to generate corresponding SIMD instructions. The intrinsic functions work as an API to SIMD assembly for the programmers. Intrinsic functions enable the programmer to explicitly request the compiler the use of such an instruction. Therefore, many C compilers provide platform-specific intrinsic functions. For ARM, *arm_neon.h* header file defines the SIMD intrinsics.

SIMD instructions and intrinsics vary from one platform to another. Therefore, application programmers often do not utilize SIMD intrinsic functions in practice as it limits the applicability of the code to the SIMD target platform. However, it is difficult to write efficient SIMD based code separately for each platform. Therefore, it is desired that an SIMD porting or translation technique be devised such that the application programmers can write SIMD based efficient applications that can be targeted for multiple platforms. The task of devising an SIMD porting techniques is made difficult by many factors. For example, for x86 and ARM ISAs, the translation and porting of SIMD instructions is not a trivial work due to the difference in instruction length and register sizes.

2.4.2 Taxonomy of SIMD Porting Techniques

The taxonomy of SIMD translation and porting techniques is presented in Figure 2.8.

Translation Technique: The techniques utilized for cross-platform translation of SIMD instructions can be either JIT or DBT. The JIT translation maintains a generic platform-independent intermediate translation of SIMD instructions that can be further translated to the target ISA on runtime. DBT also utilizes intermediate translations. How-

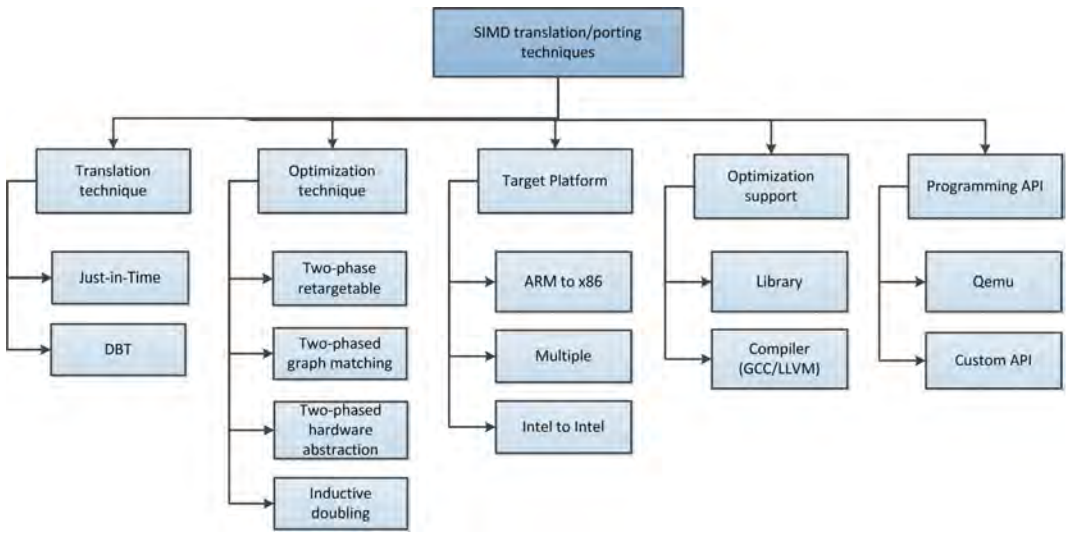


Figure 2.8: Taxonomy of SIMD Porting Techniques

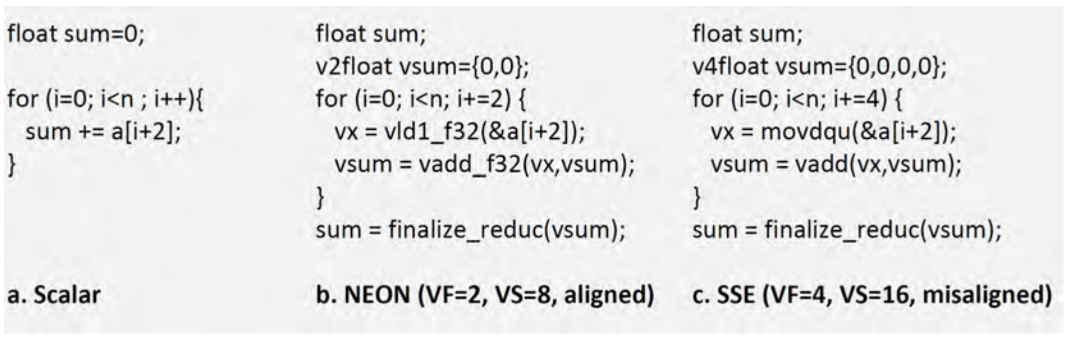
ever, the DBT IR is based on emulation tool while the JIT IR is based on a compiler technology.

Optimization Technique: The intermediate representations of both DBT and JIT lead to two-phase retargetable translation of SIMD instructions. Most of SIMD porting techniques perform two-phase retargetable translations. To-phase translations can be supported by the IR, graph matching, or hardware abstraction methods. Moreover, inductive doubling can be applied to translate 32 bit instructions with the help of 16 bit operations.

Target Platforms: The cross-platform SIMD porting techniques can target two or more than two platforms. However, the focus of our work is cross-platform emulation of ARM ISA on Intel ISA.

Optimization Support: The optimizations can be supported through compiler or a custom library. The compiler supported studies rely on the compilers (GCC or LLVM) to produce instructions that are optimized for cross-platform execution. Both GCC and LLVM provide support for x86 and ARM architectures along with support for generation of portable IR. On the contrary, the library based SIMD translation frameworks utilize custom libraries that support cross-platform programming of SIMD instructions.

Programming API: SIMD porting techniques can utilize the existing DBT tools,



```
float sum=0;

for (i=0; i<n ; i++){
  sum += a[i+2];
}
```

a. Scalar

```
float sum;
v2float vsum={0,0};
for (i=0; i<n; i+=2) {
  vx = vld1_f32(&a[i+2]);
  vsum = vadd_f32(vx,vsum);
}
sum = finalize_reduc(vsum);
```

b. NEON (VF=2, VS=8, aligned)

```
float sum;
v4float vsum={0,0,0,0};
for (i=0; i<n; i+=4) {
  vx = movdqu(&a[i+2]);
  vsum = vadd(vx,vsum);
}
sum = finalize_reduc(vsum);
```

c. SSE (VF=4, VS=16, misaligned)

Figure 2.9: SIMD Code: Scalar, ARM NEON and Intel SSE Instructions

such as Qemu or can work standalone as cross-platform programming interfaces.

2.4.3 Review of SIMD Porting Techniques

Porting of SIMD instructions across ISAs is important due to many factors. Firstly, mobile device market consists of heterogeneous architectures, such as ARM and Intel. Programmers who want their vectorized applications to support both architectures require porting of SIMD instructions. Secondly, the MCC paradigm is also based on heterogeneous processor architectures. Therefore, code offloading between heterogeneous processors requires porting or translation of SIMD instructions. Figure 2.9 provides an example of sum instruction in scalar, NEON SSE, and x86 SSE format to emphasize the difference between the semantics of SIMD intrinsics.

Optimization of SIMD instruction translations has received the attention of the research community very recently. Some of the research works are based on Qemu as the current Qemu upstream implementation has many shortcomings with respect to translation of SIMD instructions. Qemu translates SIMD instructions to multiple scalar instructions that consume more instruction cycles while ignoring the support for SIMD instructions in the host ISA. Therefore, the existing approach of DBT in Qemu leaves significant room for performance enhancement of SIMD instructions. In the below subsections, we list the studies that have ported SIMD instruction between heterogeneous ISA with particular focus on ARM NEON to Intel SSE translations

2.4.3.1 FREERIDER

FREERIDER (Manilov et al., 2015) methodology introduced a framework that enables retargeting of non-portable SIMD intrinsics between heterogeneous ISA. FREERIDER utilizes graph based matching of intrinsic functions while searching similar instructions on the target ISA. The study also utilizes descriptive language to specify the semantics of intrinsic functions. High-level code transformations are done to obtain optimized translations for target ISA. FREERIDER achieves retargeting in three steps. First, the intrinsics are defined by a custom descriptive language. Then the descriptive language is represented in graphs. The graphs are then translated to C programs of the target ISA using graph matching techniques. C header files are generated utilizing inputs from guest and target intrinsics. The header file is utilized as input to produce data flow graphs for each intrinsic function. The graphs act as IR and are annotated with input types. The graph matching stage requires the header files, data flow graphs, and application source code to perform the matching. As an output of graph matching, C code of the target ISA is generated. In the final stage, the code is translated to the target ISA intrinsics with further target-specific optimizations, such as loop unrolling.

2.4.3.2 IDISA+

IDISA+ (Huang, 2011) proposes a framework for cross-platform compatible SIMD programming. The framework supports a limited set of well-defined integer SIMD instructions that are commonly utilized in multimedia applications. The framework comprises of two components. The code generator produces portable libraries for SIMD code generation while the test suite examines the performance of the portable libraries along with their correctness analysis. To achieve optimal performance, the code generator selects the least instruction count as the best alternative among library routines. The libraries provide a high level programming interface for writing portable SIMD code. The libraries automatically select the compilation flags suitable for the target architecture. The framework is based on inductive doubling principle to generate in-register SIMD instructions. Due to the portability of IDSA+, performance loss is expected. However, the results show that the code generated by the model is slightly efficient than its hand-written counterpart.

2.4.3.3 Improving SIMD Instruction Generation in DBT

Sheng et al. (Fu, Wu, & Hsu, 2015) introduced new IR for Qemu to enhance the performance of Qemu for x86 and ARM emulation back-ends. The work added vector IR to existing TCG implementations of Qemu. The study proposed two approaches to optimize the existing DBT of SIMD instructions in Qemu. In the first approach, the NEON helper functions were modified to generate LLVM vector instructions through C intrinsic functions. However, the experiments showed that the helper function call overhead was significant. In the second approach, vector IR support is added to the TCG so that SIMD instructions are translated to corresponding vector instructions. In Qemu implementation, the IR buffers comprise of instruction op-code and parameters that the register numbers. This work modified the IR buffer to contain opcode, CPU state, and SIMD registers. The proposed framework migrated from scalar IR to hybrid IR that also contained vector IR. The model of vector IR was inspired by the LLVM IR. Results showed that the vector IR implementation produced better performance for x86 emulation on all benchmarks. However, the ARM emulation produced better performance for only three benchmarks due to lack of compiler optimization in producing guest SIMD instructions.

The same group of researchers enhanced their earlier work to efficient SIMD translations in HQEMU (Fu, Wu, Liu, et al., 2015). HQEMU enhances Qemu with LLVM optimizations that are applied to the code generated by the TCG and classified by the profiler as a candidate for optimization. The authors added two methods to enhance the SIMD translations in HQEMU. One method is to enhance the helper functions to emit vector IR, while the second approach is to utilize vector optimizations in TCG. In the first method, the pre-compiled helper functions that contained scalar LLVM IR were modified to produce vector LLVM IR which are already supported by LLVM compiler. As a result, only the LLVM optimizer in the HQEMU is modified. To reap the benefits of LLVM compiler in the second approach, TCG IR is converted into LLVM IR with two-level approach. Experiments utilized SEPC2006 and Linpack benchmarks. The ARM to x86 emulation shows 2.5X performance gains for the second approach while compared to HQEMU. However, the helper function approach only yield 1.05X performance gains while compared to the baseline.

2.4.3.4 Liquid SIMD

Liquid SIMD (Clark, Hormati, Yehia, Mahlke, & Flautner, 2007) addresses the issue of binary compatibility across heterogeneous ISAs while porting SIMD instructions. The work aims to decouple the ISA from the hardware accelerator through abstraction. The decoupling is done by delayed binding while representing SIMD with scalar instructions and utilizing light-weight dynamic translation to map the scalar representation on hardware accelerators. The delayed binding in the Liquid SIMD framework is supported by the compiler and a translation system. The compiler translates the SIMD instructions to abstract scalar instructions supported by the baseline ISA. The compiler converts the SIMD instruction to scalar instructions based on a set of syntax rules either at compile time or after compilation with a cross-ISA compiler. The compiler also provides data flow graphs of the application for the translation process. The translator identifies these data flow graphs and transforms them into the target ISA specific SIMD instructions. Liquid SIMD utilizes an abstract hardware translator for translation of SIMD instructions.

2.4.3.5 Optimizing DBT of SIMD

Researchers (Li, Zhang, Xu, & Huang, 2006) proposed a framework to port a program written for one architecture to another with optimized DBT of SIMD instructions. As SIMD registers can have different data types stored at different times, this work proposes an algorithm to track the SIMD data type. The SIMD data type tracking algorithm solves the issue of register synchronization while scanning the input code and emitting translate code. Moreover, the work also proposes three algorithms to optimize the DBT of SIMD instructions. These algorithms are; SIMD data type re-assignment algorithm, the translate-time inter-block mismatch removal algorithm, and the runtime inter-block data type mismatch removal algorithm. The type re-assignment algorithm utilizes the data type flow to re-assign data type of multi-data type instructions. The translate time algorithm utilizes flow graphs with well-defined data type flow equations to detect inter-block mismatch. SIMD code blocks are explicitly made aware of neighbor data types for this purpose. The runtime inter-block mismatch algorithm tries to correct block mismatches if the program execution flow changes. The work is based on x86 to x86 DBT of SIMD instructions. A performance optimization of 3.89% is achieved by the system for SPEC2000 integer benchmarks.

2.4.3.6 VaporSIMD

Vapor SIMD (Nuzman et al., 2011) proposes an auto-vectorizing compilation scheme that vectorizes scalar instructions such that they can be executed on multiple heterogeneous ISAs. The work auto-vectorizes multiple scalar instructions to a single SIMD instruction while meeting the constraints, such as memory alignment, the cost of loop vectorizing, and dependencies between the data elements. The proposed system utilizes split compilation based auto-vectorizing and JIT compilers which lead to two-step translation from source code to machine code. In the first compilation step, the auto-vectorizing compiler works offline while translating C source code to vectorized bytecode that is portable. The GCC offline compiler emits Common Language Infrastructure (CLI) complaint bytecode that is used by the JIT compiler. In the second compilation step, the JIT compiler (Mono) compiles the portable CLI complaint bytecode to the target machine code that can have variable SIMD semantics in online mode. The Mono is a CLI compliant VM that can target multiple ISAs with a JIT compiler. A split layer is defined for abstraction between the static and dynamic compilation steps. The split layer is equivalent to the IR abstraction in Qemu. The work is applicable to AltiVec, SSE, AVX, and NEON SIMD instructions. Due to the vector JIT compilation, the bytecode size and compilation time increases. The auto-vectorizing results in 1.5X and 1.2X speedups for SSE and AltiVec targets respectively. However, the speedup is low for ARM NEON due to the low support of auto-vectorizing in the GCC compiler.

2.4.3.7 Speeding up SIMD DBT

Luc et al. (Michel et al., 2011) proposed a 3-address IR to map guest SIMD instruction to host SIMD instructions. The 3-address IR is close to the intersection of guest and host ISAs. The 3-address IR is formed on two constraints; the number of new IR instructions should be limited so that the TCG is not burdened and enough IR instructions should be added to allow a maximum coverage of the SIMD instruction sets in the guest and target ISAs. Three cases arise from such instruction mappings: **(a)** a direct one-to-one mapping between guest and host SIMD instruction. In such case, new vector IR is added to DBT that allows passing of vector arguments to the TCG. An example of such instructions is 8 bit addition operation represented by $vadd.i8\ Qd,\ Qn,\ Qm$ in NEON and $paddb\ xmm1, xmm2$ in SSE ISA. A new IR $simd\_128\_add\_i16$ is added to the TCG in this case, **(b)** a one-to-many mapping in which the result of guest SIMD instruction is achieved by execution of multiple scalar IR instructions and no new IR is added to the DBT. An example of such a case is ARM Neon *vsra.u*32 instruction which performs a right shift on operands and accumulates the shifted results in the output register. *vsra.u*32 is translated into two IR micro-ops *simd*_128_*shr*_*i*32 and *simd*_128_*add*_*i*32. The TCG then find an equivalent for each micro-op ,i.e. *psrld* and *paddd* in the SSE ISA and **(c)** there is mapping between guest instruction and IR but there is no equivalent host instruction. An example of such instruction is 8-bit logical left shift in ARM NEON whereas Intel SSE does not support 8-bit shift operations. A speedup of 20% is achieved on average in the direct mapping case for different instructions.

2.4.3.8 MC2LLVM

Researchers (Y.-C. Guo, Yang, Chen, & Lee, 2016) proposed a framework for the translation of ARM NEON and Vector Floating Point (VFP) instructions in a DBT if the host architecture supports such instructions in hardware. Their approach is similar to that discussed in earlier subsections (Fu, Wu, Liu, et al., 2015). A machine-code-to-low-level-virtual-machine (MC2LLVM) approach is applied for the efficient translation of SIMD instructions. The guest vector instructions are translated into LLVM IR which keeps intact the NEON and VFP instructions, unlike Qemu which translates them it into scalar instructions. MCDisassembler from the LLVM is utilized to translate the guest binary to MCInst. Each MCInst is further translated to LLVM IR. Target independent optimizations are applied on LLVM IR by the LLVM optimizer. LLVM IR is then translated to the target binary with the LLVM backend. An algorithm based on FP operation input and output is utilized to detect exceptions that can cause errors in the emulation process. Experiments show that the MC2LLVM approach is 3X faster than QEMU in processing NEON and VFP instructions for various benchmarks.

Table 2.3: Comparison of SIMD Porting Techniques

SIMD porting technique	**Translation technique**	**Optimization technique**	**Target platform**	**Optimization support**	**Programming API support**
FREERIDER (Manilov et al., 2015)	JIT	Two phase graph matching	ARM to Intel	Library	Custom API
IDISA+ (Huang, 2011)	NA	Inductive doubling / NA	Multiple	Library	Custom API
Optimizing SIMD in HQEMU (Fu, Wu, & Hsu, 2015)	DBT	Two phase retargetable	ARM to Intel	Compiler	Qemu
Liquid SIMD (Clark et al., 2007)	JIT	Two phase hardware abstraction	ARM to Intel	Compiler	Custom API
Optimizing DBT of SIMD (Li et al., 2006)	DBT	Two-phase retargetable	Intel to Intel	NA	Custom API
Vapor SIMD (Nuzman et al., 2011)	JIT	Two phase retargetable	Multiple	Compiler	Custom API
Speeding up SIMD DBT (Michel et al., 2011)	DBT	Two phase retargetable	ARM to Intel	NA	Qemu
MC2LLVM (Y.-C. Guo et al., 2016)	DBT	Two phase retargetable	ARM to Intel	Compiler	Qemu

2.4.4 Comparison of SIMD Porting Techniques

We discussed seven SIMD instruction porting techniques in the above subsections. Table 2.3 compares the SIMD porting techniques on quantitative parameters identified from the taxonomy.

The frameworks discussed in the above sections have the objective of SIMD instruction translation optimization and cross-platform porting. The aforementioned frameworks utilize two techniques for SIMD instruction translation; JIT translation based on the target platform (Nuzman et al., 2011) and DBT which is retargetable for any target platform (Fu, Wu, & Hsu, 2015). The basic purpose of both techniques is to translate the code into a generic platform-independent IR which can be further translated into target specific code. However, JIT is preferred over DBT for two reasons. First, there is a lack of transparency in DBT as user or OS intervention is needed to translate the binary. Secondly, if the emulation leads to an error, accountability can not ensure the fault is with the application or the DBT.

Most of the SIMD porting techniques are two-phase retargetable (Nuzman et al., 2011). Retargetability is achieved through platform-independent IR (Y.-C. Guo et al., 2016). However, graph matching (Manilov et al., 2015) and hardware abstraction (Clark et al., 2007) techniques are also used for this purpose. Retargetable IR based SIMD porting techniques have the advantage that they can support multiple host and target platforms. On the contrary, graph based matching and hardware abstraction require addition of instruction profile to the current translation system in order to target multiple platforms.

The focus of most of the surveyed SIMD porting techniques was ARM to Intel translations. Due to recent emergence of ARM NEON ISA (Limited, 2009), all these studies are recent and relevant to the subject discussed. However, some SIMD porting frameworks also target platforms other than ARM and Intel (Nuzman et al., 2011). To perform the optimization on translations, the SIMD porting techniques are either supported by the compilers or a custom library. Compiler based support for SIMD optimizations are mostly done along with the two-phase retargetable translations. Compilers, such as GCC and LLVM, provide support for compiler based optimizations. As ARM to Intel DBT is already implemented in Qemu, most of the aforementioned frameworks are based on Qemu. However, there are serious performance drawbacks in the SIMD translations in Qemu which are addressed by researchers (Y.-C. Guo et al., 2016; Fu, Wu, Liu, et al., 2015). Moreover, some SIMD porting frameworks are based on LLVM optimization based HQEMU (Fu, Wu, & Hsu, 2015). Custom API's and programming models have also been developed for cross-platform SIMD instruction translations (Clark et al., 2007). However, Qemu based frameworks have more scalable nature than custom API's. Qemu based frameworks are feasible for complete application binaries while custom API's only translate SIMD instructions.

2.5 Open Research Issues

Generic MCC offloading frameworks are based on virtualization techniques that often overlook the benefits of native application migration. Efficient translation and porting of

SIMD instructions among heterogeneous ISAs has also eluded the considerations of the research community. There are several challenges to the native code migration of multimedia application that are rich in SIMD instructions and require DBT over heterogeneous platforms. In the following subsections, we list these challenges and open research issues.

2.5.1 MCC Code Offloading Challenges for Native Applications

The challenges to code offloading in native applications are listed as follows (Xu & Mao, 2013; Zhu, Luo, Wang, & Li, 2011).

- The biggest challenge to MCC code offloading is the handling heterogeneous hardware architectures. Though system virtualization and application virtualization address this issue through abstractions and intermediate translations, they have significant computational and communicational overhead as compared to natively executed applications.
- The cross-ISA migration of live configurations for interactive and latency sensitive applications is a complex task due to the volatile memory and CPU contexts of the smartphone applications. The context of the offloaded application on the mobile client can change leading to the nullification of offloaded operations on the server.
- Dynamic partitioning of applications at runtime and classification of methods that require resource-rich cloud services without static analysis is difficult. However, the challenge of dynamic partitioning of applications can be developed into a benefit in case of multimedia based applications. The image processing kernels in multimedia applications can be statically annotated with offload semantics. These kernels can also have clones at the cloud servers that only require input for operation rather than the entire instance of application offload.
- The MCC offloading frameworks fail if the network disconnects or falls below a

threshold network bandwidth. The issue of network disconnectivity is aggravated if the offload in enabled by system or application virtualization as the data transfer requirements increase. To address this issue, network bandwidth adoptable MCC code offloading solutions need to be developed.

- Most of MCC frameworks address the challenge of mobile client energy while ignoring the energy metrics on the cloud server. Moreover, the multimedia based kernels are resource-intensive and consume high energy. Therefore, integrated energy efficient frameworks need to be devised for MCC code offloading that consider both mobile and server energy metrics for green computation.

2.5.2 ARM DBT Challenges

The major challenge of ARM DBT solutions is to reduce the overhead of runtime translation and optimizations. Further challenges to ARM DBT translation and optimizations are listed as follows (Moore, Baiocchi, Childers, Davidson, & Hiser, 2009; Shen et al., 2014; Nimmakayala, 2015).

- DBT or emulation process executes a large number of additional instructions as compared to the native application execution. The increase in the number of executed instructions is a result of DBT management tasks, such as code translation, indirect branch resolution, and trace formation while frequently exiting the code cache at run-time. The optimization of DBT management task is a challenge for ARM emulation systems.

- DBT optimizations are beneficial for programs with longer execution times. Therefore, DBT optimizations are not suitable for ARM ISA as it is utilized in embedded and mobile devices where applications are client programs that have relatively short execution time.

- The ARM ISA has an exposed Program Counter (PC, r15) that is can be modified by high-level APIs. The DBT has to handle the frequent reads and writes of the exposed PC that leads to significant run-time overhead.
- Mobile and embedded applications are mostly interactive and have high real-time, boot-up, and response time constraints. DBT of such applications leads to higher latency and low user interactive experience.

2.5.3 SIMD Porting Challenges

The challenges to the translation and porting of SIMD instructions are listed as follows (Fu, Wu, & Hsu, 2015; Mitra et al., 2013; Li et al., 2006; Michel et al., 2011).

- SIMD instructions vary syntactically from one ISA to another. The diversity of hardware ISAs, instruction widths, and register sizes leads to the substantial difference between SIMD semantics across platforms. Therefore, an application written and compiled for one platform is highly unlikely to be portable on another platform.
- SIMD registers can hold data of different types at different times. The ability of SIMD registers to support multiple data types within the same register makes the task of DBT difficult. The DBT process may implement a type-tracking algorithm at translation time for accurate execution.
- ARM NEON functions are mostly 64-bit while Intel SSE operations are 128-bit. The porting of 64-bit functions to 128-bit registers impacts the code quality. Moreover, if an ARM SIMD function is translated to multiple Intel SSE instructions, intermediate flow can occur that requires manual handling during translation.
- A subset of Intel SSE intrinsic functions require intermediate parameters instead of constant inputs. Hence, when they are called from a wrapper function for DBT, errors are encountered.

- Even when there is a one-to-one correspondence between ARM and Intel intrinsics, the behavior may differ on inputs that are out of range. Similarly, the rounding rules of corresponding SIMD intrinsics can vary.

- ARM NEON ISA and x86 SSE ISA differ in many ways. ARM ISA is a register-register architecture while Intel x86 is register-memory architecture. Intel SSE supports double precision FP operations while ARM NEON supports only single precision FP operations. Moreover, ARM NEON has instructions that have no equivalent in Intel SSE. For example, there are no 8-bit shifts in x86 SIMD. In such case, the 8-bit data has to be packed in 16 bit and then back to 8 bit after the operation.

2.6 Conclusion

In this chapter, the research domain of native code offloading MCC frameworks was surveyed in detail. MCC offload enabling techniques and frameworks were presented along with the debate on their strengths and weaknesses. It was identified that the native code migration technique has the least computational and communicational overhead among MCC offload enabling techniques. Further, DBT techniques for ARM ISA that are essential for native code migration between heterogeneous architectures were surveyed. The ARM emulation or DBT techniques mainly focus on parallelization of execution. Hence, significant room is left for optimal translation of special instruction, such as the vector and SIMD instructions. The case of efficient cross-platform translation of SIMD instructions was surveyed for the SIMD porting techniques. Moreover, in-depth comparison and taxonomy of the state-of-the-art MCC offloading frameworks, DBT optimization techniques, and SIMD porting techniques were provided.

The main objective of the detailed literature review was to identify the potential research issues in the domain of native code offloading frameworks in MCC. Through the literature survey, multiple research problems in the domain of multimedia based native

MCC offloading frameworks were identified. One of the foremost challenges in MCC offloading frameworks is the efficient offloading of native code without dependence on system and application virtualization techniques. Native code migration brings new challenges in the form of cross-platform translation for heterogeneous mobile and cloud architectures. The challenge of translation of native code is brought to the spotlight in the case of SIMD instructions which are non-optimally translated from ARM ISA to x86 ISA in existing MCC offloading frameworks. The optimal translation of SIMD instructions in DBT systems requires re-design due to current vector-to-scalar mappings that result in higher instruction cycles and execution time. Moreover, as vectorized multimedia applications depend on SIMD intrinsics for high performance, custom libraries, and APIs need to be developed for cross-platform translation of such instructions.

CHAPTER 3: PERFORMANCE ANALYSIS OF MCC OFFLOADING TECHNIQUES

This chapter presents a case of pre-compiled native code migration based MCC offloading framework for multimedia applications by evaluating the existing MCC offloading frameworks. Most of the MCC offloading frameworks are enabled by system and application virtualization techniques. We present the case of pre-compiled code offloading by experimentally demonstrating the overhead of system virtualization, application virtualization, and compiled process code migration. Previous research works have focused on performance analysis of a single MCC offload enabling technique, such as system or application virtualization. Therefore, a collective performance analysis of the existing MCC offload enabling techniques is required. The performance analysis will identify the level of severity of the computational overhead faced by MCC offloading frameworks.

The rest of this chapter is organized as follows. Related work to the existing performance evaluations of system and application virtualization is presented in Section 3.1. In Section 3.2, the experiments undertaken for the performance analysis of MCC offload enabling techniques are described. To put forward the case of pre-compiled code offloading, performance analysis of existing MCC code offload enabling techniques is performed to highlight the computational overheads in Section 3.3. In Section 3.4, an analysis of performance enhancement in the case of SIMD instructions and the corresponding DBT overhead for process code migration across heterogeneous architectures is provided. We detail and discuss the insights gained from the performance evaluation of MCC offload enabling techniques in Section 3.5. We provide the concluding remarks of the performance evaluation in Section 3.6.

3.1 Background

In this section, we provide a review of existing performance evaluation studies of system and application virtualization. A study on Xen, KVM, and VirtualBox virtualization solutions finds the performance overhead for most High Performance Computing (HPC) applications to be around 8% (Younge et al., 2011). The study utilized HPC and SPEC benchmarks to evaluate various aspects of the system from network bandwidth to CPU performance. The overall rating of various virtualization solutions is almost similar to each other. Fernando et al. (Camargos, Girard, & Ligneris, 2008) evaluated Kqemu, KVM, Xen, OpenVZ, Linux-VServer, and VirtualBox to determine most efficient Linux server virtualization solution and the scalability of these solutions to host multiple OSs. Benchmarks that targeted different parts of the system were used. Benchmark results showed that Linux-VServer and KVM have the least overhead, which is at least 5% in most of the cases as compared to the native performance.

Dalvik and .Net runtimes are utilized in smartphones for application virtualization. As most of the smartphone devices are Android based, we will focus on Dalvik VM for application virtualization (*Smartphone OS Market Share, 2015 Q2*, 2016). Researchers have utilized Dalvik VM as a tool for code migration from mobile devices to cloud servers for computational offloading (Chun et al., 2011). Researchers (Y.-J. Kim et al., 2012) evaluated interpretation and compilation performance of Dalvik VM and a conventional Java based VM. The study found that Dalvik's register-based bytecode approach leads to a slightly better interpretation while trace-based compilation leads to significant overhead as compared to HotSpot VM. The article also studies Dalvik and Native C performance utilizing Caffeinemark benchmark and finds the Dalvik overhead to be three to six times higher as compared to Native C. Lee et al. (S. Lee & Jeon, 2010) determined overhead of Dalvik for integer, floating point, and memory access operations. The study utilized

Android emulator for experiments and found at least 66%, 40%, and 96% overhead for integer, floating point, and memory access operations respectively as compared to Native C. Researchers also estimated that the overhead of Dalvik reduces with the new Ahead-of-Time (AOT) compilation technique in Android 5.0 (Ehringer, 2010). However, all performance evaluations of Dalvik VM reveal higher performance for native code than the Java code (Jenkins, 2016).

To the best of our knowledge, the performance evaluation detailed in this chapter is the first effort to evaluate the computational overheads of the MCC offloading techniques collectively. This study can help researchers in understanding the limitations and the main causes of performance degradation in current cross-platform ARM emulation techniques. The study can also help cloud service providers in the selection of offloading mechanism for resource constrained mobile devices based on the evaluated performance parameters.

3.2 Experiments: Performance Analysis of MCC Offload Enabling Techniques

In order to determine the severity of the computational overhead in the existing MCC offloading frameworks, we perform a performance analysis. The performance analysis is executed on the MCC offload enabling techniques, namely, system virtualization, application virtualization, and DBT. We execute benchmarks application on the MCC offload enabling techniques to evaluate the computational overhead of each. The experiments on the MCC offload enabling techniques reveal their performance overhead compared to native systems.

In the subsections below, we report the details of the experimental framework for the performance analysis. To evaluate the MCC offloading techniques, we utilized various smartphones, a server device, and multiple benchmarks. We performed experiments to analyze the computational overhead of three common MCC offloading mechanisms: **(a)** system virtualization based VM migration, **(b)** application virtualization based managed code migration, and **(c)** native code based application migration. We do not quantitatively evaluate the communication overhead of MCC offloading mechanisms. The main focus of our work is to evaluate the computational overhead of MCC offloading mechanism. Therefore, network and communication devices are not a part of our experimental framework.

3.2.1 Application Benchmarks

Application benchmarks evaluate the performance of a system or framework by executing a set of instructions to reveal the execution performance. We selected the application benchmarks for our problem analysis based on multiple criteria listed below.

- We focused on multimedia and SIMD instruction based benchmarks in most of the cases to highlight overhead of cross-platform SIMD instruction execution.
- Open-source benchmarks were utilized so that the results of the experiments can be replicated, verified, and thoroughly investigated.

Aside from the criteria of selection of the benchmarks, it must be understood that we are evaluating diverse systems. Therefore, for each performance evaluation, the benchmarks, and corresponding evaluation parameters can differ. However, we evaluate each MCC offload enabling techniques with a set of common benchmarks. These benchmarks are termed as multimedia benchmarks in this research work. Additional benchmarks are utilized in evaluation when required.

The selected set of multimedia benchmarks comprises of four applications, namely, Mathlib, Speed, Linpack, and FFT. The selected set of application benchmarks for the experiments were either multimedia based on included explicit SIMD intrinsic instructions. We utilized two versions of each benchmark; **(a)** a simple version based on scalar instructions compiled without optimization and vectorizing options and **(b)** a SIMD version that replaces suitable scalar instructions in original version with SIMD intrinsics. Moreover, the SIMD version of application benchmarks is compiled with optimizing and auto-vectoring options. The set of vectorized multimedia benchmarks is rigorously utilized in the performance evaluations performed throughout this thesis. An example of part of code with scalar instructions for Linpack benchmark and corresponding vector instructions for LinpackSIMD benchmark is illustrated in Appendix B.

The Mathlib application benchmark comprises of transcendental functions that are evaluated over a range of input variables. The performance of the system measured by Mathlib is in a unit of millions of vector evaluations per second (MVIPS). The Linpack is based on a series of algebraic routines solving a system of linear equations. The unit of performance in the Linpack benchmark is Millions of Floating-point Operations per Second (MFLOPS). Speed application benchmark calculates the data reading speeds of a system in Mbytes/second. Fast Fourier Transform (FFT) is one of the most commonly used algorithms in numerical computing and multimedia based applications. The FFT performs one-dimensional Discrete Fourier Transform (DFT) on real and complex vectors. The performance of the FFT benchmark is measured in MFLOPS.

We employed the aforesaid vectorized multimedia benchmarks and traditional Phoronix test suite for system virtualization. The Phoronix test suite contains tests that target various components of the system. We selected ten tests that target various aspects of system performance, such as, network, memory, CPU and graphics subsystems (Deshane et al., 2008). The processor performance was tested with 7zip and Scimark test suites. 7zip compresses a file and calculates Millions of Instructions Per Second (MIPS) during the compression process. The Scimark and Java Scimark tests inside the Phoronix test suite consist of the LU matrix factorization. Both tests evaluate the MIPS during the LU matrix factorization. To test the system I/O throughput, AIO-Stress test was used. AIO-Stress is an asynchronous I/O benchmark that uses a 2048MB test file and a 64KB record size for read and write operations. Stream benchmark is used to test system memory (RAM) throughput. The disk performance is evaluated by the Linux kernel unpacking benchmark. PyBench tests overall system performance by executing loops and built-in function in Python. The Loopback TCP performance test measures the loopback network adapter performance using a micro-benchmark. It calculates the time to transfer 10GB Via Loopback. Cairo is a 2D vector graphics drawing library that tests the performance of the system while executing graphic rendering commands. The render benchmark evaluates the performance of video drivers.

For application virtualization, we employed multimedia benchmarks and Scimark benchmark. We chose NIST approved Scimark for this purpose as it evaluates both Native C and Java code for several scientific computations (Boisvert, Moreira, Philippsen, & Pozo, 2001). The Scimark benchmark consists of: **(a)** a Fast Fourier Transform (FFT) of 1024 size performing one-dimensional forward transform of 4000 complex numbers, **(b)** Jacobi Successive Over-relaxation (SOR) method for solving 100*100 system of linear equations, **(c)** Monte Carlo (MC) integration that approximates the value of Pi through integral, **(d)** a 1,000 x 1,000 Sparse Matrix Multiplication (SMM) with 5,000 nonzeros, and **(e)** Lower Upper (LU) matrix factorization of a dense 100x100 matrix using partial pivoting. The Scimark calculates the MFLOPS while executing the aforementioned scientific applications. The performance of DBT techniques was evaluated by a series of customized nested loops programmed in C to and multimedia benchmarks.

3.2.2 Devices

We conducted experiments on a physical workstation to evaluate system virtualization overhead. The physical workstation represents a generic system with Intel Optiplex755 server. 64 bit Linux is utilized as the host architecture. VirtualBox (a hosted virtualization solution) is installed on the server to evaluate the system virtualization overhead. 64 bit

Table 3.1: Experimental Devices for Problem Analysis

Device	**Processor**	**Memory**	**OS**
S7560	1GHz (ARMv7)	1GB	Android 4.0
MT6582	1.3GHz*4 (ARMv7)	4GB	Android 4.4
MT6589	1GHz*4 (ARMv7)	2GB	Android 4.1.2
GTL9100	1.2GHz*2 (ARMv7)	1GB	Android 4.4.4
Zen5	2GHz*2 (Intel Atom)	2GB	Android 4.3
Server (Optiplex755)	2.3GHz*4 (x86)	4GB	Linux 14.04
Emulated OMAP3 (Qemu)	1GHz max(ARMv7)	512MB	linaro-nano 3.0

Linux is used as the guest OS with the physical memory equally divided between the guest and host OS.

