Received April 19, 2017, accepted May 8, 2017, date of current version November 28, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2713818

Analysis of Vector Code Offloading Framework in Heterogeneous Cloud and Edge Architectures

JUNAID SHUJA[1], SAAD MUSTAFA[1], RAJA WASIM AHMAD[1], SAJJAD A. MADANI[1], ABDULLAH GANI[2], AND MUHAMMAD KHURRAM KHAN[3], (Senior Member, IEEE)
[1]Department of Computer Science, COMSATS Institute of Information Technology, Abbottabad 22060, Pakistan
[2]Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur 50603, Malaysia
[3]Center of Excellence in Information Assurance, King Saud University, Riyadh 12372, Saudi Arabia

Corresponding author: Muhammad Khurram Khan (mkhurram@ksu.edu.sa)

The authors extend their appreciation to the Deanship of Scientific Research at King Saud University for funding this work through Research Group Number (RG-288).

ABSTRACT Smartphones are computationally constrained compared with server devices due to their size and limited battery-based power. Compute-intensive tasks are often offloaded from smartphones to high-performance computing opportunities provided by nearby high-end cloud and edge servers. ARM architectures dominate smartphones, while x86 dominate server devices. The difference in architectures requires dynamic binary translation (DBT) of compiled code migration, which increases the task execution time on the cloud servers. Multimedia applications contain a large number of vector instructions (single instruction multiple data) that are compute and resource intensive. Vector instructions optimize application execution by parallel processing multiple data points in a single instruction. However, DBT of vector instructions losses the parallelism and optimization due to vector–scalar translations. We present and analyze a framework for pre-compiled vector instruction translation and offloading in heterogeneous compute architectures that avoids the execution overhead of compiled code offloading. The framework maps and translates ARM vector intrinsics to x86 vector intrinsics such that an application programmed for ARM architecture can be executed on the x86 architecture without any modification. We analyze the code offloading framework with static code analysis to determine the optimal compilers and corresponding compilation parameters. Moreover, we analyze the overhead of the vector instruction translator and application profiler. Furthermore, the comparative analysis based on increasing computational sizes reveals that our framework provides 78.8% energy efficiency as compared with existing code translation and offloading frameworks.

INDEX TERMS SIMD, code offloading, cloud, edge computing.

I. INTRODUCTION

Smartphones are commonly brought into service for basic computational and infotainment requirements. The expeditious increase in the power, performance, and utilization of smartphone devices has been observed in recent years [1]. Compute and resource-intensive application such as multimedia applications, voice, and image recognition, are widely utilized in smartphones. However, smartphones are relatively resource-constrained with respect to high-end devices functioning in edge and cloud networks due to their size and power limitations. The practice of computation and code offloading is often practiced in smartphone devices with the help of nearby cloud, cloudlet, edge and similar technologies. The computation and code offloading enable smartphones to function longer on limited battery power. The augmentation smartphones with server and edge network devices are called Mobile Cloud Computing (MCC) and Fog or Edge Computing respectively [2], [3].

Smartphone and high-end server Instruction Set Architectures (ISA) are heterogeneous. ARM based devices are dominantly utilized in smartphones while x86 architectures dominate the high-end server market [4], [5]. Therefore, we focus on ARM to x86 translations for mobile-cloud offload which can be further applied to other heterogeneous architecture. The techniques enabling offloading of a task from the mobile device to a high-end server while accommodating heterogeneity of architectures are: **(a)** system virtualization for Virtual Machine (VM) migrations, **(b)** application virtualization for platform independence, and **(c)** Dynamic Binary Translation (DBT) for native code migration [6]. We evaluated the overheads of offload enabling techniques in heterogeneous compute architectures in our earlier study [5].

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

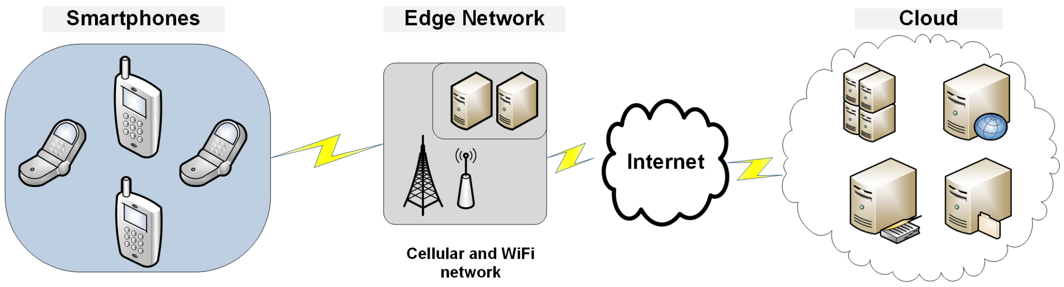

FIGURE 1. A generic scenario edge server and high-end cloud server offload.

VM migration based offloading leads to extraordinary network delay due to the size of migration load [7], [8]. Application virtualization results in high computational overhead as compared to native codes due to intermediate bytecode compilation. For native compiled code offloading, DBT is required. The overhead of DBT is very high as compared to the actual instance of instruction execution [5]. Moreover, existing DBT techniques perform vector-to-scalar translations that degrade the performance of vectorized applications [2], [9]. Figure 1 illustrates a generic scenario for a smartphone to edge and cloud offloading.

