Programmatic Effect of Optimized Smali Code on Saving Energy of Android Applications

Marwa Dahdouh
PhD Student
Dept. Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo, Syria

Amer Bouchi
Assist Professor
Dept. Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo, Syria

Souheil Khawatmi
Associate Professor
Dept. Systems and Computer Networks
Faculty of Informatics Engineering
University of Aleppo, Syria

Mouhamad Ayman Naal
Associate Professor
Dept. Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo, Syria

ABSTRACT

This paper presents the effect of saving Android application execution time on saving energy consumed by optimized applications. An algorithm for optimizing instructions on a Smali code-level proposes to provide execution time. The Smali optimization algorithm relies on replacing high execution times instructions with lower execution times ones and equivalent in behavior. MySMALI compiler is designed to support the proposed optimization algorithm and applied on Android applications. Optimized APK files are generated for optimized applications. Measurements of APKs execution times are taken. Measurements prove that the percentage of optimization in execution time is approximately 26.27%.

The paper provides code-level estimates of the energy consumption of Android applications. A programmatic method about reading operating system files is applied to determine resource consumption by the applications. Energy measurements are also recorded by a power monitor (PowerTutor) for Android-based mobile platforms. The measurements of resources (Memory, CPU, Disk) consumption prove that the optimized compiler helps to save the consumption percentage of Android applications about 19.9%. The memory consumed is provided by the optimized compiler to approximately 20000 Kbyte and 31.7 KB size of files. The time that the optimized process of application consumes from the CPU time is reduced from 26% to 5%. The results demonstrate that the providing execution times of applications can save energy consumed to approximately 8.4%, and can save the power consumption by up to 14%.

General Terms

Compiler Efficiency, Bytecode Optimization, Saving Energy, Android Applications Performance, Resource Utilization.

Keywords

Smali Code Optimization, Energy Consumption, Power Usage Measurement, Optimized Compiler, Execution Time.

1. INTRODUCTION

With the rapid development of mobile devices, energy consumption has become a more and more important issue [1], [2]. It has become necessary for software developers to take into account the energy consumed by their applications. Developers should also know the impact of application implementation on battery life [3]. Battery life is one critical computing resource for mobile applications. Energy saving has become an increasing requirement imposed to meet the needs of mobile device users in energy saving applications [4]. Measurement of software energy consumption is expensive in terms of hardware and difficult in terms of expertise [5]. it is critical to analyze the energy consumption of Android applications [6]. Several attempts have been made to determine and measure energy consumption at code-level, which helps developers know and determine the effect of code modification on the energy consumed by the application [7-9]. The impact of code obfuscation, code smells and refactoring is studied on the energy consumption of several Android applications [10-12]. Some efforts are made to understand the correlation between execution time and consumption energy of Android applications [13]. Energy saving is discussed as a result of shorter execution times [14]. Programming methodologies are used to measure the energy consumed by applications [15-17]. This research comes to meet the energy saving requirements of Android applications. For this purpose, an optimized compiler is designed to reduce the energy consumption of Android applications. The optimized compiler relied on optimizing Smali code on bytecode instructions. The optimizing process leads to reduce Android application execution time.

The rest of the paper is organized as follows. Section 2 explains the optimizing process of Smali code. Next, in section 3 The optimization algorithms supported in the compiler are explained. An overview of the method for determining resource consumption and energy measurement is described in section 4. Finally, the experimental results of energy consumption measurements are displayed in section 5. The experimental results are discussed and analyzed in section 6 and conclusion is given in section 7.

2. SMALI CODE OPTIMIZATION

The Smali language file is identified as the target file in the optimization process. The execution times of all Smali instructions are studied and compared. Instructions with the lowest execution time are specified. The proposed optimized algorithm of Smali instructions is based on replacing high execution time instructions with lower execution time instructions [18]. Android applications are written and the optimized compiler is applied to generate corresponding Smali code [19]. Special structures are proposed to be adopted in the designing of the compiler. Compound statements (conditional, jump, selection, repetition) are processed. The proposed structures are adopted in the optimization process. Structures of instruction blocks are studied in different cases (overlapping or singular). Figure 1, shows the stages of optimization process.

