UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ANDERSON LUIZ SARTOR

AndroProf: A Profiling tool for the Android platform

Graduation Project.

Prof. Dr. Antonio Carlos Schneider Beck Filho
Advisor

Ulisses Brisolara Corrêa
Co-Advisor

Porto Alegre, December of 2013.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do ECP: Prof. Marcelo Götz
Bibliotecário-Chefe do Instituto de Informática: Alexsander Borges Ribeiro

ACKNOWLEDGEMENTS

I would like to thank my advisors, Professor Antônio Carlos and Ulisses for the support and guidance during the development of this work.

Also, I would like to thank my family for the support and my girlfriend, Ana, for the encouragement and for always believing in me.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS 6

LIST OF FIGURES 8

LIST OF TABLES 9

RESUMO 10

ABSTRACT 11

1 INTRODUCTION 12

2 ANDROID OVERVIEW 13

2.1 Market share 13

2.2 Technical details 14
2.2.1 Android stack 14
2.2.2 Native code 15
2.2.3 Just-In-Time compiler 15

3 RELATED WORK 17

3.1 Traceview 17

3.2 Modification of DDMS 18

3.3 AndroScope 18

3.4 CPU cycle estimation 19

3.5 MARSSx86 20

3.6 PowerTutor 21

3.7 Sesame 21

3.8 Trepn Profiler 22

4 IMPLEMENTATION 25

4.1 QEMU modification ... 26

4.2 Graphical User Interfaces ... 27
4.2.1 Instruction categorization GUI ... 27
4.2.2 Analysis GUI ... 29

4.3 Environment setup ... 32

5 RESULTS AND CASE STUDY ... 33

5.1 Execution time, number of BBs and number of instructions comparison between ARM/MIPS and Java/JNI ... 34

5.2 Energy and performance estimation ... 36

5.3 QEMU modification impact for Java and JNI applications on ARM and MIPS architectures ... 38

5.4 JIT compiler evaluation ... 39

6 CONCLUSION AND FUTURE WORK ... 41

REFERENCES ... 42

APPENDIX A: JAVA APPLICATIONS EXECUTION TIME ... 45

APPENDIX B: JAVA AND JNI APPLICATIONS TIME, #BBS, #INSTRUCTIONS ON ARM AND MIPS ... 46

APPENDIX C: JAVA AND JNI APPLICATIONS TIME, #BBS, #INSTRUCTIONS ON ARM EMULATOR WITH JIT DISABLED ... 48

APPENDIX D: JAVA AND JNI APPLICATIONS ON MOTOROLA ATRIX WITH AND WITHOUT JIT ... 50

APPENDIX E: PROJECT DESCRIPTION (TG1) ... 51

LIST OF ABBREVIATIONS AND ACRONYMS

ACPI	Advanced Configuration and Power Interface
ADB	Android Debug Bridge
AVD	Android Virtual Device
BB	Basic Block
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CPU	Central Processing Unit
DB	Database
DBMS	Database Management System
DDMS	Dalvik Debug Monitor Server
DVM	Dalvik Virtual Machine
ER	Entity-Relationship
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDD	Hard Drive Disk
HPC	Hardware Performance Counter
IPC	Instructions Per Cycle
JIT	Just-In-Time
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LCD	Liquid Crystal Display
MVC	Model-View-Controller
OHA	Open Handset Alliance
PC	Program Counter
PID	Process Identifier

RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SD card	Secure Digital card
SDK	Software Development Kit
TB	Translated Block
VNC	Virtual Network Connection
XML	eXtensible Markup Language