Data-level parallelism or vectorization techniques such as Single Instruction Multiple Data (SIMD) are applied to increase the performance of smartphone applications. Multimedia applications are specifically coded with SIMD instructions for fast performance. A single operation is executed on multiple data points in parallel in SIMD instructions. Multiple instruction fetch and decode cycles are reduced with the help of SIMD instructions resulting in both power and time efficiency. Changing the brightness of an image is an example of SIMD instructions where the same operation is performed on all N pixels. The performance of vectorized applications with respect to scalar counterparts can be up to four times better. Up to 25% of the code in smartphone multimedia based applications can be SIMD instructions [5], [10]. SIMD intrinsics/functions are platform specific. Therefore, it is not possible to write a vectorized smartphone based application that can be offloaded to heterogeneous compute platforms [11].

In this article, we extend our work on the SIMDOM framework while analyzing various modules of the framework for optimal execution parameters [12]. In our previous effort, we focused on the comparison of the SIMDOM framework with the state-of-the-art code offloading frameworks. In this article, we focus on the analysis of the modules of SIMDOM framework for execution parameters. The SIMDOM framework deviates from conventional smartphone offload enabling techniques for optimization and acceleration of pre-compiled SIMD instructions. The SIMD instruction translation and offloading framework for heterogeneous compute architectures enables a smartphone application to offload to a high-end server. As a result, execution of hardware accelerated SIMD instructions leads to smartphone energy and execution time economy. We provide the necessary details of vector instruction translation, application profiling, and network profiling modules of the code offloading framework while focusing on the module analysis and data collection methodology. The main contributions of this article are as follows:

- We analyze a vector instruction translator algorithm for heterogeneous compute architectures. The SIMD translator algorithm maps ARM SIMD instructions to x86 SIMD instructions such that vector-to-scalar translations are minimized. The analysis of the SIMDOM framework is performed while considering optimal compilers and corresponding parameters for application compilation, instruction translation, and application profiling overhead.
- We evaluate the performance of SIMD instructions on smartphones to demonstrate the effectiveness of SIMD intrinsics through multimedia benchmarks. Moreover, we evaluate computational overhead of DBT for compiled code offloading.
- We debate the network overhead for high-end cloud server and edge server devices to deliberate on the choice of edge and cloud paradigms for increasing computational workloads.

The rest of the paper is structured as follows. Section II lists the related work corresponding to code offloading frameworks and SIMD translation techniques in heterogeneous compute architectures. In Section III, we present details of a generic code offloading framework, its basic modules, and a preliminary data collection for the framework modules. Section IV details the evaluation methodology for the SIMDOM framework including details of benchmarks and devices. The analysis of SIMDOM framework in terms of application translation and profiling overhead is presented in Section V. In Section VI, we provide a detailed discussion on the outcomes of the analysis of vector instruction translation and offloading framework and highlight the future research directions.

II. RELATED WORK

Code and computation offloading techniques formulated for cloud technologies are now common to edge and fog computing paradigms [13]. The resource-constraint of smartphone devices and heterogeneity of computing architectures is a routine in the aforementioned distributed computing paradigms. Therefore, code and computation offloading are practiced whenever possible in ubiquitous computing scenarios [14]. Smartphone application offload frameworks are mostly enabled by system and application level virtualization techniques which result in high communication and computational overhead. CloneCloud [15] and ThinkAir [16] are examples application and system virtualization based code offloading frameworks. System virtualization is not supported by traditional smartphone OS's such as Android. Moreover, application virtualization restricts the application programmer development domain and leads to almost 50% performance loss as compared to non-virtualizable native C code [5], [17]. Researchers [18] proposed an architecture-aware compiled code offloading framework for smartphones. The framework avoids the overhead of virtualization techniques while offloading native code. The framework relies on LLVM compiler for intermediate code generation to support heterogeneous smartphone and high-end server architectures. The intermediate code binaries are compiled to the native hardware by LLVM back-end compilers at runtime. As intermediate codes are machine independent, cross-platform execution of the application is enabled.

Most of the compiled code offloading frameworks are dependent on DBT techniques for execution of instructions on heterogeneous compute platforms. Qemu is the most commonly utilized cross-platform translation and execution tool for heterogeneous architectures. While translating a vector instruction from guest ISA to host ISA, Qemu ignores the support for vector instructions in the host. Hence, a vector instruction is translated to multiple scalar instructions that consume more instruction cycles. Therefore, the basic design of DBT can be further enhanced for performance [19]. Most of DBT optimization efforts work towards parallelizing the execution of DBT process [2]. HQEMU [20] enhances Qemu performance with the help of LLVM optimizations that are applied to the code generation process for multi-threaded code. Fu et al. [21] added two methods to enhance the vector instruction translations in HQEMU. The first method enhanced the Qemu helper functions while allowing them to emit vectorized intermediate code. The second method utilized vector optimizations in code generation process. The LLVM optimizer in the HQEMU is modified to convert single level vector-to-scalar translations to two level vector-to-vector translations. Similar machine-code-to-low-level-virtual-machine (MC2LLVM) approach based on LLVM DBT optimizations were proposed in [22] and [23]. Intel Atom-based Android devices include a translation layer named Houdini that performs the vector-to-vector translation of ARM ISA based applications [24]. As Houdini is limited to Intel Atom based devices and closed-source, we can not compare it further with our framework.