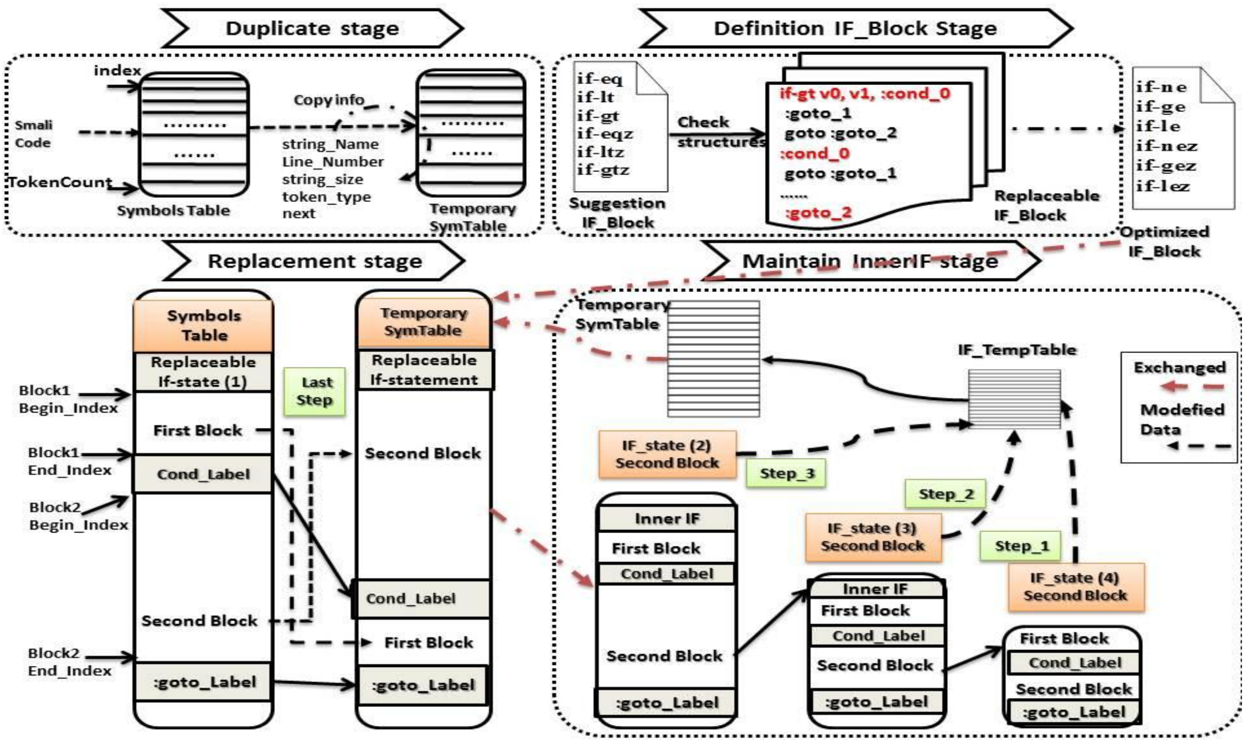

Figure 1: stages of the optimizing process on Smali code

The replacing instruction blocks process of replaceable structures is carried out as four stages. First stage, a temporary symbols table is created from the original symbols table resulting from lexical analysis phase of the compiler. The second stage, the proposed and replaceable instruction structures are identified, ensuring that an application maintains the same required behavior. The next two stages are the replacement and manipulate the internal structures. The optimization algorithm is applied at the level of structure definition and conditional blocks replacement. The instruction block that appears at the end of the input file is replaced before replacing the instruction that is presented in the first, as will be explained in the next section.

The optimization algorithm replaces conditional instruction blocks with lower execution time blocks. Conditional statements within Smali language are classified into IF and IFZ classes. A special structure is declared for each type of conditional statement. The optimization algorithm handles the structure of each class taking into account the presence of overlapping structures. Table 1, shows the proposed structures of IF Block statements.

Table 1. The proposed structures of IF instructions

Typedef	Struct declaration	Description

IFInfo_struct	char *if_name;
	int Line;
	int index;
	char *cond_label;
	int LblLine;
	int Lblindex;
	struct IFInfo_struct *next;
		Structure of each IF statement

OptIF_struct	int *If_Line;
	int *cond_Line;
	int block1_BIndex;
	int block1_EIndex;
	int block2_BIndex;
	int block2_EIndex;
	struct OptIF_struct * next;
		optimized IF Structure

3. PROPOSED OPTIMIZED ALGORITHM

The optimization process is done according to two algorithms. First algorithm analyzes the source code of the compiler's input file. Second algorithm performs the replacing instruction blocks.

3.1 Conditional Structures

The conditional structures differ according to their type (IF or IFZ) and their overlap with other structures by (if-else) statements. The optimized compiler applies the structure definition algorithm and blocks replacement algorithm. Three main conditional structures are defined as follows.

3.1.1 IF Instruction Blocks

The first block begins with IF statements (if-eq, if-lt, if-gt) and ends with a (:cond_name) statement. This block includes instructions (gotoLab_name1) and (goto :gotoLab_name2). While second block begins with an instruction that immediately follows (:cond_name) statement. This block includes instructions (goto :gotoLab_name1) and (gotoLab_name2). The statement (gotoLab_name2) defines the end of this block. The formula of this block is:

IF_statement reg$_1$, reg$_2$, :cond_name # statements

:gotoLab_name$_1$ # statements

goto :gotoLab_ name$_2$ # statements

:cond_name

goto :gotoLab_ name$_1$ # statements

:gotoLab_ name$_2$

3.1.2 IFZ Instruction Blocks

The first block begins with IFZ statements (if-eqz, if-ltz, if-gtz) and ends with a statement (:cond_name). This block includes (:gotoLab_name1). The second block begins with the instruction that follows (:cond_name) and includes (goto :gotoLab_name1).

3.1.3 IF Nested Blocks

The first block begins with IF and ends with (:cond_name). This block includes (goto :gotoLab_name1). The second block begins with the instruction that follows (:cond_name) statement and includes (goto :gotoLab_name1). The goto statement within this block exactly the same as the instruction in the first block. This block ends with (gotoLab_name1) statement. Label of goto jump is programmatically distinguished from goto/16 and goto/32 jump instructions.

3.2 Structure Definition Algorithm

This Proposed algorithm identifies the proposed and replaceable conditional block structures. The structure of (IF, IFZ) blocks and nested conditional instruction blocks are defined to be replaced. A linked list of proposed and replaceable statements is produced. The replaceable conditional instructions are added to the list according to the sequence they appear in the source code file. The instructions adding process is done at the front of the list. This makes the last replaceable instruction placed first in the list and treated first by the optimized algorithm. Figure 2, Shows steps of structures definition algorithm.