LIST OF FIGURES

Figure 2.1: Global mobile platform market share. Source: Gartner, IDC, Strategy Analytics, BI intelligence estimates, and company filings 13
Figure 2.2: Global computing platform market share. Source: Gartner, IDC, Strategy Analytics, BI intelligence estimates, and company filings 14
Figure 2.3: Android Stack. Source: Google Developers 15
Figure 2.4: JIT speed up. Source: Google I/O 2010 16
Figure 3.1: Traceview timeline panel. Source: Android Developers. 17
Figure 3.2: Traceview profile panel. Source: Android Developers 18
Figure 3.3: Modified method tracing. Source: (CHO, HWANG, et al., 2012) 19
Figure 3.4: MARSSx86 overview. Source: MARSS MICRO 2012 tutorial 20
Figure 3.5: Trepn profiler example. Source: Qualcomm developers 22
Figure 4.1: Tool overview 25
Figure 4.2: Modified QEMU emulation flow 26
Figure 4.3: AndroProf instruction categorization GUI 28
Figure 4.4: Instruction categorization GUI project 29
Figure 4.5: Instruction categorization file XML structure 29
Figure 4.6: AndroProf analysis GUI 30
Figure 4.7: Analysis GUI project 31
Figure 4.8: Database ER diagram 31
Figure 5.1: #BBs and #instructions comparison for ARM and MIPS 35
Figure 5.2: Time and power estimation 37

LIST OF TABLES

Table 3.1: Tools comparison 24
Table 5.1: Average execution time comparison 33
Table 5.2. ARM and MIPS comparison 34
Table 5.3: ARM Power and cycles information 36
Table 5.4: ARM Cortex A8 and A9 cycles comparison.. 37
Table 5.5. ARM Cortex A8 and A9 energy consumption comparison 38
Table 5.6. ARM and MIPS QEMU modification overhead 39
Table 5.7: Execution time on the emulator (ARM) with JIT disabled 39
Table 5.8: JIT speed up - Motorola Atrix 40
Table A.1: Java applications execution time on original QEMU (ARM) 45
Table A.2: Java applications execution time on modified QEMU (ARM) 45
Table B.1: Java/JNI applications on ARM (execution 1) 46
Table B.2: Java/JNI applications on ARM (execution 2) 46
Table B.3: Java/JNI applications on ARM (execution 3) 46
Table B.4: Java/JNI applications on MIPS (execution 1)... 47
Table B.5: Java/JNI applications on MIPS (execution 2)... 47
Table B.6: Java/JNI applications on MIPS (execution 3)... 47
Table C.1: Java/JNI applications on ARM without JIT (execution 1) 48
Table C.2: Java/JNI applications on ARM without JIT (execution 2) 48
Table C.3: Java/JNI applications on ARM without JIT (execution 3) 48
Table D.1: Java/JNI applications on Motorola Atrix with JIT 50
Table D.2: Java/JNI application on Motorola Atrix without JIT................................... 50

RESUMO

Este trabalho tem como objetivo o desenvolvimento de uma ferramenta de *profiling* para a plataforma móvel Android. As ferramentas atuais de *tracing* e *profiling* não acompanharam o crescimento do mercado móvel, fazendo com que a tarefa de obter dados sobre a execução de aplicativos se torne muito mais difícil. As poucas ferramentas que estão à disposição têm grandes limitações com relação a quais informações elas conseguem obter e também quanto a quantidade de informação que pode ser coletada. Além das poucas opções de ferramentas, o desenvolvimento de aplicações para sistemas embarcados já é, naturalmente, mais complexo devido às limitações do sistema, como, por exemplo, desempenho reduzido e alimentação por bateria. Portanto, o desenvolvimento de uma ferramenta que obtenha informações como: dissipação de potência, tempo de execução e outras estatísticas é extremamente necessária no desenvolvimento de aplicações para sistemas embarcados.

Este trabalho apresenta uma ferramenta multiplataforma que suporta a emulação de arquiteturas ARM e MIPS executando Android, além de suportar parcialmente a arquitetura x86. Ela obtém as informações citadas anteriormente por aplicação e ainda é capaz de obter dados de aplicações que executam tanto código nativo, quanto aplicações que executam na máquina virtual Dalvik. Para alcançar este objetivo, nós estendemos o QEMU do Android SDK e desenvolvemos ferramentas com interfaces gráficas para processar os dados coletados.