Dalvik overhead is evaluated with the help of mobile devices and Android emulators installed on standard workstations. A set of five mobile devices is used to validate comprehensive results on heterogeneous processor architectures and OSes. The reason behind the selection of multiple Android devices is that Java compilation techniques have evolved over time with upstream Android Application Programming Interfaces (API). Recent AOT compilation makes the Dalvik overhead lesser as compared to previous Android APIs that were based on Just-in-Time (JIT) compilation. The selected mobile devices utilize different versions of Android. Moreover, most of the mobile devices used in the experiments are ARM based as ARM ISA captures 90% share of mobile market (Do, 2011). Table 3.1 lists the details of mobile and server devices utilized in the experiments.

ARM ISA emulation evaluation is conducted on the Intel Optiplex755 server. Latest versions of Qemu and gem5 emulators are installed from their respective repositories. We utilized standard ARM kernels for Qemu and gem5 emulation. For Qemu, we utilized Linaro based ARM file system compiled for ARMv7 based Versatile Express board for all experiments. For gem5, we utilized ARM Embedded Linux (AEL) file system for Versatile Express board. For most of the experiments, we used gem5.opt and gem5.fast binaries from five gem5 binary options to evaluate the performance of gem5. gem5.opt provides the worst emulation time as the binary includes symbols, tracing, and assert options. On the contrary, gem5.fast provides the best emulation time as is does not provide support for the symbol, tracing, and assert routines.

3.3 Experimental Results

In this section, we will present our experimental results in three different directions. Firstly, we present results of Type-2 system virtualization overhead while benchmarking a single VirtualBox instance. Secondly, we measure the performance of Dalvik VM and compare it with Native C and various versions of Android. We also evaluate the reduction in Dalvik overhead with the upstream Android APIs. Thirdly, we evaluate ARM ISA and Android emulators through system call emulation and full system emulation performance. Moreover, we also measure the performance overhead of ARM to Intel Atom translation layer in x86 based Android devices.

3.3.1 System Virtualization

MCC offloading frameworks often utilize system level hosted virtual machines to offload a virtual instance to the cloud (Satyanarayanan et al., 2009). We evaluated the overhead of hosted virtual machines for two purposes: **(a)** to demonstrate the overhead induced by system virtualization based MCC offloading schemes, and **(b)** to calculate the lower bound for ARM emulation performance overhead. In system virtualization, an OS is hosted above a virtualization layer. However, the instructions are not translated from one ISA to another. On the contrary, system emulation requires hosting of an OS over another OS with the additional overhead of instruction translation from one ISA to another. Therefore, the overhead of x86 to x86 system virtualization provides the lower bound for ARM to x86 emulation overhead. We utilized a set of multimedia benchmarks and Phoronix test suite v3.6.1 from Linux distribution for the evaluation.

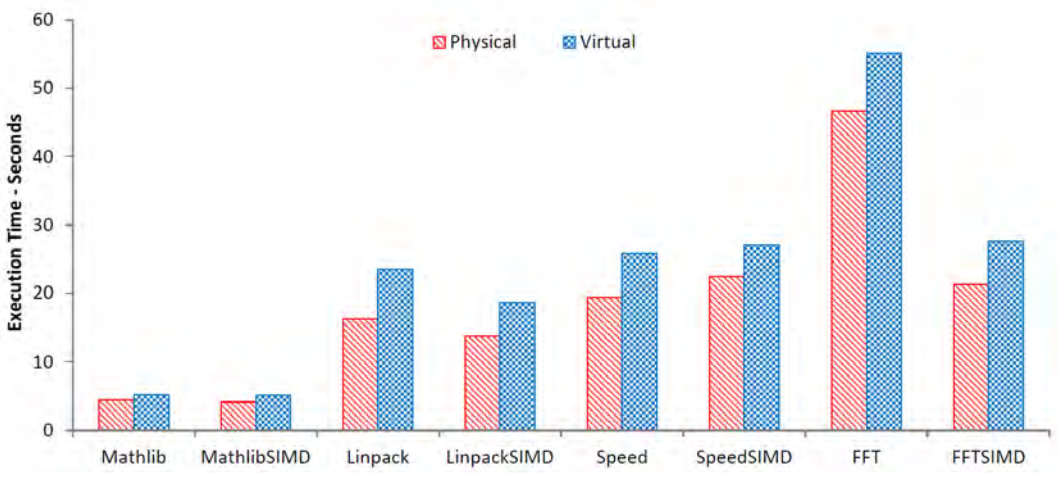


Figure 3.1: System Virtualization Evaluation with Multimedia Benchmarks

3.3.1.1 Multimedia Benchmarks

The set of multimedia benchmarks includes Mathlib, Linpack, Speed, and FFT. Each of these benchmarks executes a set of complex compute-intensive SIMD instructions. The benchmarks are executed simultaneously on physical and virtual systems so that they experience similar resource utilization levels. Both scalar and vector versions of each of the aforementioned benchmarks were utilized. The results of multimedia benchmark execution time for physical and virtual systems are shown in Figure 3.1.

The execution time of FFT benchmark is factored by seven as it skewed the bounds of the graph. There is considerable overhead for all of the benchmarks for the virtualized system. The result of eight benchmark tests reveals overhead of 14.53%, 19.06%, 30.75%, 26.66%, 25.13%, 16.50%, 15.27%, and 23.06% respectively for the virtualized system in terms of execution time. The results reveal an average overhead of approximately 21.37% for all multimedia benchmarks related tasks in a virtualized system. The overhead is considerable as the increase in execution time of the application will result in higher user response time.

Table 3.2: System Virtualization Evaluation with Phoronix Test Suite

Test	**Version**	**Target**	**Unit**	**Native**	**Virtual**	**Overhead**
7zip	compress-7zip-1.6.0	Processor	MIPS	2129	1766	17.05%
Native Scimark	scimark-2.1.2.0	Processor	MFLOPS	787.13	757.07	3.82%
Java Scimark	java-scimark-2.1.1.0	Processor	MFLOPS	1358	1296	4.51%
AIO-Stress	aio-stress-1.1.1	I/O	Mb/s	11.12	10.74	3.42%
Stream	stream-1.2.0	Memory	Mb/s	4431.96	4298.23	3.02%
Kernel Unpacking	unpack-linux-1.0.0	Disk	Seconds	24.72	27.47	10.01%
PyBench	pybench-1.0.0	System	Seconds	3.91	4.02	2.81%
TCP Loopback	network-loopback-1.0.1	Network	Seconds	72.13	316.65	77.22%
Render Bench	render-bench-1.1.2	Graphics	Seconds	30.67	41.14	25.44%
Cairo trace	cairo-pref-trace-1.0.1	Graphics	Seconds	1.49	4.16	64.18%

3.3.1.2 Phoronix Test Suite

We also utilized Phoronix test suite v3.6.1 to evaluate various aspects of system performance (Larabel & Tippett, 2013). The Phoronix test suite is commonly utilized to measure the performance of various system parameters and subsystems. However, among the available tests, we focused on graphics based benchmarks for multimedia applications. We executed the benchmarks simultaneously over both OS to evaluate the overhead of hosted virtualization. The benchmarks were executed for multiple runs to get an average estimate. We covered all aspects of system performance with processor, I/O, memory, disk, system, network, and graphics tests. Table 3.2 shows the results of these benchmarks on native physical and virtual systems.

The overhead of the virtual systems is between 2-4% for most of the benchmarks. However, two exceptions can be marked out in the form network and graphics tests which are both critical to offloading of multimedia based applications. The overhead of TCP loopback benchmark is more than 77%. We evaluated this overhead with different network configurations of the virtual machine. However, all of the scenarios depicted similar overhead. As network performance is a critical factor in MCC offloading frameworks, it can be stated that MCC offloading frameworks based on system virtualization will suffer high overhead in the form of network delay.

To test the performance for multimedia based applications, we executed two graphics based benchmarks from the Phoronix test suite. The Cairo and render bench tests resulted in 64.18% and 25.44% performance overhead for the virtualized system. The results imply that the performance of system virtualization for multimedia based applications will be low due to large overhead in the form of network and graphics parameters. The overhead of processor benchmarks i.e., 7zip, native Scimark, and java Scimark have overhead of 17.05%, 3.82%, and 4.51% respectively. The results imply that the minimum system virtualization overhead of processor related tasks will be 3-4%.

The results in this subsection show that the overhead of system virtualization for a CPU intensive task can range between 3.82% to 14.64%. Similarly, the overhead for network related tasks can be as high as 70%. Network performance can become a bottleneck as it is a critical factor for MCC frameworks that offload tasks to the cloud server over the network. The network performance bottleneck is aggravated by the fact that system virtualization requires the largest amount of data to be transferred over the network during the offloading process.

3.3.2 Application Virtualization

The following subsections present results related to the overhead of Dalvik VM. We selected Dalvik VM as Android devices dominate the majority of the smartphone market. Dalvik VM is a Java based application virtualization solution adopted in Android smartphones. The virtualization layer of Dalvik VM allows execution of an application from mobile space in the cloud without any code or binary level modifications. We chose Linpack and Scimark benchmarks for evaluation as they can be easily installed on Android devices from the Google PlayStore.

3.3.2.1 Multimedia Benchmark

Application virtual machines such as, Dalvik VM, are utilized in Android based smartphones to execute platform-independent Java applications. We argue that the overhead of application virtualization based Java as compared to native applications prohibits its

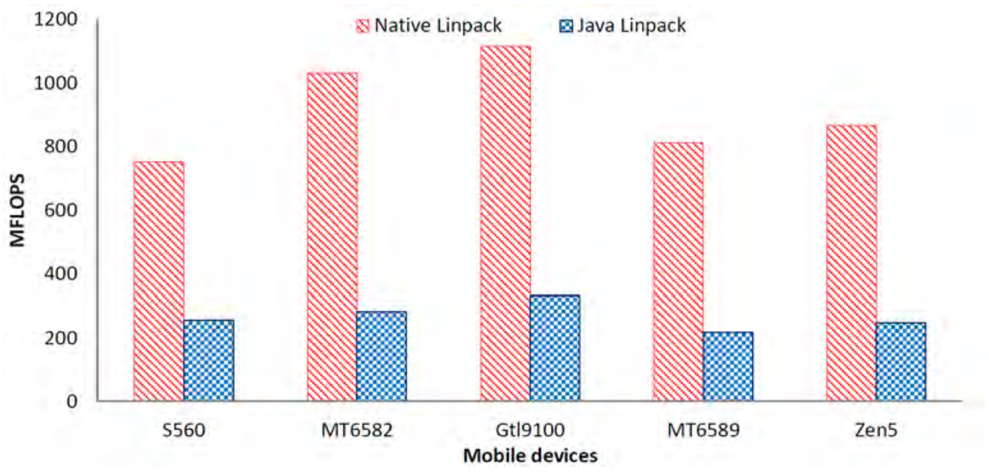


Figure 3.2: Application Virtualization Evaluation with Linpack: Performance in MFLOPS

usage in compute intensive multimedia applications. To evaluate the overhead of application virtualization, we utilized two versions of the Linpack benchmark. A Java version of the benchmark is devised from the native C version of the benchmark. The result of Java and native C Linpack performance is shown in Figure 3.2.

The result depicts that Java has a significant overhead as compared to native C code. The overhead of Java, when compared to native C, is 66.05%, 72.66%, 71.23%, 73.28%, and 71.25% respectively on the five mobile devices for MFLOPS calculation. On average the Dalvik to native C overhead is 70.89%. In modern smartphones, most of the applications are developed to run in Dalvik VM for cross-platform compatibility. Applications that are required to be offloaded through application virtualization techniques suffer from the overhead evaluated in this section. Moreover, applications that are not a candidate for offloading also generally suffer from this overhead as Java is mostly preferred for application development for Android. However, multimedia applications are compute intensive and require the support of low level C libraries for optimal performance. Therefore, the overhead of application virtualization is not desired for such applications. We did not utilize multi-threaded benchmarks. Therefore, the Linpack results on the mobile devices scale with the single processor speed. The execution time of the native and Java based

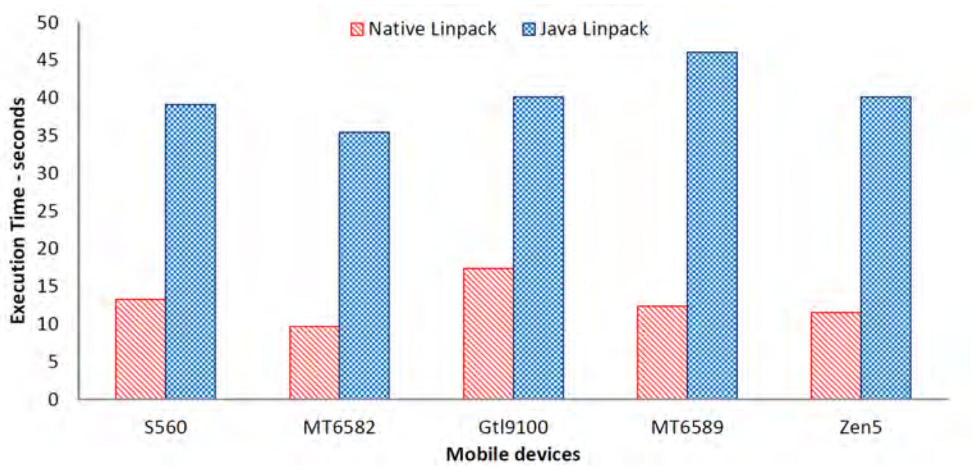


Figure 3.3: Application Virtualization Evaluation with Linpack: Execution Time

benchmarks reveal similar overhead. The result of execution time for Native and Java Linpack benchmark is depicted in Figure 3.3.

3.3.2.2 Scimark Benchmark

We also executed Scimark Benchmark to evaluate performance of the Dalvik VM. Scimark is the most commonly utilized benchmark for evaluation of application virtualization solutions. The Scimark benchmark consists of scientific applications programmed in both Java and Native C. SciMark benchmark evaluates five scientific algorithms and computes Million of Floating Point Instructions (MFLOPS) performed by the device during execution. Among the five scientific algorithms, FFT and matrix multiplication are backbone of many multimedia applications. Complete results of the Scimark benchmark for the set of mobile devices are listed in Figure 3.4.

We are interested in the difference of native C and Java results while ignoring the performance of individual devices and benchmarks. Scimark benchmark also derives composite scores based on the individual benchmark score for each device. The composite scores show the overall results of all scientific computations on the mobile devices. The native C performance compared to Java is quite high for all of the benchmarks and devices. In the composite scores, the overhead of Java as compared to native C is 44.03%,

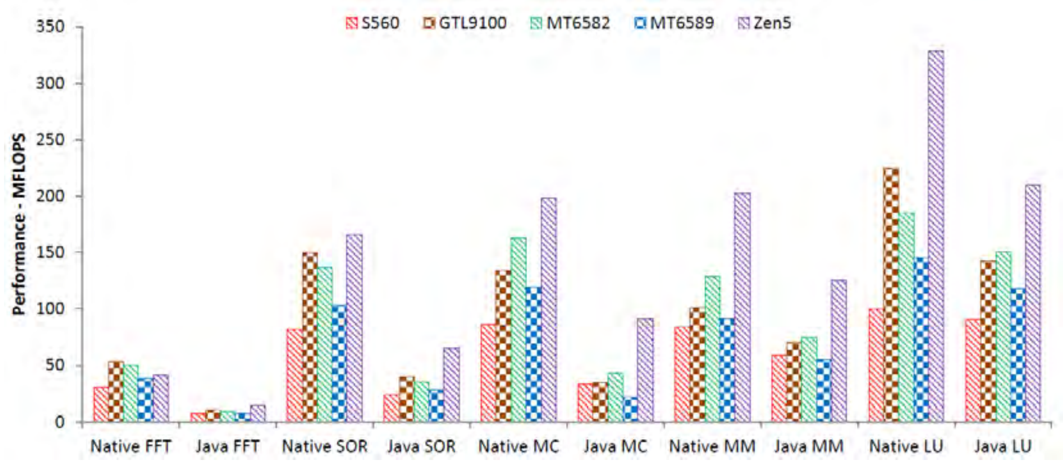


Figure 3.4: Application Virtualization Evaluation with Scimark Benchmark: MFLOPS Performance

55.18%, 52.78%, 53.85%, and 46.08% for the five mobile devices. The results of Scimark benchmark show a lower Dalvik overhead than the Linpack benchmark. However, a minimum overhead of 44.03% for the S7560 device is still significant in terms of scientific and compute-intensive applications. Moreover, our results depict a similar overhead for Dalvik Java to Native C as evaluated in an earlier study (S. Lee & Jeon, 2010).

3.3.2.3 Dalvik Compilation Method Optimization

Upstream versions of Android have optimized Java compilation methods that have lead to overall efficiency in Java performance. To evaluate the compiler optimization of Java, we executed the Whetstone benchmark on the series of Android API's, namely, API 21 (5.1.0 Lollypop), API 19 (4.4.4 KitKat), API 17 (4.1.1 Jellybean), and API 9 (2.3.7 Gingerbread). As real Android devices with old API's are no longer available in the market, we performed these experiments in the Genymotion Android emulator. The hardware configurations for the emulator were set at 1 processor core and 1536 RAM. The results of MWIPS performance for native and Java benchmarks are depicted in Figure 3.5.

The Native C performance does not deviate significantly for different Android API's. However, there are three significant changes in Dalvik performance. Firstly, the overhead of Dalvik reduces gradually with upstream Android API's. The overhead of Java as com-

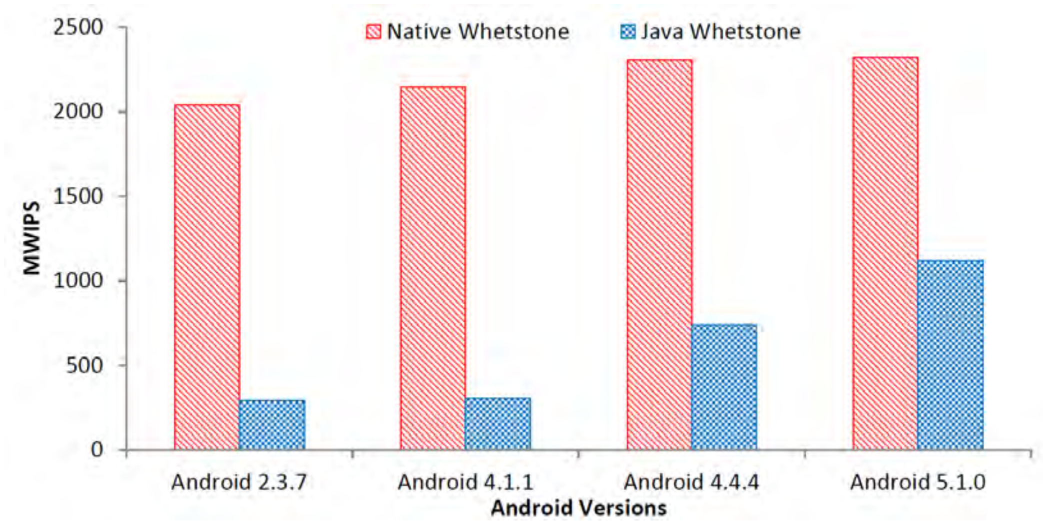


Figure 3.5: Java and Native C MWIPS Comparison for Upstream Android Versions: Performance in MWIPS

pared to Native C is calculated as 85.73%, 85.82%, 71.70%, and 51.69% respectively for Android API 9, 17, 19, and 21. Secondly, the Dalvik performance increases almost twofold while migrating from API 19 to 21. This performance increase is due to migration of Android API from JIT compilation to AOT compilation. In AOT compilation, an application Java bytecode is compiled to native code once and stored for subsequent executions. As a result, subsequent executions do not require translation to native code. The trade-off comes in the form of extra storage requirements for applications in AOT compilation. Thirdly, there is also a performance increase from Android API 17 to 19. This performance increase is due to Dalvik JIT code cache tuning, kernel samepage merging (KSM), and other optimizations that increase Java performance for Android API 19[1].

The performance of Java based application virtualization has significantly increased with the updates in the Android API. Incremental versions of Android API have included different Java compilation techniques. These Java compilation techniques have evolved from JIT compilation to AOT Compilation techniques. AOT performs better in term of CPU performance but consumes more memory for first compilation instance. However, approximately 50% overhead as compared Native C performance in AOT compilation

[1]http://developer.android.com/about/versions/kitkat.html

is still a significant factor that advocates migration from application virtualization based MCC offloading mechanisms for multimedia applications.

3.3.3 ARM ISA Emulators

ARM ISA emulators are essentially developed to test mobile code on Intel based systems. Therefore, the ARM ISA emulators tend to lag behind in performance with their physical counterparts. Qemu and gem5 are two mainstream ARM ISA emulators. In the below subsections, firstly we evaluate ARM ISA emulators based on customized applications to determine and compare their performance with eachother and physical systems. Afterward, we evaluate the ARM to Intel translations in the Android framework.

3.3.3.1 System Call Emulation

Qemu and gem5 can execute compiled process through the system call (user-mode) for multiple ISAs. A process that has been compiled for ARM ISA can be executed on the x86 ISA with the help of gem5 and Qemu system call mechanism. We used a series of nested loops programed in C to measure the performance of gem5 and Qemu system call emulation. The nested loops performed a singular integer operation for each iteration. We constructed loops with variable degree of iterations resulting in seven programs starting from 1 thousand loops to 1000 million loops. We calculated both CPU time from Unix time calls and wall clock time to measure the time of execution of these loops on Qemu and gem5 system call emulations. The CPU time is the sum of user and system time measured by the Unix time call. The wall clock time is the real world time the process took to execute. Program binaries were compiled statically with GCC compilers, i.e., gcc 4.8.4 for x86 host and arm-linux-gnueabi-gcc 4.7.3 for ARM platforms. The results of the execution times of system call emulation of qemu, gem5.opt, and gem5.fast are listed in Table 3.3.

The wall clock time of the results is impossible to measure for benchmarks that take

Table 3.3: System Call Emulation Performance: Execution Time

Number of Loops	**1K**	**10K**	**100K**	**1M**	**10M**	**100M**	**1000M**
Qemu CPU time (s)	0.060	0.064	0.067	0.072	0.141	0.818	7.924
Qemu Wall clock time (s)	NA	NA	NA	NA	NA	1.5	9.1
gem5.opt CPU time (s)	0.533	0.659	1.177	6.894	62.532	617.25	7123.12
gem5.opt Wall clock time (s)	NA	NA	1.3	7.9	65	675	7301
gem5.fast CPU time (s)	0.488	0.578	1.092	6.022	56.640	557.15	6443.03
gem5.fast Wall clock time (s)	NA	NA	1.1	6.3	57	565	6840

sub-seconds to execute. Such results have been marked as 'NA' in the above table. Moreover, floating point precision is hard to achieve for wall clock time of the benchmarks. Qemu outperforms gem5 in terms of both CPU and wall clock time for system call emulation. The reason behind the superior performance of qemu for system call emulation is that in system call qemu only emulates the target CPU without memory and I/O interfaces (Vincent & Janin, 2011). On the contrary, gem5 emulates the target host and OS with cycle accuracy during system call emulations. Qemu provides 87.7-99.87% faster execution than gem5.fast system call emulation in terms of CPU time for the seven loop programs. Qemu to gem5.fast performance ratio increases with the increase in the size of the loops. Similarly, Qemu performs 95-99.86% faster emulation than gem5.fast in terms of wall clock time. Moreover, gem5.fast provides approximately 7.22-12.64% faster emulation time than gem5.opt for the seven loop programs in terms of CPU time. Similarly, gem5.fast performs 6.31-24.05% faster than gem5.opt in terms of wall clock time. The gem5 CPU time and wall clock time results are similar as gem5 is a cycle-accurate simulator.

3.3.3.2 Full System Emulation

Qemu and gem5 can also emulate full systems with specific OS kernels. On an Intel based system, these emulators can host an ARM based kernel on emulated ARM board. Full system emulation experiments also utilized the same loops used in system call emulation. Table 3.4 lists the results of full system execution for the native device, Qemu, gem5.opt,

Table 3.4: Full System Emulation Performance: Execution Time

Number of Loops	**1K**	**10K**	**100K**	**1M**	**10M**	**100M**	**1000M**
Native CPU time (s)	0.00	0.00015	0.0017	0.0155	0.094	0.792	7.573
Qemu CPU time (s)	0.04	0.05	0.06	0.07	0.25	1.79	19.01
Qemu Wall clock time (s)	NA	NA	NA	NA	1	2.3	15
gem5.opt CPU time (s)	0.0001	0.00037	0.0007	0.001	0.04	0.45	5.90
gem5.opt Wall clock time (s)	1	1.3	3.2	11	80	900	10215
gem5.fast CPU time (s)	0.0001	0.00037	0.0007	0.001	0.04	0.45	5.90
gem5.fast Wall clock time (s)	NA	NA	1.1	6.5	65	730	7800

and gem5.fast hosts while comparing them with the physical device.

The native execution results were obtained from the GTL9100 mobile device. Qemu shows 99.7-55.75% overhead as compared to native execution with respect to CPU time. The overhead of Qemu reduces when the size of the workload is increased. On the other hand, native execution CPU time is slightly higher than gem5 CPU time due to mismatch of native and emulated hardware profiles. As gem5 is a cycle accurate emulator, the CPU time of an emulated program would match the CPU time of the program on actual hardware. A hardware board with similar specification would have consumed similar time to execute the loops. Such results are used to evaluate the accuracy of the emulator with respect to the emulated hardware (Butko, Garibotti, Ost, & Sassatelli, 2012). Therefore, due to cycle accurate emulation, gem5 CPU time is also slightly lower than Qemu CPU time. gem5.opt performs 73.69-99.75% faster than Qemu in terms of CPU time. Moreover, the gem5.opt and gem5.fast CPU time remains same on multiple executions due to cycle accurate emulation.

We are more interested in the actual world time taken by the emulator to execute the loops. In this respect, gem5 shows high overhead as compared to Qemu. For instance, the loop of 1000M linear instructions takes hours of wall clock to execute on both gem5.fast and gem5.opt as compared to few seconds on Qemu. Qemu provides 73.33-99.8% faster full system emulation than gem5.fast for the seven loop programs in terms of wall clock time. The large overhead in wall clock time of gem5 emulation contributes to its cycle

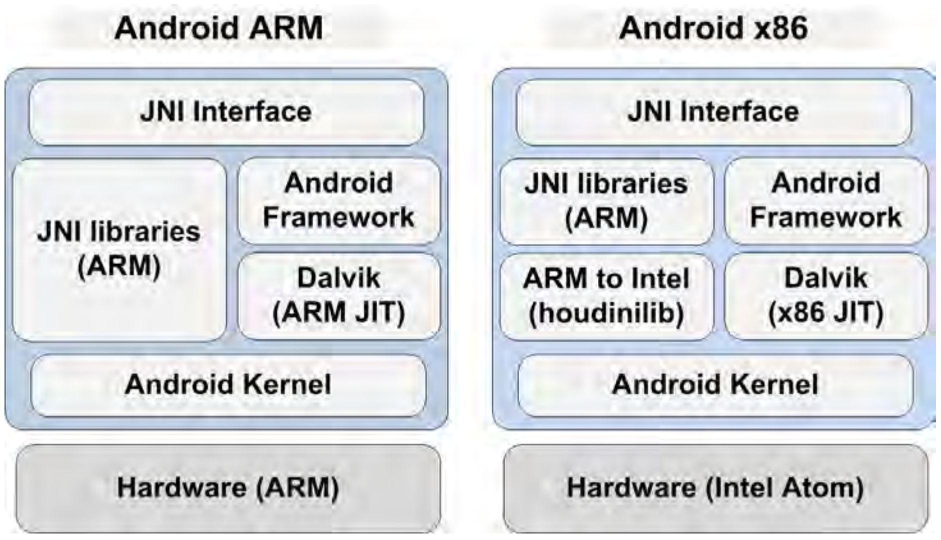


Figure 3.6: ARM and x86 based Android Framework

accurate design that precisely captures timing information of each instruction. Moreover, gem5.fast provides lower overhead than gem5.opt due to exclusion of debug and trace options. Specifically, gem5.fast executes 49.23% faster than gem5.opt in terms of wall clock time on average.

3.3.3.3 ARM to Intel Atom Emulation

Most of the mobile devices are equipped with ARM processors. However, some mobile vendors such as ASUS also utilize Intel Atom processors in mobile devices. As most of the mobile applications are compiled to execute on ARM natively, without translation these applications can not execute on Intel based mobile devices. Applications compiled with Java are compatible for both ARM and Intel based mobile devices due to application virtualization. Applications that target faster native C compilation have to address compatibility issue with both ARM and Intel architectures. To address the issue, Android versions compiled for Intel Atom processors have a compatibility layer called Houdini, which translates the ARM based applications to Intel architectures. The Houdini library effectively translates ARM instructions to Intel architecture (Choi & Lim, 2016). Figure 3.6 illustrates the difference between x86 and ARM based Android frameworks.

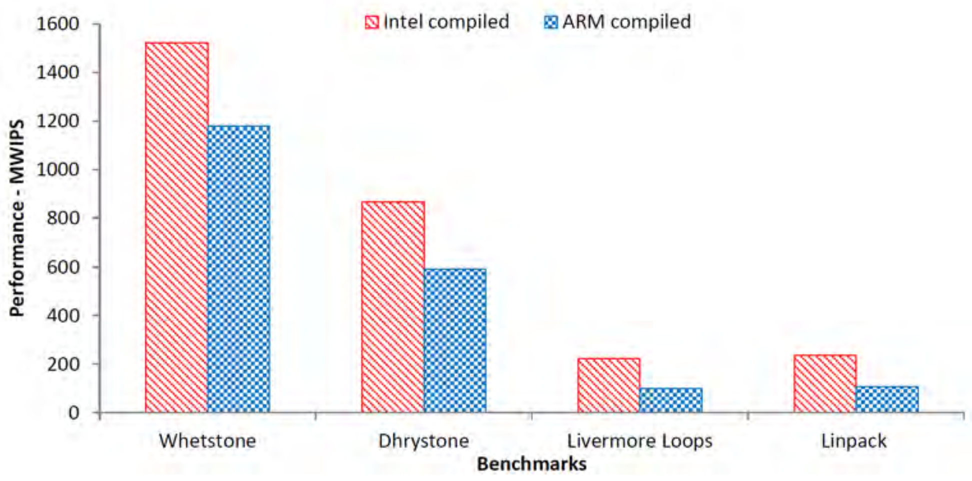


Figure 3.7: ARM to Intel Translation Overhead: Performance in MWIPS

There are several points to consider for Houdini translation in x86 Android. Firstly, Houdini provides translation for only Intel Atom based processors that are architecturally different from Intel processors used in server devices. Intel Atom processors utilize Bonnell Architecture, which translates CISC based x86 instructions into RISC like micro-ops for low energy operations. Therefore, Houdini ARM to Intel Atom translation layer is not scalable to server devices. Secondly, the Houdini translation layer is closed source. Therefore, it cannot be immediately extended for research purposes. However, the overhead of the Houdini translation can be similar to a scalable ARM to Intel translation framework. We evaluated the overhead of Houdini translation by executing four classic benchmarks that are compiled for both ARM and Intel native architectures. The classic benchmarks are executed on the Zen5 that is an Intel Atom based mobile device. The result of the performance of Intel native and ARM to Intel translated benchmarks is depicted in Figure 3.7.

The ARM compiled benchmarks go through the Houdini translation layer while the Intel compiled benchmarks are executed without any translation process. Therefore, the MWIPS executed by the ARM compiled benchmarks are always less than Intel compiled benchmarks due to the overhead of instruction translation by Houdini libraries. The overhead of Houdini translation comes out to be 31.63%, 22.58%, 55.53%, and 53.94% respectively, for Whetstone, Dhrystone, Livermore Loops, and Linpack benchmarks respectively. These results indicate a considerable overhead for ARM to Intel Atom translations in the existing x86 Android.

We evaluated the performance of ARM emulators in this subsection. The results show significant overhead during evaluation of system call emulation, full system emulation, and ARM to Intel Atom emulation for all ARM ISA emulation tools. Qemu and gem5 were utilized as the open-source ARM emulation tools. Qemu shows significantly better performance than gem5 for system call and full system emulation experiments. Therefore, Qemu is the ideal choice to be utilized as a SaaS tool for offloaded code from ARM based mobile device to Intel servers. ARM to Intel Atom translation by the Houdini layer can be considered as a best case analysis for overall ARM to Intel ISA emulation. However, on average, ARM to Intel Atom emulation overhead was found to be 40.92%.

3.4 Case for SIMD Instruction Optimizations

In this section, we forward the case of SIMD instruction optimization in heterogeneous MCC architectures. This subsection has two objectives towards its findings. Firstly, to assert the performance gain obtained using SIMD instructions, we execute the vectorized multimedia benchmarks. The performance enhancement in the case of SIMD instructions has a theoretical upper-bound equal to the depth of the SIMD vector. For example, if a SIMD instruction adds four integers of 32-bit length each in a 128-bit register, then the performance gain is equal to 4X. However, such theoretical bounds are impossible to achieve due to several reasons. Every instruction in an application is not a candidate for conversion to SIMD instructions. Moreover, the processor instruction cycles have complex dependencies leading to in or out-of-order execution of instructions resulting in execution latency.

Table 3.5: Mathlib and MathlibSIMD Comparison on Physical and Emulated Systems

Benchmark	**SIMD ratio**	**Execution time (sec)**	**Performance (MVIPS)**
Mathlib (physical)	17.16%	7.28	19.31
MathlibSIMD (physical)	24.41%	5.53	57.73
Mathlib (emulated)	17.16%	34.21	9.33
MathlibSIMD (emulated)	24.41%	29.95	11.94

The second objective of the findings presented in this section is to evaluate the overhead of DBT for native code offloading of multimedia benchmarks. Our assumption is that the current implementation of SIMD instructions in the ARM emulators is not efficient and has high overhead. Hence, when a SIMD based application is offloaded to the cloud server, the cross-platform execution leads to higher instruction count and lower performance due to non-optimal translations. In the below subsections, we analyze the SIMD translation in the current Qemu implementation. We compare and execute the benchmarks on the physical ARM based mobile device (GTL9100) and emulated ARM board (OMAP3 emulated in Qemu on Optiplex755 server). The GCC ARM compiler (arm-linux-gnueabihf-gcc-4.7) is used in all of the experiments in this subsection. Optimization flags for vector generation, such as *O3* and *ftree-vectorize* were used in case of SIMD versions of application benchmarks. We draw three inferences from the aforementioned results: **(a)** first for the case of SIMD instruction performance, **(b)** second for the general overhead of Qemu, and **(c)** third for the SIMD overhead of Qemu.

3.4.1 Mathlib

We executed the Mathlib benchmark on the physical and emulated devices. The result of the Mathlib execution time and SIMD instruction is listed in table 3.5.

As the target architecture is ARM in both physical and emulated cases, the binaries contain same number of SIMD instructions. Due to utilization of SIMD intrinsics and auto-vectorization flags, the percentage of SIMD instructions is higher in SIMD benchmarks than the basic benchmarks. The SIMD application benchmark shows 24.03% and

Table 3.6: Linpack and LinpackSIMD Comparison on Physical and Emulated Systems

Benchmark	**SIMD ratio**	**Execution time (sec)**	**Performance (MFLOPS)**
Linpack (physical)	0.75%	18.57	56.98
LinpackSIMD (physical)	3.15%	15.23	370.37
Linpack (emulated)	0.75%	161.58	15.31
LinpackSIMD (emulated)	3.15%	144.50	21.79

12.45% time efficiency on the physical and emulated system respectively compared to basic benchmarks. Similarly, the SIMD benchmark shows 66.55% and 21.85% performance improvement on the physical and emulated respectively compared to basic benchmarks. The time and performance efficiency is the result of employment of SIMD intrinsics and compiler optimization flags. The SIMD intrinsics result in higher SIMD instruction count and lower application execution time. However, it must be noted that the time efficiency provided by SIMD benchmark is approximately twice as high for the physical systems than the emulated system due to vector-to-scalar translations of Qemu. The emulated ARM device executing on Qemu leads to 78.72% and 81.53% overhead in terms of time and 51.68% and 79.31% overhead in terms of performance for Mathlib and MathlibSIMD benchmarks respectively compared to native execution.

3.4.2 Linpack

We executed scalar and vector versions of single precision 64-bit Linpack benchmark on both physical and emulated devices. The results of the experiment are listed in Table 3.6.

There are three implications of the aforestated result. Firstly, LinpackSIMD produces 84.61% (approximately 6X) and 17.98% performance enhancements as compared to Linpack benchmark on the physical device in terms MFLOPS and execution time respectively. The performance gain is quite substantial in the case of vectorized version of the Linpack benchmark. The performance gain is mainly due to increase in the SIMD instructions in the application binary. The number of SIMD instructions are 76.19% higher in the vectorized version of the Linpack benchmark than the non-vectorized version.

Secondly, the LinpackSIMD benchmark leads to 94.11% and 89.46% speedup on native execution as compared to emulated execution in terms of MFLOPS count and execution time respectively. However, if we make a comparison on the basis of BogoMIPS values of the systems, the LinpackSIMD benchmark leads to 37.19% and 33.96% speedup on native execution as compared to emulated system in terms of MFLOPS count and execution time respectively. BogoMIPS is a measure of the CPU speed in terms of MIPS. The server device has a BogoMIPS value of 4654.87 while Qemu has the BogoMIPS value of 471.61. The results point to the fact that the ARM to x86 translation or emulation has significant overhead which can be reduced with efficient translation and optimizations. The scalar Linpack benchmark leads to similar efficiency for the native execution, i.e., 88.50% and 73.13% speedup as compared to emulated system in terms of MFLOPS and execution time.