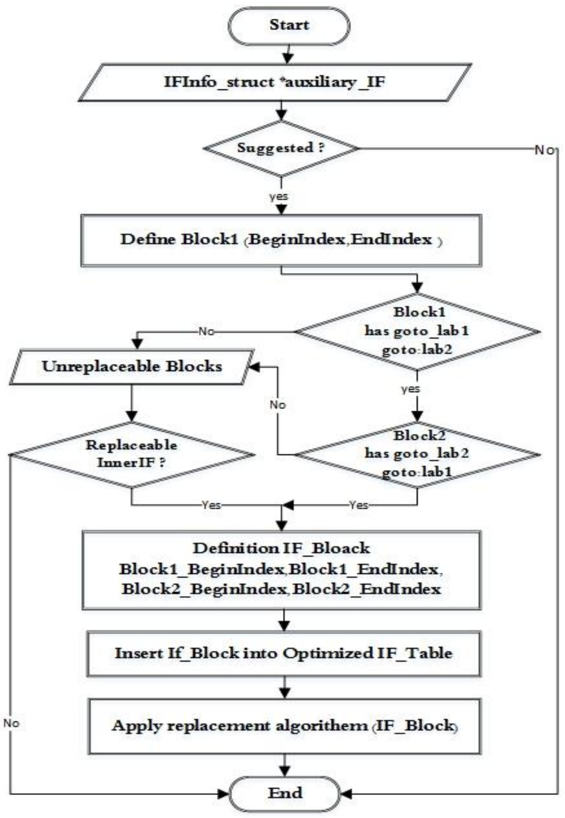
Start
IFInfo_struct *auxiliary_IF
Suggested ?
yes
No
Define Block1 (BeginIndex,EndIndex)
Block1 has goto_lab1 goto:lab2
No
yes
Unreplaceable Blocks
No
Replaceable InnerIF ?
Block2 has goto_lab2 goto:lab1
Yes
Yes
Definition IF_Bloack Block1_BeginIndex,Block1_EndIndex, Block2_BeginIndex,Block2_EndIndex
Insert If_Block into Optimized IF_Table
Apply replacement algorithem (IF_Block)
No
End

Figure 2. Flowchart of Structure definition algorithm

3.3 Blocks Replacement Algorithm

This Proposed algorithm replaces the high execution time instructions with the lower execution time instructions. The algorithm changes the structure of conditional instructions within the linked list that results from the definition algorithm. The conditional instruction that is included at the end of the compiler's input file is first handled according to the replacement algorithm. Each replaceable structure is divided into two instruction blocks. The replacement process is done by switching the order of instructions within the blocks. So, the instructions for the first block are placed where the instructions for the second blocks are placed and vice versa. The resulting instruction structures are stored within the temporary symbols table structure to keep the original symbols table information unchanged. Figure 3, Shows steps of the optimized algorithm for replacing (IF, IFZ) instructions blocks with the suggested equivalent blocks. Optimized applications rely on the conditional statements (if-ge, if-le, if-nez, if-gez, if-lez).

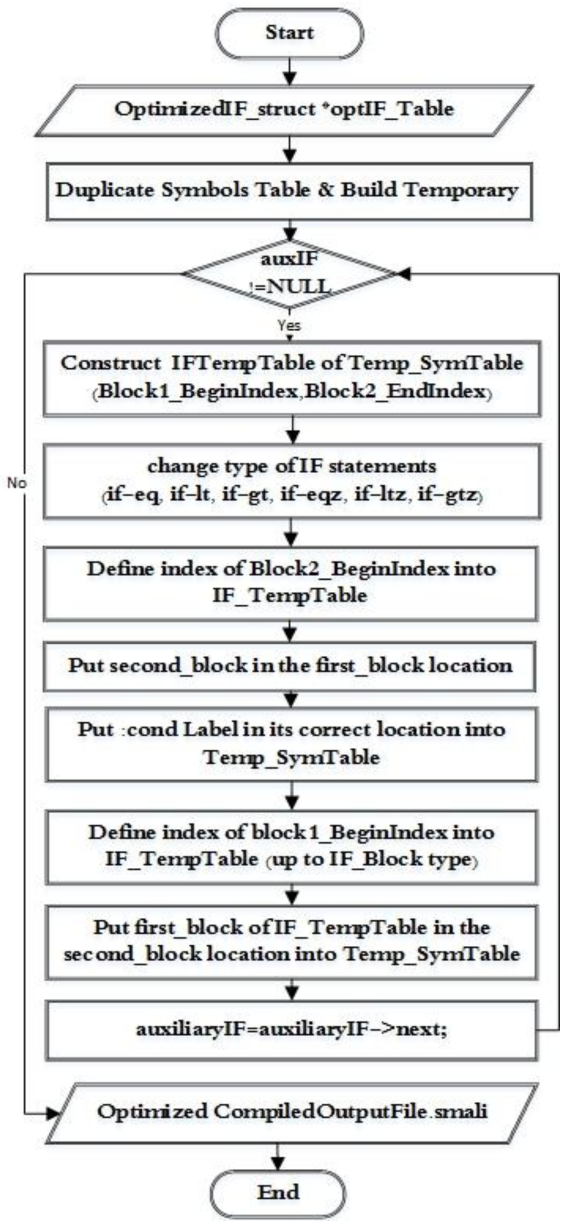
Start
OptimizedIF_struct *optIF_Table
Duplicate Symbols Table & Build Temporary
auxIF !=NULL
Yes
No
Construct IFTempTable of Temp_SymTable (Block1_BeginIndex,Block2_EndIndex)
change type of IF statements (if–eq, if–lt, if–gt, if–eqz, if–ltz, if–gtz)
Define index of Block2_BeginIndex into IF_TempTable
Put second_block in the first_block location
Put :cond Label in its correct location into Temp_SymTable
Define index of block1_BeginIndex into IF_TempTable (up to IF_Block type)
Put first_block of IF_TempTable in the second_block location into Temp_SymTable
auxiliaryIF=auxiliaryIF–>next;
Optimized CompiledOutputFile.smali
End

Figure 3. Flowchart of IF blocks replacement algorithm

4. ENERGY ACCOUNTING

In this paper, the effect of optimizing the Smali code on resources and energy consumption is studied. The energy measurement methods differ between Hardware-based and Software-based [20], [21]. The power consumed by Android applications can be determined by reading system files [22], [23]. Power monitor tools are used to take accurate measurements of the energy consumed [24], [25]. There are several challenges to measuring and calculating source code energy consumption [26]. PowerTutor is a real time system and application power monitor. It provides accurate real-time power consumption estimates for power components including CPU and LCD display as well as GPS, Wi-Fi [27], [28]. In this paper, PowerTutor (V1.4) is used to enables real-time information over all time about energy consumed by unit of joules and power calculation by unit of mille-Watt.