The SIMDOM framework is different from previous vector code translation and offloading techniques in heterogeneous environments as it works on the non-compiled code. The SIMD translator utilizes a custom header file to map ARM vector instructions to x86 vector instructions enabling a multimedia application to offload seamlessly to the high-end server. In this article, we analyze each profiler module of SIMDOM framework and describe the data collection methodology. After deliberating on the basic modules of the SIMDOM framework, we analyze the SIMDOM framework from various aspects. First, we debate the effect of vector code on smartphone applications. Afterward, the LLVM and GCC compilers are compared for vector instruction generation. The overhead of vector instruction translation and application profiling overhead is debated in edge network and high-end server scenarios. The edge network and high-end servers are utilized in the evaluation to deliberate on the trade-off of resource proximity and network overhead. This work extends our previous efforts of surveying the MCC offloading frameworks [2], analyzing the problem of SIMD instruction translation in existing MCC offload enabling techniques [5], and comparing the SIMDOM framework with state-of-the-art code offloading frameworks [12].

III. SIMDOM FRAMEWORK

In this section, we detail the overall algorithm of the SIMDOM framework and the corresponding flow diagram. Furthermore, the modules of the SIMDOM framework are detailed along with the preliminary data collection for the framework analysis. The experimental setup utilized for the data collection is detailed in Section IV. We skip the details of SIMD translator as it was presented in previous article [12].

The SIMDOM framework starts by checking the cloud/edge server 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. 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 2.

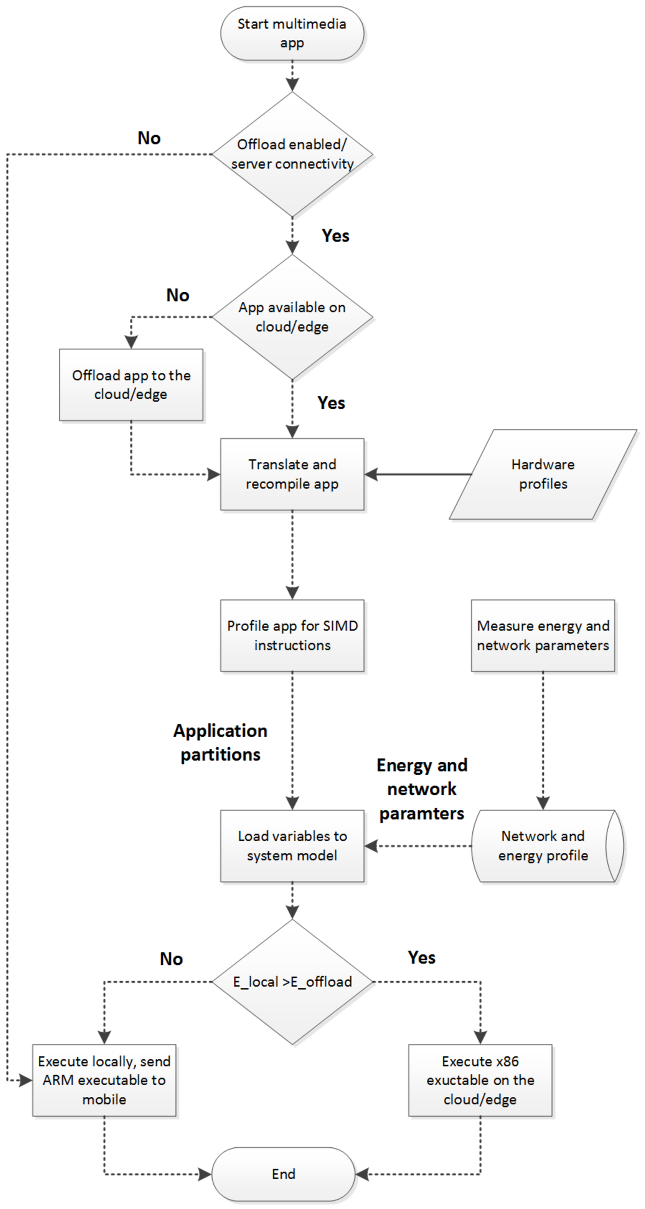
Start multimedia app
Offload enabled/ server connectivity
No
Yes
App available on cloud/edge
No
Offload app to the cloud/edge
Yes
Translate and recompile app
Hardware profiles
Profile app for SIMD instructions
Measure energy and network parameters
Application partitions
Energy and network paramters
Load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

FIGURE 2. Flow diagram of SIMDOM: A framework for pre-compiled multimedia application offload.

A. APPLICATION PROFILER

The application profiler is the basic component of a code offloading framework in heterogeneous compute architectures. The basic task of an application profiler is to determine the optimal execution parameters of the application in both local and remote execution scenarios. The optimal execution parameters generally define the compiler parameters that lead to the most vectorized profile of the application. Further, if the application is to be partitioned, the application profiler determines the vector part of the application to be executed on the cloud, and the scalar part of the application to be executed on the local device. The application profiler decides on optimal offload parameters through two steps. First, the application profiler utilizes application code binaries obtained through the 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 is conducted. The static analysis results in calculation of the percentage of SIMD instructions in both ARM and x86 binaries for the corresponding compilers. 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 profile. However, the same is not true for 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. We found out through profiling of candidate benchmarks that only a subset of 20-30 SIMD instructions are utilized repeatedly. Therefore, the profiling program was limited to the active subset to reduce profiling overhead. Table 1 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.

TABLE 1. 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**

Our framework provides three simplistic application partition options based on the 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 ideal candidate for the offload. However, complete application can also be offloaded if it is programmed with a focus on higher utilization of SIMD intrinsics. Multimedia applications are usually candidate for complete application offload. Therefore, application partitioning is relatively simple task for SIMD based multimedia applications.