Thirdly, the increase in number of SIMD instructions translates well into performance gains for the physical mobile device. The SIMD benchmark leads to 84.61% and 17.98% efficiency in terms of MFLOPS and execution time on the mobile device. On the contrary, the performance gain on the emulated device in terms of MFLOPS and execution time are 29.73% and 10.57% respectively for the vector version of the benchmark. This result points to the fact that the ARM to x86 translation of SIMD instructions in Qemu is not efficient and can be enhanced with optimized translation framework.

3.4.3 Speed

We utilized the memory speed benchmark to further our case for SIMD translation optimization for cross-platform execution. Similar configurations were used as in the previous experiments. The results of executing the benchmark on physical and emulated device for various inputs are listed in Figure 3.8.

The three inferences listed in case of Linpack can be drawn from these results too.

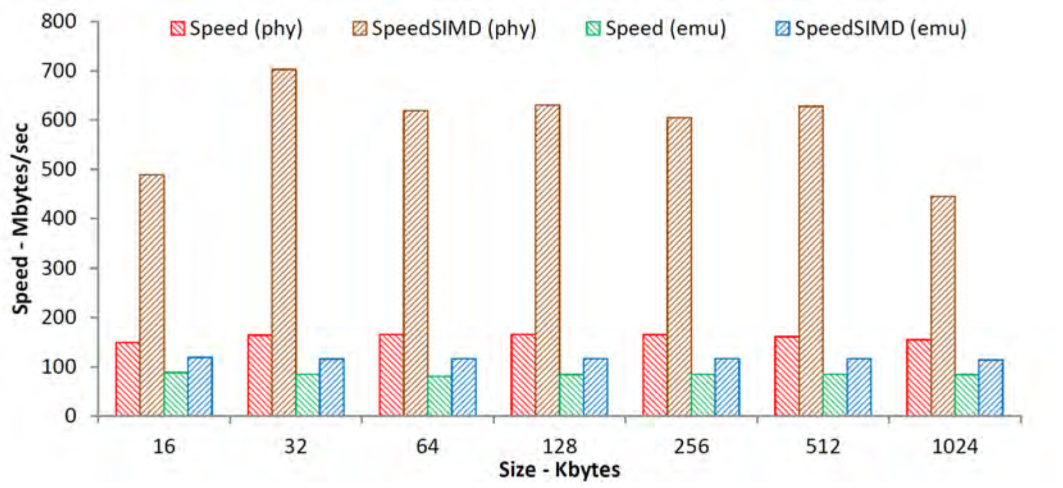


Figure 3.8: Speed and SpeedSIMD Comparison on Physical and Emulated Systems

Firstly, the Speed and SpeedSIMD binaries produced 17.49% and 11.92% SIMD instructions respectively. The memory benchmark produces a lower number of SIMD instructions while utilizing optimization and vectorizing flags. Compilers often tend to produce a lower number of SIMD instructions if the auto-vectorization options are applied to vectorized code (Maleki et al., 2011).

Secondly, the results show that the performance enhancement achieved by SpeedSIMD as compared to the Speed benchmark ranges from 76.67% to 65.16% for various input sizes on the physical device. The average performance enhancement in the case of SpeedSIMD benchmark is 72.66% on the physical device which is similar to the previous results of Linpack benchmark. On the contrary, the vectorized versions of the benchmarks produce only 27.53% better performance on average on the emulated device. Hence, it can be asserted that a part of SIMD performance gain is lost on the emulated systems due to non-optimal instruction translation.

Thirdly, the physical device produces 47.42% and 80.16% better performance (MB/s) than the emulated ARM system for the basic and SIMD benchmark respectively. However, if we make the comparison between the physical and emulated system based on the BogoMIPS values, the physical system performs 18.74% and 31.68% better than

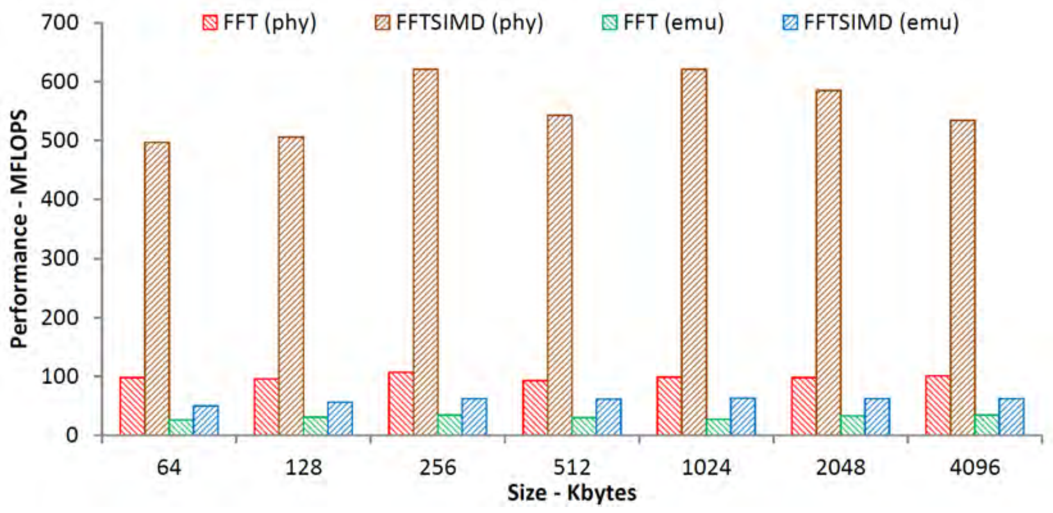


Figure 3.9: FFT and FFTSIMD Comparison on Physical and Emulated Systems

the emulated system for the non-vectorized and vectorized benchmarks respectively.

3.4.4 FFT

To establish our case of multimedia application based SIMD instruction optimization, we evaluated the performance of FFT benchmark on physical and emulated devices. FFT is also the backbone of many multimedia based applications, such as JPEG and MPEG encoding. Similar to previous experiments, we utilized two version of FFT benchmark, i.e., FFT and FFTSIMD. The results of the performance (MFLOPS) with different input sizes (64KB to 4096KB) on physical and emulated devices are shown in Figure 3.9.

The FFT and FFTSIMD benchmarks produce 1.05% and 11.02% SIMD instructions respectively when compiled by the GCC-ARM compiler. The results show that for all input sizes, the FFSIMD always performs better in terms of MFLOPS performance than the FFT benchmark on the physical device. Overall, the FFTSIMD performs 80.28% to 84.05% better than the FFT benchmark for different input sizes on the physical device. On average, the FFTSIMD performs 82.18% better than FFT benchmarks for all input sizes on the physical device. However, the performance gain on the emulated system reduces to 48.5% on average. The results show that on average 41.05% of performance lost is witnessed by the emulated systems due to non-optimal vector-to-scalar translation of SIMD instructions.

The physical system performs 68.92% and 89.62% better than the emulated system on average for FFT and FFTSIMD benchmarks. As the benchmark is optimized, the performance of the emulated system further decreases. This result also points out to the fact that the SIMD instructions are non-optimally translated by the DBT engine of Qemu. If we compare the physical and emulated system on the BogoMIPS values, the physical system performs 27.24% and 35.42% better than the emulated system for the FFT and FFTSIMD benchmarks respectively.

All of the results listed in this section show that the SIMD based applications lead to considerable performance optimizations as compared to basic versions of the same applications. In most of the cases, the performance gain was more than 70%. The performance gain is due to SIMD intrinsics and auto-vectorizing options utilized in SIMD benchmarks. On mobile and embedded devices, performance and time optimizations are particularly important due to several resource constraints, such as battery lifetime. The mobile battery life can be increased if the application utilizes lesser time and the hardware supports execution of vector instructions. Therefore, SIMD based applications and corresponding hardware support (e.g., ARM NEON) are important for the better performance of the mobile devices. Moreover, the aforementioned results show considerable performance overhead in the DBT of compiled code offloading. Furthermore, the SIMD benchmarks do not achieve the same performance gain on the emulated systems as compared to the physical systems. Therefore, optimization is required in the heterogeneous cross-platform execution of multimedia applications.

3.5 Discussion

In this chapter, we analyzed the computational performance overhead of the MCC offloading techniques. System virtualization suffers from performance overhead that is 4%

for computational tasks while more than 70% for network related tasks. For the multimedia benchmarks, a virtualized system suffers from 38% overhead on average as compared to the non-virtualized system. The overhead as compared to native execution is due to the virtualization layer that manages resources and instruction execution over real hardware. Similarly, application virtualization can lead to a minimum 66% overhead for multimedia applications due to application sandboxing and intermediate bytecode translations. Moreover, system and application virtualization techniques suffer from this overhead at both the mobile and cloud server side as the offloaded mobile instance has to execute inside a virtualization solution on both ends. Therefore, the performance overhead will double while considering both mobile and cloud executions. On the contrary, native code migration (compiled or pre-compiled) suffers from performance overhead on the cloud server side only due to the heterogeneity of architectures. Moreover, code migration/offloading leads to minimal communicational overhead as compared to system and application virtualization techniques due to the small size of the migration instance. Therefore, code migration best fits in most of the MCC offloading scenarios. However, selection of best mechanism to offload computation from a particular mobile device depends on multiple factors, such as network bandwidth, cloud/cloudlet proximity, and computation overhead.

The performance of an ISA emulator (DBT) can be several times lower than the native system for compiled code offloading. Qemu and gem5 are the two candidates for ARM ISA emulation. Our results show that for speed and high performance, Qemu is the obvious choice as it leads to lower execution time as compared to gem5. There are two main reasons behind the performance difference between Qemu and gem5. Firstly, Qemu was developed with the design objective of high performance and fast emulation. gem5, on the contrary, was developed with the design goal of first order performance accuracy in hardware and software system simulation. Secondly, gem5 is a cycle accurate simulator as compared to functional accurate Qemu. The cycle accuracy of gem5 leads to more simulation overhead as compared to Qemu. A functionally accurate simulator, such as Qemu, focuses on what a processor does and not how the processor does it. Moreover, a cycle-accurate simulator has to emulate the ISA according to the actual hardware and software semantics with accurate timing information (Yeh, Tseng, & Chiang, 2010). Both Qemu and gem5 are open-source emulators permitting further modification and optimization.

The native code offloading mechanisms demand that the cloud server performs the offloaded tasks is relatively lesser time so that the overall process is beneficial for the mobile client. Qemu is the best candidate for ARM ISA emulation due to lower performance overhead than gem5. However, Qemu shows approximately 70-80% overhead compared to native execution for compute intensive benchmarks. Moreover, we demonstrated from experiments that the SIMD instruction translation in current Qemu leads to 3-4X overhead when compared to native execution. Therefore, there is significant scope for improvement in the case of SIMD instructions. The performance evaluation performed in this chapter leads to many meaningful research directions. They are summarized as follows,

- There are overheads in all of the analyzed computational offload enabling techniques. Research can be carried out on any of the MCC offload enabling techniques, i.e., system virtualization, application virtualization, and DBT to reduce their overhead for efficient offloading.

- SMID instruction translation from ARM to x86 is not efficient in Qemu. Vectorized multimedia applications can have significant performance speedup with the efficient translation of guest SIMD instructions to host SIMD instructions in Qemu.

- The communication overhead of the MCC offloading techniques needs to be quantitatively investigated. The computation and communication performance evaluation will result in a comprehensive analysis of MCC offloading schemes. Modeling of both communicational and computational overhead is necessary to support an automated decision based MCC offloading framework.

3.6 Conclusion

In this chapter, the case for pre-compiled code translation for MCC offloading techniques was presented. The MCC offloading mechanisms can be effective for the mobile users in terms of energy savings. The mobile device can spend time in the idle state that consumes less energy while a task is offloaded to the cloud. MCC offloading mechanisms can only be effective with respect to execution time if the offloaded process executes faster on the cloud server than the mobile device. As observed in our performance evaluation, the overhead in terms of execution time for system virtualization, application virtualization, and ARM emulation is quite high as compared to native execution.

MCC offload enabling techniques, namely, system virtualization, application virtualization, and ARM ISA emulation were evaluated against a set of application benchmarks. MCC offload enabling techniques were experimentally analyzed and their performance overhead was determined. In most of the cases, such as Dalvik VM based application virtualization, the overhead was found to be as high as 70% when compared to native performance. The overhead for system virtualization was evaluated at 38% as compared to a non-virtualized system for multimedia benchmarks. Similarly, the DBT leads to approximately 70% overhead for compiled code offloading. The evaluation of the performance overheads leads us to the conclusion that instead of existing system virtualization, application virtualization, and emulation techniques, pre-compiled code offloading should be adopted. The pre-compiled code offloading based MCC framework should provide lower performance overhead as compared to the existing MCC offloading frameworks. Moreover, the framework needs to address the cross-platform execution and translation of multimedia applications that are rich in SIMD instructions.

CHAPTER 4: A FRAMEWORK FOR SIMD INSTRUCTION TRANSLATION AND OFFLOADING IN MCC: SIMDOM

This chapter presents the details of our framework for cross-platform translation and execution of SIMD instruction based applications in MCC. The framework for **SIMD** instruction translation and **O**ffloading in **M**CC (SIMDOM) comprises of two main components, namely, a SIMD translator and a application offloader. The SIMD translator maps ARM NEON SIMD instruction set to Intel x86 SSE instruction set such that the offloaded application seamlessly executes on the cloud server. SIMD intrinsics vary from platform to platform. Therefore, it is difficult to write an application based on SIMD intrinsics that can execute on multiple heterogeneous platforms, as in the case of MCC. SIMDOM framework facilitates execution of SIMD instructions in heterogeneous MCC architectures. SIMDOM framework offloads pre-compiled code to the cloud while avoiding the high overhead of emulation. Moreover, SIMDOM framework is vector application centric where SIMD instructions and their efficient translations are utilized for performance enhancements.

In the following sections, details of the SIMDOM framework are provided. In section 4.1, the overview of SIMD instruction based MCC offloading framework is presented. Section 4.2 describes various components of the framework, such as SIMD translator and application profiler in detail along with the corresponding algorithms. In section 4.3, the system model of the proposed framework based on time and energy optimizations is formulated. Section 4.4 provides the high-level flow diagram and algorithm for the SIMDOM framework. Section 4.5 provides the concluding remarks for this section.

4.1 SIMDOM Framework

In this section, we provide the overview of the SIMDOM framework in the form of framework features, high-level system architecture, and the framework design assumptions.

4.1.1 SIMDOM Features

The aim of the SIMDOM framework is to execute vectorized application is heterogeneous MCC architectures. The SIMDOM framework has three features that distinguish it from existing MCC offloading frameworks. These features are listed below.

1. The SIMDOM framework enables SIMD based multimedia applications to achieve efficiency in execution time and energy consumption while offloading to a cloud or cloudlet server.

2. The SIMDOM framework allows multimedia applications programmed for ARM NEON architecture to execute unmodified on x86 SSE architecture. Hence, the framework allows for cross-platform portability of SIMD instructions. A SIMD based multimedia application programmed for ARM based architecture can be executed seamlessly on a cloud server without the overhead of binary translation.

3. The SIMDOM framework has lower computational and communicational overhead due to the utilization of native code offloading and recompilation techniques.

4.1.2 System Architecture

We devise a framework for SIMD instruction translation and offloading in MCC (SIMDOM) to achieve the objective of cross-platform application execution in heterogeneous ISAs. The SIMDOM framework is enabled by native code offloading technique rather than conventional system and application virtualization techniques. Moreover, the code offloading is performed at the pre-compiled application level rather than the compiled application. The offloading of the compiled code leads to the overhead of binary translation. On the contrary, pre-code offloading requires translation of native libraries from one ISA to another. The SIMDOM framework performs recompilation and translation of the library that supports and defines SIMD intrinsics for the ARM ISA.

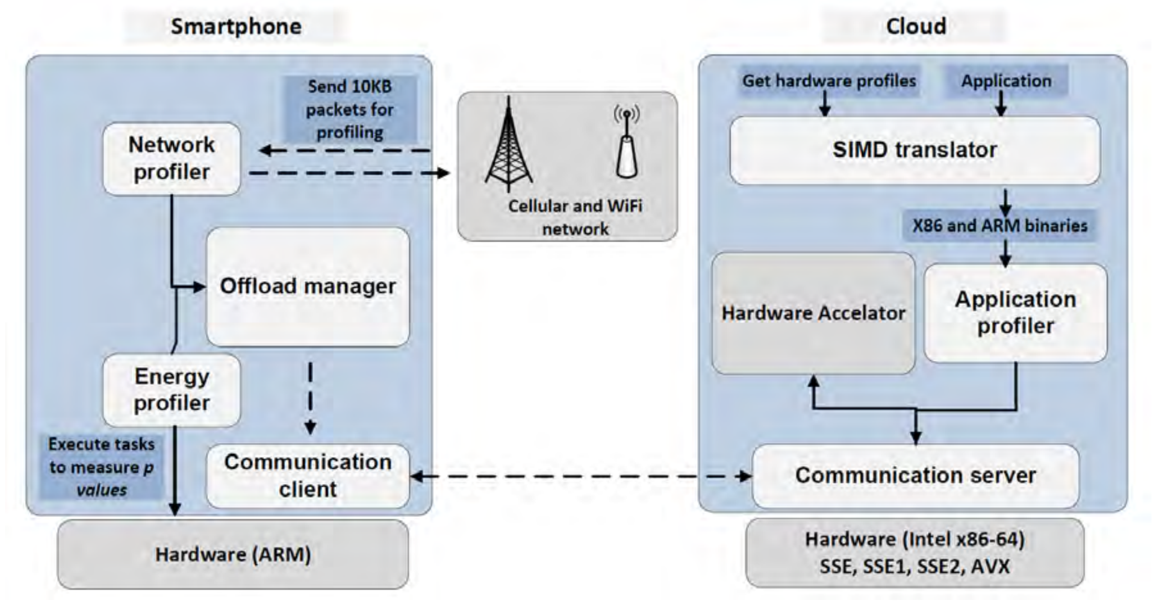


Figure 4.1: SIMDOM: A Framework for SIMD Instruction Based Multimedia Application Offloading in MCC

Our framework for cross-platform (ARM to x86) translation and execution of SIMD instructions is composed of five modules; **(a)** a SIMD translator for ARM intrinsic functions that enables execution of ARM based vectorized multimedia applications on x86 architectures, **(b)** an application profiler for static analysis of application to determine the optimal application partition for offload, **(c)** an energy profiler that measures the energy parameters of the system as input to the offload manager, **(d)** a network profiler that profiles that state of the network, **(e)** and offload manager that decides upon the feasibility of code offload based on inputs from other modules. The offload manager also acts as a client-server communication module. The proposed framework is depicted at the high-level system in Figure 4.1.

4.1.3 Assumptions

The SIMDOM framework is based on multiple assumptions. The SIMD translator works on ARM NEON intrinsic functions. Therefore, we assume that the smartphone application is programmed with the ARM NEON intrinsic functions to exploit the efficient translations of SIMDOM framework. The multimedia benchmarks and applications we tested were all programmed in C language. However, the SIMD translator module can be integrated into any Android, iOS, or Java project with simple header file (NEON-to-SSE) inclusions. The SIMDOM framework also assumes that the application is available on a cloud server for static application analysis. The SIMDOM framework does not take into account the mobile battery drainage and location profile, i.e., the amount of battery left and the distance of mobile from the cloud server while deciding upon the feasibility of the offload.

4.2 Components of SIMDOM

We will first describe the ARM NEON and x86 SSE SIMD profiles that are the target of our framework. The detail of SIMD profiles is necessary to understand the challenges and technical moralities of the SIMD translator. Afterward, we will explain each component of the SIMDOM framework in detail.

4.2.1 SIMD Profiles

SIMD instructions were first introduced to the ARMv6 architecture in 2009. The ARMv6 SIMD instructions operated on multiple 16-bit or 8-bit values packed into standard 32-bit general purpose registers. ARMv7 and ARMv8 enhanced SIMD instructions to 64-bit and 128-bit with the advanced NEON ISA. The ARM NEON ISA supports a comprehensive set of SIMD instructions while sharing some features with ARM Vector Floating Point (VFP) unit. ARM NEON supports 8-bit, 16-bit, 32-bit, and 64-bit integers. Moreover, ARM NEON also supports 32-bit single precision floating point values. 32 and 64-bit registers support the SIMD instructions that can be accessed from both ARM NEON and VFP co-processor. ARM NEON registers can be utilized in two configurations; firstly, as thirty-two double-word registers of 64-bit each (D0-D31) and secondly as sixteen quad-word registers (Q0-Q15) of 128-bit width each. However, data from the same register with different configurations can be accessed in the same instruction. Therefore, NEON instructions can have variable size input and output registers (Huang, 2011; Limited, 2009). ARM NEON is like a co-processor added to the general purpose CPU. NEON instructions execute in their own 10-stage instruction pipeline which can dispatch two NEON instructions per cycle. The processor can initiate some NEON instructions in the NEON pipeline while the normal pipeline executes scalar instructions. An example of ARM NEON instruction is $VADD.I32\ q1, q2, q3$ which adds four 32-bit integers.

Currently, Intel Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) architectures support up to 256-bit and 512-bit of SIMD instructions respectively. The series of SSE instructions sets is based on the previous MultiMedia eXtensions (MMX). The SSE ISA contains eight 64-bit registers (MM0-MM7) and eight 128-bit registers (XMM0-XMM7). The support for MMX registers and instructions was migrated to XMM registers in SSE2 ISA (Lomont, 2011). An example of x86 SIMD instruction is *_mm_add_ss* that adds 128-bit single precision values. Due to the difference in instruction length and register sizes, the translation and porting of SIMD instructions across Intel and ARM ISAs is not a trivial work.

4.2.2 SIMD Translator

The first phase of the SIMDOM framework comprises of the SIMD translator module on the cloud server. The SIMD translator requires that the smartphone application is available at the cloud server for translation and subsequent profiling. The task of SIMD instruction translator is made complex by the fact that SIMD instructions in compiled code can be obtained by multiple methods. Firstly, SIMD instructions can be hand-written in the assembly code. Although the performance of such hand-written code can be high, it is less readable and can lead to conflicts in the instruction pipeline. The second method of generating SIMD instructions in assembled code is to use compiler optimizations and auto-vectorization options. Auto-vectorization method solely depends on the capability of the compiler to generate SIMD instructions. Lastly, intrinsic functions can be used in C code to generate SIMD instructions. The compiler in-lines the intrinsic functions into assembly code. For ARM, *arm_neon.h* header file defines the SIMD intrinsics. Smartphone application developers utilize the SIMD intrinsics defined in *arm_neon.h* for execution of efficient vectorized code. The SIMD translator is applicable to the two latter methods of generating NEON instructions where a custom header file manages ARM NEON to Intel SSE translations. However, the proposed SIMD translations can be extended to the low level assembly code by replacing the corresponding assembly codes generated by the SIMD translator. We devised a best fit algorithm to map ARM NEON instructions to x86 SSE instruction. The SIMD translator is detailed in Algorithm 1.

Algorithm 1 ARM NEON to x86 SSE Translation Algorithm

```
 1: Input: ARM application, target ARM NEON and host x86 SSE SIMD profiles
 2: Include NEON-to-SSE header file
 3: for $Target\ Instruction = 0$ to $EOF$ do
 4:    if one-to-one map (target ins., host ins. = True) then
 5:       Optimize/pack 32 and 64 bit target ins. to 64 and 128 bit host ins
 6:       Produce bit masks if required
 7:       Copy operands and map registers
 8:       Calculate instruction overflow/check bounds
 9:       Output: Vector instruction corresponding to target ins.
10:    else
11:       Perform target ins. to host ins. mapping through multiple instructions (one-to-
          many)
12:       Find least number of host ins. for target ins.
13:       Produce bit masks if required
14:       Copy operands and map registers
15:       Calculate instruction overflow/check bounds
16:       Print user warning for serial implementation
17:       Output: Sequence of Vector/scalar instructions corresponding to target ins.
18:    end if
19: end for
20: Return x86 application
```

The SIMD translator works offline (on the cloud server). It translates the ARM NEON ABI (NEON intrinsic functions) to equivalent x86 SSE instructions. The offline optimization results in translation of approximately 1700 ARM intrinsic functions into equivalent x86 SSE instructions (*ARM® NEON™ Intrinsics Reference*, 2014; Lomont, 2011). The SIMD translations are defined in an NEON-to-SSE header file that must be included in the application code scope that is to be offloaded to the cloud. The SIMD translation reworks *arm_neon.h* header file that defines the ARM NEON intrinsics such that the compiler produces x86 target code rather than ARM code. The NEON-to-SSE header file first includes the header definition files of x86 SSE versions. If no SSE version is defined, then SSE2 is selected as default translation target as SSE4 is only available in newer versions of x86 ISA. Then the NEON-to-SSE file type defines both ARM NEON and x86 SSE data types utilized as inputs in the SIMD intrinsic function translations. Afterward, all the NEON intrinsic functions defined in *arm_neon.h* header file are declared, redefined, and implemented as x86 SSE functions. The SIMD translator generates an ARM application binary and an x86 application binary. The ARM binary is generated with the help of ARM compiler that does not require the SIMD instruction translation or re-compilation. The x86 binary is generated with the help of x86 compiler and the SIMD translator while mapping the SIMD instructions.

The SIMD translator makes the correspondence between ARM NEON intrinsics (as defined in *arm_neon.h*) and x86 SSE intrinsics (up to SSE4.2). However, some of these translations are not ideal due to the challenges listed in the previous section. For example, where one-to-one correspondence does not exist between SIMD instructions, serial implementation through multiple instructions is followed, which may result in low performance.

Precisely, the SIMD translator translates ARM NEON vector instructions to corresponding x86 SIMD instructions in three ways: **(a)** one-to-one mapping where one-to-one correspondence exists between ARM NEON and x86 SSE instructions, **(b)** one-to-many mapping with limited overhead, and **(c)** one-to-many mapping with serial implementation and low performance. The first case of ARM to x86 SIMD translations is represented by

```
int8x8_t vqadd_s8(int8x8_t a, int8x8_t b);
// VQADD.S8 d0,d0,d0
_NEON2SSE_INLINE int8x8_t vqadd_s8(int8x8_t a, int8x8_t b)
{
        int8x8_t res64;
        return64(_mm_adds_epi8(_pM128i(a),_pM128i(b)));
}
```

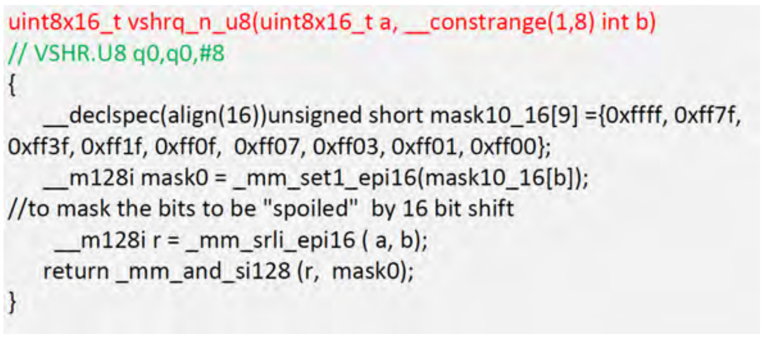
Figure 4.2: SIMD Translator: Sample Code for Case 1: One-to-One Vector Mapping

the branch taken code in Algorithm 1. The latter two cases are represented by the branch not taken code. The examples of the aforementioned cases are provided below.

- **Case 1:** An example of one-to-one mapping is *vadd_s*8 instruction which is mapped to *_mm_add_epi*8 instruction with simple input passing. Approximately 50% of ARM NEON instructions can be translated to x86 SSE instructions with one-to-one mappings. Figure 4.2 illustrates an example of the SIMD translator for simple one-to-one mapping.

- **Case 2:** ARM NEON instruction vector shift right by constant (*vshrq_n_u*8) has no corresponding equivalent in x86 SSE. Therefore, the translation is done with the help of bit masking on SSE 16-bit vector shift SSE instructions. The 8-bit input is initially masked or packed and converted to 16-bit (or higher) variable. The 16-bit ARM instruction is then mapped to 16-bit x86 instruction and unmasked or unpacked through logical operations. Around 45% of the NEON SIMD functions are implemented using one-to-many mapping if the performance effective implementation is possible. This case has limited overhead as the one-to-many translations are mostly vector to vector. Figure 4.3 illustrates an example of the SIMD translator for one-to-many vector mappings.

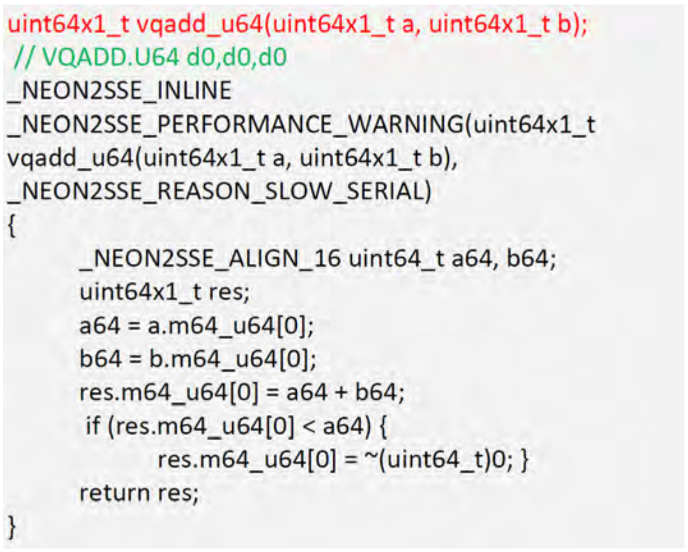
- **Case 3:** In the third case, the SIMD translator implements ARM NEON functions using serial implementation while issuing the corresponding low performance



```
uint8x16_t vshrq_n_u8(uint8x16_t a, __constrange(1,8) int b)
// VSHR.U8 q0,q0,#8
{
    __declspec(align(16))unsigned short mask10_16[9] ={0xffff, 0xff7f,
0xff3f, 0xff1f, 0xff0f,  0xff07, 0xff03, 0xff01, 0xff00};
    __m128i mask0 = _mm_set1_epi16(mask10_16[b]);
//to mask the bits to be "spoiled"  by 16 bit shift
     __m128i r = _mm_srli_epi16 ( a, b);
    return _mm_and_si128 (r,  mask0);
}
```

Figure 4.3: SIMD Translator: Sample Code for Case 2: One-to-Many Vector Mapping

compiler warning. The low performance in this case occurs as the SIMD translator can not map the NEON SIMD instruction to a corresponding vector instruction. Moreover, serially implemented instructions need to store data from vector registers to memory, process them serially and load them again to registers. For example, the result 64-bit vector saturating add (*vqadd_u*64) has to be checked for bounds. Therefore, after the translation of the instruction to corresponding SSE intrinsics, the upper and lower bounds of the result are checked leading to a serial implementation. In both cases of one-to-many mapping, the x86 SSE instructions utilized for translation are not necessarily SIMD instructions. Figure 4.4 illustrates example of the SIMD translator for serial one-to-many mappings with a performance overhead.



```
uint64x1_t vqadd_u64(uint64x1_t a, uint64x1_t b);
 // VQADD.U64 d0,d0,d0
_NEON2SSE_INLINE
_NEON2SSE_PERFORMANCE_WARNING(uint64x1_t
vqadd_u64(uint64x1_t a, uint64x1_t b),
_NEON2SSE_REASON_SLOW_SERIAL)
{
        _NEON2SSE_ALIGN_16 uint64_t a64, b64;
        uint64x1_t res;
        a64 = a.m64_u64[0];
        b64 = b.m64_u64[0];
        res.m64_u64[0] = a64 + b64;
         if (res.m64_u64[0] < a64) {
                res.m64_u64[0] = ~(uint64_t)0; }
        return res;
}
```

Figure 4.4: SIMD Translator: Sample Code for Case 3: Serial Implementation with One-to-Many Vector/Scalar Mapping

The ARM NEON to Intel x86 SSE translator is not commutative, i.e., one-to-one mapping between all instructions does not exist. Therefore, the reverse of ARM NEON to x86 SSE translations is not possible through the SIMDOM framework. As a result of SIMD translations, two executable and static binaries are produced each for ARM and x86 platforms. The executable produced for the x86 architecture does not require DBT when offloaded for execution on cloud server.

4.2.3 Application Profiler

The application profiling is performed to help the offload manager in determination the execution parameters such as local and remote application partition. The application profiler decides on optimal offload parameters through two steps. First, the application profiler utilizes application binaries obtained through the SIMD translator module for static analysis. The static analysis of the application binaries is performed with the $object-dump$ commands. As vectorizing capabilities of compilers differ, a comparison of GCC and LLVM compilers along with the corresponding compilation flags is done to decide the optimal compilation parameters for the static binaries. The optimal compilation parameters for the application should lead to the highest percentage of SIMD instructions in the application binary. The static analysis results in the calculation of the percentage of SIMD instructions in both ARM and x86 binaries. The calculation of SIMD instructions in the application binary is trivial for x86 architecture as the SIMD instruction is marked with *xmm* register tags. Therefore, a simple system command (grep -c 'xmm') can count the number of SIMD instructions in x86 application profile. However, the same is not true for the ARM binaries as SIMD instruction count based on NEON register tagging results in incorrect findings. Therefore, a profiling program is devised that takes the ARM binary as input and calculates the SIMD instructions as the sum of all ARM NEON instructions. However, as we found out through profiling of candidate benchmarks that only a subset of 20-30 SIMD instructions is produced repeatedly in the application benchmarks. Therefore, the profiling program was limited to the commonly occurring subset of active instructions to reduce profiling overhead. Application profiling based on all SIMD instructions in the corresponding hardware architectures leads to undesired overhead. On the contrary, limiting the application profiling to a subset of popular or active instructions can lead to 99.7% accuracy with limited overhead (Jeffery, 2009). The total number of lines was calculated with the system command $wc - l$ while excluding the common file headers for ARM and x86 binaries.

As a general rule, the application is feasible for offload if the percentage of SIMD instructions in x86 binary is higher than the ARM binary. The percentage of SIMD instructions in x86 binaries is often higher than corresponding ARM binaries due to two reasons. Firstly, the one-to-many mappings of ARM NEON intrinsics in SIMD translator results in more instructions (vector) in the x86 binaries. Secondly, the advanced vectorizing capabilities of x86 compilers often leads to higher percentage of SIMD instructions in x86 binaries. However, as a result of ARM NEON to x86 SSE translation, not all resultant instructions are SIMD. Some NEON instructions require scalar overflow and bound checking.

Our framework provides three simplistic application partition options based on the vectorized multimedia application profile: **(a)** the offloading does not save energy so no instruction is offloaded, **(b)** the complete application is offloaded to the server, or **(c)** only the SIMD intrinsics are offloaded to the server. Most of the SIMD instructions present in the static binaries are due to intrinsic functions. Therefore, the SIMD intrinsics are usually an ideal candidate for translation and offload. However, the complete application can also be offloaded if it is programmed with a focus on higher utilization of SIMD intrinsics. Multimedia applications are usually a candidate for complete application offload. Therefore, application partitioning is a relatively simple task for SIMD based multimedia applications. Other than native code SIMD intrinsics, SIMD instructions can be produced in the application binary due to auto-vectorized compilation. Moreover, we found that specific application compilation parameters lead to better SIMD output. Such parameters include auto-vectorizing, optimization, native/target ISA information and SIMD co-processor version variables that are passed to the application binary on the compilation. Most of the parameters for application compilation are initialized at the first step of the framework through get hardware profile function.

4.2.4 Energy Profiler

The basic purpose of the energy profiler is to profile the energy consumption of the device corresponding to various tasks. Hardware, software, and hybrid energy profiling techniques are used on the smartphone to measure the energy utilization corresponding to application execution. Hardware based energy profiling methods deploy a power meter with smartphone battery to measure the energy drain during application execution. Software based methods model smartphone subsystems, their power ratings, and their energy consumption behavior in various states to map an application profile for power consumption calculations (Ahmad et al., 2015; Hoque, Siekkinen, Khan, Xiao, & Tarkoma, 2015). We employed PowerTutor (L. Zhang et al., 2010) which is the most commonly utilized, accurate, and open-source power estimation tool for Android based smartphones. PowerTutor estimates the energy consumption of an application for multiple system parameters and subsystems such as CPU and WiFi in various power states (idle, busy) to find subsystem baseline power ratings. To calculate the p_i of the device, the energy profiler samples the device energy consumption in an idle state while limiting the number of background processes in LCD off state. To measure the power consumption of the device while executing computational applications p_m, the energy profiler executes multimedia applications. The energy profiler sends and receives data from the cloud to estimate the value of

p_c for Wi-Fi networks. The energy profiler provides these values as input to the offload module for the offload decision process. The device base power ratings are calculated once by the energy profiler and updated on each offload event to keep the updated device energy profile. The SIMDOM framework does not consider the device battery levels in the decision of offloading.

4.2.5 Network Profiler

The network profiler is responsible for providing the offload module with the input values of network parameters. The two main parameters required for the offload module are the up-link and down-link throughput of the network. The network throughput depends on multiple parameters such as wireless link bandwidth, latency, number of hops, data transfer protocol, radio state, etc. The network profiler periodically measures the throughput and Round Trip Time (RTT) to the cloud and cloudlet servers using the Wi-Fi network and saves the historical results for future inputs of offload module. Moreover, during an offload operation, the network profiler measures the current state of network and updates previous values (mean) through a simple sliding window protocol. The sliding window protocol keeps a record of the last 20 RTT measurements while allocating highest weight to the latest value. To measure the network parameters (up-link and down-link throughput), the network profiler executes a client-server data transfer program similar to code and data offloading. The program sends data from the mobile device to the remote and local server and measures the RTT. The data transfer program is executed multiple times to get a mean value of RRT. The RTT value is utilized in deriving the throughput of the network.