5. RESOURCES CONSUMPTION

CPU usage and memory consumption by the process are defined during the execution period. The Smali instructions that will be optimized with the compiler are included within a specific process.

5.1 System Files

The files in root of /proc/ have various information about the overall state of the system. The information in a /proc/ file is generated on the fly when the file is read. The /proc/[PID] pseudo-file system is created in order access to a ton of kernel data accessible. Three files are read.

5.1.1 /proc/uptime

This file includes the uptime of a system and the time spent in idle process (in seconds).

5.1.2 /proc/[PID]/stat

This file includes status information about the process and informs how many jiffies have been executed by a single process.

5.1.3 /proc/stat

This file tells how many jiffies the CPU has executed in total.

5.2 Programming Methods

Some methods are programmed to determine the effect of optimizing code on energy saving and resources consumption. The methods use data from operative system files stored under /proc/. Three methods are written to measure resources consumption by a specific process.

5.2.1 Get_processTime()

The process consumption time of CPU is determined. Process identifier is defined by calling (android.os.Process.myPid()). The first value (uptime) from "/proc/uptime" is used. The values (utime, stime, cutime, cstime, starttime) from "/proc/[PID]/stat" are used to calculate elapsed CPU time spent in user and kernel code, measured in jiffies. Both hertz and jiffies are converted to seconds in the calculations. A jiffie is clock tick, system's hertz is a number of ticks per second.

5.2.2 GetMemory_Usage()

The process identifier is defined by calling (getProcessMemoryInfo(PID)). The Memory consumption of a specific process can be determined by calling: (dalvikPrivateDirty, dalvikSharedDirty, dalvikPss, nativePrivateDirty, nativeSharedDirty, nativePss, otherPrivateDirty, otherSharedDirty, otherPss, getTotalPrivateDirty, getTotalSharedDirty, getTotalPss).

5.2.3 Get_processTime()

The "/proc/stat" and "/proc/[PID]/stat" files are read to calculate the CPU usage of a process. The line of "/proc/[PID]/stat" file contains (52) numerical values with different meaning. The line from "/proc/stat" file contains (9) different values.

The consumption measurements are taken from the moment the process is performed for 100 seconds. The values are recorded over equal 3-second intervals. Programmatically, CountDownTimer(100000,500) is declared and its methods (onTick, onFinish) are overridden.

6. EXPERIMENTAL RESULTS

Several applications are implemented that include different cases of conditional instructions. Some applications include overlapping structure that is repeated a number of times. All proposed conditional instructions are studied and tested within iterative loops.

The corresponding Smali instructions is generated for original APKs using a reverse engineering tool (APK_Easy Tool v1.55). MySMALI compiler is used to generate optimized Smali code. Table 2, shows the original and optimized code. The optimized code contains an if-eqz statement repeated three times with overlapping structures.

Table 2. Comparison original and optimized Smali code

source code	Optimized Code
if-eqz v13, :cond_0	if-nez v13 , :cond_0
.line 194	const-wide/16 v13 , 0x2
const-wide/16 v8, 0xa	cmp-long v13 , v8 , v13
goto :goto_1	if-nez v13 , :cond_1
.line 196	const-wide/16 v13 , 0x3
:cond_0	cmp-long v13 , v8 , v13
const-wide/16 v13, 0x2	if-nez v13 , :cond_2
cmp-long v13, v8, v13	const-wide/16 v8 , 0x28
if-eqz v13, :cond_1	.line 204
.line 197	:cond_2
const-wide/16 v8, 0x14	.line 200
goto :goto_1	const-wide/16 v8 , 0x1e
.line 199	goto :goto_1
:cond_1	.line 202
const-wide/16 v13, 0x3	:cond_1
cmp-long v13, v8, v13	.line 197
if-eqz v13, :cond_2	const-wide/16 v8 , 0x14
.line 200	goto :goto_1
const-wide/16 v8, 0x1e	.line 199
goto :goto_1	:cond_0
.line 202	.line 194
:cond_2	const-wide/16 v8 , 0xa
const-wide/16 v8, 0x28	goto :goto_1
.line 204	.line 196
:goto_1	:goto_1

A case study is performed for applications that include IFZ statements. The if-eqz instruction is studied in several cases and repeated a different number of times with (else) statement. The execution times of applications are measured with an average of 1000 repetitions per execution. Figure 4, shows a comparison of execution times between APK files before and after applying the optimized compiler. The results show that the execution time of if-eqz APKs can be saved about 543.7 nano seconds. The optimization percentage of execution time is up to 26.27.

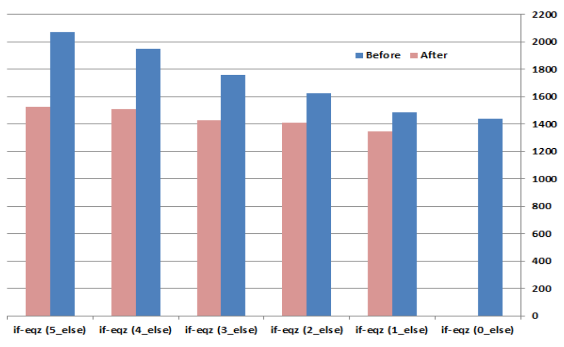

Figure 4: Comparison execution times of (if-eqz) APKs

All conditional instructions are studied in a similar way. Figure 5, shows a comparison of APK files for applications with nested if-ltz structures. The results of if-ltz APKs, show that it is possible to provide 395.4ns of execution time and the percentage of optimization may be up to 19.8%.