B. ENERGY PROFILER

Energy profiler profiles the energy consumption of the device corresponding to various instances of task execution. Hardware, software, and hybrid energy profiling techniques are used on the smartphone to measure the energy utilization corresponding to executing application. 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 [25], [26]. We employed PowerTutor [27] 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 Wi-Fi in various power states (idle, busy) to find subsystem baseline power ratings. The energy profiler samples the device energy consumption in idle state (p_i) while limiting the number of background processes in LCD off state. The energy profiler executes multimedia applications to measure the power consumption of the device while executing computational applications (p_m). The energy profiler sends and receives data from the cloud/edge server to estimate the network energy cost (p_c) for the Wi-Fi networks. The energy profiler provides these values as input to the offload module for the offload decision process. The SIMDOM framework does not consider the device battery levels in the decision of offloading. The results of data collection for the energy profiler are presented in Table 2 while executing the vector benchmarks.

TABLE 2. Experimental evaluation for p_i and p_m.

Benchmark	Input	p_i mean	p_m mean
MathlibSIMD	all	28mW	585mW
SpeedSIMD	all	27mW	578mW
LinpackSIMD	1200	27mW	576mW
FFTSIMD	all	25mW	571mW

The mean values in Table 2 are generated over a run of five trails. We created a data transfer utility that sends data to the remote servers 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. The mean values for the down-link and up-link p_c were found to be 485mW and 461mW respectively.

C. NETWORK PROFILER

The network profiler determines up-link and down-link throughput of the network that are fed to the offload module. 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 high-end cloud and edge servers using Wi-Fi network and saves the historical results as future inputs to 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.

TABLE 3. Experimental evaluation for RTT and throughput.

Connection	RTT minimum	RTT maximum	RTT mean	Throughput maximum
Edge-up	8.85ms	16.48ms	12.53ms	5.23Mbyte/sec
Edge-down	1.52ms	18.10ms	4.17ms	10.22Mbyte/sec
Cloud-up	7.90ms	20.03 ms	15.78ms	4.15Mbyte/sec
Cloud-down	2.43ms	28.01ms	14.94ms	4.38Mbyte/sec

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. To measure the network parameters (up and downlink 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. The maximum throughput for the link can be calculated as,

$$throughput \leq \frac{RWIN}{RTT} \tag{1}$$

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 edge and cloud server. The values of minimum, maximum, average, mean deviation of the RTT, and maximum throughput based on the average RTT for cloud and edge server are listed in table 3.

D. OFFLOAD MANAGER

The offload manager decides upon the feasibility of code offload for cloud and edge servers. 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 also selects the application partition which is most suitable for offload. The offload manager acquires the evaluation parameters and feeds them to the system model detailed in [12]. The offload manager maintains a connection with the remote cloud and edge servers 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). The offload manager sends an offload request to the cloud server after the determination of the offload parameters.

IV. EVALUATION METHODOLOGY

The SIMDOM framework is designed to enable execution of multimedia applications on heterogeneous compute architectures. A prototype system is developed and deployed

TABLE 4. Experimental devices.

Device	Processor	Memory	OS	BogoMIPS
Mobile device - Samsung Galaxy S2 (LE)	1.2GHz*2 (ARMv7)	1GB	4.4.4	1194.54
Edge Server (ES)	2.3GHz*4 (x86)	4GB	Linux 14.04	4654.87
Cloud Server (CS)	2.4GHz*8 (x86)	32GB	Linux 14.04	4788.05
Qemu Edge Server (QES)	1GHz max (ARMv7)	512MB	linaro-nano 3.0	471.61
Qemu Cloud Server (QCS)	1GHz max (ARMv7)	512MB	linaro-nano 3.0	591.76

on a high-end OpenStack cloud server and an edge server to evaluate the SIMDOM framework. A local network server located in close proximity of the smartphone device within the same LAN constituted as the edge server. A remote server located far from the smartphone device within another network acted as the high-end cloud server. Multiple application benchmarks based on SIMD instructions were formulated to analyze the performance of SIMDOM. 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.

A. DEVICES

We utilized multiple communicate and compute scenarios to test our framework rigorously in mobile cloud and edge environments. The specification of the devices and their computational resources is provided in Table 4.

The mobile device is equipped with a Li-Ion 1650mAh removable battery. We incorporated Wi-Fi network based communications 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.

B. APPLICATION BENCHMARKS

We utilized four application benchmarks for our experimental evaluation. Several constraints restricted the selection of application benchmarks for our framework evaluation. These contraints were **(a)** presence of SIMD intrinsic functions in application benchmarks, **(b)** preference for open-source benchmarks for cross-validation of our results, and **(c)** execution of benchmark on Qemu for comparison. We utilized a set of multimedia benchmarks comprising of four applications, namely, Mathlib, Speed, Linpack, and FFT in the experiments. We utilized two versions of each benchmark; **(a)** a simple version based on scalar instructions compiled without compiler optimizations and **(b)** a SIMD version that replaces scalar instructions in the original version with SIMD intrinsics. Moreover, the SIMD version of benchmarks is compiled with optimizing and auto-vectoring option.

V. RESULTS AND DISCUSSION

This section presents the empirical analysis of the SIMDOM framework. The SIMDOM framework enables execution of SIMD based applications in heterogeneous compute architectures. An application offloaded from mobile device is recompiled, translated, and executed on cloud server while the mobile device waits in low-power state. The empirical analysis is conducted to determine the optimal application execution parameter in both local and remote scenarios. The five subsections below find answers to the following five research questions through empirical analysis.