4.2.6 Offload Manager

The basic task of the offload manager is to decide upon the feasibility of code offload. The offload manager decides the feasibility of application code offloading based on the inputs from the energy profiler, application profiler, and network profiler. The offload manager selects the application partition which is most suitable for offload based on the system model for energy efficiency feasibility described in the next section.

The offload manager is also assigned two tasks of server communication. Firstly, it establishes a connection with remote cloud server to get the hardware profile for precise SIMD translations. The server hardware profile is used to map ARM NEON instruction to cloud server architecture. The server profiles may differ between various versions of the SSE (SSE2, SSE3, SSSE3, etc). Secondly, the offload manager is tasked to communicate application offload instances between the cloud and the smartphone. Upon identification of remote server execution part, the offload manager sends an offload request to the cloud server. The cloud server executes the code on the x86 hardware accelerator and returns result to the offload manager. The offload manager is responsible for the complete correspondence with the cloud server while the remote execution part is offloaded and sent back to the smartphone client.

4.3 System Model

In this section, we describe our system model based on feasibility of energy-aware code offloading framework for heterogeneous MCC architectures. Due to increasing usage of vectorized multimedia applications, the smartphone energy consumption and battery drainage have increased. Therefore, the basic objective of MCC offloading frameworks is to save smartphone energy. The objective of energy saving in MCC frameworks also corresponds to mobile users preferences obtained from user surveys (Falaki et al., 2010). Moreover, the mobile battery technology has not been able to keep pace with the power consumption of smartphones. Therefore, MCC offloading is a solution to the increasing user utilization and corresponding power consumption of smartphones. Previous efforts spent in modeling of MCC frameworks did not consider mobile and cloud ISA hetero-

Table 4.1: Symbol Table

Symbol	**Definition**
p_m	Smartphone power rating during local computation
p_u	Smartphone power rating for sending data
p_d	Smartphone power rating for receiving data
p_i	Smartphone power rating for waiting in idle state
I	Total number of application instructions
$I_l(host_{CPU})$	Application instructions executed locally on host architecture (smartphone)
$I_r(target_{CPU})$	Application instructions executed remotely on the target architecture (cloud)
D_r	Network down-link throughput
U_r	Network up-link throughput
s_m	Computational power (speed) of smartphone in MIPS
s_s	Computational power (speed) of server in MIPS
CPI_m	Cycles per instruction of smartphone
CPI_s	Cycles per instruction of server

geneity (Kumar, Liu, Lu, & Bhargava, 2013). We define a system model for heterogeneous cross-platform code offload in MCC. The list of symbols utilized in the model are described in Table 4.1.

The energy and time models of the system are related to each other. The energy model is derived from the time model while considering the base power of the task. We define the energy model of the system below. The corresponding time model can be obtained by excluding the base power (p) variable from the equations. For example, the time required to execute an application of I instructions on the mobile can be formulated as,

$$T = \frac{I \times CPI}{s_m} \tag{4.1}$$

The energy spent to perform a user task of I computations on the smartphone can be derived from Equation 4.1 as,

$$E_{local} = \frac{p_m \times I \times CPI_m}{s_m} \tag{4.2}$$

The value of CPI depends on the composition of benchmark application, the underlying hardware architecture, and the chip design. Based on SIMD instructions, we partition the application into local and remote execution parts. Suppose that a subset of the application, I_r instructions are offloaded to the cloud for remote execution. In this case, the energy consumption of the smartphone is the sum of: **(a)** energy spent to send application data I_r for remote execution, **(b)** energy spent on local execution of remaining I_l instructions, **(c)** energy spent in wait time while server performs cross-platform translation of I_r instructions, **(d)** energy spent in wait while the cloud server executes the translated I_r instructions, and **(e)** energy spent while receiving the result of I_r instructions. Incorporating these factors of offload scenario leads to the equation,

$$E_{offload} = E_{send(I_r)} + E_{exec(I_l)} + E_{wait(I_r \overset{translate}{\rightarrow} I_r)} + E_{wait(exec(I_r))} + E_{rec(res)} \tag{4.3}$$

where,

$$E_{send(I_r)} = \frac{p_u \times I_r}{U_r} \tag{4.4}$$

$$E_{exec(I_l)} = \frac{p_m \times I_l \times CPI_m}{s_s} \tag{4.5}$$

$$E_{wait(I_r \overset{translate}{\rightarrow} I_r)} = \frac{p_i \times \sum_{n=1}^{K} (I_r(host_{CPU}) \overset{translate}{\rightarrow} I_r(target_{CPU})) \times CPI_s}{s_s}, \quad \forall n, target_{CPU} \neq host_{CPU} \tag{4.6}$$

$$E_{wait(exec(I_r))} = \frac{p_i \times I_r \times CPI_s}{s_s} \tag{4.7}$$

$$E_{rec(res)} = \frac{p_d \times res}{D_r} \tag{4.8}$$

The aforementioned equations define the energy consumption of code offload scenario in case of heterogeneous mobile and cloud architectures. In the case of homogeneous architectures, the energy spent on the translation of instructions from the host architecture to the offload target architecture is nullified. The overhead of cross-platform code translation is directly proportional to the number of instructions that require translation. The translation of instructions also depends on the cloud server computational power and memory capacity. However, we ignore the memory capacity of a cloud server in our model due to resource-rich nature of cloud server and due to the limited number of memory pages required during the process of code translation. We assume that the cloud server send backs the result of offloaded computations (I_r) to the smartphone in the form of *res* which is an architecture independent variable that does not require translations. The major factors driving the $E_{offload}$ are the energy spent on local execution of I_l and the energy spent while waiting for translation and execution of I_r instructions on the cloud server. The computational offload is beneficial to the smartphone user in terms of energy if,

$$E_{local} > E_{offload} \tag{4.9}$$

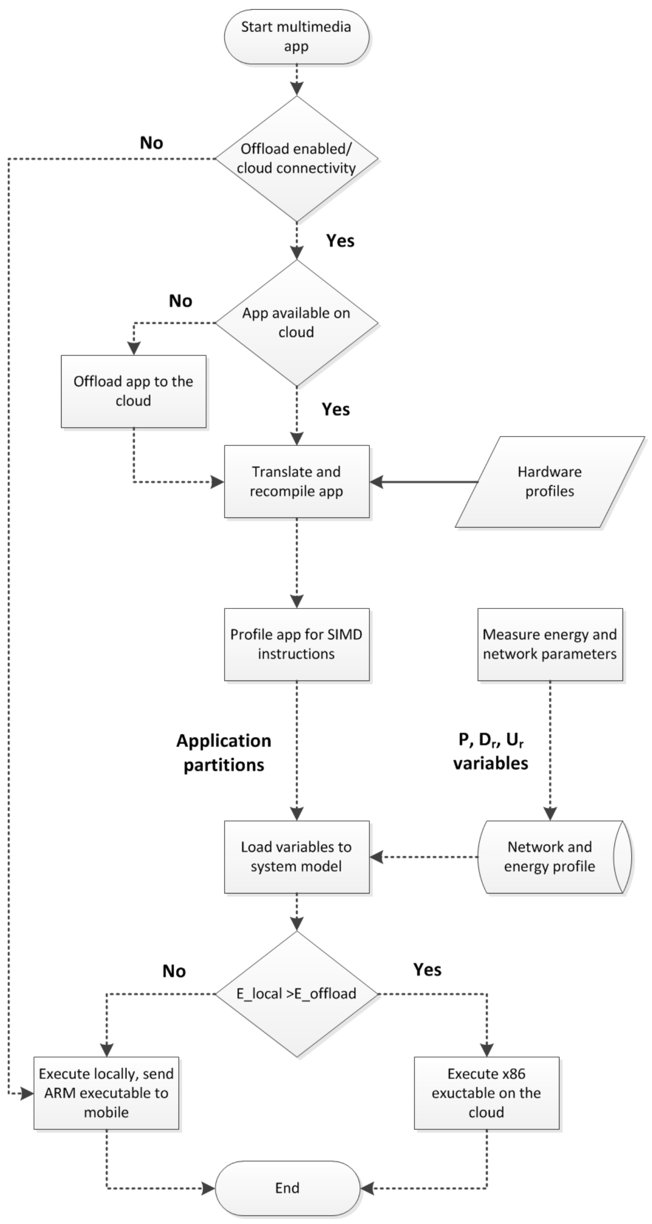
To decide upon the feasibility of offload, the offload decision module can chose among the minimum of E_{local} and $E_{offload}$ values. As the SIMDOM framework is based on pre-compiled code offloading, the variables, such as I_r and *res* are in the range of KBytes for most of the application benchmarks. Hence, the energy spent on sending and receiving data to the cloud is minimal and can be ignored or given lesser weighting (Kumar et al., 2013). The time taken to send or receive data over a wireless network link depends on multiple parameters. The undertaking of the complete model of a wireless network link leads to unnecessarily complex scenario (Bianchi, 2000). Therefore, we have followed a simplistic model in the aforementioned system model which is adopted by many MCC offloading frameworks (Altamimi, 2015). The energy consumed by the $E_{send(I_r)}$ and $E_{rec(res)}$ functions is a product of the base function energy and the time consumed while sending and receiving data. The time taken while sending or receiving the data to the cloud, $T_{(NT)}$, depends on the link throughput as defined in equation 4.10.

$$T_{NT} = \frac{I_r/(Data\_size)}{throughput} \tag{4.10}$$

4.4 SIMDOM Algorithm

We described the components of the SIMDOM framework and the system model in previous sections. In this section, we detail the overall algorithm of the SIMDOM framework and the corresponding flow diagram.

The SIMDOM framework starts by checking the cloud/cloudlet connectivity. Afterward, the application is offloaded to the cloud if it is already not available at the cloud server for analysis. The SIMD translator takes the application, the host (ARM based mobile device) and the target (cloud server, x86 SSE version) hardware profile as inputs. The SIMD translator translates the SIMD intrinsics of the ARM application to the corresponding x86 intrinsics. After translation, the SIMD translator generates two executable files of the application through re-compilation. One executable is for the ARM architecture (mobile device) and the other executable is for x86 architecture (cloud server). The x86 executable is generated through re-compilation of application with the help of an x86 compiler and the SIMD translator. These executable files are provided to the application profiler for calculation of respective SIMD instructions. The application profiler also defines the application partitions for local and remote execution. Meanwhile, the network profiler sends data packets to the local and remote cloud server to measure the RTT and throughput. The energy profiler executes tasks on the system to measure the base values of energy while the device is in idle, compute, and offload (Wi-Fi send and receive) state.



Start multimedia app
Offload enabled/ cloud connectivity
No
Yes
App available on cloud
No
Offload app to the cloud
Yes
Translate and recompile app
Hardware profiles
Profile app for SIMD instructions
Measure energy and network parameters
Application partitions
P, D$_r$, U$_r$ variables
Load variables to system model
Network and energy profile
No
E_local >E_offload
Yes
Execute locally, send ARM executable to mobile
Execute x86 exuctable on the cloud
End

Figure 4.5: Flow Diagram of SIMDOM: A Framework for Pre-compiled Multimedia Application Offload

The measurements of the application, network, and energy profiler are used by the offload module to decide the feasibility of offload decision. The flow diagram of the proposed SIMDOM framework is presented in Figure 4.5.

The main purpose of the SIMDOM framework is to enable a vectorized multimedia applications to execute on cloud and provide energy efficiency to the mobile device. To offload the computations from the mobile device to the cloud server, the SIMDOM framework performs a sequential procedure to decide upon the feasibility of code offload. The algorithm of SIMDOM framework is provided in Algorithm 2.

Algorithm 2 SIMDOM Framework: Algorithm

```
 1: Input: ARM application, hardware profiles, energy profile, network profile
 2: if Cloud connectivity = TRUE then
 3:     if Application available on cloud = TRUE then
 4:         SIMD Translator: Pass arguments to NEON-to-SEE translation algorithm
 5:         Recompile application to ARM and x86 executable
 6:         Application Profiler: Profile application for SIMD instruction
 7:         Energy Profiler: Measure $p_m$, $p_u$, $p_d$, and $p_i$ parameters
 8:         Network Profiler: Measure $T_{NT}$ through RTT and throughput for mobile to
            cloud/cloudlet server connection
 9:         Update: Energy and network parameters to database
10:         Get: Average of network parameters through sliding window update
11:         Pass: application, energy, and network parameters to system model
12:         if $Energy_{offload} > Energy_{local}$ then
13:             Send ARM executable to mobile for local execution
14:         else
                Execute x86 app on cloud and send result to mobile device
15:         end if
16:     else
17:         Offload application to the cloud
18:         Go to: Step 4
19:     end if
20: else
21:     Execute locally
22: end if
```

4.5 Conclusion

In this chapter, the SIMDOM framework for seamless SIMD instruction translation and offloading in MCC was presented. The basic objective of the SIMDOM framework is to enable offloading of SIMD instructions across heterogeneous mobile and server platforms with the help of re-compilation techniques. The SIMDOM framework consists of a SIMD instruction translator module that translates the ARM NEON intrinsic library to x86 SSE intrinsics. Hence, SIMD instruction based applications programmed for smartphones can execute on cloud servers without dependence on the virtualization technologies. The SIMD translator avoids vector-to-scalar translations that decrease the performance of existing native code offloading frameworks. As a result, offloaded vectorized applications can efficiently execute of the cloud server without performance loss. After translation, the SIMDOM framework decides upon the feasibility of cloud offload based on inputs from three profilers. These profilers are the network profiler that measures the network conditions, application profiler that calculates the application partitions, and energy profiler that measures the energy consumption of mobile subsystems. Based on the inputs provided by the profilers, the offload manager decides upon the feasibility of offload if $E_{local} > E_{offload}$.

CHAPTER 5: EVALUATION

This chapter presents the evaluation process utilized for evaluating the SIMDOM framework. The main purpose of this chapter is to detail the data collection methods, experimental setup, and parameters of the system model for the performance analysis of the proposed algorithms. The performance evaluation analyzes different components of the SIMDOM framework, such as the SIMD translator, application profiler, and network profiler. The evaluation process also describes how the experiments were performed and what were the assumptions undertaken in evaluation of the SIMDOM framework. Moreover, the data collection process for the analysis of SIMDOM framework is also described. Furthermore, mathematical analysis of the system model for the variable bounds is also performed.

The rest of the chapter is organized as follows. In Section 5.1, the process of evaluation is described at a high level. To further the details, Section 5.1.1 lists the experimental setup while Section 5.1.2 lists the mobile and server devices utilized in the evaluation. The application benchmarks and their input data are listed in Section 5.1.3. The data collection methods for experimental and mathematical evaluation of SIMDOM framework are presented in Section 5.2. The semantic accuracy and overhead of the SIMD translator is analyzed in Section 5.3. Section 5.4 details the SIMDOM application profiling overhead for x86 and ARM platforms. In Section 5.5, bounds for various variables of our mathematical model are derived. Section 5.6 provides cases studies evaluating energy consumption of the system and application virtualization based MCC offloading frameworks. The concluding remarks of this chapter are provided in Section 5.7.

5.1 Evaluation Process

The SIMDOM framework is designed to enable execution of vectorized multimedia applications on heterogeneous MCC architectures. The SIMD framework recompiles the mobile application for the cloud server along with the translation of the SIMD library. An efficient algorithm is developed to translate ARM SIMD instruction to x86 SIMD instructions while minimizing vector-to-scalar mappings. The SIMDOM framework receives input from the application, network, and energy profilers for the feasibility decision of cloud offload.

A prototype system is developed and deployed on a cloud and a cloudlet server to evaluate the SIMDOM framework. A local resource server located in close proximity of the mobile device constituted as a cloudlet and a remote server located far from the mobile device acted as the cloud. Multiple application benchmarks based on SIMD instructions are defined to analyze the performance of SIMDOM framework. Each application benchmark has two versions, i.e., a basic version and a SIMD version where some of the scalar instructions are replaced by the SIMD instructions. The data for benchmark applications is collected through application profiling tools. The data for energy profiler is collected through the PowerTutor framework. Client/server tests are performed to collect data for the network profiler. In the subsections below, we focus on the details of the experimental setup. The details encompass the devices and application benchmarks utilized in the experiments.

5.1.1 Experimental Setup

We utilized real-time experimental setup to evaluate the SIMDOM framework. There are multiple reasons behind the usage of real-time experimental setup. Firstly, simulation tools in the field of MCC are not mature and do not provide the technical capabilities to carry out research work of this nature. Secondly, simulation tools "simulate" the real-time parameters that leads to overhead and probabilistic estimations. Therefore, simulation tools are more vulnerable to result skewing and estimations that can lead to low accuracy. On the contrary, real-time system analysis provides in-depth knowledge of the system

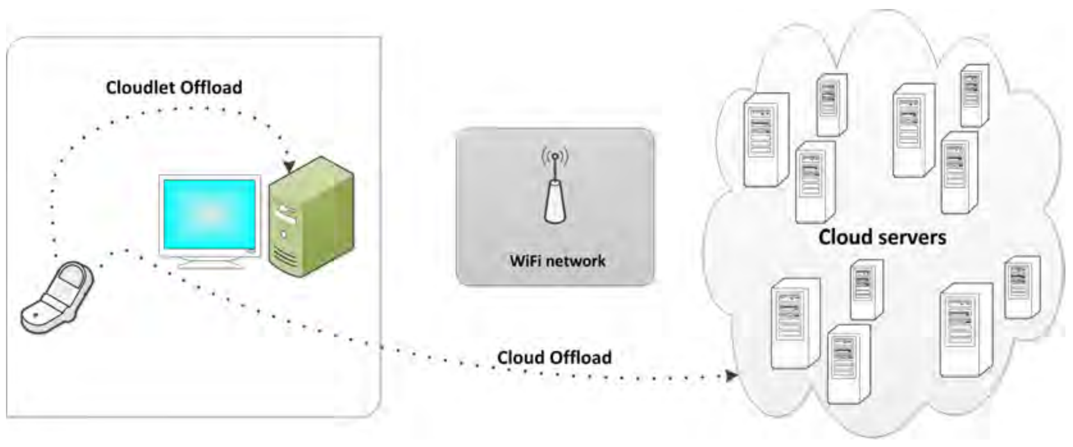


Figure 5.1: Experimental Setup for Evaluation of SIMDOM Framework

parameters that effect the performance of the framework. The experimental setup for the evaluation of the SIMDOM framework is presented in Figure 5.1.

A local server located in the same network of the mobile device constitutes the cloudlet execution scenario. A remote server located in a building away from the mobile device location constitutes as the cloud. The cloudlet and cloud server execute a prototype of the SIMDOM framework. Both the cloudlet and cloud server also run an instance of Qemu that provides for comparative analysis for the SIMDOM framework in form of native code translations and offloading.

5.1.2 Experimental Devices

We utilized multiple compute devices to test the SIMDOM framework rigorously. The specification of the devices and their computational resources is provided in Table 5.1.

Table 5.1: Experimental Devices

Device	**Processor**	**Memory**	**OS**	**BogoMIPS**
Mobile device - Samsung Galaxy S2 (LE)	1.2GHz*2 (ARMv7)	1GB	Android 4.4.4	1194.54
Local Server - Optiplex755 (LS)	2.3GHz*4 (x86)	4GB	Linux 14.04	4654.87
Remote Server - OpenStack (RS)	2.4GHz*8 (x86)	32GB	Linux 14.04	4788.05
Qemu Local Server (QLS)	1GHz max (ARMv7)	512MB	linaro-nano 3.0	471.61
Qemu Remote Server (QRS)	1GHz max (ARMv7)	512MB	linaro-nano 3.0	591.76

The mobile device is equipped with a Li-Ion 1650mAh removable battery. We incorporated Wi-Fi network in our framework evaluation. The smartphone device that offloads data and computations to the servers is equipped with Wi-Fi 802.11 a/b/g/n communication interface. The local and remote servers are equipped with wired Ethernet interfaces with maximum achievable speeds of 100Mbps.

We utilized PowerTutor power estimation framework to analyze the energy spent by the smartphone while performing different tasks (L. Zhang et al., 2010). PowerTutor is an Android based online power estimation framework that accurately measures the energy of various smartphone components. The PowerTutor provides various measurements for the experiments in the form of instantaneous power and energy consumption for different components of the mobile device. The source of PowerTutor is available for modification so that the tool can be customized according to the specifications of other mobile devices [1].

The performance of native execution of SIMD instructions was measured on ARM mobile device. The performance of binary translation of SIMD instructions was measured on the ARM emulated device utilizing Qemu. As the processing capabilities of these devices vary drastically, we also compare the performance on BogoMIPS values of the systems. BogoMIPS is a measurement of CPU speed made by the Linux kernel when it boots (E. Kim, Eom, & Yeom, 2012). The BogoMIPS is also not an accurate indicator of the performance of a system. However, the resultant evaluations are accurate enough to be considered as a scientific baseline (Camarasu-Pop, Glatard, & Benoit-Cattin, 2016).

5.1.3 Application Benchmarks

We utilized four application benchmarks for our experimental evaluation. The selection of suitable application benchmarks for the SIMDOM framework depended on three fac-

[1]https://github.com/msg555/PowerTutor

tors. Firstly, as we focused on vectorized multimedia applications, the benchmarks had to include SIMD intrinsic functions in them. The scope of our framework (SIMD intrinsic functions) limited the number of benchmarks that could be utilized in the evaluation. Secondly, the SIMD translator works on C intrinsic functions. Therefore, the benchmarks need to be developed in C code. Thirdly, we preferred open-source benchmarks for cross-validation of our results. Application programmers often do not utilize SIMD intrinsic functions as their usage restricts the application to a single platform. Therefore, there are only a few readily available standard multimedia benchmarks which make explicit use of SIMD intrinsic functions.

Based on the aforementioned criteria, we were able to select four benchmarks, i.e., Mathlib, Linpack, Speed, and FFT for our experimental evaluation. We devised two versions of each benchmark, i.e., a basic version and a SIMD version. The SIMD versions contain SIMD intrinsics and are compiled with vectorization flags. The code that differentiates the scalar version of the application benchmark from vector version for the Linpack is listed in Appendix B as an example. We provide the details of these benchmarks in below subsections.

5.1.3.1 Mathlib

The Mathlib library provides single precision sin, cos, log, and exp calculations for float vectors based on NEON intrinsics. The NEON math library has an equivalent library implementation for the Intel SSE architecture. The basic function of the math library is to calculate the transcendental functions (sines and exponential) over a range of floating point values (e.g., $-1000 \times pi$ to $1000 \times pi$) and evaluate the millions of vector evaluations/second (MVIPS) executed by the system. Moreover, the library finds the maximum deviation from the mean values for the aforementioned transcendental functions.

5.1.3.2 Linpack

Linpack Benchmark is based on a sequence of linear algebra routines. The benchmark normally operates on 200x200 matrices. The test problem requires the user to set up a random dense matrix A of size N = 200, and a right hand side vector B which is the product of A and a vector X of all 1's. The first task is to compute an LU factorization of A. The second task is to use the LU factorization to solve the linear system of the form provided in equation 5.1,

$$Ax = b; A\varepsilon R^{n\times n}; x, b\varepsilon R^{n} \tag{5.1}$$

The performance of the system is measured in MFLOPS while solving the above stated equation (Dongarra, Luszczek, & Petitet, 2003).

5.1.3.3 Speed

The Speed benchmark calculates the data reading speeds of a system in Mbytes/second carrying out calculations on arrays of cache and memory data of variable sizes. Calculations are presented by the equation 5.2,

$$x[m] = x[m] + s \times y[m] \tag{5.2}$$

where m is the input size read from the memory. The value of m varies from 16KB to 1024KB while x and y parameters are occupied by binary numbers (Wu, Krish, & Pellizzoni, 2013).

5.1.3.4 FFT

Fast Fourier Transform (FFT) is one of the most commonly used algorithms in numerical computing and multimedia based applications, such as JPEG and MPEG encoding for compression (Pommier, 2012; de Carvalho Jr, Rosan, Bianchi, & Queiroz, 2013; Don-

Table 5.2: Application Benchmarks

Benchmark	**Input**	**Input (values of N)**
Mathlib/MathlibSIMD	Float points multiple of π	$[0\pi, 1\pi]$, $[1000\pi, 1000\pi]$
Linpack/LinpackSIMD	Integer matrices	200, 400, 600, 800, 1000, 1200, 1400
Speed/SpeedSIMD	Integer matrices	16, 32, 64, 128, 256, 512, 1024, 4096, 16384, 65536
FFT/FFTSIMD	Real and complex number	64, 96, 128, 160, 192, 256, 384, 480, 512, 640, 768, 800, 1024, 2048, 2400, 4096, 8192, 9216, 16384, 32768, 262144, 1048576

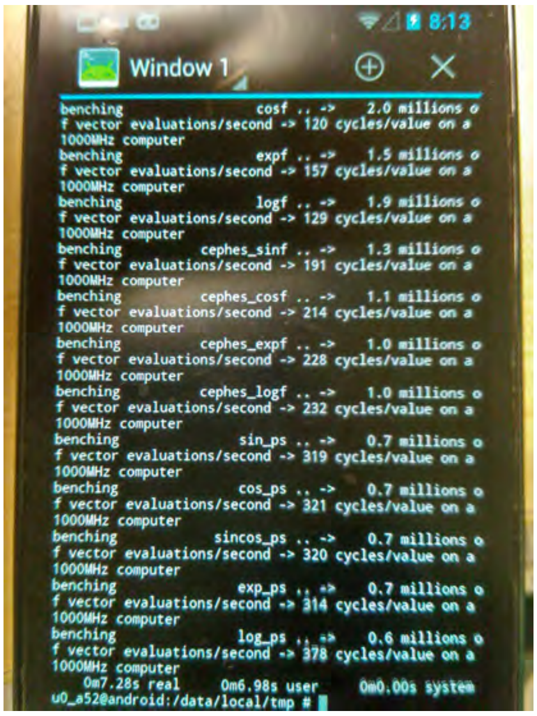
garra & Luszczek, 2005). Although there are multiple implementations of FFT available, we chose the one with SIMD intrinsics that can be utilized in performance evaluation of the SIMD translator. FFT performs one-dimensional Discrete Fourier Transform (DFT) of size m, of single precision real and complex vectors represented by equation 5.3.

$$Z_k \leftarrow \sum_{j}^{m} z_j e^{2i\frac{jk}{m}}; 1 \leq k \leq m \tag{5.3}$$

The performance of the FFT benchmark is measured in MFLOPS. The details of the input variables of the benchmarks are listed in table 5.2.

5.2 Data Collection for Model Validation

We collected the data for evaluation of the SIMDOM framework by rigorously testing the capabilities of compute and communicate interfaces of the experimental setup. We measured all results with the mobile battery at full capacity to remove the energy bugs. Moreover, we executed the experimental analysis multiple times (ten in most of the cases) to measure the mean and standard deviation values for the parameters. We utilize rounded nearest integer values in all of the derived parameters as the power measuring setup (PowerTutor) provides only integer value outputs. Figure 5.2 depicts the experimental setup of smartphone utilized for data collection. In the below subsections, the detail of data collection and derivation for system parameters is provided.



```
benching                    cosf .. ->    2.0 millions o
f vector evaluations/second -> 120 cycles/value on a
1000MHz computer
benching                    expf .. ->    1.5 millions o
f vector evaluations/second -> 157 cycles/value on a
1000MHz computer
benching                    logf .. ->    1.9 millions o
f vector evaluations/second -> 129 cycles/value on a
1000MHz computer
benching             cephes_sinf .. ->    1.3 millions o
f vector evaluations/second -> 191 cycles/value on a
1000MHz computer
benching             cephes_cosf .. ->    1.1 millions o
f vector evaluations/second -> 214 cycles/value on a
1000MHz computer
benching             cephes_expf .. ->    1.0 millions o
f vector evaluations/second -> 228 cycles/value on a
1000MHz computer
benching             cephes_logf .. ->    1.0 millions o
f vector evaluations/second -> 232 cycles/value on a
1000MHz computer
benching                  sin_ps .. ->    0.7 millions o
f vector evaluations/second -> 319 cycles/value on a
1000MHz computer
benching                  cos_ps .. ->    0.7 millions o
f vector evaluations/second -> 321 cycles/value on a
1000MHz computer
benching               sincos_ps .. ->    0.7 millions o
f vector evaluations/second -> 320 cycles/value on a
1000MHz computer
benching                  exp_ps .. ->    0.7 millions o
f vector evaluations/second -> 314 cycles/value on a
1000MHz computer
benching                  log_ps .. ->    0.6 millions o
f vector evaluations/second -> 378 cycles/value on a
1000MHz computer
    0m7.28s real     0m6.98s user     0m0.00s system
u0_a52@android:/data/local/tmp #
```

Figure 5.2: Experimental setup for measurement of energy coefficients

5.2.1 Idle Power

We sampled the device energy consumption while limiting the number of background processes and turning off the LCD to calculate the p_i of the device. When a computationally intensive task is offloaded to the cloud for energy efficiency, the mobile user usually turns the LCD off for maximum benefits. The mobile LCD is switched on once the results of task offloading are received. To create a similar scenario, we turned the mobile screen off after starting the energy profiler while the user is in the wait state. We found that the average value of p_i varied with the time it took for the offloaded task to be completed. For instance, simple loop executions of 1M instructions take few milliseconds to execute on the remote server. Therefore, the time that mobile device is in an idle state and the LCD off time is minimal that leads to higher p_i values. As we are focusing on multimedia benchmarks that are compute intensive, we utilized the SIMD versions of the benchmarks, i.e., MathlibSIMD, SpeedSIMD, LinpackSIMD (1200), and FFTSIMD benchmarks for our evaluation. These benchmarks take a considerable amount of time to execute on the local server. We add statistical analysis to each result in the form of

Table 5.3: Experimental Evaluation for p_i

Benchmark	**Input**	p_i **mean**	p_i **std. dev.**	**95% confidence interval**
MathlibSIMD	all	28mW	0.89	±0.78
SpeedSIMD	all	27mW	0.83	±0.73
LinpackSIMD	1200	27mW	1.22	±1.07
FFTSIMD	all	25mW	1.14	±1.00

the sample mean, sample standard deviation, and 95% confidence interval values. The findings of these experiments are presented in table 5.3 for five experimental runs.

Based on the aforementioned evaluation, we found out that the mean p_i of our device is 26.75mW with a standard deviation of 1.25 (based on mean sample population). The 95% confidence interval for the value of p_i is ±1.22. We also consider an alternate case for p_i of the device. In the alternate case, the LCD of the mobile devices is turned off after some pre-defined time of inactivity. Most of the smartphones provide the users multiple choices of when to turn the device into sleep or idle mode. We consider the most energy efficient case where the device is shifted to the idle state after 15 seconds of inactivity. Therefore, for offloaded codes that take less than 15 seconds from the time the device offloads task to the time the device receives the results, we consider a different value of p_i. We turned on the LCD and measured the value of p_i while no task is being performed. The p_i of the device in this case is 402mW.

5.2.2 Computing Power

We executed MathlibSIMD, SpeedSIMD, LinpackSIMD (1200), and FFTSIMD benchmarks to measure the power consumption of the device while executing computational applications p_m. The findings of the experiment are listed in table 5.4.

Table 5.4: Experimental Evaluation for p_m

Benchmark	**Input**	p_m **mean**	p_m **std. dev.**	**95% confidence interval**
MathlibSIMD	all	585mW	2.73	±2.39
SpeedSIMD	all	578mW	2.07	±1.82
LinpackSIMD	1400	576mW	2.40	±2.10
FFTSIMD	all	571mW	2.12	±1.86

Table 5.5: Experimental Evaluation for p_c

Utility	**Data transferred**	p_c **mean**	p_c **std. dev.**	**95% confidence interval**
Down-link	100 KBytes, 0.2s interval, 50s sample	485mW	2.58	±2.26
Up-link	100 KBytes, 0.2s interval, 50s sample	461mW	3.27	±2.87

We observed that the average p_m values for the benchmarks are similar. The p_m value decreases as the computational size and execution time of the benchmark increases. Based on the aforementioned measurements, the average mean value of p_m for our mathematical model is 577.5mW with a standard deviation of 5.80 based on the mean sample population. The confidence interval is ±5.68 for 95% confidence level.

Our case of SIMD based instructions is also advocated from these results. The benchmarks take variable time for execution on the mobile device. The FFTSIMD and FFT benchmarks took 46.02sec and 463.47sec respectively while performing the same operations over the same input on the mobile device. The FFT benchmark took 90.07% more time in execution as compared to FFTSIMD. However, the instantaneous energy consumption was similar for both FFT and FFTSIMD. The only difference in the programs was the inclusion of SIMD intrinsic functions in the FFTSIMD benchmark. The total energy consumption of the mobile device is approximately ten times higher while executing the FFT benchmark. The SIMD instructions do not affect the instantaneous energy but lead to lesser total execution time and total energy.

5.2.3 Wi-Fi Power

We created a data transfer utility that sends data to the local server and remote cloud to estimate the value of p_c for Wi-Fi subsystem. The utility sends ICMP packets of 100 KBytes each with a delay interval of 0.2 seconds for total time interval of 50 seconds to our remote server. We created a similar utility to measure the down-link power consumption of the Wi-Fi subsystem. The mean and standard deviation values for the down-link and up-link p_c are listed in Table 5.5.

Table 5.6: Experimental Evaluation for RTT and Throughput

Connection	**RTT minimum**	**RTT maximum**	**RTT mean**	**RTT std. dev.**	**Throughput maximum**
Cloudlet-up	8.85ms	16.48ms	12.53ms	2.24ms	5.23Mbyte/sec
Cloudlet-down	1.52ms	18.10ms	4.17ms	6.41ms	10.22Mbyte/sec
Cloud-up	7.90ms	20.03 ms	15.78ms	3.25ms	4.15Mbyte/sec
Cloud-down	2.43ms	28.01ms	14.94ms	8.14ms	4.38Mbyte/sec

5.2.4 Wi-Fi Throughput

A number of parameter are involved in the measurement of the capability of a wireless connection based mobile device to offload data to the cloud server. The end-to-end throughput of the link depends on parameters such as wireless link bandwidth, latency, number of hops, data transfer protocol, radio state, etc. Our network profiler periodically measures the throughput and Round Trip Time (RTT) to the server and saves the historical results for future inputs to the offload decision module. To measure the network parameters (up and down-link throughput) we conducted experiments with client-server data transfer programs similar to code and data offloading programs. The program sends data of size 100KB from the mobile device to the remote server and measures the delay. We conducted similar experiments to measure the down-link capacity of the network from the remote server. The maximum throughput for the link can be calculated as,

$$throughput \leq \frac{RWIN}{RTT} \tag{5.4}$$

where $RWIN$ is the window size of the data transfer protocol. We conducted similar experiments to measure the down-link and uplink capacity of the network for the both remote and local server. The values of minimum, maximum, average, mean deviation of the RTT, and maximum throughput based on the average RTT for cloud and cloudlet server are listed in table 5.6.

The results depict that the RTT and throughput to the cloudlet are superior than that of the cloud server. Therefore, the energy and time cost of the cloud server will be higher

Table 5.7: Experimental Evaluation for *l*

Benchmark	**Total instructions - x86**	**Total instructions - ARM**
FFT	1179874658407	343021290747
FFTSIMD	76014564517	32864558129
Linpack	58864968467	13502810905
LinpackSIMD	49419351005	12660729092
Speed	69721726315	19506996423
SpeedSIMD	80479847838	20831136457
Mathlib	12623014875	3356970193
MathlibSIMD	11755307177	2540303047

than the cloudlet.

5.2.5 Application Instructions

The number of instructions in a program are not deterministic and can vary on a number of parameters. The source code of a program is not an exact measure of how the program is executed as a number of dynamic system libraries are involved in the overall execution. There are multiple application profiling software that provide the detailed information of application instructions, cache hit rates, memory fetches, etc. There are a number of program instrumentation tools available such as, performance counters for Linux (PCL), Google perf, and Valgrind. We used callgrind tool of the Valgrind framework to calculate the total number of instructions executed by our benchmarks (Nethercote & Seward, 2007). Valgrind is the most commonly used tool in research regarding the program performance, memory allocation, and OS interactions. The number of instructions executed by an application varies on the base of several parameters, such as compiler tool-chain, compilation flags, and target hardware architecture. For ARM applications, we utilized a cross-compiled version of Valgrind. The details of the number of instructions for the application benchmarks is provided in table 5.7.

The evaluation of application instructions through profiling tools shows that the number of instructions on x86 ISA is 2-10 times more than the ARM ISA for various benchmarks. The foremost reason behind this fact is that we translate an ARM based application to the x86 ISA. During translation, about 50% of the SIMD instructions result in one-to-many mappings. Hence, the total number of instructions for the x86 ISA increases. The highest increase in the number of instructions is witnessed for the FFT benchmark as it contains the highest number and percentage of SIMD instructions.

Another observation from this evaluation is that for all benchmark application, except for speed, the number of instructions executed by the vectorized versions is less than that of normal versions. The optimization flags introduce SIMD instructions and optimize code wherever possible. Therefore, the optimized code application has higher instruction density and lower instruction count. However, the speed benchmark provides variable behavior in this regard. The number of instructions increases for the optimized benchmarks. The reason behind the higher instruction count for speed benchmark is that the application behaves abnormally while the vectorizing flags are imposed on code that is already vectorized.