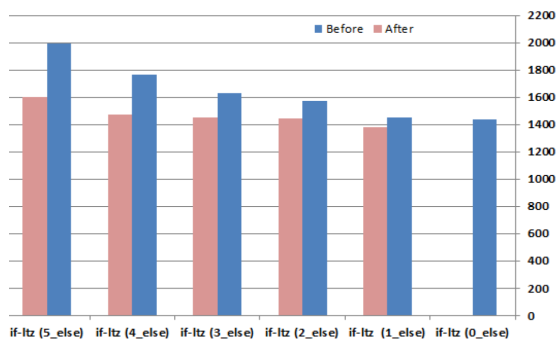

Figure 5: Comparison execution times of (if-ltz) APKs

Another case study is performed for applications that include IF (if-eq, if-lt, if-gt) statements. Figure 6, shows a comparison of APK files for applications with nested (if-eq).

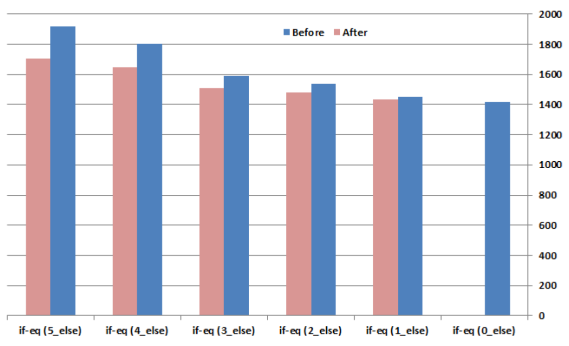

Figure 6: Comparison execution times of (if-eq) APKs

In figure 7, a comparison of APKs for applications with nested (if-lt) structures is displayed. The percent optimization of if-lt case ranges from 3.27% to 19.045% percent.

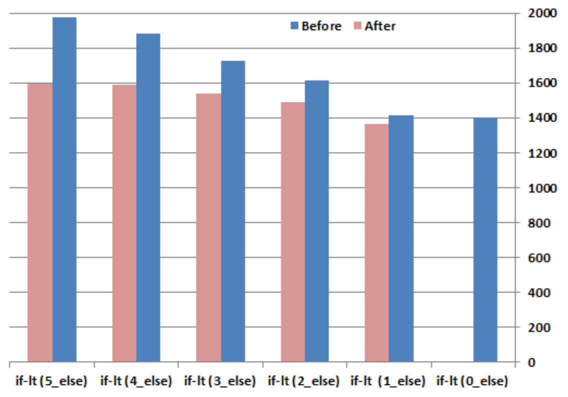

Figure 7: Comparison execution times of (if-lt) APKs

Table 3, shows the percentage of optimization the if-lt statement. The structure of a one-way conditional instruction isn't replaceable because it causes a change in instructional behavior.

Table 3. Comparison averages of execution times between original and optimized APKs (nano_second)

cases	Before	After	profit time	percent %
if-lt (0)	1402.71	---	---	---
if-lt (1)	1416.12	1369.71	1402.711	3.276
if-lt (2)	1615.48	1489.08	46.406	7.824
if-lt (3)	1731.25	1544.08	126.401	10.810
if-lt (4)	1887.04	1589.70	187.163	15.757
if-lt (5)	1977.48	1600.85	297.343	19.045

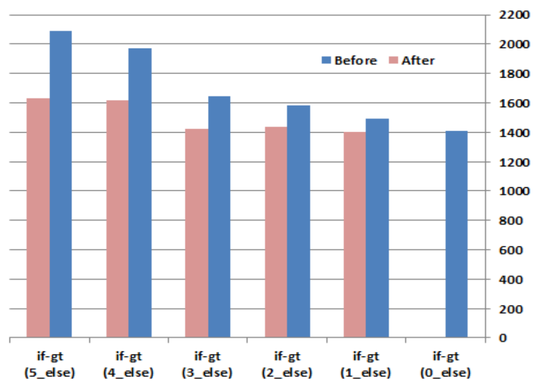

Figure 8: Comparison execution times of (if-gt) APKs

Figure 8, shows a comparison of APK files for applications with nested (if-gt). In (if-gt) case study, the results demonstrate that the optimization process for the If statement provided execution time of approximately 457.55ns and the percentage of optimization is 21.89%.

As a result, cases study shows that optimized files consume less execution time. Then, resources consumption of applications is studied before and after the Smali code optimization process, and the results are compared. Memory consumption, process time consumed, and CPU consumption are studied. Consumption measurements are recorded for 100 seconds and values are taken at equal intervals of 3 seconds. Resource consumption are studied for applications with the if-eqz instruction. Conditional if-eqz instruction is repeated within five overlapping structures. The study demonstrated that the execution of application more than once gives the same results for memory consumption values. As shown in Figure 9.

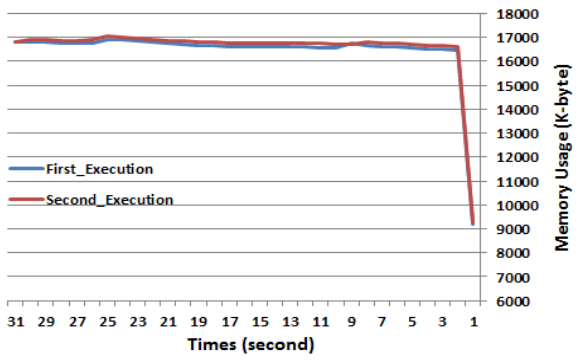

Figure 9: Process memory consumption of Android application before optimization process

In figure 10, the results show a decrease in memory consumption by the files resulting from the optimized application. The profit of memory is about 20000 Kbyte. Figure 11, displays a comparison of CPU time that the process takes before and after the code optimization. Figure 12, Shows a comparison of CPU usage before and after the optimization process. The results confirm a reduction in resource consumption of application with optimized instructions.