- How much the vector applications are efficient over their scalar counterparts and how efficient are current native code translation frameworks?
- Which compiler (GCC or LLVM) provides better code vectorization for ARM and x86 architectures?
- Which compiler has higher code vectorization and translation overhead?
- What is the overhead of application profiling before the offload decision?
- What is the impact of benchmark computation size on the efficiency of SIMDOM framework in mobile edge and cloud scenarios?

A. SIMD OPTIMIZATIONS

In this section, we present the case of SIMD instruction optimization in heterogeneous MCC architectures by revealing the overhead of SIMD instruction translation in Qemu. We utilize SIMD centric FFT and scientific computation centric Mathlib benchmarks to implicate the application optimizations of SIMD based benchmarks. 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. However, such theoretical bounds are impossible to achieve due to dependencies in instruction cycles leading to in or out-of-order execution of the instructions. 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. We compare and execute the benchmarks on the physical ARM based mobile device (Samsung Galaxy S2) and emulated ARM board (OMAP3 emulated in Qemu on Optiplex755 server). GCC compiler was utilized in compilation of the application benchmarks utilized in below subsections.

1) MATHLIB

We executed the Mathlib benchmark on the physical and emulated devices. The physical device represents the normal execution of the application benchmark while the emulated device represents the ARM to x86 translations. The result

TABLE 5. Mathlib and MathlibSIMD comparison on ARM physical and translated 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

of the Mathlib execution time and SIMD instruction is listed in table 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 version of the benchmark shows 24.12% and 12.45% time efficiency on the physical and emulated system respectively compared to basic scalar benchmark. 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 inefficient 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.

2) 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 ARM physical and ARM emulated devices are shown in Figure 3.

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 ARM emulated system reduces to 48.5% on average.

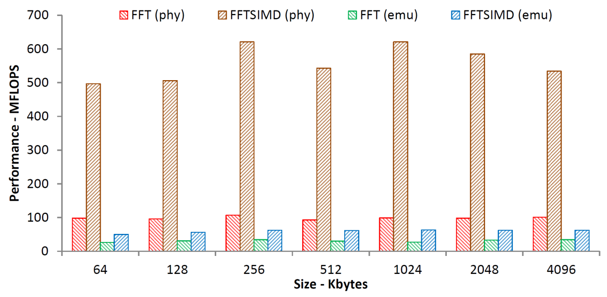

FIGURE 3. FFT and FFTSIMD comparison on ARM physical and translated systems.

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 translation 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 subsection 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. Moreover, the aforementioned results show considerable performance overhead in the translation of compiled code offloading. 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.

B. COMPARISON OF COMPILERS FOR SIMD INSTRUCTION

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

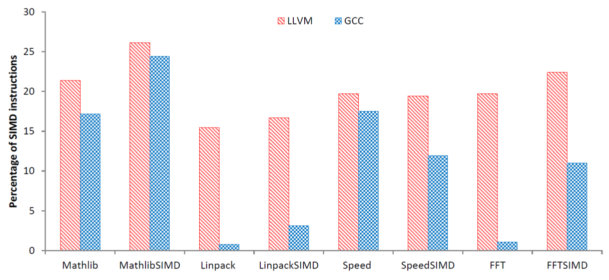

FIGURE 4. Analysis of ARM GCC and LLVM\clang compilers for application benchmarks.

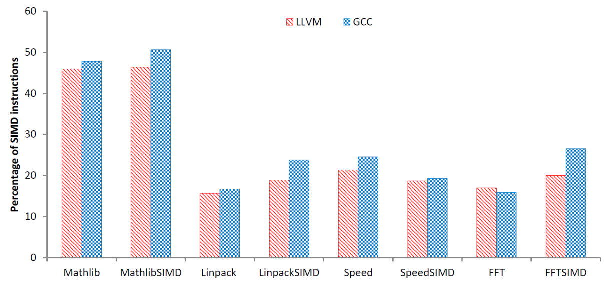

FIGURE 5. Analysis of x86 GCC and LLVM\Clang compilers for application benchmarks.

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 4 and 5 provide the comparison of ARM and x86 compilers for the production of SIMD instructions for 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 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 [23], [28].

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 edge 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.

C. SIMD TRANSLATOR OVERHEAD

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 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 edge server are listed in Figure 6.

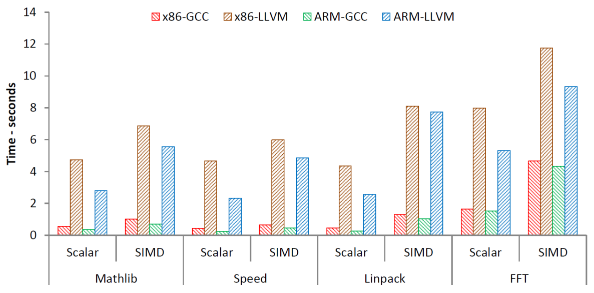

FIGURE 6. SIMD translator overhead on edge server: comparison of compilers.

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 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.

We found that the application translation overhead is approximately 10% lower on the cloud server than edge server for most of the benchmarks due to its higher computational capability. 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.

D. APPLICATION PROFILER OVERHEAD

The overhead of application profiler occurs in two main tasks. Firstly, the overhead occurs during compilation and translation of the application as listed in previous subsection. Secondly, the overhead occurs while profiling the application for the percentage of SIMD instructions. 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 edge server is illustrated in Figure 8.

The high percentage of SIMD instructions is achieved with auto-vectorization flags. Therefore, the overhead of profiling the SIMD benchmark is high compared to basic counterparts. 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 [29],

$$I = \frac{T \times MIPS \times 10^6}{CPI} \tag{2}$$

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.