5.2.6 Computational Power and CPI

The measure of the capability of a device to execute instructions (eg., Millions of Instructions Per Second, MIPS) is also not a deterministic process. The value of MIPS depends on several parameters such as the processor speed and cycles per instruction (CPI). However, modern processors execute multiple instructions in one cycle. Therefore, MIPS is not a simple count of processor speed. Several programs have been written to calculate the MIPS value of a device. The most popular among these is the BogoMIPS. BogoMIPS is a measurement of CPU speed made by the Linux kernel when it boots (E. Kim et al., 2012). The BogoMIPS is also not an accurate indicator of the performance of a system. However, the resultant evaluations are accurate enough to be considered as a scientific baseline (Camarasu-Pop et al., 2016). Another method to measure the MIPS rating of a processor is provided as (Stallings, 2000),

$$MIPS \leq \frac{f}{CPI \times 10^6} \tag{5.5}$$

The MIPS value provided by the BogoMIPS program and Equation 5.5 are approximately equal. We will utilize the BogoMIPS values of the devices as a measure of their MIPS capability in our experiments. The value of BogoMIPS have been listed in table 5.1. The value of CPI depends on the composition of benchmark application, the underlying hardware architecture, and chip design. For our mobile device (Samsung Galaxy S2, ARMv7, Cortex-A9) the average CPI is found to be 1.6 while CPI value for the Intel servers is found to be 1.3 after evaluation of multiple benchmarks (Blem, Menon, Vijayaraghavan, & Sankaralingam, 2015).

5.3 Data Collection and Analysis of SIMD Translator

In this section, we perform data collection for the SIMD translator. The data collection for the SIMD translator is performed to analyze the SIMD translator for its semantic accuracy and translation overhead. First, we perform data collection to identify the set of active instructions produced in the application binary of the application benchmarks. The set of active instructions is utilized to reduce the application profiling overhead. Further, we analyze the semantic accuracy and the overhead of the SIMD translations.

5.3.1 Active Instructions

The NEON ISA comprises of more than a thousand SIMD instructions. However, most of the benchmarks and multimedia applications include only a subset of these instructions. These popular or active instructions can be utilized to lower to the overhead of SIMD translation, re-compilation, and application profiling. The active NEON instructions and their corresponding x86 translations can be pre-fetched to reduce the overhead of dynamic

Table 5.8: Data Collection for SIMD Translator: ARMv-7 Active NEON Instructions

Instruction	**Mathlib**	**Linpack**	**Speed**	**FFT**
vld	1163	705	380	7210
vstr	509	428	275	4391
vmov	670	96	56	638
vmul*	623	180	64	1311
vadd*	461	56	42	1372
vsub	218	6	1	942
vdiv	43	22	13	37
vabs	23	13	0	6
vpush	39	9	5	29
vpop	57	13	9	50
vdup	1	7	3	2
vneg	74	9	2	157
Total SIMD	**3881**	**1544**	**850**	**163145**

translation. We analyzed our benchmarks through static binary analysis (object dumps) for the identification of the set of active instructions. Table 5.8 lists the set of active NEON instructions for the ARMv-7 architecture collected from the GCC compiler with optimization and vectorization flags for the application benchmarks.

The active instruction set comprises of vector arithmetic instructions (VADD, VSUB, VDIV, VMUL, VMLA, VMIN, VMAX, VABS, VNEG), vector load/store instructions (VLD, VSTR, VMOV), and miscellaneous logical and comparative operations for the ARM NEON ISA. Other than the instructions mentioned in the table above, *vmin*, *vmax*, *vand*, and *vsh*∗ also commonly occur in some binaries but were found in very small numbers in the aforementioned benchmarks. Therefore, they are excluded from the table but included in the count of total SIMD instructions.

5.3.2 Semantic Accuracy

The SIMDOM framework is based on a translator that resides on the server for SIMD instruction translations. The SIMD translator exploits a pre-compiled multimedia smartphone (ARM) application such that it can be executed on the x86 server. The main translation challenge of multimedia applications is the SIMD intrinsics which vary from ARM to x86 ISA. However, the ARM NEON to Intel SSE translations can be semantically in-

Table 5.9: NEON Math Library: Semantic Analysis of SIMD Translator

Function	**Range**	**Maximum deviation (original) (sec)**	**Maximum deviation (SIMD translator)**	**Relative difference %**
$sin(x)^2+cos(x)^2-1$	$[0\pi, 1\pi]$	$1.78814e^{-07}$	$1.78814e^{-07}$	0
$sin(x)^2+cos(x)^2-1$	$[-1000\pi, 1000\pi]$	$1.78814e^{-07}$	$1.78814e^{-07}$	0
$x-\log(exp(x))$	$[-60, 60]$	$1.19209e^{-07}$	$1.19209e^{-07}$	0

correct if the translations do not address the challenges of ISA heterogeneity. The task of testing each translation in the SIMD translator that translates nearly 1700 instructions is complex. Therefore, we tested the semantic accuracy of the active NEON instructions listed in the previous subsection with different input parameters.

To test the semantic accuracy of our SIMD translator, we utilized the FFT and Mathlib benchmarks based on SIMD intrinsics. We observed that for the FFTSIMD benchmark, the results of SIMD translator matched completely with the original output in terms of 1-dimensional FFT calculations. We tested the FFT benchmark with variable inputs and achieved consistent results regarding the accuracy of SIMD translations.

The Mathlib benchmark finds the maximum deviation from the mean values for the transcendental functions. The values of transcendental functions can be utilized to evaluate the accuracy of the SIMD translator. Table 5.9 lists the results of the maximum deviation of the transcendental functions for the original Mathlib code and the translated code obtained from the SIMD translator.

For all functions of the Mathlib benchmark, the maximum deviation calculated over a range of values by the original and translated code is same. The aforementioned results show that the SIMD translator achieves 100% semantic accuracy while translating the NEON instructions to SSE instructions.

5.3.3 Overhead of SIMD Translator

In this subsection, we examine the overhead of SIMDOM translator. SIMD translator is the most dynamic runtime element in our offload framework along with application

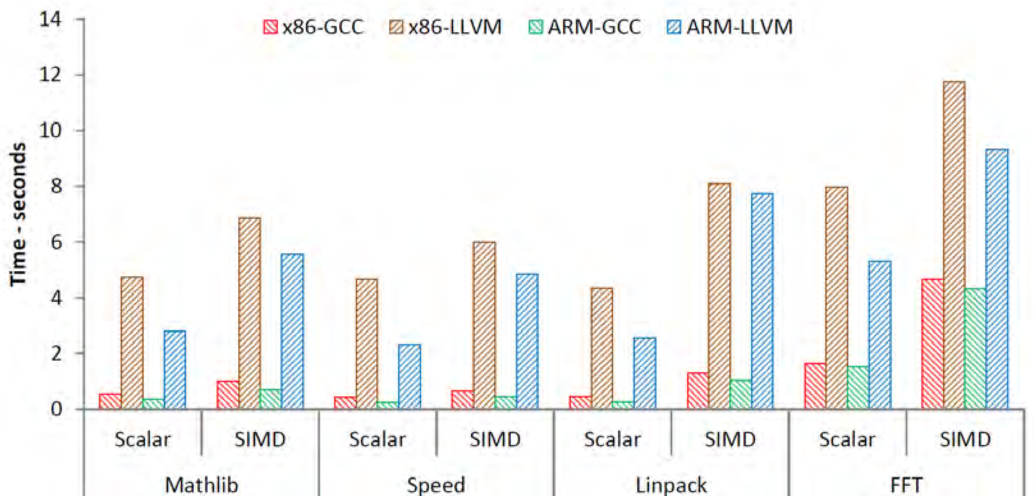


Figure 5.3: SIMD Translator Overhead on Cloudlet: Comparison of Compilers

profiler. The SIMD translator and application profiler have to provide inputs to the offload manager for the offload feasibility evaluation. The SIMD translator and application profiling tasks that can lead to overhead include compilation of application for the ARM and x86 ISA on multiple configurations and translation of SIMD instructions. The overheads of compilation (for ARM ISA) and recompilation along with translation (for x86) for GCC and LLVM compilers on cloudlet are listed in Figure 5.3.

The compilation time of FFTSIMD benchmark for LLVM\Clang compiler has been factored by five to provide equivalent perspective to the rest of the results. There are several repercussions of the preceding result. The most important implication of the result is that the compilation time for the LLVM\Clang compiler is 83.23% and 81.91% higher than the GCC compiler for x86 and ARM ISAs respectively on average. The GCC compiler can be the choice of the SIMDOM framework for lower translation and compilation overhead of both ARM and x86 ISA.

The compilation time for the SIMD benchmarks is higher than basic benchmarks. As the percentage of SIMD instructions is high with auto-vectorization and optimization flags, it is necessary to utilize the extra compilation flags for investigation of application vectorization properties. The overhead of simple benchmarks without optimization flags

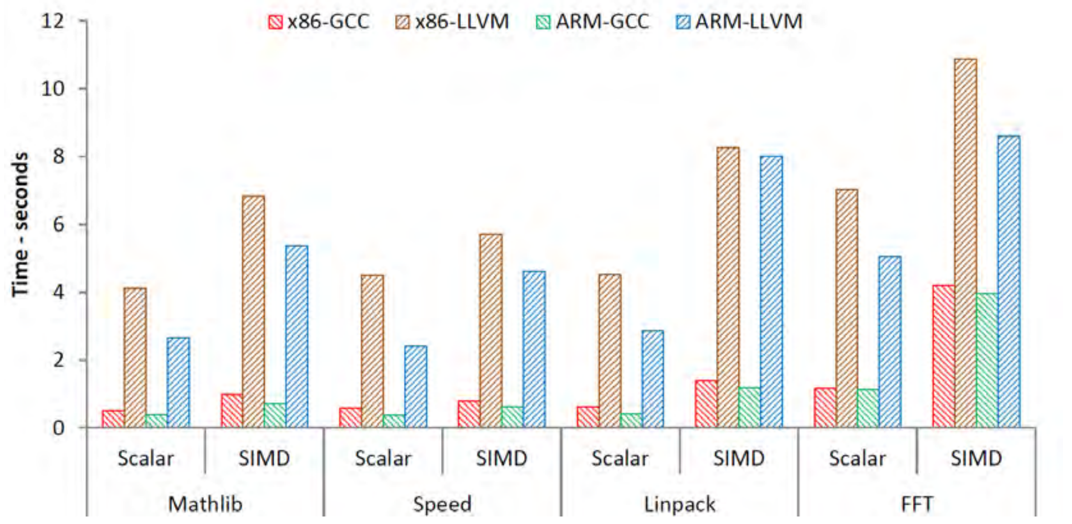


Figure 5.4: SIMD Translator Overhead on Cloud: Comparison of Compilers

can be ignored in the overall profiling overhead as it does not lead to the optimal case of SIMD instruction generation. For the auto-vectorized benchmarks, the overall overhead is the sum of ARM and x86 compilation times and the ARM to x86 SIMD translation time. The application compilation time is highest for FFT as the application is in the form of a library consisting of multiple source files. For Linpack and Speed benchmarks, the compilation overhead for both ARM and x86 platforms is low. Moreover, the compilation time for the x86 ISA is higher than ARM for all benchmarks as it also includes the SIMD intrinsic translation from NEON to SSE ISA. The overheads of compilation (for ARM ISA) and recompilation along with translation (for x86) for GCC and LLVM compilers on cloud server are listed in Figure 5.4.

The application translation overhead is lower on the cloud server for most of the benchmarks due to its higher computational capability. The implications derived from the cloudlet analysis can be noted in the cloud too. We also investigated the variance in compilation overhead for the Linpack benchmark based on variable inputs. We found that the input matrix size does not affect the compilation overhead of the Linpack benchmark significantly.

5.4 Data Collection and Analysis of Application Profiler

In this section, we perform the data collection and analysis of the application profiler. The data for the application profiler is collected through static code analysis of the multimedia benchmarks against multiple ARM and x86 compilers (GCC and LLVM\Clang). Further, the evaluation of the overhead of application profiling is performed. We also wanted to include the ARMCC compiler in our analysis. However, its commercially paid license did not allow for free research purpose utilization. The data collected through static code analysis and profiling overhead analysis helps the application profiler determine the optimal parameters for application offloading for the SIMDOM framework.

5.4.1 Static Code Analysis

Static code analysis of the candidate offload application helps the application profiler to collect data for optimal generation of SIMD instructions. The parameters considered by the application profiler can be the selection of compiler, compilation flags, and target architectures. The application profiler investigates these parameters to find an ideal combination of the compiler and corresponding compilation flags that lead to the highest percentage of SIMD instructions in the program binary. Our application profiler provides three possibilities of application partition to the offload manager: **(a)** zero application partition if offloading does not lead to energy efficiency, **(b)** full application partition where complete application is offloaded, and **(c)** partial application partition where only SIMD instructions are offloaded. The static code analysis provides the compiler parameters such that the program binary has the highest percentage of SIMD instructions for the two latter cases of application partition. The static code analysis results in the identification of partial application partition that only comprises of SIMD instructions. The static code analysis is based on the source dumps of the program binaries. Figure 5.5 and 5.6 provide the comparison of ARM and x86 compilers for the production of SIMD instructions for

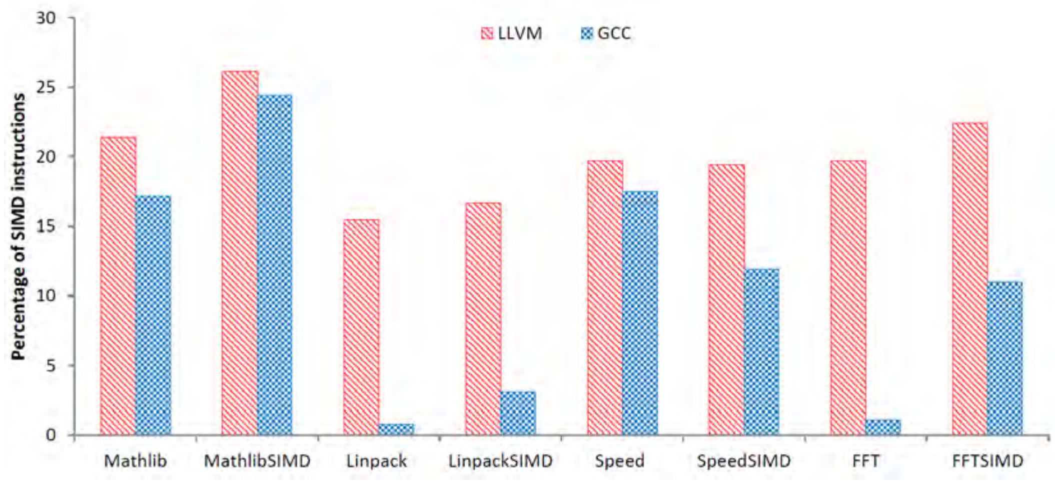


Figure 5.5: Analysis of ARM GCC and LLVM\Clang Compilers for Application Benchmarks

multimedia benchmarks.

The static code analysis provides comparative analysis of the vectorizing capabilities of ARM and x86 compilers. There are multiple implications of the aforementioned results. Based on these implications, recommendations are forwarded to the SIMDOM framework regarding parameters for efficient remote and local execution of multimedia applications.

Firstly, the x86 compilers fare far better than the ARM compilers in the generation of SIMD instructions. The x86 compilers produce an average of 26.80% SIMD instructions

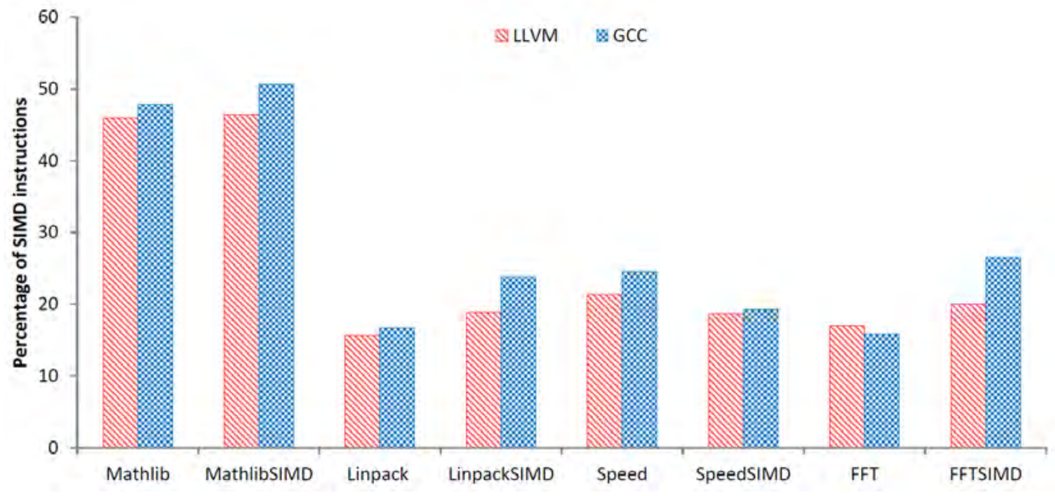


Figure 5.6: Analysis of x86 GCC and LLVM\Clang Compilers for Application Benchmarks

as compared to 15.49% for ARM compilers for all benchmarks. This observation points to the fact that the support for vector instruction generation in ARM compilers is not as efficient as the x86 compilers. Several recent studies carried out on vectorizing compilers support this finding (Fu, Wu, & Hsu, 2015; Maleki et al., 2011).

Secondly, the major deficiency in the case of ARM compilers comes from the GCC compiler. LLVM produces more efficient vectorizing code than the GCC compiler for the ARM architecture. GCC produces 10.86% of SIMD instructions as compared to 20.11% for the LLVM compiler on average for all benchmarks on ARM architectures. Hence, the LLVM compiler is 45.99% efficient in the production of SIMD instructions than the GCC compiler for ARM ISA. On the contrary, GCC performs marginally better than LLVM compiler for x86 architectures. GCC produces 28.12% SIMD instructions for x86 architecture as compared to 25.48% SIMD instruction produced by the LLVM compiler on average for all benchmarks. The GCC compiler is 9.38% efficient than the LLVM compiler for the production of SIMD instruction for x86 ISA.

Thirdly, FFT, Linpack, and Mathlib benchmarks produce better SIMD instruction count on optimization and vectorizing flags. However, the Speed benchmark produces lower SIMD instruction count with vectorizing flags for both x86 and ARM ISAs. Forcing a compiler to produce vectorized assembly code for a program that already contains vector intrinsics can lead to such unusual performance. However, there are performance gains in case of the optimized versions due to overall optimization of the program binary.

Fourthly, the Mathlib benchmark provides the highest percentage of SIMD instructions for both ARM and x86 binaries. The Mathlib is comprehensively programmed with NEON intrinsics such that the transcendental function calculations are exclusively performed by vector instructions. On the contrary, the remaining benchmark applications insert NEON intrinsics wherever possible. Moreover, the Mathlib benchmark produces the least SIMD instruction increase on optimization flags due to native vectorized code that does not require optimization flags for vector generation.

The static analysis reveals that for local execution, the supporting compiler should be LLVM\Clang for the ARM ISA for a higher percentage of SIMD instructions. In case the code is offloaded to the x86 cloud or cloudlet server, GCC compilers should be used. However, this use of LLVM\Clang is contradictory to our previous findings of the SIMD translator overhead. The SIMD translator overhead revealed that the LLVM\Clang has approximately 80% higher compilation overhead than the GCC for the x86 ISA. In case lower translation overhead is desired, the GCC compiler should be utilized for both x86 and ARM ISAs in the SIMDOM framework. On the contrary, for long term optimizations where the translation overhead can be ignored, LLVM compiler for the ARM ISA and GCC compiler for the x86 ISA should be utilized to produce highly optimized and vectorized code.

5.4.2 Overhead of Application Profiler

In this subsection, we examine the overhead of the application profiler. The overhead of application profiler occurs in two main tasks. Firstly, the overhead occurs during compilation and translation of the application. The overhead of application profiling for different compilers and ISAs has been listed in Section 5.3.3. Secondly, the overhead occurs while profiling the application for the percentage of SIMD instructions as executed in Section 5.4.1. In this subsection, we detail the overhead of application profiling generated from examination of application binaries for optimal offload parameters. The application profiler provides inputs to the offload manager for the offload feasibility evaluation in the form of optimal application partition with the highest percentage of SIMD instructions and the corresponding profiling overhead. The overhead of application profiler for cloudlet server is illustrated in Figure 5.7.

The high percentage of SIMD instructions is achieved with auto-vectorization flags.

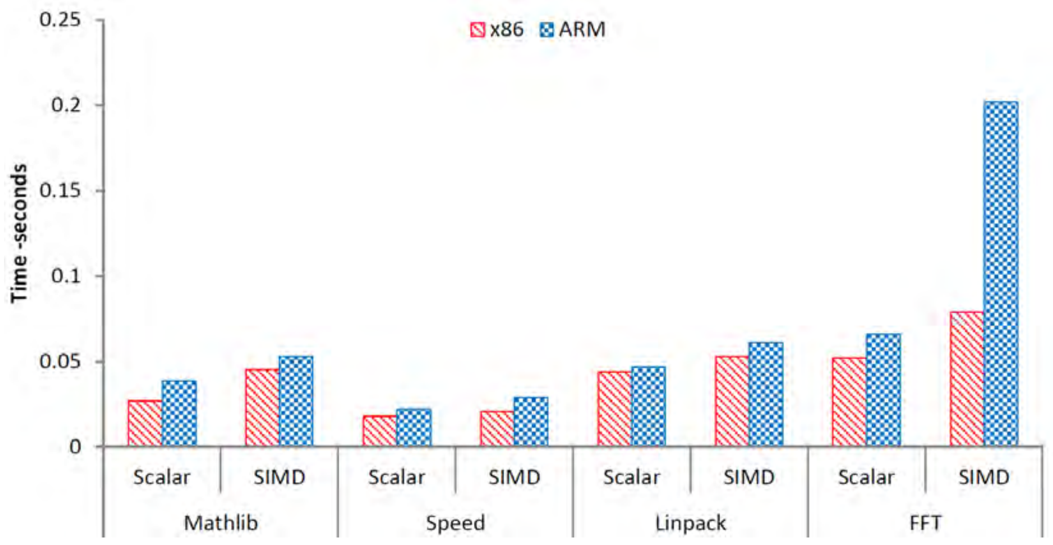


Figure 5.7: Application Profiler Overhead on Cloudlet

Therefore, the overhead of profiling the SIMD benchmark is high compared to basic counterparts. The cloud server provides relatively lower overhead for the application profiler depicted in Figure 5.8.

The application profiling overhead can be used to infer the number of instructions executed by the server while profiling the application. The equation is provided as (Stallings, 2000),

$$I = \frac{T \times MIPS \times 10^6}{CPI} \tag{5.6}$$

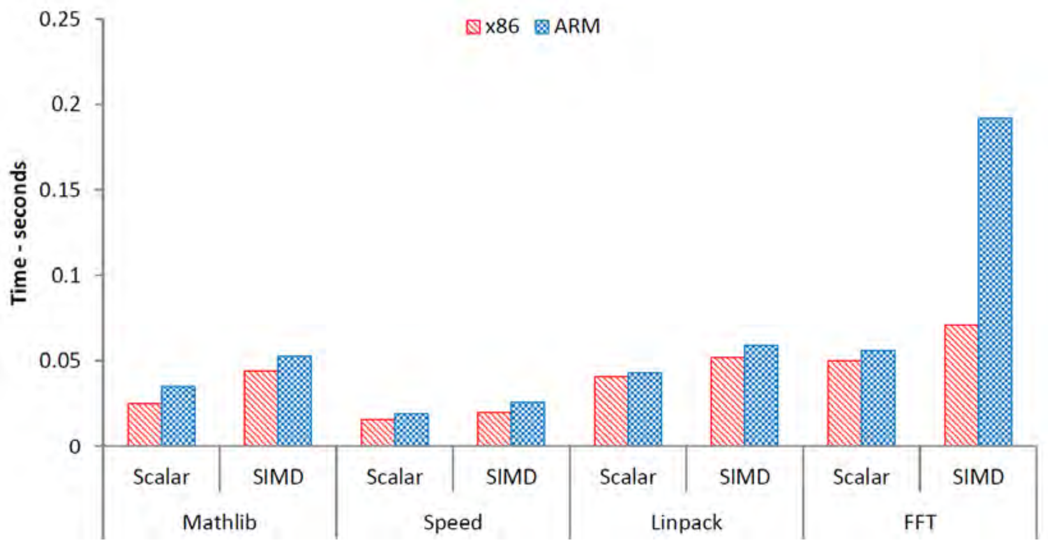


Figure 5.8: Application Profiler Overhead on Cloud

As an example, consider the FFT benchmark for local server,

$$I = 3.26sec \times 4654.87 \times 10^6 = 11672981692$$

We will utilize the aforementioned formulation to validate our mathematical model. The network and energy profiler modules do not incur overhead in terms of time on the SIMDOM framework as they execute in parallel to collect the required data on the mobile device.

5.5 Model Bounds

In this section, we derive the mathematical bounds for some of the variables in the system model.

5.5.1 Bounds for Application Partitioning

The optimal application partition is often hard to find in a MCC offload scenario. We calculate the minimum ratio $\frac{I_r}{I}$ such that offload leads to energy efficiency for the mobile device. Based on equation 4.9,

$$E_{local} > E_{offload}$$

$$E_{local} > E_{send(I_r)} + E_{exec(I_l)} + E_{wait(I_r \overset{translate}{\rightarrow} I_r)} + E_{wait(exec(I_r))} + E_{rec(res)}$$

As we assume in SIMDOM framework that the application is available on the server for profiling, the energy spent on sending and receiving the data from the server is minimal. Therefore, $E_{send(I_r)}$ and $E_{rec(res)}$ can be ignored in the calculation of bounds for application partitioning. Moreover, for our simplistic case where most of the code is executed on the server, $E_{exec(I_l)}$ can also be ignored. Therefore,

$$E_{local} > E_{wait(I_r \overset{translate}{\rightarrow} I_r)} + E_{wait(exec(I_r))}$$

We further assume that the to re-compile and translate the application in the SIMDOM framework, 10% of the total instructions are further required. That is, $(I_r \overset{translate}{\rightarrow} I_r)$ adds 10% more instructions to the total I_r instructions. Our assumption is supported by a simple calculation. For the FFT benchmark, the number of instructions executed by the SIMDOM framework to re-compile and translate the application (15160911590 instructions) are approximately 10% of the total instructions (162487172630 instructions). Assuming the code is offloaded to the cloudlet,

$$\frac{p_c \times I \times CPI_m}{s_m} > \frac{p_i \times I_r \times 0.1 \times CPI_s}{s_s} + \frac{p_i \times I_r \times CPI_s}{s_s}$$

$$\frac{p_c \times I \times CPI_m}{s_m} > \frac{p_i \times CPI_s \times I_r \times 1.1}{s_s}$$

$$\frac{s_s}{s_m} > \frac{p_i \times CPI_s}{p_c \times CPI_m} \times \frac{1.1 \times I_r}{I}$$

Replacing the variables with the data collected in previous sections,

$$\frac{I_r}{I} < 94.56$$

The aforementioned derivation concludes that the ratio of offloaded instruction to total instruction should be less than 94.56 in order to achieve energy efficiency during code offload. For example, if the total code base of the application is 100 instructions, more than 6 instructions should be offloaded to the server while ignoring the overhead of network delay and communication. For the SIMDOM framework, the aforementioned derivation puts bound on I_r. If the percentage of SIMD instructions in an application is less than 6%, then it is not feasible to offload only SIMD instructions to the server. If we consider the remote server instead of the local server, the ratio $\frac{s_s}{s_m}$ increases. Hence, for remote server,

$$\frac{I_r}{I} < 97.28$$

Moreover, the overhead of wireless communications can not be ignored in case of remote server.

5.5.2 Bounds for Server Speed

The ratio of $\frac{s_s}{s_m}$ is often debated for the feasibility of MCC offload. The higher the difference between speed (or computational power expressed in MIPS) of the mobile and server devices, the higher are the resultant gains in terms of energy and execution time. We again take the example of FFT benchmark for the local server. Based on the assumption debated in previous case, we derive from equation 4.9,

$$\frac{p_c \times I \times CPI_m}{s_m} > \frac{p_i \times I_r \times CPI_s}{s_s} + \frac{p_i \times I_r \times 0.1 \times CPI_s}{s_s}$$

$$\frac{s_s}{s_m} > \frac{p_i \times CPI_s \times I_r \times 1.1}{p_c \times CPI_m \times I}$$

$$\frac{s_s}{s_m} > 0.27$$

The aforementioned derivations points to the fact that based on our assumption of negligible network overhead, the server power can be approximately 4 times lesser than the mobile device and still achieve energy efficiency while offloading. There are two reasons behind the lower feasibility bound of $\frac{s_s}{s_m}$. Firstly, the low ratio of $\frac{p_i}{p_c}$ allows for the server speed to be so low. We found through the data collection in Section 5.2 that the $\frac{p_i}{p_c}$ ratio is approximately 0.05. While the mobile device offloads tasks to the cloud, its energy in wait state is very low as compared to the state where it performs local computations. Therefore, the offload decision remains feasible even if the server speed is lower than the mobile device. Secondly, the ratio of $\frac{I_r}{I}$ is also low that influences lower value for the ratio of $\frac{s_s}{s_m}$.

5.6 Case Studies

In this section, we present the case studies of MCC offload enabling techniques, i.e., system virtualization and application virtualization. The objective of these case studies is to analyze the communicational and computational overheads of the MCC offload enabling techniques.

5.6.1 Case Study: System Virtualization

System virtualization is largely utilized in offloading of tasks from the mobile device to the cloud server. In this case study, we take six scenarios of system virtualization based MCC offloading and derive their overhead; **(a)** VM based cloudlet, **(b)** VM based cloud, **(c)** uncompressed VM overlay based cloudlet, **(d)** uncompressed VM overlay based cloud, **(e)** compressed VM overlay based cloudlet, and **(f)** compressed VM overlay based cloud as described in the concept paper of VM based MCC offloading (Satyanarayanan et al., 2009). The basic $E_{offload}$ equation for the system virtualization based MCC offloading is given as,

$$E_{offload} = E_{send(VM)} + E_{wait(exec(VM))} + E_{rec(VM)}$$

Table 5.10: Energy Consumption of System Virtualization Based MCC Offloading Techniques

Scenario	Local server (cloudlet)			Remote server (cloud)		
	E_{send}	E_{rec}	$E_{offload}$	E_{send}	E_{rec}	$E_{offload}$
System VM	705.16J	358.51J	1063.67J	888.67J	885.84J	1774.51J
Uncompressed VM overlay	36.32J	19.55J	55.87J	45.77J	45.63J	91.40J
Compressed VM overlay	13.69J	7.37J	21.06J	17.25J	17.20J	34.45J

We ignore the $E_{wait(exec(VM))}$ and derive only the energy spent on receiving and sending the VM in all the cases discussed below. We consider the initial VM sizes and ignore the delta values of the VM sent between cloud and mobile devices during the process of VM migration. For the VM overlay cases, the basic $E_{offload}$ equation is,

$$E_{offload} = E_{send(VM)} + E_{compress(VM)} + E_{wait(uncompress(VM)+exec(VM))} + E_{rec(VM)} + E_{uncompress(VM)} \quad (5.7)$$

We do not know the exact parameters for compression of the VM such as, the compression algorithm and the input size. Therefore, we ignore the energy consumed in compression of the VM at multiple instances. We are only left to contemplate E_{send} and E_{rec}. We utilize the system variables derived for our model for the local server (cloudlet) and remote server (cloud) in Section 5.2. For the VM scenarios, the most common VM size for a Linux based 8GByte. For the VM overlay scenarios, we take the VM size values from the pilot study (Satyanarayanan et al., 2009). The compressed and uncompressed VM overlay migration sizes have been taken as the average of six applications listed in the pilot study. The results of energy consumption of VM based MCC offloading techniques are depicted in Table 5.10.

Due to the proximity of cloudlet server, the energy spent on cloudlet offloading is less than the remote server offloading in every case. The aforementioned energy derivations for system virtualization based MCC offloading are quite high as compared to local execution. Although we ignored several energy consuming tasks such as, compression and decompression and evaluated $E_{offload}$ only based on the energy of communicational tasks. The lowest energy consumption is achieved by compressed VM overlay based offloading. However, the compressed VM overlay scenario is evaluated on the base of only $E_{send(VM)}$ and $E_{rec(VM)}$ values while ignoring the energy spent on compressing and compressing the VM at both mobile client and server. The aforementioned results reveal that the system virtualization based MCC offloading techniques lead to considerable overhead in terms of network energy consumption. Therefore, even if the VM overlay and compression techniques are applied, it is hard to find a scenario where system virtualization based MCC offloading can lead to energy efficiency for the mobile devices.

5.6.2 Case Study: Application Virtualization

Application virtualization is the most commonly utilized technique for enabling of MCC offloading. However, several overheads exist towards true implementation of and application virtualization based MCC offloading framework. Such overheads are, but not limited to, determination of optimal application partition, memory synchronization between the mobile device and cloud server, and byte code interpretation. We list the aggregate of all such overheads as $E_{opt(VM)}$. The equation of offload energy in case of application based virtualization offloading can be described as,

$$E_{offload} = E_{opt(VM)} + E_{send(VM)} + E_{wait(exec(VM))} + E_{rec(VM)}$$

Where $E_{opt(VM)}$ can be afforded by the mobile device or the cloud server based on the specific implementation of the offloading framework. As there are no standard specifications of $E_{opt(VM)}$ in MCC offloading frameworks, we ignore the value and determine the energy consumption of application virtualization frameworks based offloading based on the network operations. We obtain the mean values of VM send (1130KByte) and receive

Table 5.11: Energy Consumption of Application Virtualization Based MCC Offloading Techniques

Scenario	Local server (cloudlet)			Remote server (cloud)		
	E_{send}	E_{rec}	$E_{offload}$	E_{send}	E_{rec}	$E_{offload}$
Application virtualization	0.097J	0.013J	0.11J	0.122J	0.036J	0.158J

(294.3KByte) from the COMET framework as it is a standard reference for application virtualization based offloading (Gordon, Jamshidi, Mahlke, Mao, & Chen, 2012). Table 5.11 lists the energy consumption of application virtualization based MCC offloading technique based on our case study.

The $E_{send(VM)}$ and $E_{rec(VM)}$ cost of application virtualization based solutions is comparable to, but not less than that of native code offloading based solutions such as SIMDOM which will be evaluated in the next chapter. However, we ignored several factors of computational overhead such as, $E_{opt(VM)}$ and $E_{wait(exec(VM))}$ as their are no standard specifications in existing MCC frameworks. Incorporating detailed parameters in the case study will most likely lead to an increase in the value of $E_{offload}$.

5.7 Conclusion

In this chapter, evaluation process for the SIMDOM framework was described in detail. The evaluation process consisted of the experimental setup, devices, benchmarks, and data. The data collection methodology and its statistical analysis for the energy, network, and application modules were defined. Based on the collected data, the SIMD translator and application profiler modules were evaluated. The feasibility bounds for the SIMDOM framework were also determined.

We found that the application benchmarks produce a set of active instructions that occur frequently in the application binary. The overhead of the application profiler can be reduced while profiling only for the set of active instructions. The analysis of the SIMD translator revealed 100% semantic accuracy in translation of instructions from ARM ISA to x86 ISA. The LLVM\Clang compiler produced approximately 46% higher number of SIMD instructions for the ARM ISA than the GCC compiler. However, the LLVM\Clang compiler also led to 81%-83% higher compilation overhead compared to the GCC compiler. Therefore, for short-term optimization, GCC compiler is preferred for the application execution on both mobile and cloud devices. The analysis of application partition bounds shows that at least 10% of application instructions should be offloaded to a cloud server in order to gain energy efficiency. The analysis of server speed reveals that the offload server can be four times less powerful and still provide energy efficiency to the mobile device while ignoring the energy consumed by the network components.

CHAPTER 6: RESULTS AND DISCUSSION

This chapter presents the empirical evaluation of the SIMDOM framework and provides the main findings of our research. The SIMDOM framework enables execution of SIMD based applications in heterogeneous MCC architectures. An application offloaded from a mobile device is recompiled, translated, and executed on cloud server while the mobile device waits in a low-power state. We implement a prototype of the SIMDOM framework on a cloudlet and a cloud server. The empirical evaluation and comparison of the SIMDOM framework is performed on five scenarios of application execution, i.e., **(a)** local execution (LE) on the mobile device, **(b)** SIMDOM execution on local server (LS), **(c)** SIMDOM execution on remote server (RS), **(d)** Qemu execution on local server (QLS), and **(e)** Qemu execution on cloud server (QRS). The SIMDOM scenarios represent our framework prototype of pre-compiled native code offloading. The Qemu scenarios present the case of compiled native code offloading and translation in MCC. The cloud and cloudlet servers are utilized to deliberate on the proximity of resource and network overhead. Moreover, scalar and SIMD versions of the benchmarks are utilized to analyze the efficiency of SIMD translation in SIMDOM and Qemu frameworks.

Our findings are divided into three sections. Firstly, validation of the system model presented in Section 4.3 is performed with the empirical results in Section 6.1. In Section 6.2, empirical analysis of the SIMDOM framework is performed for parameters such as energy, execution time, and performance gain. The comparative analysis of SIMDOM framework with the existing state-of-the-art MCC translation and offloading frameworks is presented in Section 6.3. The section also contains the analysis of the impact of device sleep time, application partition, and application computational size on the energy consumption. The impact of application partition and computational size on the execution time is also analyzed. The concluding remarks for the chapter are provided in Section 6.4.

6.1 Framework Validation

In this section, we validate the system model of the SIMDOM framework. We answer the validation question: Does the developed model accurately represent the operational SIMDOM framework? The validation of the developed system model is performed by comparing its results with the empirical results. We validate our system model based on two variables, i.e., energy and execution time as listed in Section 4.3.