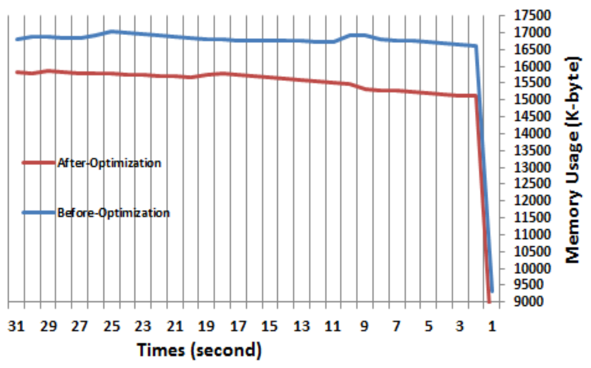

Figure 10: Process memory consumption of Android application before and after the optimization

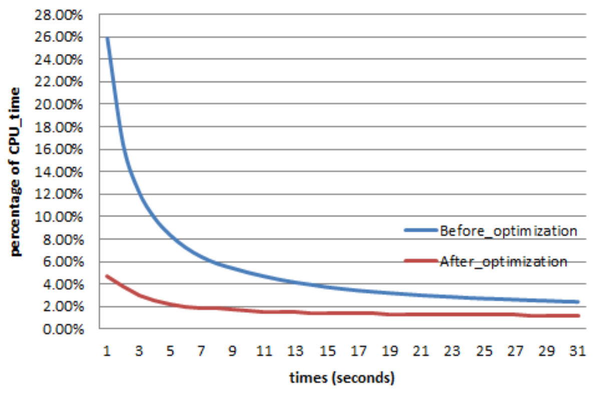

Figure 11: Comparison of CPU time consumption

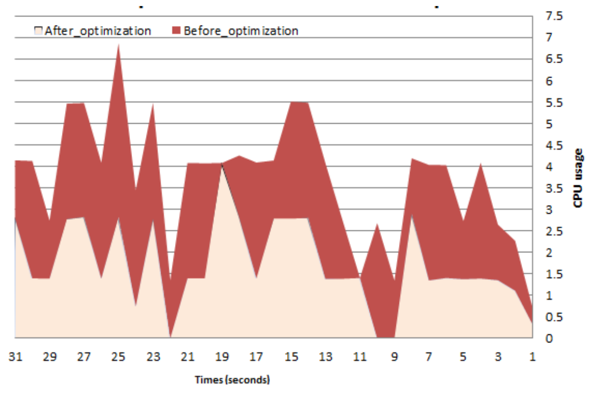

Figure 12: Comparison of CPU usage of process

To make a quantitative comparison of power consumption, a PowerTutor application is used. Consumption is studied for an application that includes conditional instructions with repeated structures. The required instructions are programmed into a method *onHandleWork* of a class *JobIntentService*. IntentService is advanced background service that create a separate background thread. The thread executes the instructions during the execution period of an application.

The measurements of energy consumption are recorded and compared. Figure 13, shows a comparison power calculation percentage between original and optimized APKs. The comparison of energy consumption measurements for optimized applications demonstrates that optimized code consumes less power, as figure 14 shows. Figure 15, illustrates that power consumption measured by mille-Watt is also decreased compared to that of unimproved applications.

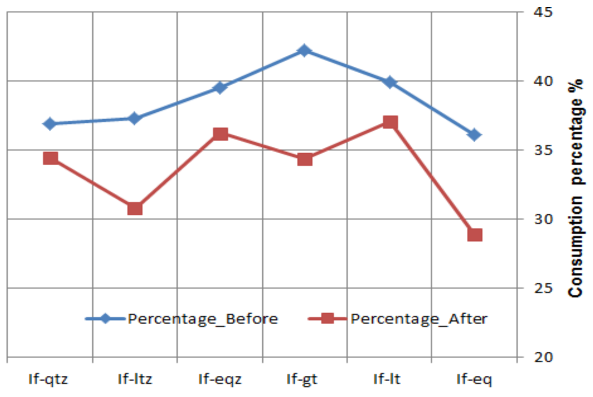

Figure 13: Comparison power calculation percentage

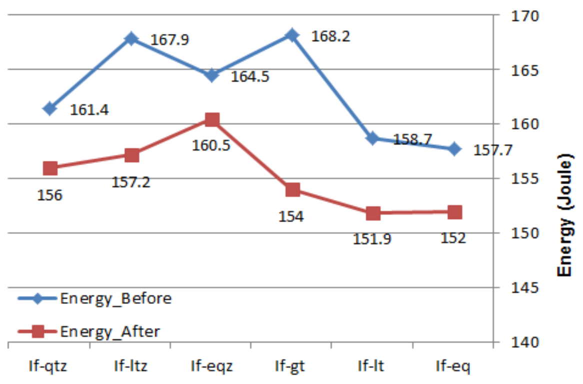

Figure 14: Comparison of energy consumption

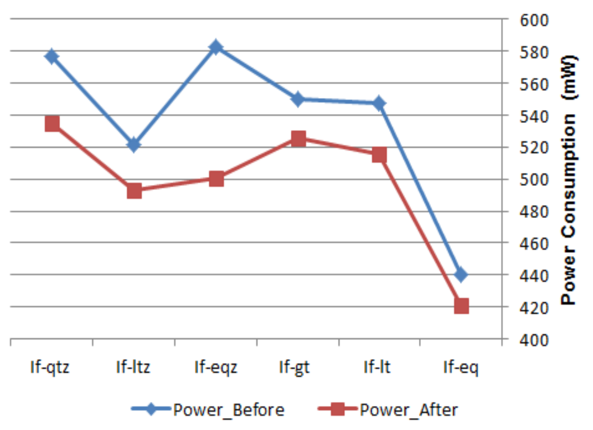

Figure 15: Comparison of power consumption (mw)