E. IMPACT OF COMPUTATIONAL SIZE

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. The results of this section aim to quantify the of application benchmark (computational) size on the efficiency of the MCC offloading frameworks. Five execution scenarios were created with the help of a smartphone, a remote cloud and an edge server. The smartphone represented local execution (LE) when code is not offloaded. Two of the scenarios represented SIMDOM prototype of edge and cloud server (ES and CS respectively). Two scenarios represented compiled code translation and offloading based on Qemu framework for edge and cloud servers (QES and QCS respectively). Qemu provides the state-of-the art comparison as most widely utilized heterogeneous architecture translation framework [30].

1) ENERGY

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

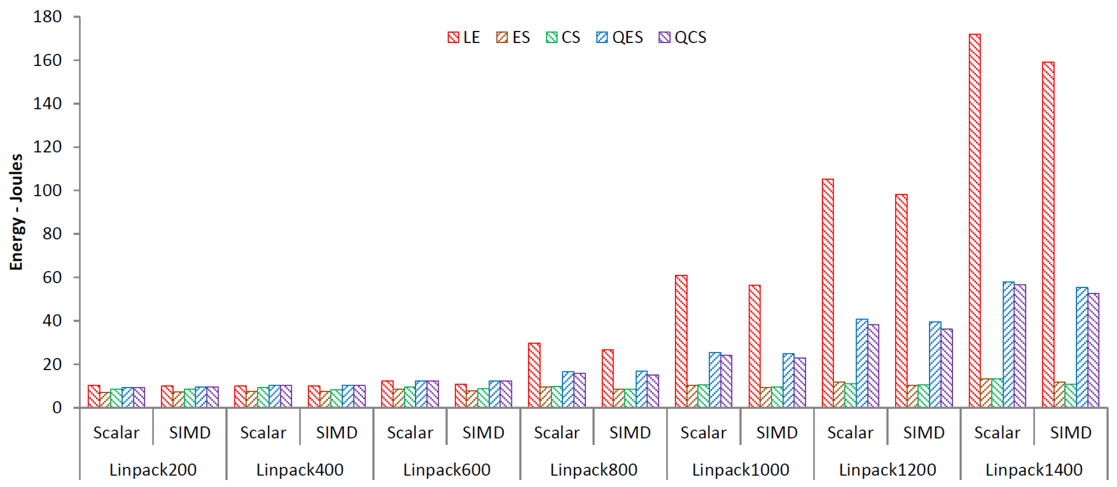

FIGURE 7. Energy consumption: linpack benchmark on variable input matrices of size $N * N$.

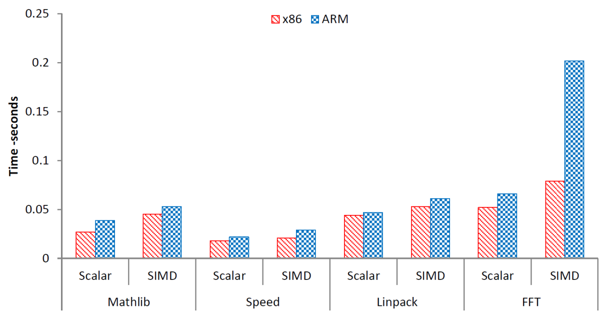

FIGURE 8. Application profiler overhead on edge server.

of performance. The result of the energy consumption of the Linpack benchmark for different execution scenarios are depicted in Figure 7.

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 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 edge server 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. On average, SIMDOM edge server provides 59.17% and 48.51% energy efficiency than local execution and Qemu edge server execution.

The comparison of cloud and edge servers for SIMDOM framework depicts that for lower computations (200 × 200), the edge server provides 17.57% energy efficiency. On the contrary, for higher computations (1400 × 1400), the cloud server provides 8.50% energy efficiency. Hence, edge and fog technologies are proffered for lower computations while cloud technologies are preferred for large computations.

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, 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.

2) TIME

We investigate the application execution time for variable size inputs. We utilize the Linpack benchmark as its input matrices can be varied. The result of the execution times of the Linpack benchmark are depicted in Figure 9.

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 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

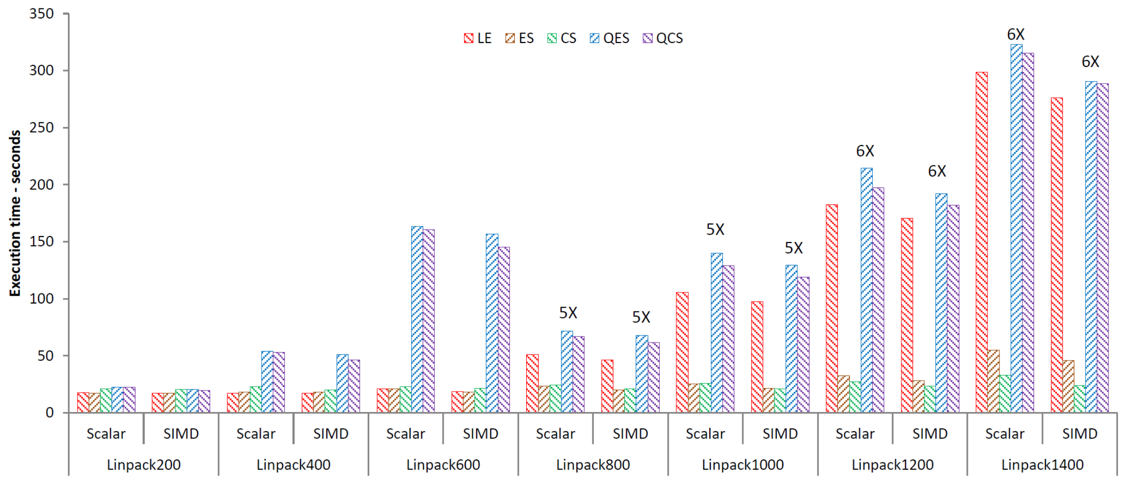

FIGURE 9. **Execution time of linpack benchmark on variable input matrices of size $N * N$.**

the input matrix of 1200 × 1200 and 1400 × 1400. The stable performance of the SIMDOM framework is due to the computational power of the edge 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 edge provides 66.24% efficiency than the local execution. For the input matrices 200 × 200, 400 × 400, and 600 × 600, the SIMDOM framework on edge server 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.