6.1.1 Energy

We validate the system model of the SIMDOM framework for the energy consumption. We take FFT benchmark as an example. In the local MCC-disabled execution scenario,

$$E_{local} = \frac{0.571W \times 346645907377 \times 1.6}{1194.54 \times 10^6/s} = 268.05J$$

For the cloudlet offload scenario, we consider the case when the complete program is offloaded to the cloud. On the base of the FFT benchmark code size (I_r), smartphone power rating for sending data (p_u), and network up-link throughput (U_r) we find the value of $E_{send(I_r)}$ as 0.51J. We assume for our re-compilation framework that for the total number of instructions executed in translations and re-compilation can be calculated from the total time overhead of the SIMDOM framework. In case of cloud server, the parameters of D_r, U_r, s_s, and T_{ovr} change in the system model. For cloudlet Qemu scenario, the only change from the SIMDOM cloudlet scenario is the value of s_s. For the cloudlet, the value of s_s is 4654.87 MIPS while for the emulated device the s_m value is 471.61 MIPS. We assume that the overhead of binary translation is represented in the complete time of application execution. For the remote Qemu server, the value of s_s is 591.76. Figure 6.1 presents the comparison of empirical and mathematical energy calculation for the FFT benchmark in various scenarios of execution.

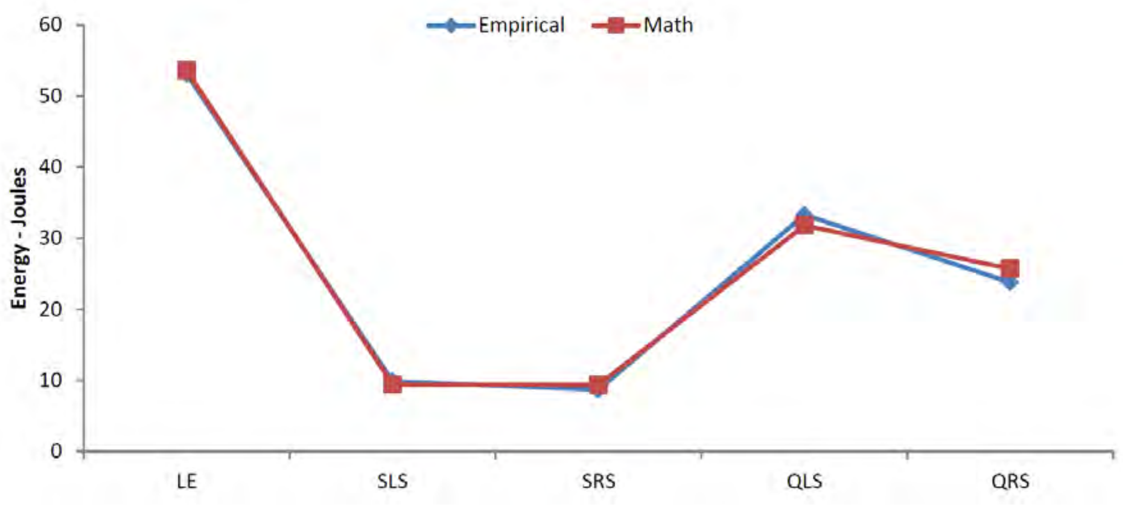


Figure 6.1: Mathematical Validation of SIMDOM Framework for FFT Benchmark: Energy Consumption

The x-axis represents the execution scenarios while the y-axis represents the energy consumption in Joules. The energy consumed by MCC-disabled execution is the highest among the evaluated scenarios and normalized for the graph by dividing the actual value by five. The mathematical and empirical energy calculations for FFT benchmark differ slightly in the aforementioned scenarios. The absolute difference is 0.94%, 4.56%, 7.59%, 4.45%, and 7.98% respectively between the mathematical and empirical results for the five scenarios. The least difference is in the case of local energy consumption as it does not involve the dynamic and fluctuating nature of wireless medium. All other scenarios involve Wi-Fi communications which can vary from time to time based on multiple parameters. The highest difference in mathematical and empirical results is in the case of Qemu calculations. We did not model the memory management operations of the server that executes Qemu and its corresponding tasks. Therefore, the imperfections in the Qemu offloading model leads to higher error between mathematical and empirical data. Moreover, as Qemu is an emulated system, the computational power of the Qemu also fluctuates leading to higher difference between mathematical and empirical results. We utilized Pearson correlation coefficient to find the linear correlation between the data sets

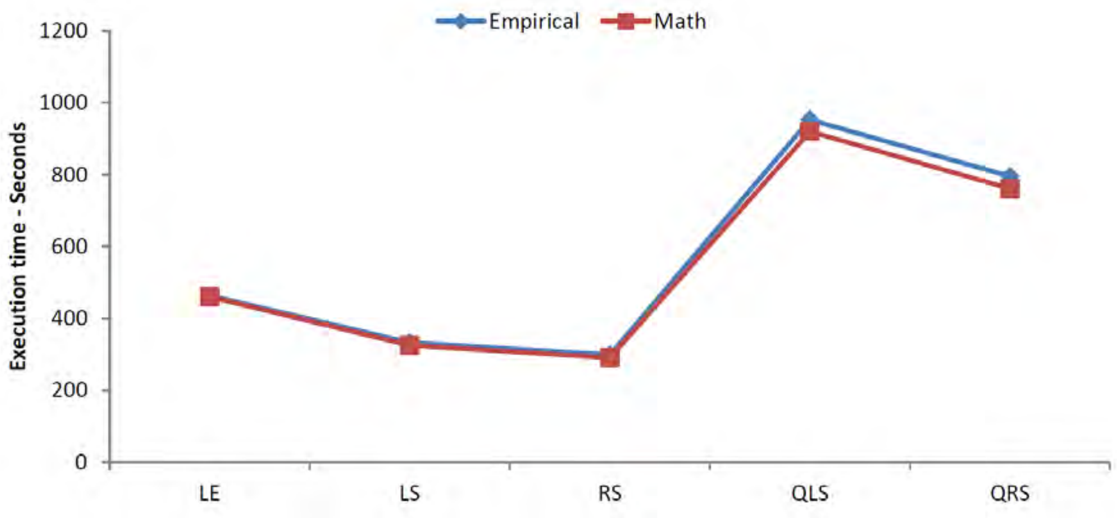


Figure 6.2: Mathematical Validation of SIMDOM Framework for FFT Benchmark: Execution Time

of mathematical and empirical results. The value of Pearson correlation coefficient lies between 0 and 1. The closer the value of Pearson correlation coefficient to 1, the higher is a correlation between the two datasets. The Pearson correlation for our mathematical and empirical data sets is 0.993462, indicating a strong correlation.

6.1.2 Execution Time

We perform the validation for the execution time in this subsection. Similar to previous case, we chose the FFT benchmark to validate the empirical results. Figure 6.2 presents the validation of execution time for the SIMDOM framework.

The x-axis represents the execution scenarios while the y-axis represents the execution time in seconds. The aforesaid result depict that the mathematical and empirical data differ slightly for the execution time. The absolute difference between empirical and mathematical results is 0.63%, 2.28%, 2.73%, 3.42%, and 4.29% for the five execution scenarios. The mathematical model produces lower values for execution time as compared to the empirical results. The mathematical model is simplistic as it ignores the complexity of most of the parameter of the Wi-Fi network. Moreover, the mathematical model does not represent the internal memory management details of Qemu. Therefore, the difference in mathematical and empirical values for the Qemu scenario is high. Furthermore, the absolute difference between mathematical and empirical values for the execution time is lesser than those for the energy. The reason is that the energy model includes the energy profiling framework in the form of PowerTutor which can lead to further inaccuracy in empirical results. The 0.999762 Pearson correlation value for the data sets indicates a strong correlation further validating our model and empirical results.

6.2 Comparison of SIMDOM for Application Benchmarks

The SIMDOM framework provides energy and time efficiency for the offloaded mobile application. In this section, we compare the SIMDOM framework on multiple application benchmarks to verify its performance. The analysis is presented for energy consumption, execution time, and MFLOPS performance. The comparison is performed on both versions of the four application benchmarks. The performance analysis of the SIMDOM benchmark is performed on the prototype implemented on cloud server.

6.2.1 Energy

We compare the energy consumption of the SIMDOM framework in this subsection for the application benchmarks. Figure 6.3 presents the energy consumption of SIMDOM framework for the scalar and SIMD versions of the benchmarks.

The x-axis represents the application benchmarks while the y-axis present the energy consumption in Joules. The result shows that the energy consumed by the FFT benchmark is highest among the application benchmarks. The FFT benchmark has higher computational load compared to the other benchmarks leading to higher energy consumption. The rest of application benchmarks have similar energy consumption pattern due to approximately same amount of computational workload. The energy consumption of the SIMD version of benchmarks is 17.44% lower than the scalar benchmarks. The lower energy

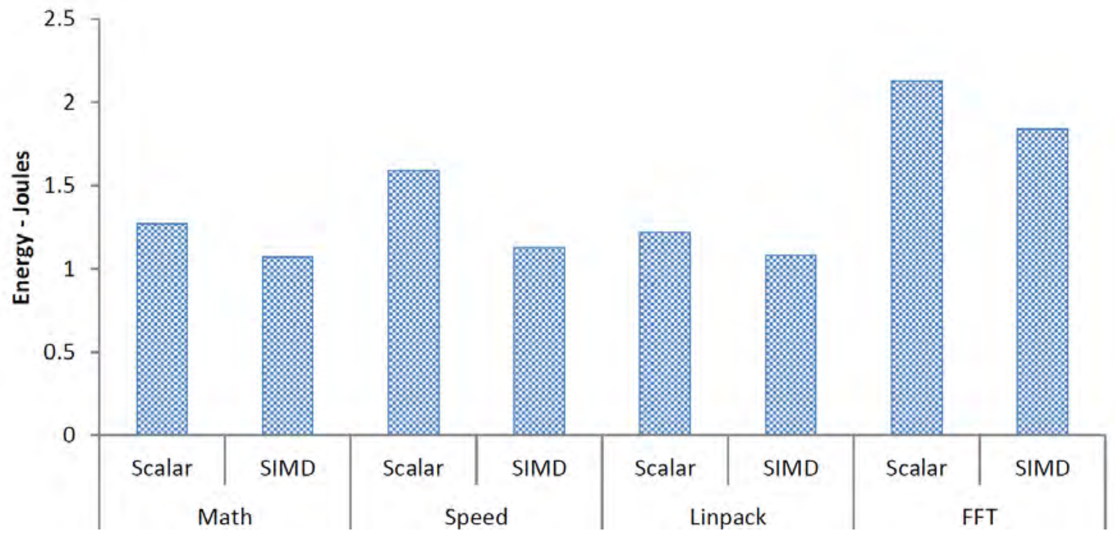


Figure 6.3: Comparison of SIMDOM Framework for Application Benchmarks: Energy Consumption

consumption by the SIMD benchmarks verifies that the SIMD instructions have been efficiently translated by the SIMDOM framework. The average energy consumption by the application benchmarks is 1.41 while the standard deviation is 0.36. The confidence interval for energy consumption of the application benchmarks is ± 0.25.

6.2.2 Execution Time

Figure 6.4 presents the comparison for execution time of application benchmarks.

The x-axis represents the application benchmarks selected for evaluation, while the y-axis represents the execution time of the benchmarks. The execution time of FFT benchmark is scaled by a factor of five to provide a reasonable comparison for all of the evaluated application benchmarks. The execution time of the FFT benchmark is highest among the application benchmarks due to a higher number of instructions and computational workload. The Mathlib benchmark provides the lowest execution time as it has the lowest number of instructions among the evaluated application benchmarks. The execution time of the SIMD benchmarks is 34.58% lower than the scalar benchmarks. The SIMD benchmarks achieve lower execution time as they are efficiently translated by

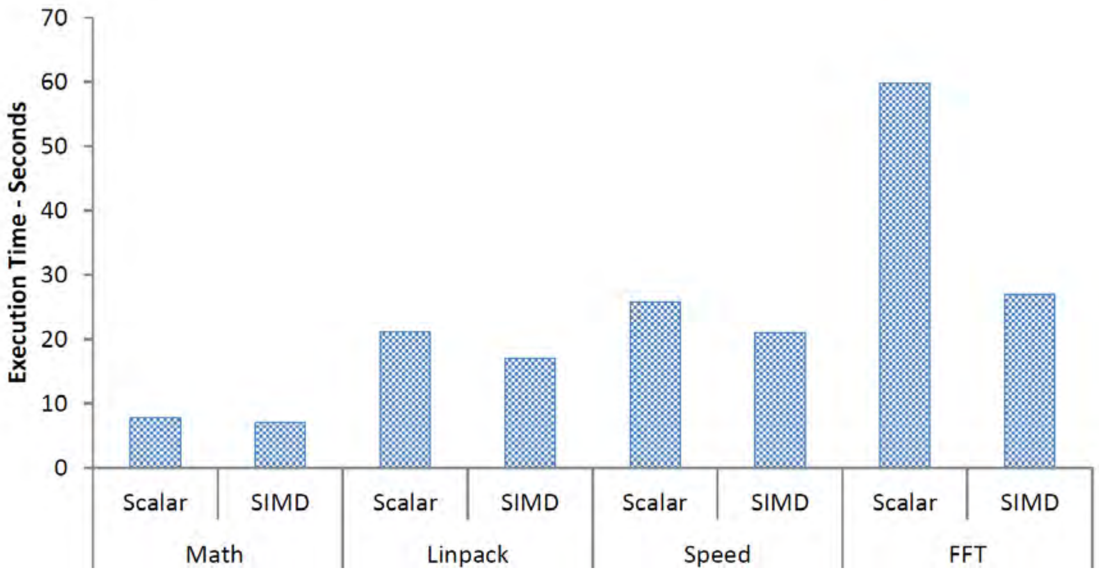


Figure 6.4: Comparison of SIMDOM Framework for Application Benchmarks: Execution Time

vector-to-vector translations of the SIMDOM framework. The mean execution time of the evaluated application benchmarks is 53.26 while the population standard deviation is 93.21. The higher standard deviation occurs due to the higher execution time of FFT benchmarks. The standard deviation of the dataset is reduced to 7.44 while excluding the FFT benchmark. The confidence interval of the execution time for all the application benchmarks is ±64.59.

6.2.3 Performance Gain

Figure 6.5 presents the performance gain of SIMDOM framework in terms of MFLOPS.

The MFLOPS performance is evaluated on the Linpack benchmark which provides variable matrix inputs. The matrix inputs vary from 200 x 200 to 1200 x 1200. The average performance for all inputs of the Linpack benchmarks is 764.08 MFLOPS while the standard deviation is 358.63. The basic and SIMD version of benchmarks lead to contrasting performance, hence, resulting in higher standard deviation. The confidence interval is ±202.59 for the evaluated benchmarks. The SIMD versions of the benchmarks produce 63.74% efficient performance than the basic versions.

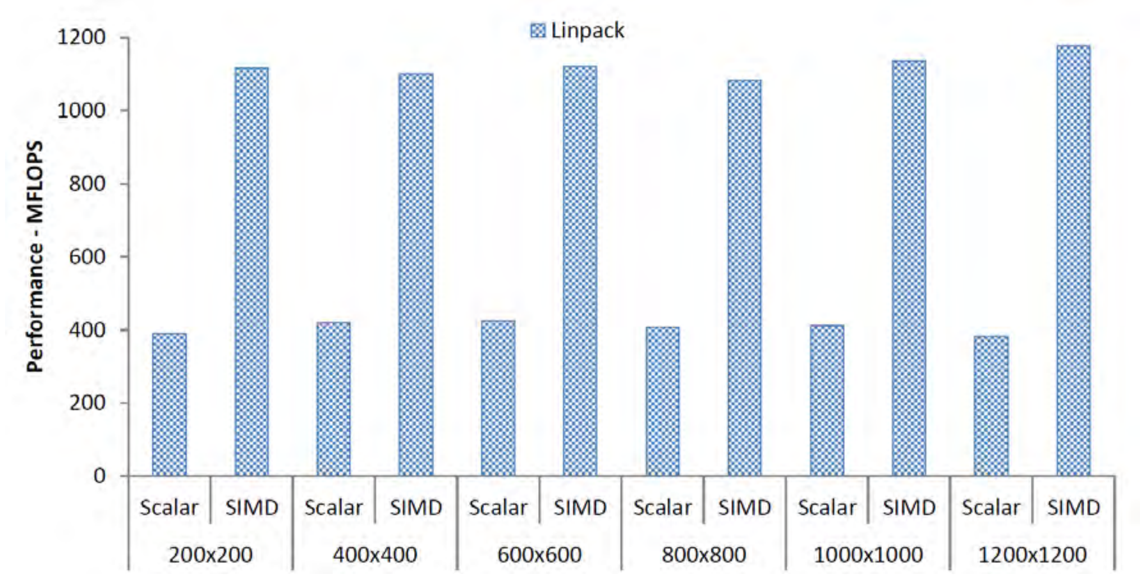


Figure 6.5: Comparison of SIMDOM Framework for Application Benchmarks: MFLOPS Performance

In the above subsections, we analyzed the performance of SIMDOM framework on various application benchmarks. The performance of the SIMDOM framework in cross-platform translation and offloading of SIMD instructions can be judged from the difference in SIMD and scalar version of the benchmarks. The SIMD benchmarks result in 17.44%, 34.58%, and 63.74% energy, time, and MFLOPS performance efficiency for the SIMDOM framework.

6.3 Comparison of SIMDOM with Qemu Offloading Framework

In this section we present the operational and experimental results of the SIMDOM framework while comparing it with the state-of-the-art MCC code offloading frameworks. We quantify and investigate each of the illustrated result from the following five aspects.

- Is code offloading feasible in terms of energy and time for the evaluated benchmarks?
- How much is pre-compiled code offloading based SIMDOM framework efficient than compiled code offloading frameworks?

- Under what conditions are the local server scenarios efficient than the remote server scenarios?
- What effect has the benchmark size on the efficiency of the MCC offloading frameworks?
- How efficiently does the SIMDOM framework handle translation of SIMD instructions as compared to the Qemu framework?

The SIMDOM framework comparison and evaluation is performed on five scenarios of application execution, namely, LE, LS, RS, QLS, and QRS as detailed at the start of the chapter. The scenario of local execution provides the base case where the application executes without the support of MCC. The SIMDOM scenarios represent our framework prototype of pre-compiled native code offloading. The Qemu scenarios present the case of compiled native code offloading in MCC. The compiled code experiments are performed with Qemu and provide a comparative analysis of the SIMDOM framework. We chose Qemu for comparison with the SIMDOM framework as it provides the existing state-of-the-art implementation of native code translation in MCC. Moreover, the cloud and cloudlet servers are considered to deliberate on the proximity of resource and network overhead. Furthermore, scalar and SIMD versions of the benchmarks are utilized to analyze the efficiency of SIMD translation in SIMDOM and Qemu frameworks. While other benchmarks execute for all inputs, the Linpack benchmark is based on the input of a 600×600 matrix.

6.3.1 Energy

We measured the energy consumption of application benchmarks in various operational scenarios. We first evaluated the SIMDOM framework for the base case where the mobile device goes to idle state immediately after offloading the computations and LCD is

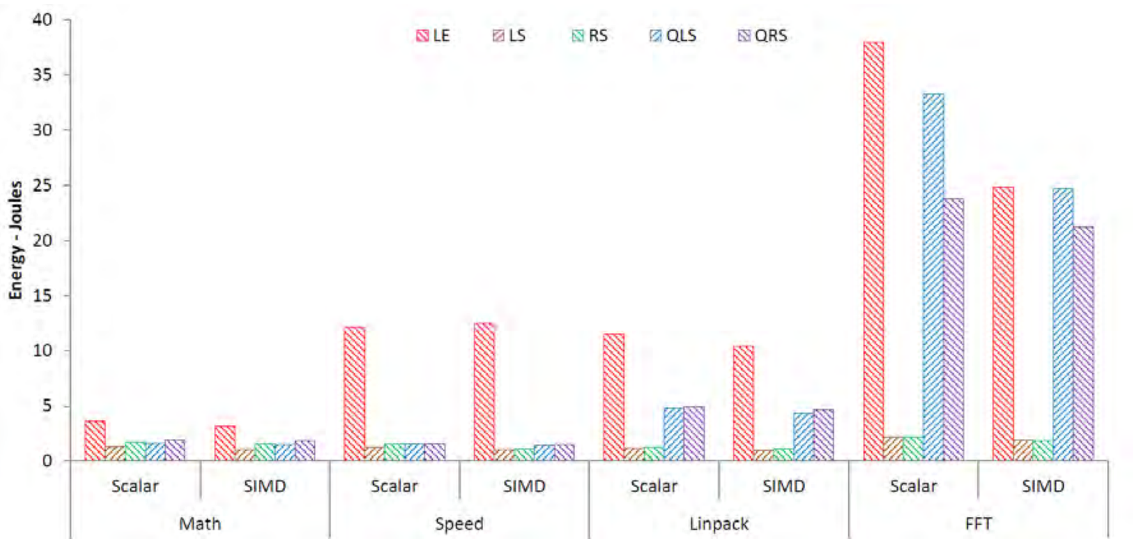


Figure 6.6: Energy Consumption: LCD Sleep Time = 0

turned off. The result of the energy consumption for device sleep time = 0 is presented in Figure 6.6.

The result of FFT benchmark for local execution has been factored by seven to provide comparable perspective for other benchmarks. The results depict that the energy consumption in the case of local computation on the smartphone is highest for all benchmarks. On the contrary, the lowest energy consumed is by the pre-compiled code offloading of SIMDOM framework. SIMDOM and Qemu frameworks provide 85.66% and 61.09% energy efficiency respectively as compared to local MCC-disabled execution on average for all benchmarks. Similarly, SIMDOM consumes 55.99% and 48.23% lesser energy than the Qemu framework in the cloudlet and cloud server scenarios. The energy efficiency in case of offloading (either SIMDOM or Qemu based) increases with the increase in the computational size of the benchmark. For example, the SIMDOM framework on cloudlet server leads to 63.96% energy efficiency for the Mathlib benchmark as compared to the 92.23% for the FFTSIMD benchmark. This implies that for large benchmarks such as FFT, the energy efficiency gained by offloading is higher as the mobile device remains in idle low-power state for longer time intervals.

In most of the cases, the energy consumption of cloud server is higher than the cloudlet for both SIMDOM and Qemu based offloading. There are two reasons behind high energy consumption of the cloud server. Firstly, the network latency of the remote cloud server is higher than the cloudlet server leading to more waiting time for code offload and result feedback. Secondly, for smaller inputs or low execution time benchmarks, the network latency and overhead always dominates the computational overhead. For example, the FFT benchmark which has a higher number of computations leads to 2.71% and 22.35% energy efficiency for cloud execution as compared to cloudlet execution for SIMDOM and Qemu frameworks respectively. On the contrary, the rest of the benchmarks with a smaller number of instructions lead to 15.15% and 12.87% energy efficiency for the cloudlet execution as compared to cloud execution for SIMDOM and Qemu frameworks respectively.

The SIMD benchmarks are energy efficient in most of the scenarios of local and remote execution as compared to scalar benchmarks. For local MCC-disabled execution, the SIMD benchmarks provide 13.03% energy efficiency as compared to scalar benchmarks. The SIMDOM and Qemu frameworks provide 16.88% and 11.63% energy efficiency respectively for the SIMD benchmarks as compared to scalar benchmarks on the cloudlet. The higher percentage of energy efficiency for the SIMDOM framework in case of SIMD benchmarks points to the fact that the SIMD instructions are efficiently translated. Similarly, the SIMDOM and Qemu frameworks provide 15.55% and 6.61% energy efficiency respectively for the SIMD benchmarks as compared to scalar benchmarks on the cloud server. Hence, the SIMDOM efficiency for SIMD instruction translation remains stable for both cloud and cloudlet execution.

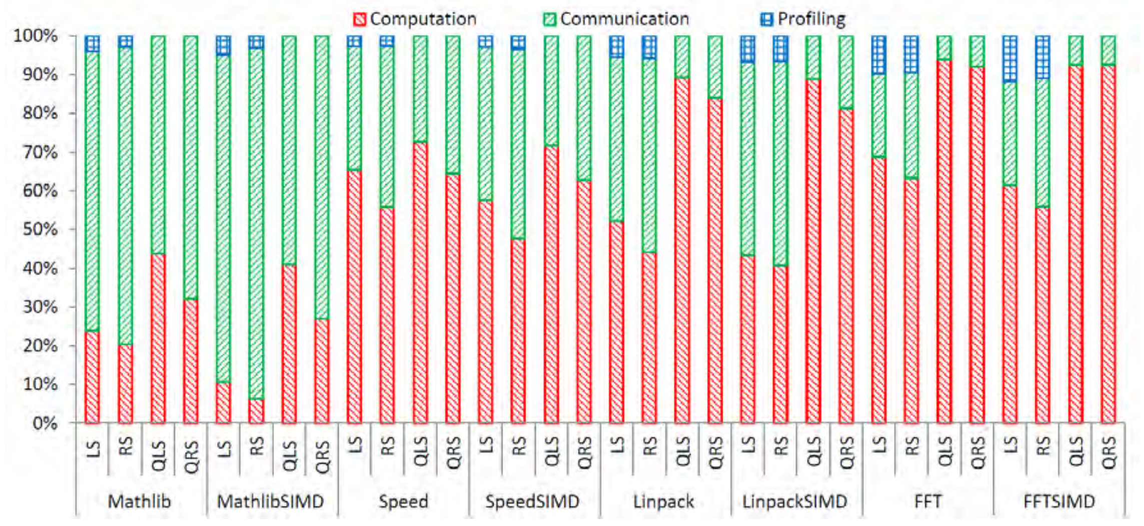


Figure 6.7: Energy Distribution: LCD Sleep Time = 0

6.3.1.1 Energy Distribution

In this subsection, we will analyze the distribution of energy among the energy consuming components. In case of code offloading, the energy consumption can be contributed to three major categories; **(a)** energy consumed during execution of the benchmark, i.e., computational energy, **(b)** energy consumed during offloading of data and receiving of results, i.e., communicational energy, and **(c)** energy spent while profiling the code at server, i.e., profiling energy. We exclude the case of local MCC-disabled execution in our evaluation as all the energy is spent in computations in that scenario. Figure 6.7 presents the energy distribution for the evaluated benchmarks in a 100% stacked column chart.

There are multiple implications of the aforementioned result for energy distribution. Firstly, the profiling energy contributes only for the SIMDOM framework where we profile application for SIMD instructions and recompile the application for corresponding ARM and x86 architectures. The profiling energy is higher for large computation benchmarks. For FFT benchmark which is a library (consists of multiple files), the profiling overhead is significantly high (16.5%). On the contrary, the profiling overhead is very low (approximately 2.2%) for the Mathlib benchmark.

Secondly, the computational energy consumption is higher in all of the cases than the communicational energy. However, for the Mathlib benchmark application, the communicational energy is high. The reason behind the low ratio of computational energy is that SIMDOM framework sends and receives data as C code files that have small network footprint. However, in the case of benchmarks (Speed, Linpack, and Mathlib) where computations are low, the communicational energy matches the computational energy on some levels. With the increase in the computational size of the benchmark, the contribution of communicational energy decreases. Moreover, the communicational energy is higher for cloud server scenarios than the cloudlet scenarios. Furthermore, the profiling overhead is lower in the case of cloud server that has more computational power. Another implication of the aforementioned result is the difference in SIMD and scalar versions of the benchmarks. As the computations are efficient in the SIMD version of the benchmark, the computational energy decreases while the share of communication energy increases in the overall distribution. However, the decrease in computational energy for SIMD benchmarks is more significant for the SIMDOM framework as compared to Qemu.

6.3.1.2 Impact of Sleep Time

The results presented in above sections are based on the assumption that the mobile is switched to idle low-power state and the LCD is turned off as soon as the computations are offloaded to the cloud server. In a real-time scenario, a mobile user has multiple options for the mobile idle state. The mobile device utilized in our evaluation has six idle state options, i.e., 15 seconds, 30 seconds, 1 minute, 2 minute, 5, minute, and 10 minute. In this section, we explore other options for mobile device idle state and analyze its effects on mobile device power consumption. For local smartphone execution, the mobile LCD always remains on as long as the task is being performed on the device.

Case 1: Sleep Time = 15s. We analyze the energy consumption of SIMDOM frame-

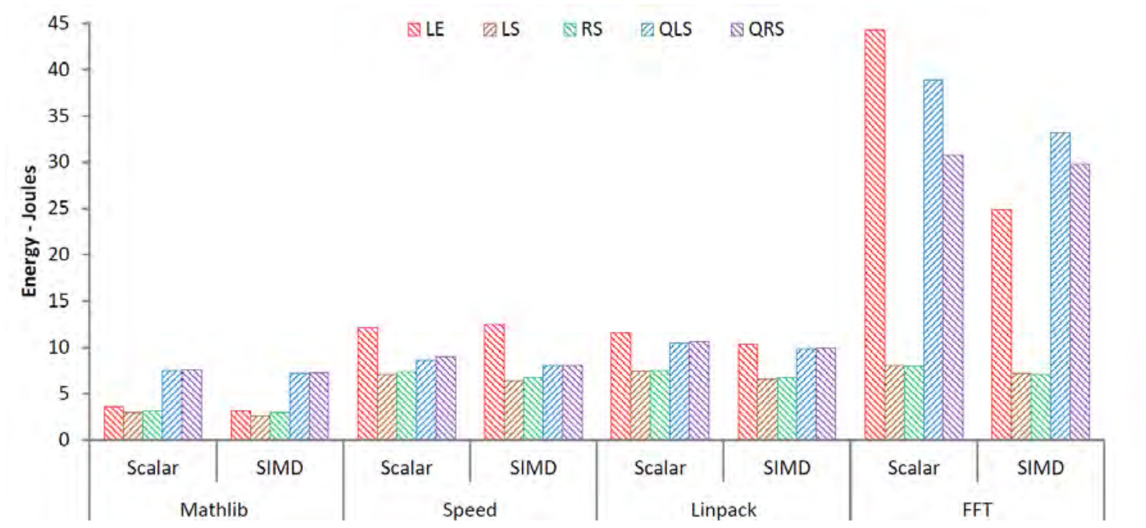


Figure 6.8: Energy Consumption: Sleep Time = 15s

work when the device sleep time is set to 15 seconds. Figure 6.8 presents the results of energy consumption for application execution in the mobile device sleep time configuration of 15 seconds.

The result of FFT benchmark for local execution has been factored by six to provide comparable perspective to other benchmarks. There are multiple ramifications of the aforesaid result. The result depicts that for all instances other than local mobile execution, the energy consumption has increased from the base case of device sleep time. The reason behind the static energy of mobile local execution is that during execution of the task, the mobile device always remains in power on state. The SIMDOM framework consumes less energy than both local mobile and Qemu framework scenarios. For example, the SIMD framework on cloudlet consumes 46.02% less energy than the local execution on average for all benchmarks. The energy efficiency increases with the size of the benchmark as the FFT provides the highest efficiency (96.99%) while the Mathlib provides the least efficiency (15.36%). Similarly, the SIMDOM framework is 47.34% energy efficient than the Qemu framework on average for all benchmarks on the cloudlet. The energy consumption of SIMDOM framework is less in the case of cloudlet server than the cloud server due to low network overhead of offloading for Linpack, Speed, and

Mathlib benchmarks. On average, the SIMDOM is 2.70% energy efficient in the cloudlet execution as compared to cloud execution due to the higher communicational overhead of the cloud server. However, the cloud server energy consumption is lower than cloudlet server energy consumption in case of FFT benchmark as its higher computational requirements dominate the network overhead. Moreover, a similar pattern is found for cloud and cloudlet energy consumption in the case Qemu framework.

Qemu framework also provides energy efficiency to the mobile client for all benchmarks except for both scalar and SIMD versions of Mathlib and SIMD version of FFT benchmark. For the rest of benchmarks, the Qemu cloudlet provides 18.18% energy efficiency as compared to local MCC-disabled execution on average. The Qemu framework does not provide energy efficiency for Mathlib as the computations are small in number and better off performed locally than executed with the overhead of profiling and offloading. With the increase in the number of computations in benchmarks, the energy efficiency for the QLS scenario increases. However, the vectorization of FFT benchmark leads to compact code that does not provide energy efficiency in case of the Qemu framework.

The SIMD versions of the benchmarks provide higher energy efficiency than the scalar versions. The local execution, SIMDOM framework on cloudlet, and Qemu framework on cloudlet provide 15.37%, 11.19%, and 7.80% energy efficiency respectively for the SIMD benchmarks as compared to scalar benchmarks. It can be noted that the energy efficiency for the SIMDOM framework increases with respect to the Qemu execution, thereby quantifying the efficiency of the SIMD translator. Qemu provides lower efficiency than both the local execution and SIMDOM framework for the SIMD based benchmarks. Therefore, it can be stated that the Qemu framework loses some of the efficiency of the SIMD instructions while translating the applications by vector-to-scalar mappings.

Case 2: Sleep Time = 30s. We analyze the energy consumption of the evaluated

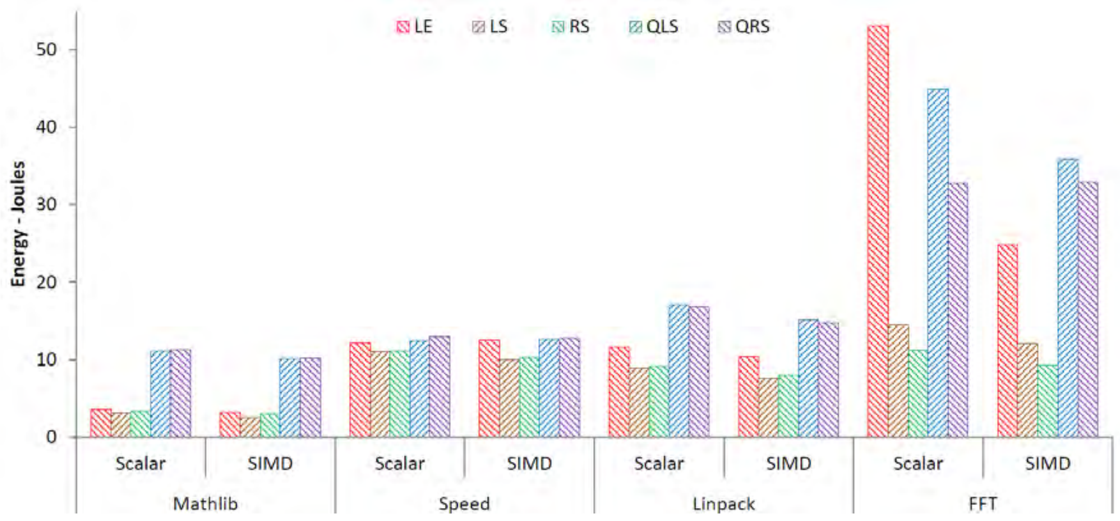


Figure 6.9: Energy Consumption: Sleep Time = 30s

benchmarks for the case of device sleep time of 30 seconds. Figure 6.9 presents the results of energy consumption of all application benchmarks for device sleep time of 30 seconds.

The result of FFT benchmark for local execution has been factored by five to provide comparable perspective to other benchmarks. There are three inferences of the aforementioned results. Firstly, for Mathlib, Speed, and Linpack benchmarks, the Qemu framework does not deliver energy efficiency. The Qemu framework also does not provide energy efficiency for the SIMD version of FFT benchmark. Hence, the offload decision is not feasible in terms of energy for compiled code for these benchmarks. The Qemu framework provides 35.42% higher energy consumption for the aforementioned benchmarks on average for the cloudlet as compared to local execution. The compiled code offloading results in 83.08% and energy efficiency for the scalar version on the cloudlet for FFT benchmark.

The second inference is that the SIMDOM framework of pre-compiled code offloading leads to energy efficiency for all benchmark as compared to local execution and compiled code offloading scenarios. The SIMDOM framework provides 32.09% and 29.76% energy efficiency than local execution for the cloudlet and cloud servers respectively on

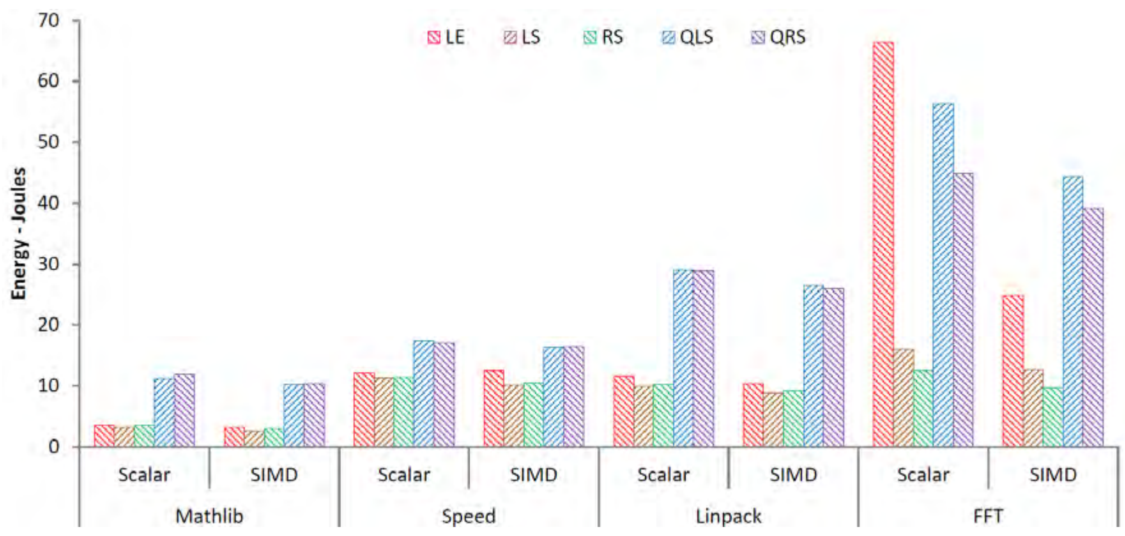


Figure 6.10: Energy Consumption: Sleep Time = 60s

average for all benchmarks. Hence, the code can be offloaded even if the device sleep time is less than or equal to 30 seconds in SIMDOM framework for all benchmarks. However, the energy gains are marginal for the Mathlib benchmark (13.96% and 6.42% respectively for cloudlet and cloud server). As the benchmark computational size increases for FFT benchmark, the SIMDOM framework leads to higher energy efficiency (50.24%). Moreover, the SIMDOM framework leads to 51.24% and 50.39% energy efficiency than the Qemu on average for the cloudlet and cloud servers respectively.