Table 4. Size of files after and before optimization

Files	SizeBefore	Size After	Diff Size
If_ltState.apk	**1.5 MB**	**1.5 MB**	**-31.6 KB**
res	294.7 KB	344.9 KB	50.2 KB
classes.dex	2.2 MB	2.2 MB	6.6 KB
AndroidManifest	2.2 KB	2.6 KB	352 B
resources.arsc	212.2 KB	212.4 KB	144 B
META-INF	97.1 KB	96.2 KB	-930 B

The effect of optimizing the Smali code on file size and saving disk space is studied. The results confirmed a slight saving in the size of application files after optimizing the code. As Table 4, displays.

7. DISCUSSION AND ANALYSIS RESULTS

APK files of conditional statements for original and optimized applications are generated and executed. Energy measurements are recorded every 5 minutes period. Energy metrics are compared for original and optimized APK files. Table 5, shows the results of the comparison of consumption Percentage. The results display that the percentage of optimization in consumption power ranges from 6.7 to 19.9 percent.

Table 5. Comparison of consumption percentage (%)

Statement case	Before	After	profit	Optimization percent %
If-eq	36.1	28.9	7.2	19.944
If-lt	39.9	37.0	2.9	7.268
If-gt	42.2	34.3	7.9	18.720
If-eqz	39.5	36.2	3.3	8.354
If-ltz	37.3	30.8	6.5	17.426
If-gtz	36.9	34.4	2.5	6.775

Table 6, shows the results of the comparison of energy consumption. The percentage of optimization ranges from 2.43 to 8.44%.

Table 6. Comparison of energy consumption (Joule)

Statement case	Before	After	profit	Optimization percent %
If-eq	157.7	152.0	5.7	3.614
If-lt	158.7	151.9	6.8	4.284
If-gt	168.2	154.0	14.2	8.442
If-eqz	164.5	160.5	4	2.431
If-ltz	167.9	157.2	10.7	6.372
If-gtz	161.4	156.0	5.4	3.345

Table 7, shows a comparison of power calculation and the percentage of optimization ranges from 4.3 to 14.06%.

Table 7. Comparison of power measurements (mille-W)

Statement case	Before	After	profit	Optimization percent %
If-eq	440	421	19	4.318
If-lt	548	516	32	5.839
If-gt	550	526	24	4.363
If-eqz	583	501	82	14.065
If-ltz	522	493	29	5.555
If-gtz	577	535	42	7.279

The *'AVLoadingIndicatorView'* application is adopted as benchmark [29]. The Smali files are optimized at code-level of conditional and selection instructions.

Table 8. Comparison of consumption measurements

Benchmark APKs	percentage	Energy	Power
Original APK	0.9 %	1.3 J	118 mW
Optimized APK	0.4 %	526.0 mJ	58 mW

The APK files are generated according to the different cases studied (original, optimized) of APKs. Table 8, displays a comparison between consumption measurements of benchmark APK files.

Then, the benchmark is modified at Java and Smali code-level. The Smali code modification is done by adding a Smali class file. The class (small_code.smali) file includes hungry-energy instructions into the method *(.method static test()V)*.

In this case study, The added class (*.class public Lcom/wang/avi/sample/small_code;*) includes the conditional statements (if-eqz, if-ltz, if-gtz). Each statement is repeated with five overlapping structures. The invoke-static (*test*) statement is called into (*onCreate*) virtual method. The (*onCreate*) method is declared in Smali code as: *(.method protected onCreate(Landroid/os/Bundle;)V).* The statement of invoking a method of class into another class is: *invoke-static { }, Lcom/wang/avi/sample/small_code;->test()V.*

The APK files of a modified benchmark after adding the hungry-instructions and optimizing the modified code only are generated. Power measurements are taken by PowerTutor for modified files before and after optimizing the added code. Comparing the energy consumption values proves the increased energy consumption by the application after adding the most energy-consuming instructions. Table 9, displays a comparison between consumption measurements of benchmark APK files in this case study.

Table 9. Comparison of consumption measurements

Benchmark APKs	percentage	Energy	Power
Modified Smali code	41.5 %	159.3 J	542 mW
Optimized hungry-instructions	36.4 %	147.7 J	519 mW

The results prove the correct performance of the optimized compiler and its effect on saving energy consumed by applications, as in figure 16. The experiments demonstrate the efficiency of the optimized compiler in the various cases studied from the benchmark.

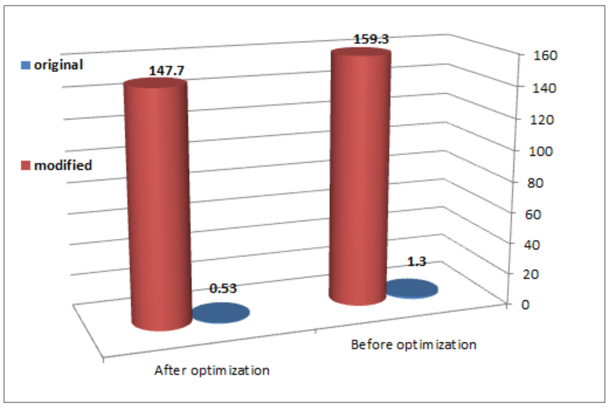

Figure 16: Comparison of energy consumption in joules Unit for 'AVLoadingIndicatorView' APKs

8. CONCLUSION

In this paper, An optimized compiler is used to optimize Smali code and generate saving-energy Android Applications. The effect of code modification is studied by optimizing the Smali language instructions. Execution time of Smali instructions are measured. The instructions with the lowest execution time are specified. The optimized compiler relies on the lowest execution time instructions. A comparison of resource consumption between APK files before and after using the optimized compiler proved energy saving of application.