After comparing edge and cloud servers for time efficiency for SIMDOM framework, we find that for lower computations (200 × 200), the edge server provides 17.76% time efficiency. On the contrary, for higher computations (1400 × 1400), the cloud server provides 66.50% time efficiency. Therefore, for low computations, nearby edge servers can be more feasible than the cloud servers in terms of execution time. There are two reasons behind the higher feasibility of edge network for lower computations. Firstly, for smaller workloads, the network overhead of reaching the distant cloud dominates the computational power. Secondly, the edge and cloud servers utilized in our experimental setup have similar computational power (BogoMIPS). Any increase in the computational power of the cloud server will result in higher time and energy efficiency than the edge server.

The SIMD versions of the benchmarks show significantly lower execution times than the basic versions. The efficiency of the SIMD versions increases 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 edge server. On the contrary, the Qemu provides 9.63% time efficiency for the SIMD versions on the edge server.

VI. CONCLUSION

In this article, we presented and analyzed a vector instruction offloading framework (SIMDOM) in heterogeneous compute architectures. The analysis of the SIMDOM framework was conducted on subjects of vector instruction generation, the overhead of vector instruction generation, compilation, active vector instructions in multimedia applications, and application profiling. 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 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 for both ARM and x86 architectures. Therefore, for short-term optimization, GCC compiler is preferred for the application execution on both mobile and cloud devices. Further, the GCC compiler is 9.38% efficient in the production of SIMD instructions than the LLVM compiler for x86 ISA. Overall, the x86 compilers produce an average of 26.80% SIMD instructions as compared to 15.49% for ARM compilers for all benchmarks. Therefore, in terms of vector instructions, pre-compiled code offloading to high-end servers can be preferred. The experimental results also revealed that for smaller workloads, the edge server provided higher time and energy efficiency as compared to the cloud server. However, for larger workloads, the cloud server yields higher efficiencies. The SIMDOM framework leads to higher SIMD instruction translation efficiency compared to Qemu translator as revealed by SIMD to scalar application benchmark energy and time ratios.

REFERENCES

[1] E. Ahmed, A. Gani, M. Sookhak, S. H. Ab Hamid, and F. Xia, "Application optimization in mobile cloud computing: Motivation, taxonomies, and open challenges," *J. Netw. Comput. Appl.*, vol. 52, pp. 52–68, Jun. 2015.

[2] J. Shuja *et al.*, "Towards native code offloading based mcc frameworks for multimedia applications: A survey," *J. Netw. Comput. Appl.*, vol. 75, pp. 335–354, Nov. 2016.

[3] M. Peng, S. Yan, K. Zhang, and C. Wang, "Fog-computing-based radio access networks: Issues and challenges," *IEEE Netw.*, vol. 30, no. 4, pp. 46–53, Jul./Aug. 2016.

[4] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generat. Comput. Syst.*, vol. 29, no. 1, pp. 84–106, 2013.

[5] J. Shuja, A. Gani, A. Naveed, E. Ahmed, and C.-H. Hsu, "Case of arm emulation optimization for offloading mechanisms in mobile cloud computing," *Future Generat. Comput. Syst.*, to be published, doi: http://dx.doi.org/10.1016/j.future.2016.05.037

[6] E. Ahmed, A. Gani, M. K. Khan, R. Buyya, and S. U. Khan, "Seamless application execution in mobile cloud computing: Motivation, taxonomy, and open challenges," *J. Netw. Comput. Appl.*, vol. 52, pp. 154–172, Jun. 2015.

[7] J. Shuja *et al.*, "A survey of mobile device virtualization: Taxonomy and state-of-the-art," *ACM Comput. Surv.*, vol. 49, no. 1, p. 1, Apr. 2016.

[8] J. Shuja *et al.*, "Survey of techniques and architectures for designing energy-efficient data centers," *IEEE Syst. J.*, vol. 10, no. 2, pp. 507–519, Jun. 2016.

[9] S. T. Nimmakayala, "Exploring causes of performance overhead during dynamic binary translation," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. Kansas, Lawrence, KS, USA, 2015.

[10] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms," in *Proc. IEEE 27th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum (IPDPSW)*, May 2013, pp. 1107–1116.

[11] S. Manilov, B. Franke, A. Magrath, and C. Andrieu, "Free rider: A tool for retargeting platform-specific intrinsic functions," in *Proc. 16th ACM SIGPLAN/SIGBED Conf. Lang., Compil. Tools Embedded Syst. (CD-ROM)*, 2015, p. 5.

[12] J. Shuja *et al.*, "Simdom: A framework for SIMD instruction translation and offloading in heterogeneous mobile architectures," *Trans. Emerg. Telecommun. Technol.*, to be published, doi: http://dx.doi.org/10.1002/ett.3174

[13] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 12, pp. 3590–3605, Dec. 2016.