Thirdly, as evaluated in earlier cases, the SIMD benchmarks provide higher energy efficiency than their scalar counterparts. The local execution, SIMDOM framework on cloudlet, and Qemu on cloudlet provide 17.7%, 14.15%, and 10.58% energy efficiency for the SIMD benchmarks respectively as compared to scalar benchmarks. The higher efficiency for the SIMDOM framework as compared to Qemu proves the effectiveness of the SIMD translator.

Case 3: Sleep Time = 60s. We analyze the energy consumption of the evaluated benchmarks for all scenarios with the device sleep time set to 60 seconds. Figure 6.10 presents the results of energy consumption for device sleep time of 60 seconds.

The result of FFT benchmark for local execution has been factored by four to provide comparable perspective to other benchmarks. The aforementioned results reveal that the Qemu framework scenarios do not provide energy efficiency as compared to local execution except for scalar FFT benchmark. The Qemu cloudlet execution leads to 34.54% higher energy consumption than that of the local execution while including the negative result of scalar FFT benchmark. On the contrary, the SIMDOM framework provides 28.14% and 26.36% energy efficiency as compared to local executions for cloudlet and cloud servers respectively. Similar to previous cases, the FFT benchmark provides the highest energy efficiency, i.e., 93.99% for cloudlet server and 95.30% for cloud server. On the contrary, Mathlib provides the least energy efficiency, i.e., 10.33% and 1.67% respectively for cloudlet and cloud SIMDOM framework.

The SIMDOM framework provides 61.70% and 63.92% energy efficiency on average for all benchmarks than the Qemu framework for cloudlet and cloud server execution respectively. The cloud server execution provides considerable energy efficiency than the cloudlet server only for the FFT benchmark. The SIMDOM framework in the cloud server scenario provides 26.94% energy efficiency than the cloudlet server for the FFT benchmark. For the remaining applications, the cloud server gains are negligible. Similar to previous cases, the SIMD benchmarks provide more efficiency than the scalar versions for the SIMDOM framework. The SIMDOM framework and Qemu on cloudlet provides 14.75% and 11.11% energy efficiency respectively for the SIMD benchmarks as compared to scalar benchmarks.

In the above subsection, we analyzed the energy efficiency of computational offloading frameworks in various scenarios of mobile device sleep time. We observed that by increasing mobile device sleep time, the energy efficiency of the offloading frameworks decreases. The SIMDOM framework provides 85.66% energy efficiency for immediate device sleep after offloading as compared to 28.14% for sleep time = 60 seconds scenario for the cloudlet execution. For benchmarks that take longer than 60 seconds to execute on the cloud server, the energy efficiency will further decrease upon increase of mobile device sleep time. However, the most energy efficient case is where the mobile device is kept in sleep mode after computational offloading. The Qemu offloading framework provides energy efficiency for all benchmarks in the sleep time = 0 seconds case. For the case of sleep time = 15 seconds, the Qemu provides energy efficiency for three benchmarks. Similarly, for the case of sleep time = 30 seconds, the Qemu provides energy efficiency for only scalar version of FFT benchmark. However, for sleep time = 60 seconds, the Qemu framework provides energy efficiency for scalar FFT while leading to 34.54% higher energy consumption for all benchmarks on average.

In all the evaluated scenarios, the SIMD benchmarks provide higher efficiency as compared to scalar benchmarks for the SIMDOM framework. The SIMDOM framework provides approximately 14% energy efficiency for the SIMD benchmarks as compared to scalar benchmarks while Qemu provides only 7% energy efficiency.

6.3.1.3 Impact of Application Partitioning

The SIMDOM framework provides three cases for application partition and offloading, i.e., no application partition in case the offload decision is not feasible, full application partition and offloading, and partial application offloading for only SIMD instructions. In the above sections, we investigated the SIMDOM framework where the complete application is offloaded to the server without partition. In this subsection, we will investigate the SIMDOM framework for partial application partition.

In the simplest of the cases, the complete application can be offloaded and executed on the cloud. However, the offload manager can decide upon the partial execution of the application based on the input from the application profiler. The application profiler profiles the application after compiling it for ARM and x86 architectures. The simplest partition of the application code during compilation can be based on the NEON intrinsics.

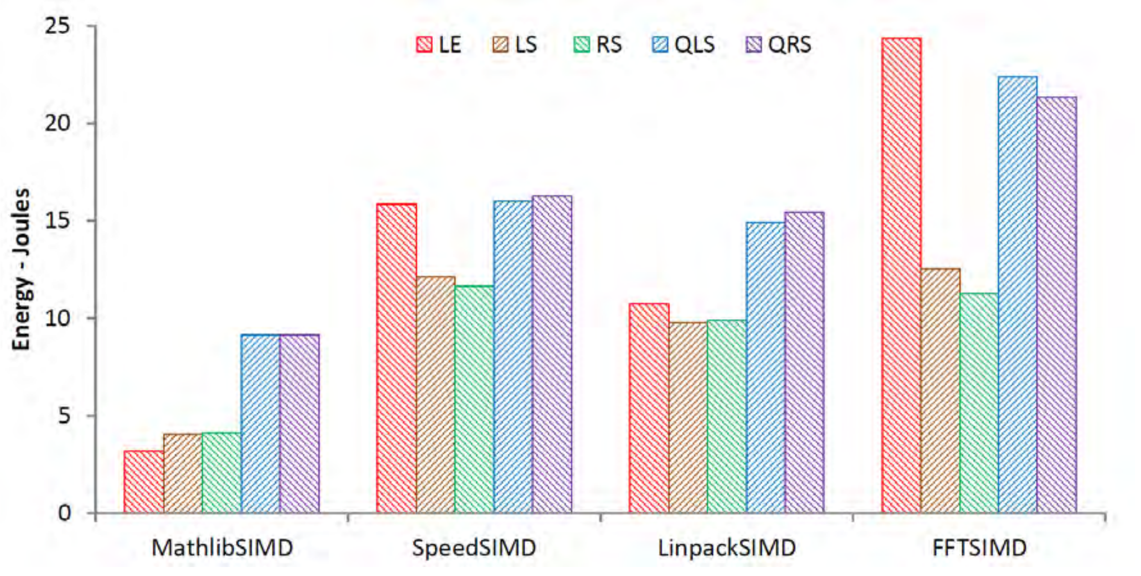


Figure 6.11: Energy Consumption: The Case of Application Partition

The NEON intrinsics can be translated to SSE code and executed on the server, while the scalar code can be executed on the mobile device. We analyze only SIMD benchmarks for application partitioning as scalar benchmarks do not provide an opportunity for partitioning based on SIMD instructions. Figure 6.11 illustrates the energy consumption of application partition based offloading.

There are multiple ramifications of the above stated result. Firstly, the SIMDOM framework does not provide energy efficiency for benchmarks with lower computation, such as the Mathlib benchmark. Through application profiling, we found that the Mathlib benchmark produces 26.14% and 46.37% SIMD instructions for the ARM and x86 ISAs respectively. The percentage of SIMD instructions in the Mathlib benchmark were found to be highest among the evaluated benchmarks. However, Mathlib benchmark does not gain from the application partition as the total number of SIMD instructions are low. The Linpack and Speed benchmark provide marginal energy efficiency in this scenario. On the contrary, the total number of SIMD instructions in the FFT benchmark are quite high as compared to other benchmarks. Hence, the only benchmark application gaining considerable energy efficiency (93.94%) from the application partition is the FFT benchmark.

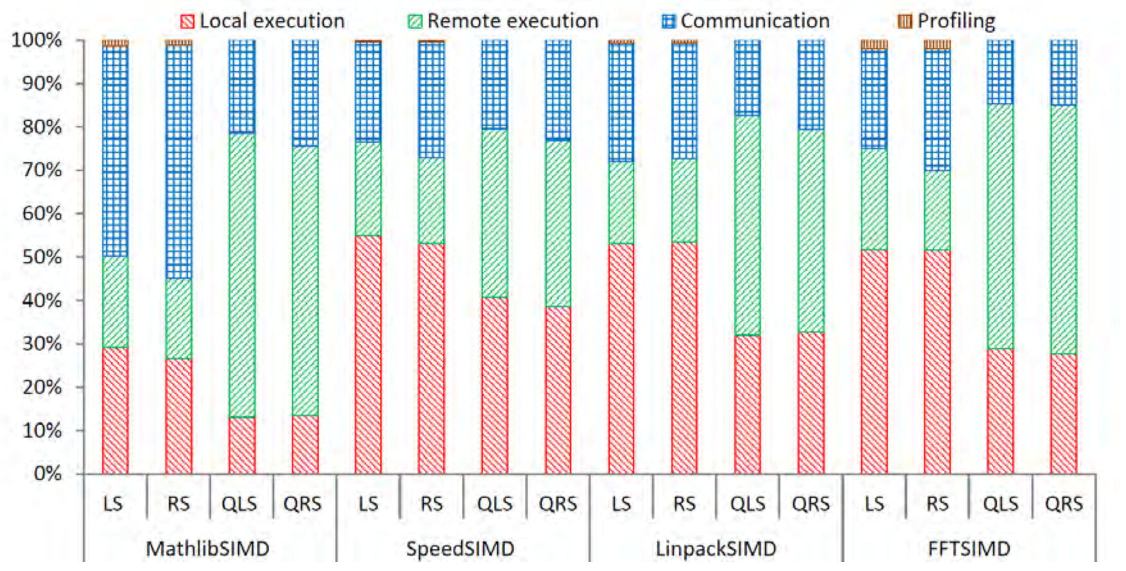


Figure 6.12: Energy Distribution: The Case of Application Partition

Compared to the corresponding scenario of full application offloading, the SIMDOM energy gains corresponding to local execution decrease from 42.85% to 28.28% on average for all benchmarks.

Secondly, the Qemu framework does not provide energy efficiency for any of the benchmarks except for the FFT. Qemu consumes 31.42% higher energy than the local MCC-disabled execution on average for all benchmarks. Moreover, Qemu provided energy efficiency for three benchmarks in the full application offloading scenario compared to only one benchmark in application partitioning scenario. Thirdly, the SIMDOM is 39.61% energy efficient than the Qemu framework on average for all benchmarks for the cloudlet scenario. To illustrate the distribution of energy among local and remote computing, we analyze the result in further detail. The energy distribution of the benchmarks for the case of application partition is presented in Figure 6.12.

The total energy is distributed among local, remote, communicate, and remote profiling components. The communicational energy contributes significantly to the overall energy as compared to the scenario presented in Figure 6.7. The communicational energy contribution is particularly higher for smaller benchmarks due to offloading of data and synchronization of distributed application execution results. Due to the increase in the communicational energy, the contribution of energy utilized in profiling of the application decreases. Moreover, for the SIMDOM framework, the local execution energy consumption is higher than the remote execution energy consumption as the SIMDOM efficiently translates and executes the SIMD instruction over the cloud servers. The Qemu framework provides the contrary scenario.

Theoretically, as the application is divided into a scalar instruction part and a vector instruction part, the overall energy efficiency of the application should increase from a singleton execution scenario. On the contrary, the energy efficiency of the offloading scenarios decreases. There are two main contributors to this fact. Firstly, the communicational energy increases as described above. Secondly, after the completion of local execution, the mobile device waits for the result of the offloaded part without going into sleep mode immediately. The device waiting for the results contributes to the remote execution energy and increases the overall energy of the application partition based offload scenario.

6.3.1.4 Impact of Computational Size

We investigated the energy efficiency of the SIMDOM framework for variable size inputs. We utilized the Linpack benchmark as its input matrices can be varied. The Linpack benchmark was compiled to operate on 200×200, 400×400, 600×600, 800×800, 1000×1000, 1200×1200, and 1400×1400 matrices. As a result, $N \times N$ basic operations are performed in each benchmark instance. The matrix size of 1400×1400 and greater lead to negative results in terms of performance. The result of the energy consumption of the Linpack benchmark for different execution scenarios are depicted in Figure 6.13.

The energy was measured with the mobile device configuration of sleep mode after 15 seconds of inactivity. The energy consumption in various scenarios does not increase

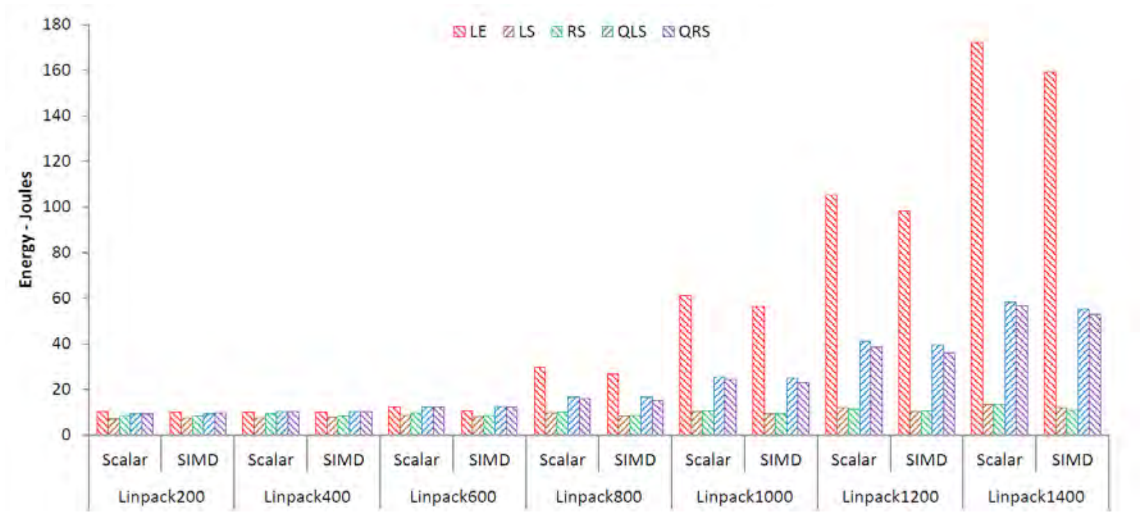


Figure 6.13: Energy Consumption: Linpack Benchmark on Variable Input Matrices of Size $N*N$

significantly until the matrix size of 600×600. Afterward, the energy spent on local execution increases exponentially with the increase in the size of the input. The energy consumption of the Qemu framework also increases linearly with the increase in matrix size. On the contrary, the energy consumption of SIMDOM framework remains linearly stable and does not increase significantly with the increase in the size of the input matrix. Both pre-compiled code SIMDOM and compiled code Qemu offloading frameworks lead to the energy efficiency for larger input sizes. However, for smaller inputs, the energy gains are marginal. The energy efficiency of the cloudlet based SIMDOM framework as compared to local execution increases from 26.63% for 200×200 matrix to 92.21% for 1400×1400 matrix. Similarly, the energy efficiency of the SIMDOM framework as compared to Qemu increases 22.75% for 200×200 matrix to 78.80% for 1400×1400 matrix.

The SIMD version of the benchmarks leads to considerable energy efficiency as compared to basic benchmarks. The SIMD version efficiency for the local execution on a mobile device is 6.69%. However, the translated code by the SIMDOM framework provides better SIMD to basic version energy efficiency ratio of 8.58%. On the contrary,

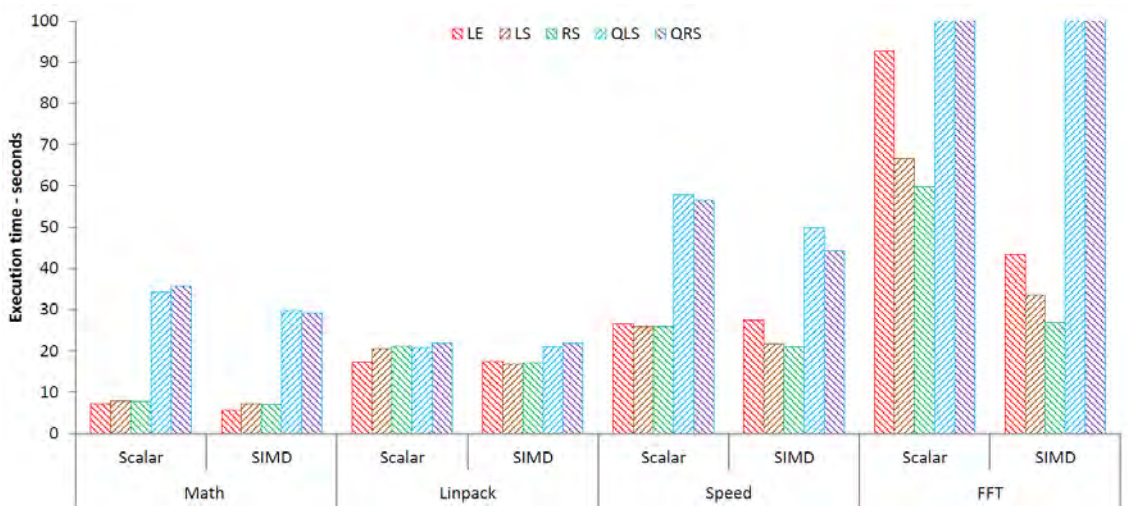


Figure 6.14: Execution Time of Evaluated Benchmark Applications

Qemu provides only 2.21% energy efficiency while providing inefficient vector-to-scalar translation for the SIMD benchmarks. These ratios also quantitatively assert the efficiency of SIMD translations in the SIMDOM framework.

6.3.2 Execution Time

The second most important factor in the evaluation of a MCC offloading framework is the execution time of the applications. The execution time of the offload framework should also be efficient than the local computations to provide greater benefits to the cloud users. In this section, we investigate the execution time of the benchmarks in different execution scenarios for the case where the complete application is offloaded to the server. The total execution time includes the overhead of application offloading and profiling. In Figure 6.14, we present the execution times of the evaluated benchmarks in various scenarios of application execution.

The execution time of the FFT benchmark scale out of the figurative graph limits. Therefore, all scenarios of the FFT benchmark and Qemu scenarios of FFTSIMD benchmark are scaled by a factor of 5. Still, FFT-QLS and FFT-QRS is out of bound of the graph. The FFT benchmark values have been scaled to provide a comparative perspective to other benchmarks.

The Mathlib benchmark leads to the lowest execution times while the FFT variations lead to the highest execution times. The Qemu framework does not provide time efficiency for any of the benchmark applications. The Qemu framework leads to 56.68% and 54.90% higher execution time on average for all benchmarks for cloud and cloudlet scenarios respectively. The inefficient translation of SIMD instructions and the overhead of DBT result in the higher execution times for the Qemu framework. On the contrary, the SIMDOM framework leads to time efficiency for all the benchmarks except for Mathlib, MathlibSIMD, Linpack, and LinpackSIMD in the cloudlet and cloud server scenarios. On average, the SIMDOM framework provides 3.93% and 8.01% time efficiency as compared to the local execution for the cloudlet and cloud offloading respectively while including the negative results. The time efficiency of the SIMDOM framework increases with the increase in the computational size of the benchmark. For instance, the time efficiency of FFTSIMD benchmark is 22.46% and 37.69% for the cloudlet and cloud server respectively for SIMDOM framework. Therefore, the offloading is favorable in terms of execution time for those benchmarks that have a large number of computations. On the contrary, smaller applications provide negligible or no time efficiency that can be further degraded by the variable wireless medium.

The SIMDOM framework leads to 57.30% and 56.53% time efficiency than the Qemu framework for the cloudlet and cloud server scenarios respectively. The time efficiency of the SIMDOM framework compared to Qemu is higher for Mathlib (76.70%) and FFT (93.14%) benchmarks that have a higher percentage of SIMD instructions. The reason behind higher time efficiency of SIMD-rich benchmarks is that the Qemu applies non-optimal translations to vector instructions leading to one-to-many mappings in most of the cases. Moreover, the higher time efficiency in case of Mathlib and FFT benchmark shows that the SIMDOM framework provides an efficient solution for translation and offloading of vector instructions.

The cloud server provides efficient execution times than the cloudlet, particularly for large benchmarks. However, few exceptions occur for the small benchmarks. The higher execution time on the cloud server can be contributed to higher CPU load, network latency, or benchmark computational size. On average, the SIMDOM framework on cloud server provides 4.16% time efficiency than the cloudlet including the negative results. The lower time efficiency is attributed to the fact that the computational power (MIPS) of the cloudlet and cloud server are quite similar.

The SIMD benchmarks provide considerably lower execution time than the scalar benchmarks. The SIMD benchmarks lead to 31.72%, 35.54%, and 12.37% time efficiency when compared to scalar benchmarks for local execution, SIMD framework on cloudlet, and Qemu framework on cloudlet respectively. The lowest time efficiency is provided by Qemu with SIMD benchmark execution time similar to the scalar benchmarks. On the contrary, SIMDOM framework leads to the highest time difference in SIMD and scalar benchmarks while mapping most of the ARM vector instructions to x86 vector instructions.

6.3.2.1 Time Distribution

We further investigate the execution time of the benchmarks by illustrating the distribution across time consuming components, i.e., computing, profiling, and communication. Figure 6.15 provides the distribution of the execution time among these components.

The profiling time distribution is only for SIMDOM framework due to activities of the application profiler and SIMD translator. The time distribution of local execution is not illustrated as it consists entirely of local computations. The time spent on profiling the application increases with the size of the benchmark and has the highest proportion for the FFT benchmark. The higher percentage of SIMD instructions also leads to higher

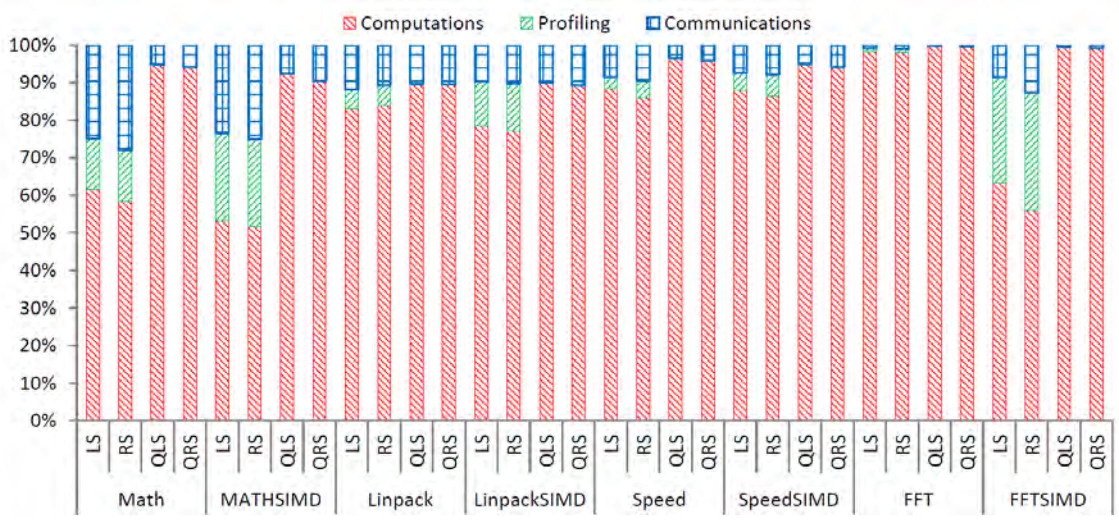


Figure 6.15: Execution Time Distribution of The Evaluated Application Benchmarks

profiling overhead for the Mathlib benchmark. It can be further noted that the time spent in profiling the SIMD versions of the application is higher than the scalar versions. The reason is that the SIMD version often requires additional compilation flags that lead to higher SIMD instruction count and higher profiling overhead. The time spent on communicating the FFT benchmark is also highest as it consists of multiple libraries rather than singleton source files in case of other benchmarks. However, in the overall distribution, the communication time share is dominated by profiling and computational time. The contribution of communication time is less for the Qemu scenarios than SIMDOM scenarios. The Qemu framework leads to higher number of computations, hence, dominating the communication time.

6.3.2.2 Impact of Application Partitioning

In this subsection, we investigate the impact of application partitioning on the execution time of the application. The application partition is based on NEON intrinsics and performed by statically annotating code of the benchmark. We analyze only SIMD benchmarks for application partitioning as scalar benchmarks do not provide an opportunity for partitioning based on SIMD instructions. Figure 6.16 illustrates the result of application

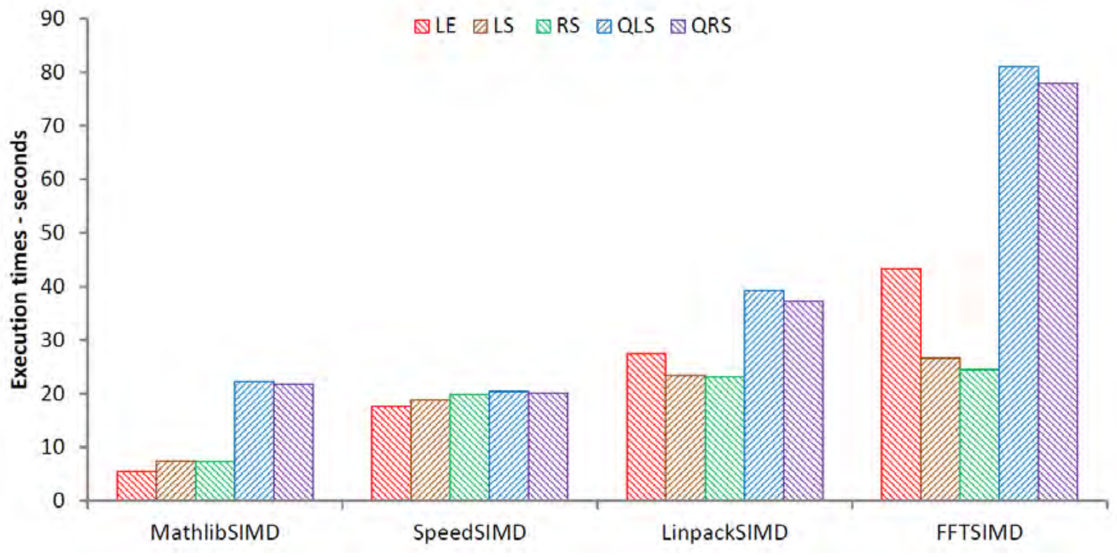


Figure 6.16: Execution Time: The Case of Application Partition

partition based offloading.

The execution time of the FFT benchmark scale out of the figurative graph limits. Therefore, the Qemu scenarios of FFTSIMD benchmark are scaled by a factor of five. The Qemu framework does not provide time efficiency for any of the benchmarks. While the base case of full application offloading leads to 56.68% higher execution times, the partial application partitioning leads to 52.15% higher execution time for the Qemu framework on cloudlet. As Qemu does not handle SIMD instructions optimally, it does not gain significant advantage from the partitioning of the application.

The SIMDOM framework provides time efficiency for Linpack and FFT benchmarks. Including the negative results, the SIMDOM framework provides 5.23% and 5.87% time efficiency for the cloudlet and cloud server respectively as compared to the local execution. The execution time efficiency of SIMDOM framework shows an increase from 3.93% to 5.23% and decrease from 9.33% to 5.87% for cloudlet and cloud server respectively when compared to the case of full application offloading for the SIMD benchmarks. The application partitioning and parallel execution of scalar and vector instances lead to higher time efficiency for only FFT benchmark. The FFT benchmarks gain 38.61% time efficiency as it has the highest number of SIMD instructions. The rest of the benchmarks have a small number of instructions and the overhead of instruction offloading dominates the efficiency of parallel execution.

In case of energy and execution time, the application partitioning led to lesser efficiencies when compared to the case of full application offloading. However, the execution time and energy show higher efficiencies in case of partial application offloading for the FFT benchmark. As the number of computations and vector instructions increase for the FFT benchmark, the parallel execution of code is efficient. However, for smaller benchmark applications, as the application is executing on the mobile device and expecting results from the server, its network components remain in the active state. Therefore, benchmarks that are not specifically compute-intensive result in higher energy consumption and execution time in case of application partitioning.

6.3.2.3 Impact of Computational Size

We investigate the application execution time for variable size inputs. We utilize the Linpack benchmark as its input matrices can be varied. The Linpack benchmark was compiled to operate on 200×200, 400×400, 600×600, 800×800, 1000×1000, 1200×1200, and 1400×1400 matrices. The result of the execution times of the Linpack benchmark are depicted in Figure 6.17.

The Qemu inputs are scaled to fit the figurative bounds of graph and provide a better illustration for all input sizes. The execution time of Qemu for matrix sizes 800×800 and 1000×1000 has been scaled by a factor of five while that for matrix sizes 1200×1200 and 1400×1400 has been scaled by a factor of six.

There are multiple ramifications of the aforementioned result. The execution time in local MCC-disabled does not increase until the input size of 600×600. Afterward, the increase in matrix size leads to exponential increase in the execution time. Similarly, the

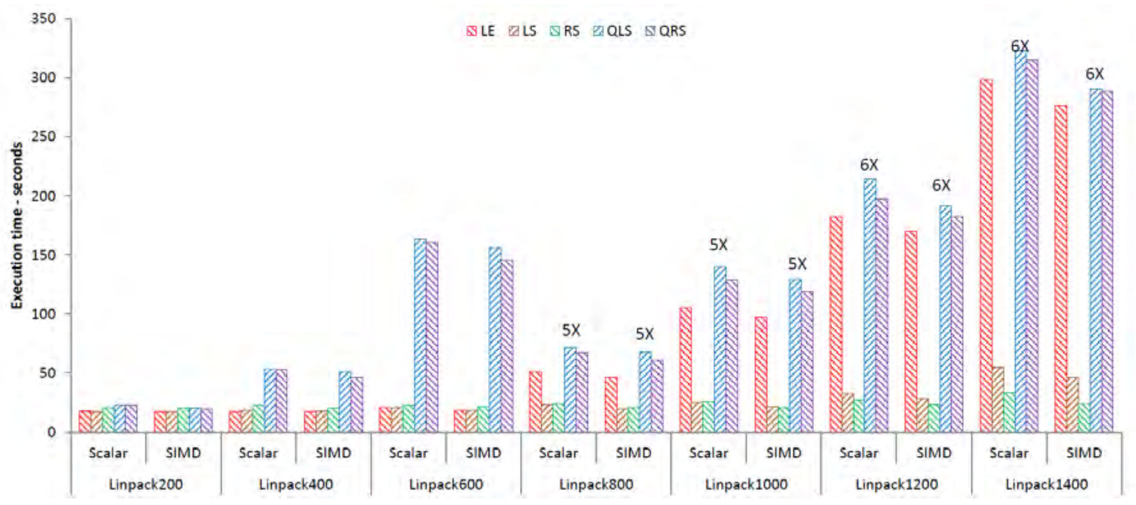


Figure 6.17: Execution Time of Linpack Benchmark on Variable Input Matrices of Size $N*N$

SIMDOM framework does not show any significant increase in execution time with the increase of matrix size. The only significant increase in execution time of the SIMDOM framework occurs for the input matrix of 1200×1200 and 1400×1400. The stable performance of the SIMDOM framework is due to the computational power of the cloudlet and cloud servers that can sustain the input increase gracefully to provide efficient execution. On the contrary, Qemu does not sustain performance on the increase in the size of input matrix as the execution time of Qemu scenarios increases exponentially.

The SIMDOM framework does not provide time efficiency for the small matrix inputs. The SIMDOM framework provides time efficiency as compared to local execution after the input size is increased to 800×800 and beyond. For the input matrices 800×800 to 1400×1400, the SIMDOM framework on cloudlet provides 66.24% efficiency than the local execution. For the input matrices 200×200, 400×400, and 600×600, the SIMDOM framework on cloudlet leads to 0.54% time overhead as compared to local execution. Therefore, the time efficiency of SIMDOM framework increases with the increase in the size of the input. The SIMD versions of the benchmarks show significantly lower execution times than the basic versions. The efficiency of the SIMD versions increases

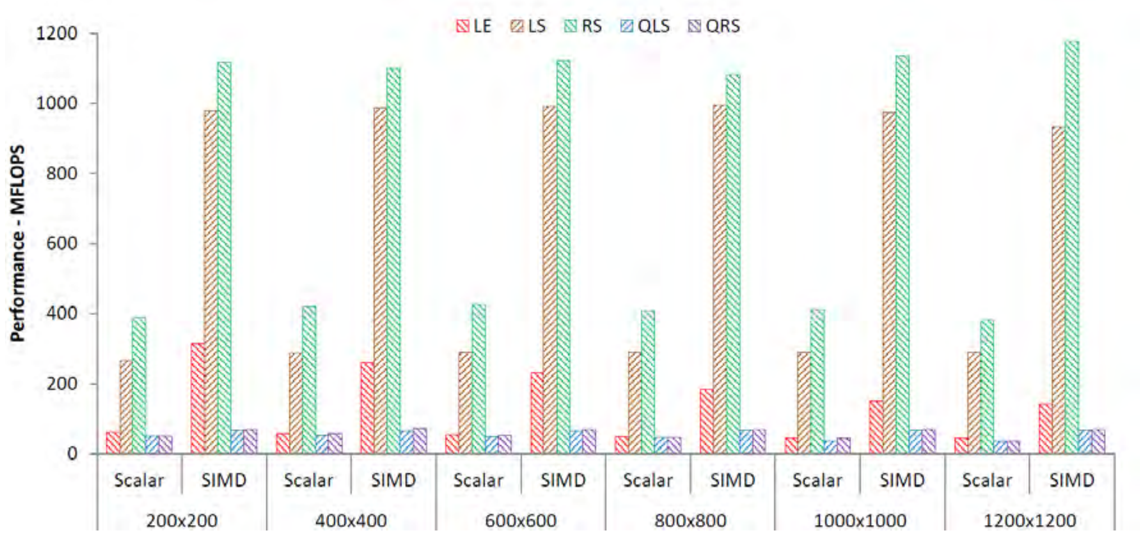


Figure 6.18: Performance of Linpack Benchmark on Variable Input Matrices of Size $N*N$

with the increase in the size of the benchmark. However, the efficiency is more significant for the SIMDOM framework than the Qemu as SIMDOM efficiently translates the SIMD instructions. The SIMD versions of the benchmarks provide 14.32% time efficiency than the scalar version for the SIMDOM framework on the cloudlet. On the contrary, the Qemu provides 9.63% time efficiency for the SIMD versions on the cloudlet.

6.3.3 Performance Gain

The Linpack benchmark also analyzes the performance of the system. The performance measured by the Linpack benchmark is dependent on the calculation of $x[i] = x[i] + c * y[i]$ for a system of linear equations based on matrices of variable inputs. The performance of the system is measured in MFLOPS, i.e., Million of Floating Point instructions per Second. The Linpack benchmark iterates through loops while solving the aforementioned equation. As a result, the number of floating point instructions executed over a time duration are calculated. Figure 6.18 presents the performance of Linpack benchmark in different execution scenarios.

The performance of cloud server for all input sizes scale out of the figurative graph limits. Therefore, all scenarios of the cloud server (RS) have been divided by a factor of

3.3. On the other hand, the Qemu performance is too low to be figuratively represented in the graph. Therefore, the Qemu performance for both cloudlet (QLS) and cloud server (QRS) are multiplied by three. The Linpack benchmark for 1400×1400 matrix input is not included in the results as it leads to negative performance on all execution instances.

The SIMDOM framework provides 79.93% and 95.16% higher performance than the local execution for cloudlet and cloud server respectively on average. Similarly, the SIMDOM cloud server provides 76.55% higher performance than the cloudlet server on average for all benchmarks. Contrary to the cases of energy and time efficiency, which were 13.60% and 5.99% respectively, the performance efficiency of the cloud server is relatively high than the cloudlet. The cloud server provides higher performance than the cloudlet due to its abundance of resources, such as the number of processing cores.

The Qemu framework provides the lowest performance. The local execution is 79.96% better in performance than the Qemu cloudlet on average for all benchmarks. Similarly, the SIMDOM cloudlet is 96.23% performance efficient than the Qemu cloudlet on average for all benchmarks. The increase in the matrix size effects the performance of the application execution, particularly, for resource-constrained environments. For example, the performance of Linpack benchmark after the input size of 600 decreases on the mobile device. However, the SIMDOM cloudlet and cloud server sustain their performance for higher input matrices.

The SIMD versions of the benchmarks provide 74.56%, 70.78%, and 29.45% higher performance than the scalar benchmarks for the local, SIMDOM cloudlet, and Qemu cloudlet scenarios respectively. The SIMD benchmarks have the highest performance as compared to scalar benchmarks in local execution. The SIMDOM cloudlet provides a comparatively equivalent performance for SIMD benchmarks. However, the Qemu does not show higher performance for SIMD benchmarks as compared the local execution and SIMDOM framework as it translates most of the vector instructions to scalar instructions.