The results demonstrate that the optimized applications with lower execution times consume less power. Energy consumption and resources usage measurements are taken for original and optimized applications by using powerTutor. Experiments prove that reducing execution time by optimization Smali code works to save energy consumed by Android applications. The consumption percentage is saved between 6.7% to 19.9%. An open-source application 'AVLoadingIndicatorView' is used in the experiments to validate the results of the optimized compiler. The application instructions are modified at the smali code-level. APK files for the benchmark application confirmed the validity of the results and the quality of the optimized compiler.

In this research, the results demonstrate the ability to reduce the energy consumed of Android applications by reducing the execution time.

As for the future work, A tool can be built to monitor power consumption at the Smali code-level in a programming stage. The tool identifies the hungry-energy instructions and helps developers produce less-consuming Android applications.

9. REFERENCES

[1] Claas Wilke et al, 2013, "Energy-Aware Development and Labeling for Mobile Applications", Technischen University Dresden.

[2] Shuai Hao, Ding Li, William G. J. Halfond, Ramesh Govindan, 2013, "Estimating Mobile Application Energy Consumption using Program Analysis", IEEE.

[3] Grace Metri, 2014, "Energy Efficiency Analysis And Optimization For Mobile Platforms", Wayne State University Dissertations.

[4] Irene Manotas Gutierrez, 2017, "Developing A Software Engineer's Energy-Optimization Decision Support Framework", University of Delaware.

[5] Anton Georgiev, Alberto Sillitti, Giancarlo Succi, 2016, "Open Source Mobile Virtual Machines: An Energy Assessment of Dalvik vs. ART", HAL Id: hal-01373061.

[6] Xueliang Li John P. Gallagher, 2016, "Energy Optimization of Source Code Guided by a Fine-Grained Energy Model", arXiv:1605.05234v1.

[7] Cagri Sahin, Mian Wan et al, 2016," How does code obfuscation impact energy usage?", JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS.

[8] Abhijeet Banerjee, Abhik Roychoudhury, 2016, "Automated Re-factoring of Android Apps to Enhance Energy-efficiency", ACM.

[9] Shuai Hao, Ding Li et al, 2012," Estimating Android Applications' CPU Energy Usage via Bytecode Profiling", IEEE.

[10] Hina Anwar et al, 2019, "Evaluating the impact of code smell refactoring on the energy consumption of Android applications", 45th Euromicro Conference on SEAA.

[11] Shaiful Chowdhury et al, 2017, "An exploratory study on assessing the energy impact of logging on Android applications", Springer Science.

[12] Fabio Palomba et al, 2018, "On the Impact of Code Smells on the Energy Consumption of Mobile Applications", Information and Software Technology.

[13] Marco Couto, J_acome Cunha et al, 2015,"Analyzing and Classifying Energy Consumption in Android Applications", Science of Computer Programming.

[14] Luis Corral, Anton B. Georgiev, Alberto Sillitti, Giancarlo Succi, 2014, "Can execution time describe accurately the energy consumption of mobile apps? An experiment in Android", ACM.

[15] Marco Couto, Tiago et al, 2014, "Detecting Anomalous Energy Consumption in Android Applications", Springer International Switzerland.

[16] Roberto Verdecchia et al, 2018, "Empirical Evaluation of the Energy Impact of Refactoring Code Smells", EPiC Series in Computing,Volume 52, Pages 365-383.

[17] Abhijeet Banerjee, Abhik Roychoudhury, 2016, "Future

of Mobile Software for Smartphones and Drones: Energy and Performance", National University of Singapore.

[18] Marwa DAHDOUH, Amer BOUCHI, Souheil KHAWATMI, Mouhamad Ayman NAAL, 2019, "Structural Analysis of Smali Language to Enhance Performance of Android Applications", Research Journal-University of Aleppo, Volume 151.

[19] Marwa Dahdouh, Mouhamad Ayman Naal, Souheil Khawatmi, Amer Bouchi, 2019, "Design an Optimized Compiler to Enhance Performance of Android Applications", IJCA.

[20] Xueliang Li, John P. Gallagher, 2015, "A Top-to-Bottom View: Energy Analysis for Mobile Application Source Code", Roskilde University arXiv:1510.04165v1.

[21] Ruben Saborido, Foutse Khomh et al, 2018, "An App Performance Optimization Advisor for Mobile Device App Marketplaces", Sustainable Computing: Informatics and Systems.

[22] Inmaculada Ayala et al, 2019, "An Energy Efficiency Study of Web-Based Communication in Android Phones", Hindawi,Scientific Programming

[23] Yan Hu et al, 2017, "Lightweight Energy Consumption Analysis and Prediction for Android Applications", Science of Computer Programming.

[24] Luis Cruz et al, 2019, "EMaaS: Energy Measurements as a Service for Mobile Applications", IEEE/ACM 41st International Conference on Software Engineering.

[25] MOHAMMAD ASHRAFUL HOQUE et al, 2015, "Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices", ACM.

[26] Ding Li, Shuai Hao, William G.J. Halfond, Ramesh Govindan, 2013, "Calculating Source Line Level Energy Information for Android Applications", ACM.

[27] Lide Zhang, Birjodh Tiwana et al, 2010,"Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones", ACM.

[28] Lin-Tao Duan et al, 2013, "Energy analysis and prediction for applications on smartphones", Journal of Systems Architecture.

[29] https://github.com/81813780/AVLoadingIndicatorView/blob/master/apk/app-debug.apk. [Accessed 1/9/2019].