[14] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[15] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.

[16] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 945–953.

[17] C. Lichtenau, P. Oehler, and P. H. Roth, "Systems for a cloud, analytics, mobile and social era," in *Proc. CHIPS*, vol. 2. 2016, p. 125.

[18] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim, "Architecture-aware automatic computation offload for native applications," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 521–532.

[19] L. Michel, N. Fournel, and F. Pétrot, "Speeding-up SIMD instructions dynamic binary translation in embedded processor simulation," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, 2011, pp. 1–4.

[20] D.-Y. Hong *et al.*, "HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores," in *Proc. 10th Int. Symp. Code Generat. Optim.*, 2012, pp. 104–113.

[21] S.-Y. Fu, J.-J. Wu, P. Liu, D.-Y. Hong, and W.-C. Hsu, "Simd code translation in an enhanced HQEMU," in *Proc. IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2015, pp. 507–514.

[22] Y.-C. Guo, W. Yang, J.-Y. Chen, and J.-K. Lee, "Translating the ARM neon and VFP instructions in a binary translator," *Softw. Pract. Exper.*, vol. 46, no. 12, pp. 1591–1615, Dec. 2016.

[23] S.-Y. Fu, J.-J. Wu, and W.-C. Hsu, "Improving SIMD code generation in QEMU," in *Proc. Design, Autom. Test Eur. Conf. Exhibit.*, 2015, pp. 1233–1236.

[24] M. Choi and S.-H. Lim, "x86-android performance improvement for x86 smart mobile devices," *Concurrency Comput., Pract. Exper.*, vol. 28, no. 10, pp. 2770–2780, 2016.

[25] R. W. Ahmad, A. Gani, S. H. A. Hamid, F. Xia, and M. Shiraz, "A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues," *J. Netw. Comput. Appl.*, vol. 58, pp. 42–59, Dec. 2015.

[26] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao, and S. Tarkoma, "Modeling, profiling, and debugging the energy consumption of mobile devices," *ACM Comput. Surv.*, vol. 48, no. 3, p. 39, 2015.

[27] L. Zhang *et al.*, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Oct. 2010, pp. 105–114.

[28] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *Proc. Int. Conf. Parallel Archit. Compil. Techn. (PACT)*, 2011, pp. 372–382.

[29] W. Stallings, *Computer Organization and Architecture: Designing for Performance*. Hoboken, NJ, USA: Pearson Education India, 2000.

[30] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. FREENIX Track, USENIX Annu. Tech. Conf.*, 2005, pp. 41–46.

JUNAID SHUJA received the M.S. degree from the COMSATS Institute of Information Technology (CIIT), Abbottabad, in 2012, and the Ph.D. degree from the University of Malaya in 2017. He was an Assistant Professor with CIIT. His primary research interest is ARM emulation and SIMD instruction cross-platform execution and other research interests encompass topics, such as data center energy efficiency, sustainable cloud computing, ARM and GPU-based servers for energy efficient cloud computing, and cloud computing in general.

SAAD MUSTAFA received the B.S. and M.S. degrees in computer science from the COMSATS Institute of Information Technology, Abbottabad, Pakistan, in 2007 and 2010, respectively, where he is currently pursuing the Ph.D. degree with the Department of Computer Science. His current research interests include resource management, energy efficient systems, cloud computing, and wireless networks.

RAJA WASIM AHMAD received the master's degree from the COMSATS Institute of Information Technology, Abbottabad, Pakistan, under the COMSATS Merit Scholarship Program, and the Ph.D. degree in computer science from the University of Malaya, under the Bright Spark Scholarship Program. He is currently an Assistant Professor with the COMSATS Institute of Information Technology. His current research interests include mobile application energy profiling, energy efficient computational offloading, cloud resource allocation, VM migration, network performance, application's QoS on low bandwidth networks, and energy efficient cloud data centers.

SAJJAD A. MADANI received the M.S. degree in computer science from the Lahore University of Management Sciences and the Ph.D. degree from the Vienna University of Technology. He is currently with the COMSATS Institute of Information Technology as an Associate Professor. He has authored or co-authored over 70 papers in peer-reviewed international conferences and journals. His areas of interest include low power wireless sensor network and green computing.

ABDULLAH GANI received the bachelor's and master's degrees from the University of Hull, U.K., and the Ph.D. degree from the University of Sheffield, U.K. He is currently a Full Professor with the Department of Computer System and Technology, University of Malaya. He has vast teaching experience in various educational institutions locally and abroad, including schools, teaching college, the Ministry of Education, and universities. He has authored over 150 academic papers in conferences and respectable journals. He is currently involved in mobile cloud computing with a High Impact Research Grant of U.S. 1.5 million for the period of 2011–2016.

MUHAMMAD KHURRAM KHAN (SM'12) is currently a Full Professor with the Center of Excellence in Information Assurance, King Saud University, Saudi Arabia. He has edited seven books and proceedings published by Springer-Verlag and IEEE. He has authored or co-authored over 300 papers in international journals and conferences, and he is an Inventor of several patents. His current research interests include cybersecurity, biometrics, multimedia security, Internet of Things, cloud computing security, and digital authentication. He is a fellow of the IET (U.K.), a fellow of the BCS (U.K.), a fellow of the FTRA (South Korea), a member of the IEEE Technical Committee on Security & Privacy, and a member of the IEEE Cybersecurity Community. He was a recipient of several national and international awards for his research contributions. In addition, he has been granted several national and international funding projects in the field of Cybersecurity.