6.4 Conclusion

In this chapter, we discussed the performance of SIMDOM framework from various dimensions of device sleep time, application partition, and increasing input sizes. We compared the energy consumption of proposed system model with the empirical data and found 0.94% of error in the best case for local execution and 7.98% error in the worst case for Qemu cloud execution. The higher error rate in model validation is contributed to memory page dynamics of Qemu and wireless network fluctuations. Similarly, for execution time, a best case error of 0.63% and worst case error of 4.29% was found for MCC-disabled and Qemu cloud execution validity, respectively. The Pearson correlation coefficient lies near 1, hence, validating the correlation between the data of mathematical model and empirical results for energy and execution time.

The SIMDOM framework was evaluated on three main parameters of execution, i.e., energy, execution time, and MFLOPS performance. Comparison with state-of-the-art Qemu based compiled code translation and offloading framework was performed. The SIMDOM framework provides higher energy efficiency than both local MCC-disabled execution and Qemu based offloading. The SIMDOM framework provides 85.66% and 55.99% energy efficiency than the local device and Qemu based offload scenarios respectively in the base case of mobile sleep state. As the mobile sleep state is delayed to one minute, the energy efficiency of SIMDOM framework as compared to local MCC-disabled execution decreases to 28.14%. The cloud server provides energy efficiency comparative to cloudlet only for high computation benchmarks, such as the FFT. Moreover, the SIMD versions of the benchmarks lead to higher energy efficiency for the SIMDOM framework as compared to the Qemu framework, hence, quantifying the efficiency of the SIMD translator. Furthermore, the results show that SIMD instruction based application partitioning leads to energy efficiency only for the FFT benchmark that has higher ratio and number of SIMD instructions.

The SIMDOM framework provides time efficiency only for benchmark applications with higher computational requirements. The SIMDOM framework provides 3.93% and 8.01% time efficiency as compared to the local execution for the cloudlet and cloud server offloading respectively while including the negative results. Moreover, the SIMD instruction based application partitioning leads to higher time efficiency for only the FFT benchmark. For the rest of the benchmarks, the overhead of cloud offloading dominates the efficiency of parallel execution of scalar and vector partitions of the benchmark. The SIMDOM framework provides 79.93% and 95.16% higher MFLOPS performance on the cloudlet and cloud server respectively. Moreover, the results depict that the SIMDOM framework on cloud server provides 76.55% higher MFLOPS performance than the cloudlet due to the higher computational power of the former. On the contrary, the Qemu based code offloading does not provide efficient MFLOPS and execution time performance for any of the benchmark applications and provides energy efficiency in only some of the application execution scenarios.

CHAPTER 7: CONCLUSION

This chapter presents the overall conclusion of the research work and emphasizes the qualitative features of the SIMDOM framework. The conclusive analysis is performed by reflecting on the research objectives set in the first chapter of the thesis. Future work and research contributions are also highlighted.

The rest of the chapter is organized as follows. In Section 7.1, we reexamine the research aim and objectives listed in Section 1.4 of the thesis. Section 7.2 lists the contributions of this research work. Section 7.3 deliberates on the significance of this work among existing MCC offloading frameworks. In section 7.4, the scope and limitations of this research work are elaborated. At last, Section 7.5 details the research directions in which this work can be further enhanced.

7.1 Research Objectives

This research work aimed to solve the problem of translation and offloading of vector instructions in heterogeneous MCC architectures. In section 1.4, we set four research objectives. We investigate the completeness of these research objective as follows.

Objective 1: To study the MCC offloading frameworks from the perspective of offload enabling techniques to gain insights to the performance limitations of current state-of-the-art solutions.

The first objective of our research was to study and critically analyze the recent state-of-the-art MCC offloading frameworks such that insights are gained leading to their performance limitations. This research objective was accomplished by a thorough survey in the direction of MCC offloading frameworks, techniques for application execution in heterogeneous MCC architectures, and translation of SIMD instructions in cloud environments. We performed an extensive literature review of our research field through online databases, such as IEEE, ACM, Elsevier, and Web of Science. We organized the literature, devised taxonomies, and provided a qualitative comparison for MCC code offloading, cross-platform application execution, and SIMD instruction translations techniques.

The purpose of this exercise was to identify the open research issues and challenges in SIMD instruction based MCC code offloading frameworks. We found that current cross-platform MCC offloading frameworks do not handle SIMD instructions efficiently. The most commonly utilized cross-platform execution tool, Qemu, translates SIMD instructions to scalar instructions. Therefore, an efficient SIMD instruction translation framework was required in MCC offloading domain.

To investigate the overhead of MCC offload enabling techniques to reveal inefficiency in SIMD instruction translations.

The second objective of this research work was to analyze and investigate the overhead in current MCC offload enabling techniques. System virtualization, application virtualization, and native code offloading are the three fundamental offload enabling techniques utilized in heterogeneous MCC architectures. We investigated the aforementioned MCC offload enabling techniques from the perspective of multimedia applications. The investigation revealed high computational overhead for the application virtualization and native code based offloading techniques. On the other hand, system virtualization techniques are not feasible for wireless access based mobile devices. We further examined the cross-platform execution of native code for SIMD instructions. We found that Qemu, the existing cross-platform framework for native code, does not optimally translate the SIMD instructions and leads to performance loss.

To design and develop an MCC offloading framework based on dynamic mapping of SIMD instructions that supports heterogeneity of computing architectures while providing energy and time efficiency to mobile devices.

The third objective of this research work was to develop an integrated vector instruction translation and offloading framework for heterogeneous MCC architectures. For the vector instruction translation, we devised a vector-to-vector instruction mapping algorithm based on recompilation technique. The native vector library of the mobile device is mapped to that of the cloud server, i.e., ARM NEON intrinsics are translated to x86 SSE intrinsics. An integrated SIMD instruction translation and offloading framework in MCC (SIMDOM) was detailed based on the SIMD instruction translator and offloading modules. The application, network, and energy profilers of the SIMDOM framework were elaborated in detail that provide inputs to the system model for the decision of offload feasibility.

To evaluate the proposed framework for energy and time efficiency and compare it with the state-of-the-art MCC code offloading frameworks.

The fourth objective of this research was to verify the effectiveness of the proposed SIMDOM framework in translation and offloading of vectorized applications. We developed a system model for offloading and execution of SIMD based applications that leads to performance and energy efficiency. We validated the model by comparing its results with the empirical results. We further analyzed the SIMDOM framework for energy, execution time, and performance efficiency on multiple vectorized application benchmarks. We also compared the performance of the SIMDOM framework with the existing cross-platform native code translation framework of Qemu. Moreover, we investigated the performance of SIMDOM framework inefficient translation of the SIMD instructions.

The SIMDOM framework leads to 55.99%, 57.30%, and 96.23% energy, time, and performance efficiency respectively as compared to the Qemu framework in the base case of cloudlet execution. Similarly, the SIMDOM framework provides 85.66%, 3.93%, and 79.93% energy, time, and performance efficiency respectively compared to the local MCC-disabled execution. The efficiency of SIMDOM framework was investigated in different scenarios of device sleep time, application partitioning, and variable application input sizes. The application partitioning leads to higher time and energy efficiency for only large computational applications, such as FFT. Applications with a higher number of computations gain more benefits from computational offloading in general. Moreover, SIMDOM framework provides higher scalar-to-SIMD application performance difference (14% approximately compared to the base case of 7% for Qemu) by efficiently mapping vector-to-vector instructions.

7.2 Contributions

In this section, we highlight the contributions of this research work. The contributions in terms of the scholarly articles are listed separately in Appendix A. This research work contributes to the body of knowledge in following aspects.

- **Taxonomy of MCC Offload Enabling Techniques:** We developed taxonomies from the existing literature for MCC offloading frameworks, cross-platform native code execution, and SIMD instruction translation. The taxonomies were derived from reviewing recent state-of-the-art research works in the corresponding dimensions. Moreover, critical analysis of the selected state-of-the-art research work was performed. The comprehensive literature review led to the identification of open research issues.

- **SIMD Translator Algorithm:** We devised a SIMD translator algorithm for heterogeneous ARM and x86 architectures. The SIMD translator algorithm maps ARM SIMD instructions to x86 SIMD instructions such that vector-to-scalar translations are minimized. The translated code is efficient due to the high percentage of vector-to-vector translated instructions.

- **Integrated MCC Offloading Framework:** We devised a SIMD translator based MCC offloading framework, SIMDOM. The SIMDOM framework is based on re-compilation technique and direct vector-to-vector instruction mapping. The SIMD translator was integrated with the offloading module to enable smartphone applications to execute efficiently and seamlessly on heterogeneous ARM and x86 architectures.

- **Mathematical Model:** We developed a mathematical model for SIMDOM framework of translation and offloading of mobile applications. The mathematical model captures dynamics of the system based on system variables, such as device power rating, application instructions, device cycle per instruction efficiency, and network throughput. The mathematical model decides upon the feasibility of the cloud offload based on energy consumption and execution time parameters.

7.3 Significance of The Work

The SIMDOM framework provides multiple significant features that distinguish it from the existing MCC code offloading frameworks discussed as follows.

First, the SIMDOM framework focuses on multimedia applications. In particular, SIMD instruction based applications are translated and executed on cloud and cloudlet servers. Previous research works have not focused on multimedia application offloading in MCC.

Second, the SIMDOM framework is a pre-code offloading framework. The code is offloaded before compilation and recompiled for the server architecture along with the translation of SIMD system libraries. The majority of existing MCC offloading frameworks are dependent on application virtualization that has high computational overhead for compute-intensive multimedia applications.

Third, the SIMDOM framework incorporates the heterogeneity of MCC architectures through recompilation and translation of architecture-specific SIMD libraries. Existing code migration frameworks overlook the heterogeneity of the underlying ISAs which is not a realistic assumption in the MCC paradigm. Moreover, due to pre-compiled code migration, the SIMDOM framework avoids the overhead of binary translations faced by existing native code offloading frameworks.

7.4 Scope and Limitations

The SIMDOM framework provides high energy and performance efficiency for the SIMD instruction based applications with MCC cloudlet and cloud server support. The SIMDOM framework enables ARM-based native applications to be translated and executed on x86 based servers. The SIMD translator can be integrated into any application development platform including Android and iOS provided that the NEON-to-SSE header files are integrated at all steps of application development.

The SIMDOM framework is limited to native applications and can not be applied to virtualization based MCC offloading frameworks. Moreover, the SIMDOM framework targets SIMD instructions of ARM and x86 architecture and can not be applied to other ISAs, such as MIPS. Similarly, the SIMDOM framework does not provide the reverse x86 to ARM translations. The current implementation of the SIMD translator works on the NEON intrinsics functions. However, the proposed SIMD translations can be extended to the low-level assembly code by replacing the original code with corresponding assembly codes generated by the SIMD translator. Furthermore, SIMDOM framework does not take into account the mobile battery and user location profile while deciding upon the feasibility of cloud offload.

7.5 Future Work

This research was an effort to contribute to the existing body of knowledge in the field of MCC. However, a single thesis is not enough to address all the challenges to a particular domain. In the following lines, we detail the possible future works that can progress the domain of MCC code offloading.

1. A joint cloud and mobile resource efficiency framework can be formulated such that the MCC ecosystem works in a sustainable manner. The SIMDOM framework is based on the efficiency of the mobile device while ignoring the resource utilization at the cloud server. We will formulate and investigate a joint cost optimization framework for the MCC paradigm in our future work.

2. The SIMDOM framework utilizes pre-compiled code for translation and mapping of SIMD intrinsic functions. However, often for an application executing on the mobile, the code is compiled. The only cross-platform execution framework for compiled code is Qemu. The Qemu translates most of the SIMD instructions to scalar instructions, hence, leading to lower performance. As Qemu codebase is larger than 100000 lines of code, significant research effort is required to modify the current Qemu codebase and implement an efficient compiled code based MCC offloading framework.

3. The SIMDOM framework did not consider the utilization levels of resources in the system model and empirical results. The cloud and cloudlet servers can be analyzed for performance based on a number of client offload requests such that the point of resource over-utilization is found. Moreover, an offload request control algorithm can be devised to regulate the cloud/cloudlet server performance based on its resource utilization levels.

4. In this Ph.D. study, we proposed a framework for translation of ARM-based applications such that they can execute on x86 ISA based cloud servers. However, the application of translation of x86 based applications to ARM ISA also exist. A framework to provide the reverse translation of SIMDOM needs to be explored.

REFERENCES

Abadal, S., Martínez, R., Solé-Pareta, J., Alarcón, E., & Cabellos-Aparicio, A. (2016). Characterization and modeling of multicast communication in cache-coherent many-core processors. *Computers & Electrical Engineering*, *51*, 168 - 183.

Ahmad, R. W., Gani, A., Hamid, S. H. A., Xia, F., & Shiraz, M. (2015). A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues. *Journal of Network and Computer Applications*, *58*, 42–59.

Ahmed, E., Gani, A., Khan, M. K., Buyya, R., & Khan, S. U. (2015). Seamless application execution in mobile cloud computing: Motivation, taxonomy, and open challenges. *Journal of Network and Computer Applications*, *52*, 154–172.

Ahmed, E., Gani, A., Sookhak, M., Ab Hamid, S. H., & Xia, F. (2015). Application optimization in mobile cloud computing: Motivation, taxonomies, and open challenges. *Journal of Network and Computer Applications*, *52*, 52–68.

Altamimi, M. (2015). *A task offloading framework for energy saving on mobile devices using cloud computing* (Unpublished doctoral dissertation). University of Waterloo.

Arm® neon™ intrinsics reference. (2014). Retrieved from `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0073a/IHI0073A_arm_neon_intrinsics_ref.pdf`

Balan, R. K., Gergle, D., Satyanarayanan, M., & Herbsleb, J. (2007). Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on mobile systems, applications and services* (pp. 272–285).

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *Usenix annual technical conference, freenix track* (pp. 41–46).

Bianchi, G. (2000). Performance analysis of the ieee 802.11 distributed coordination function. *IEEE Journal on selected areas in communications*, *18*(3), 535–547.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., ... others (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, *39*(2), 1–7.

Blem, E., Menon, J., Vijayaraghavan, T., & Sankaralingam, K. (2015). Isa wars: Understanding the relevance of isa being risc or cisc to performance, power, and energy on modern architectures. *ACM Transactions on Computer Systems (TOCS)*, *33*(1), 3.

Boisvert, R. F., Moreira, J., Philippsen, M., & Pozo, R. (2001). Java and numerical computing. *Computing in Science & Engineering*, *3*(2), 18–24.

Butko, A., Garibotti, R., Ost, L., Lapotre, V., Gamatie, A., Sassatelli, G., & Adeniyi-Jones, C. (2015). A trace-driven approach for fast and accurate simulation of many-core architectures. In *Design automation conference (asp-dac), 2015 20th asia and south pacific* (pp. 707–712).

Butko, A., Garibotti, R., Ost, L., & Sassatelli, G. (2012). Accuracy evaluation of gem5 simulator system. In *Reconfigurable communication-centric systems-on-chip (recosoc), 2012 7th international workshop on* (pp. 1–7).

Camarasu-Pop, S., Glatard, T., & Benoit-Cattin, H. (2016). Combining analytical modeling, realistic simulation and real experimentation for the optimization of monte-carlo applications on the european grid infrastructure. *Future Generation Computer Systems*, *57*, 13–23.

Camargos, F., Girard, G., & Ligneris, B. (2008). Virtualization of linux servers. In *Proceedings of the linux symposium* (Vol. 2008).

Choi, M., & Lim, S.-H. (2016). x86-android performance improvement for x86 smart mobile devices. *Concurrency and Computation: Practice and Experience*, *28*(10), 2770–2780.

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., & Patti, A. (2011). Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on computer systems* (pp. 301–314).

Cisco. (2015). *Cisco visual networking index: Global mobile data traffic forecast update, 2015–2020* (Tech. Rep.). Sisco.

Clark, N., Hormati, A., Yehia, S., Mahlke, S., & Flautner, K. (2007). Liquid simd: Abstracting simd hardware using lightweight dynamic mapping. In *High performance computer architecture, 2007. ieee 13th international symposium on* (pp. 216–227).

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., & Bahl, P. (2010). Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on mobile systems, applications, and services* (pp. 49–62).

Dall, C., & Nieh, J. (2014). Kvm/arm: the design and implementation of the linux arm hypervisor. *ACM SIGARCH Computer Architecture News*, *42*(1), 333–348.

de Carvalho Jr, A. D., Rosan, M., Bianchi, A., & Queiroz, M. (2013). Fft benchmark on android devices: Java versus jni. *Nexus*, *7*, 1.

Deshane, T., Shepherd, Z., Matthews, J., Ben-Yehuda, M., Shah, A., & Rao, B. (2008). Quantitative comparison of xen and kvm. *Xen Summit, Boston, MA, USA*, 1–2.

Ding, J.-H., Chang, P.-C., Hsu, W.-C., & Chung, Y.-C. (2011). Pqemu: A parallel system emulator based on qemu. In *Parallel and distributed systems (icpads), 2011 ieee 17th international conference on* (pp. 276–283).

Do, V. (2011). *Security services on an optimized thin hypervisor for embedded systems* (Unpublished doctoral dissertation). Faculty of Engineering LTH at Lund University.

Dongarra, J. J., & Luszczek, P. (2005). *Introduction to the hpcchallenge benchmark suite*. Lawrence Berkeley National Laboratory.

Dongarra, J. J., Luszczek, P., & Petitet, A. (2003). The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, *15*(9), 803–820.

Ehringer, D. (2010). *The dalvik virtual machine architecture* (Vol. 4; Tech. Rep.). Google.

Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R., & Estrin, D. (2010). Diversity in smartphone usage. In *Proceedings of the 8th international conference on mobile systems, applications, and services* (pp. 179–194).

FELLOWS, K. M. (2014). *A comparative study of the effects of parallelization on arm and intel based platforms* (Unpublished doctoral dissertation). University of Illinois at Urbana–Champaign.

Fernando, N., Loke, S. W., & Rahayu, W. (2013). Mobile cloud computing: A survey. *Future Generation Computer Systems*, *29*(1), 84–106.

Flinn, J. (2012). Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, *7*(2), 1–103.

Flores, H., Hui, P., Tarkoma, S., Li, Y., Srirama, S., & Buyya, R. (2015). Mobile code offloading: from concept to practice and beyond. *Communications Magazine, IEEE*, *53*(3), 80–88.

Fu, S.-Y., Wu, J.-J., & Hsu, W.-C. (2015). Improving simd code generation in qemu. In *Proceedings of the 2015 design, automation & test in europe conference & exhibition* (pp. 1233–1236).

Fu, S.-Y., Wu, J.-J., Liu, P., Hong, D.-Y., & Hsu, W.-C. (2015). Simd code translation in an enhanced hqemu. In *Ieee international conference on parallel and distributed systems (icpads).*

Gordon, M. S., Jamshidi, D. A., Mahlke, S., Mao, Z. M., & Chen, X. (2012). Comet: code offload by migrating execution transparently. In *Presented as part of the 10th usenix symposium on operating systems design and implementation* (pp. 93–106).

Guo, Q., Chen, T., Chen, Y., & Franchettit, F. (2015). Accelerating architectural simulation via statistical techniques: A survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *35*(3), 433 - 446.

Guo, Y.-C., Yang, W., Chen, J.-Y., & Lee, J.-K. (2016, December). Translating the arm neon and vfp instructions in a binary translator. *Software: Practice and Experience*, *46 (12)*, 1591–1615.

Hong, D.-Y., Hsu, C.-C., Yew, P.-C., Wu, J.-J., Hsu, W.-C., Liu, P., ... Chung, Y.-C. (2012). Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the tenth international symposium on code generation and optimization* (pp. 104–113).

Hong, D.-Y., Wu, J.-J., Yew, P.-C., Hsu, W.-C., Hsu, C.-C., Liu, P., ... Chung, Y.-C.

(2014). Efficient and retargetable dynamic binary translation on multicores. *Parallel and Distributed Systems, IEEE Transactions on*, *25*(3), 622–632.

Hoque, M. A., Siekkinen, M., Khan, K. N., Xiao, Y., & Tarkoma, S. (2015). Modeling, profiling, and debugging the energy consumption of mobile devices. *ACM Computing Surveys (CSUR)*, *48*(3), 39.

Hsu, C.-C., Hong, D.-Y., Hsu, W.-C., Liu, P., & Wu, J.-J. (2015). A dynamic binary translation system in a client/server environment. *Journal of Systems Architecture*, *61*(7), 307–319.

Huang, H. (2011). *Idisa+: A portable model for high performance simd programming* (Unpublished doctoral dissertation). SIMON FRASER UNIVERSITY.

Jeffery, A. (2009). *Using the llvm compiler infrastructure for optimised, asynchronous dynamic translation in qemu* (Unpublished doctoral dissertation). University of Adelaide.

Jenkins, I. R. (2016). *Android benchmarking for architectural research* (Unpublished master's thesis). THE FLORIDA STATE UNIVERSITY COLLEGE OF ARTS AND SCIENCES.

Jiang, W., Mei, C., Huang, B., Li, J., Zhu, J., Zang, B., & Zhu, C. (2005). Boosting the performance of multimedia applications using simd instructions. In *Compiler construction* (pp. 59–75).

Khan, A. R., Othman, M., Madani, S. A., & Khan, S. U. (2014). A survey of mobile cloud computing application models. *Communications Surveys & Tutorials, IEEE*, *16*(1), 393–413.

Kim, E., Eom, H., & Yeom, H. Y. (2012). Asymmetry-aware load balancing for parallel applications in single-isa multi-core systems. *Journal of Zhejiang University SCIENCE C*, *13*(6), 413–427.

Kim, Y.-J., Cho, S.-J., Kim, K.-J., Hwang, E.-H., Yoon, S.-H., & Jeon, J.-W. (2012). Benchmarking java application using jni and native c application on android. In *Control, automation and systems (iccas), 2012 12th international conference on* (pp. 284–288).

Kosta, S., Aucinas, A., Hui, P., Mortier, R., & Zhang, X. (2012). Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 proceedings ieee* (pp. 945–953).

Koukoumidis, E., Lymberopoulos, D., Strauss, K., Liu, J., & Burger, D. (2011). Pocket cloudlets. *ACM SIGARCH Computer Architecture News*, *39*(1), 171–184.

Kumar, K., Liu, J., Lu, Y.-H., & Bhargava, B. (2013). A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, *18*(1), 129–140.

Kumar, K., & Lu, Y.-H. (2010). Cloud computing for mobile users: Can offloading computation save energy? *Computer*(4), 51–56.

Larabel, M., & Tippett, M. (2013). *Phoronix test suite.*

Lee, G., Park, H., Heo, S., Chang, K.-A., Lee, H., & Kim, H. (2015). Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th international symposium on microarchitecture* (pp. 521–532).

Lee, S., & Jeon, J. W. (2010). Evaluating performance of android platform using native c for embedded systems. In *Control automation and systems (iccas), 2010 international conference on* (pp. 1160–1163).

Li, J., Zhang, Q., Xu, S., & Huang, B. (2006). Optimizing dynamic binary translation for simd instructions. In *Proceedings of the international symposium on code generation and optimization* (pp. 269–280).

Limited, A. (2009). *Introducing neon™ development article* (Tech. Rep.). ARM Holdings.

Lomont, C. (2011). Introduction to intel advanced vector extensions. *Intel White Paper*.

Maleki, S., Gao, Y., Garzaran, M. J., Wong, T., & Padua, D. A. (2011). An evaluation of vectorizing compilers. In *Parallel architectures and compilation techniques (pact), 2011 international conference on* (pp. 372–382).

Manilov, S., Franke, B., Magrath, A., & Andrieu, C. (2015). Free rider: A tool for retargeting platform-specific intrinsic functions. In *Acm sigplan notices* (Vol. 50, p. 5).

Michel, L., Fournel, N., et al. (2011). Speeding-up simd instructions dynamic binary translation in embedded processor simulation. In *Design, automation & test in europe conference & exhibition (date), 2011* (pp. 1–4).

Mitra, G., Johnston, B., Rendell, A. P., McCreath, E., & Zhou, J. (2013). Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *Parallel and distributed processing symposium workshops & phd forum (ipdpsw), 2013 ieee 27th international* (pp. 1107–1116).

Moore, R. W., Baiocchi, J. A., Childers, B. R., Davidson, J. W., & Hiser, J. D. (2009). Addressing the challenges of dbt for the arm architecture. In *Acm sigplan notices* (Vol. 44, pp. 147–156).

Mustafa, S., Nazir, B., Hayat, A., Madani, S. A., et al. (2015). Resource management in cloud computing: Taxonomy, prospects, and challenges. *Computers & Electrical Engineering*, *47*, 186–203.

Nethercote, N., & Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Acm sigplan notices* (Vol. 42, pp. 89–100).

Nimmakayala, S. T. (2015). *Exploring causes of performance overhead during dynamic binary translation* (Unpublished doctoral dissertation). University of Kansas.

Niu, J., Song, W., & Atiquzzaman, M. (2014). Bandwidth-adaptive partitioning for distributed execution optimization of mobile applications. *Journal of Network and Computer Applications*, *37*, 334–347.

Nuzman, D., Dyshel, S., Rohou, E., Rosen, I., Williams, K., Yuste, D., … Zaks, A. (2011). Vapor simd: Auto-vectorize once, run everywhere. In *Proceedings of the 9th annual ieee/acm international symposium on code generation and optimization* (pp. 151–160).

Oh, H.-S., Kim, B.-J., Choi, H.-K., & Moon, S.-M. (2012). Evaluation of android dalvik virtual machine. In *Proceedings of the 10th international workshop on java technologies for real-time and embedded systems* (pp. 115–124).

Othman, M., Khan, A. N., Abid, S. A., Madani, S. A., et al. (2015). Mobibyte: an application development model for mobile cloud computing. *Journal of Grid Computing*, *13*(4), 605–628.

Penneman, N., Kudinskas, D., Rawsthorne, A., De Sutter, B., & De Bosschere, K. (2016). Evaluation of dynamic binary translation techniques for full system virtualisation on armv7-a. *Journal of Systems Architecture*.

Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., & Ioannidis, S. (2014). Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the seventh european workshop on system security* (p. 5).

Pommier, J. (2012). *Pretty fast fft.* Online. Retrieved from `https://bitbucket.org/jpommier/pffft`

Rehman, M. H. u., Liew, C. S., Wah, T. Y., Shuja, J., & Daghighi, B. (2015). Mining personal data using smartphones and wearable devices: A survey. *Sensors*, *15*(2), 4430–4469.

Robinson, G., & Weir, G. R. (2015). Understanding android security. In *Global security, safety and sustainability: Tomorrow's challenges of cyber security* (pp. 189–199). Springer.

Sartor, A. L., Lorenzon, A. F., & Beck, A. (2015). The impact of virtual machines on embedded systems. In *Computer software and applications conference (compsac), 2015 ieee 39th annual* (Vol. 2, pp. 626–631).

Satyanarayanan, M. (2015). A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets. *ACM SIGMOBILE Mobile Computing and Communications Review*, *18*(4), 19–23.

Satyanarayanan, M., Bahl, P., Caceres, R., & Davies, N. (2009). The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, *8*(4), 14–23.

Satyanarayanan, M., Schuster, R., Ebling, M., Fettweis, G., Flinck, H., Joshi, K., & Sabnani, K. (2015). An open ecosystem for mobile-cloud convergence. *Communications Magazine, IEEE*, *53*(3), 63–70.

Shaukat, U., Ahmed, E., Anwar, Z., & Xia, F. (2016). Cloudlet deployment in local wireless networks: Motivation, architectures, applications, and open challenges. *Journal of Network and Computer Applications*, *62*, 18–40.

Shen, B.-Y., Hsu, W.-C., & Yang, W. (2014). A retargetable static binary translator for the arm architecture. *ACM Transactions on Architecture and Code Optimization (TACO)*, *11*(2), 18.

Shuja, J., Bilal, K., Madani, S. A., Othman, M., Ranjan, R., Balaji, P., & Khan, S. U. (2016, June). Survey of techniques and architectures for designing energy-efficient data centers. *IEEE Systems Journal*, *10*(2), 507-519.

Shuja, J., Gani, A., Ahmad, R. W., Muhammad, H. u. R., Ahmed, E., Khan, K., & Ko, K. (2016, November). Towards native code offloading based mcc frameworks for multimedia applications: A survey. *Journal of Network and Computer Applications*, *75*, 335–354.

Shuja, J., Gani, A., Bilal, K., Khan, A. U. R., Madani, S. A., Khan, S. U., & Zomaya, A. Y. (2016, April). A survey of mobile device virtualization: Taxonomy and state of the art. *ACM Comput. Surv.*, *49*(1), 1:1–1:36.

Shuja, J., Gani, A., & Madani, S. A. (2014, December). A qualitative comparison of mpsoc mobile and embedded virtualization techniques. In *International conference of global network for innovative technology (ignite-2014), penang, malaysia.*

Shuja, J., Gani, A., Naveed, A., Ahmed, E., & Hsu, C.-H. (2016). Case of arm emulation optimization for offloading mechanisms in mobile cloud computing. *Future Generation Computer Systems*, -. doi: http://dx.doi.org/10.1016/j.future.2016.05.037

Shuja, J., Gani, A., Shamshirband, S., Ahmad, R. W., & Bilal, K. (2016). Sustainable cloud data centers: A survey of enabling techniques and technologies. *Renewable and Sustainable Energy Reviews*, *62*, 195–214.

Smartphone os market share, 2015 q2. (2016). online. Retrieved from `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`

Stallings, W. (2000). *Computer organization and architecture: designing for performance.* Pearson Education India.

Statista. (2015). *Facts and statistics about cloud computing.* Retrieved from `http://www.statista.com/topics/1695/cloud-computing/`

ur Rehman, M. H., Sun, C., Wah, T. Y., Iqbal, A., & Jayaraman, P. P. (2016). Opportunistic computation offloading in mobile edge cloud computing environments. In *2016 17th ieee international conference on mobile data management (mdm)* (Vol. 1, pp. 208–213).

Vincent, C., & Janin, Y. (2011). Proot: a step forward for qemu user-mode. In *1st international qemu users' forum* (p. 41).

Wang, Z., Liu, R., Chen, Y., Wu, X., Chen, H., Zhang, W., & Zang, B. (2011). Coremu: a scalable and portable parallel full-system emulator. *ACM SIGPLAN Notices*, *46*(8), 213–222.

Whaiduzzaman, M., Naveed, A., & Gani, A. (2016). Mobicore: Mobile device based cloudlet resource enhancement for optimal task response. *IEEE Transactions on Services Computing*, *PP*(99), 1-1. doi: 10.1109/TSC.2016.2564407

Wu, Z. P., Krish, Y., & Pellizzoni, R. (2013). Worst case analysis of dram latency in multi-requestor systems. In *Real-time systems symposium (rtss), 2013 ieee 34th* (pp. 372–383).

Xu, Y., & Mao, S. (2013). A survey of mobile cloud computing for rich media applications. *IEEE Wireless Commun.*, *20*(3), 1–0.

Yadav, R., & Bhadoria, R. S. (2015). Performance analysis for android runtime environment. In *Communication systems and network technologies (csnt), 2015 fifth international conference on* (pp. 1076–1079).

Yaqoob, I., Ahmed, E., Gani, A., Mokhtar, S., Imran, M., & Guizani, S. (2016). Mobile ad hoc cloud: A survey. *Wireless Communications and Mobile Computing*, *16*(16), 2572–2589.

Yeh, T.-C., Tseng, G.-F., & Chiang, M.-C. (2010). A fast cycle-accurate instruction set simulator based on qemu and systemc for soc development. In *Melecon 2010-2010 15th ieee mediterranean electrotechnical conference* (pp. 1033–1038).

Younge, A. J., Henschel, R., Brown, J. T., Von Laszewski, G., Qiu, J., & Fox, G. C. (2011). Analysis of virtualization technologies for high performance computing environments. In *Cloud computing (cloud), 2011 ieee international conference on* (pp. 9–16).

Yousafzai, A., Chang, V., Gani, A., & Noor, R. M. (2016). Directory-based incentive management services for ad-hoc mobile clouds. *International Journal of Information Management*, *36*(6, Part A), 900 - 906.

Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., & Yang, L. (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth ieee/acm/ifip international conference on hardware/software codesign and system synthesis* (pp. 105–114).

Zhang, X., Guo, Q., Chen, Y., Chen, T., & Hu, W. (2015). Hermes: a fast cross-isa binary translator with post-optimization. In *Proceedings of the 13th annual ieee/acm international symposium on code generation and optimization* (pp. 246–256).

Zhao, J., Nagarakatte, S., Martin, M. M., & Zdancewic, S. (2012). Formalizing the llvm intermediate representation for verified program transformations. In *Acm sigplan notices* (Vol. 47, pp. 427–440).

Zhu, W., Luo, C., Wang, J., & Li, S. (2011). Multimedia cloud computing. *Signal Processing Magazine, IEEE*, *28*(3), 59–69.

APPENDIX A: LIST OF PUBLICATIONS

International Scholarly Publications: This PhD venture has led to the publication and submission of multiple Journal and conference papers both as the first author and as a co-author. The articles published as first-author fulfill the requirements of the BSP scholarship and the Ph.D. thesis at University Malaya. The following list provides the complete details of the international research publications in ISI-indexed journals produced during this research venture.

First Author Published Journal Articles

1. **Shuja, J.**, Gani, A., et al., "Towards native code offloading based MCC frameworks for multimedia applications: A survey" Journal of Network and Computer Applications, 75 (2016): 335-354.

2. **Shuja, J.**, Gani, A., Naveed, A., Ahmed, E., Hsu, C.-H, "Case of ARM emulation optimization for offloading mechanisms in Mobile Cloud Computing" Future Generation Computer Systems, Accepted May 2016. http://dx.doi.org/10.1016/j.future.2016.05.037.

3. **Shuja, J.**, Gani, A., Bilal, K., Khan, A. U. R., Madani, S. A., Khan, S. U., Zomaya, A. Y., "A survey of mobile device virtualization: Taxonomy and state of the art" ACM Computing Surveys, 49(1), 1:1–1:36. 2016.

4. **Shuja, J.**, Bilal, K., Madani, S., Othman, M., Ranjan, R., Balaji, P., Khan, S.,"Survey of techniques and architectures for designing energy-efficient data centers" Systems Journal, IEEE, 10(2), 507-519. 2016.

5. **Shuja, J.**, Gani, A., Shamshirband, S., Ahmad, R. W., Bilal, K., "Sustainable cloud data centers: A survey of enabling techniques and technologies" Renewable and Sustainable Energy Reviews, 62, 195–214, 2016.

First Author Submitted Journal Articles

1. **Shuja, J.**, Gani, A., et al., "SIMDOM: A Framework for SIMD Instruction Translation and Offloading in Heterogeneous MCC Architectures" Submitted to Transactions on Emerging Telecommunication Technologies, 10 September 2016.

First Author Accepted Conference Articles

1. **Shuja, J.**, Gani, A., Madani, S. A., " A qualitative comparison of MPSoC mobile and embedded virtualization techniques" In International conference of global network for innovative technology (IGNITE-2014), Penang, Malaysia, 2014, arXiv preprint arXiv:1605.01168.

Journal Articles Accepted as a Co-author

1. Rehman, M. H. u., Liew, C. S., Wah, T. Y., **Shuja, J.**, Daghighi, B, "Mining personal data using smartphones and wearable devices: A survey" Sensors, 15(2), 4430–4469, 2015.

Conference Articles Accepted as a Co-author

1. Liaqat, M.; Ninoriya, S.; **Shuja, J.**; Ahmad, R. W. Gani, A. "Virtual Machine Migration Enabled Cloud Resource Management: A Challenging Task" arXiv preprint arXiv:1601.03854, 2016

Journal Articles Submitted as a Co-author

1. Atta ur Rehman Khan, Mazliza Othman, Abdul Nasir Khan, **Junaid Shuja**, Saad Mustafa, "Computation Offloading Cost Estimation in Mobile Cloud Application Models", submitted in Wireless Personal Communications, April 2016.

2. Atta ur Rehman Khan, Mazliza Othman, **Junaid Shuja**, Abdul Nasir Khan, Saad Mustafa, Shahbaz Akhtar, "Behavioral Trends and Mindset of Smartphone Users

Towards Adopting the Mobile Cloud Computing Paradigm", submitted in Universal Access in the Information Society, April 2016.

APPENDIX B: SAMPLE CODE FOR LINPACK AND LINPACKSIMD BENCHMARKS

Linpack:

```
m = n % 4;

If (m !=0)

{

for (i=0; i<m ; i++)

    dy[i] = dy[i] + da*dx[i];

If (n < 4) return;

}

for (i=m; i<n ; i+4)

{

 dy[i] = dy[i] + da*dx[i];

 dy[i+1] = dy[i+1] + da*dx[i+!];

 dy[i+2] = dy[i+2] + da*dx[i+2];

 dy[i+3] = dy[i+3] + da*dx[i+3];

}

#endif
```

LinpackSIMD:

```
#ifdef NEON

float cf[4];

float32x4_t x41, y41, c41, r41;

float32_t   *ptrx1 = (float32_t *)dx;

float32_t   *ptry1 = (float32_t *)dy;

float32_t   *ptrc1 = (float32_t *)cf;
```

```
for (i=0; i<4; i++)
{
cf[i] = da;
}
m = n % 4;
If (m !=0)
{
for (i=0; i<m ; i++)
    dy[i] = dy[i] + da*dx[i];
If (n < 4) return;
}
ptrx1 = ptrx1 + m;
ptry1 = ptry1 + m;
c41 = vld1q_f32(ptrc1);
for (i = m; i < n; i=i+4)
{
x41 = vld1q_f32(ptrx1);
y41 = vld1q_f32(ptry1);
r41 = vmlaq_f32(y41, x41, c41);
vst1q_f32(ptry1, r41);
ptrx1 = ptrx1 + 4;
ptry1 = ptry1 + 4;
}
#endif
```