Além disso, nós avaliamos o impacto da nossa implementação em relação ao tempo de execução de diversos benchmarks e nós fizemos um estudo de caso comparando diferentes arquiteturas, aplicações escritas puramente em Java e aplicações com partes em código nativo, bem como o impacto do *JIT compiler* na execução. Todos estas comparações através do uso da ferramenta desenvolvida.

Palavras-Chave: Aplicações Android, Emulador Android, QEMU, ferramenta de *profiling*, JNI.

ABSTRACT

This work aims to develop a profiling tool for the Android platform. Current tools for mobile development are very limited in which and how much information they can trace or profile. They are also scarce when compared to general-purpose development tools. This makes the development of embedded applications even a harder task to be accomplished, due to its hard constraints, such as limited performance and power budget. Therefore, a tool that provides information such as power dissipation, execution time and other statistics is mandatory when it comes to develop embedded applications.

This work presents a multiplatform tool that fully supports ARM and MIPS architectures, and partially supports x86 architecture executing Android. It provides the aforementioned information per application and it is also able to trace all applications native code, including that generated by Dalvik Virtual Machine. To accomplish this, we extended Android SDK's QEMU, and we developed graphical user interfaces to process the traced data.

In addition, we evaluated the impact of this implementation in relation to the execution time of several benchmarks and we present a case study comparing different architectures, applications written purely in Java and applications that use native code and the impact of the JIT compiler in the execution. All these comparisons through the usage of the developed tool.

Keywords: Android applications, Android Emulator, QEMU, profiling tool, JNI.

1 INTRODUCTION

Mobile devices have hard constraints. Physical resources like storage, processing capacity, memory occupation and power supply are critical for these systems. For instance, mobile systems have only a few gigabytes of storage, so applications must be developed considering that, as well as the application's storage usage after it is delivered to the user. Moreover, a number of applications must run concurrently in an environment that is not optimized for performance, but rather for energy consumption. The latter must be kept as low as possible to maintain an acceptable battery lifetime. Therefore, mobile systems' developers must think differently from general-purpose ones regarding the optimization of their applications. For this reason, the use of profiling and monitoring tools is extremely important to help ensure these requirements are met at earlier stages of the development process. With these tools, it is possible to guarantee that the application will not consume excessive power, will not use more memory than is strictly necessary with useless or not optimized data and will not overuse the device's processor.

Considering this scenario, in this work we propose a multiplatform tool that aims to provide valuable information about Android applications. Information that today is not available, such as power dissipation estimation, statistics about executed basic blocks and instructions, and CPU cycle estimation.

Android is a Linux-based mobile software platform that is mainly used in smartphones, tablets and that will also be popular in other devices in a near future, such as smart TVs and watches. It is the world's most popular mobile platform (the cut it had in the smartphone market in 2011 was about 50% and, just one year later, it reached almost 70% (IDC, 2013)). Hence, the proposed tool can pave the way for designers to meet these requirements (i.e. power dissipation, processing capacity, and storage), so most likely the developed application will efficiently run on a mobile device.

As a case study, a comparison between different architectures and between applications written purely in Java and applications that use Java Native Interface (JNI) is made. In addition, the Just-In-Time (JIT) compiler is evaluated.

The remaining of this work is organized as follows: Section 2 presents an Android overview. Section 3 presents related works. Section 4 presents the implementation and the environment setup of the proposed tool. The related experimental results and the case study comparing Java and JNI applications for different architectures, with or without the JIT compiler mechanism, are given in Section 5. Section 6 makes the conclusion.

2 ANDROID OVERVIEW

Android is an open-source software stack for mobile devices (ANDROID-PROJECT, 2013), developed by OHA (Open Handset Alliance) consortium, which is composed of mobile operators, handset manufacturers, semiconductor companies, software companies and commercialization companies, such as Google, ARM, Intel, Samsung, HTC, Motorola, Qualcomm, NVIDIA and many others (OHA, 2013).

2.1 Market share

Figure 2.1 shows the growing popularity of Android in the market. This growth in mobile platforms is impressive: in these days, it holds 70% of the market share, followed by iOS (mobile software platform developed by Apple Inc.), with about 20%.

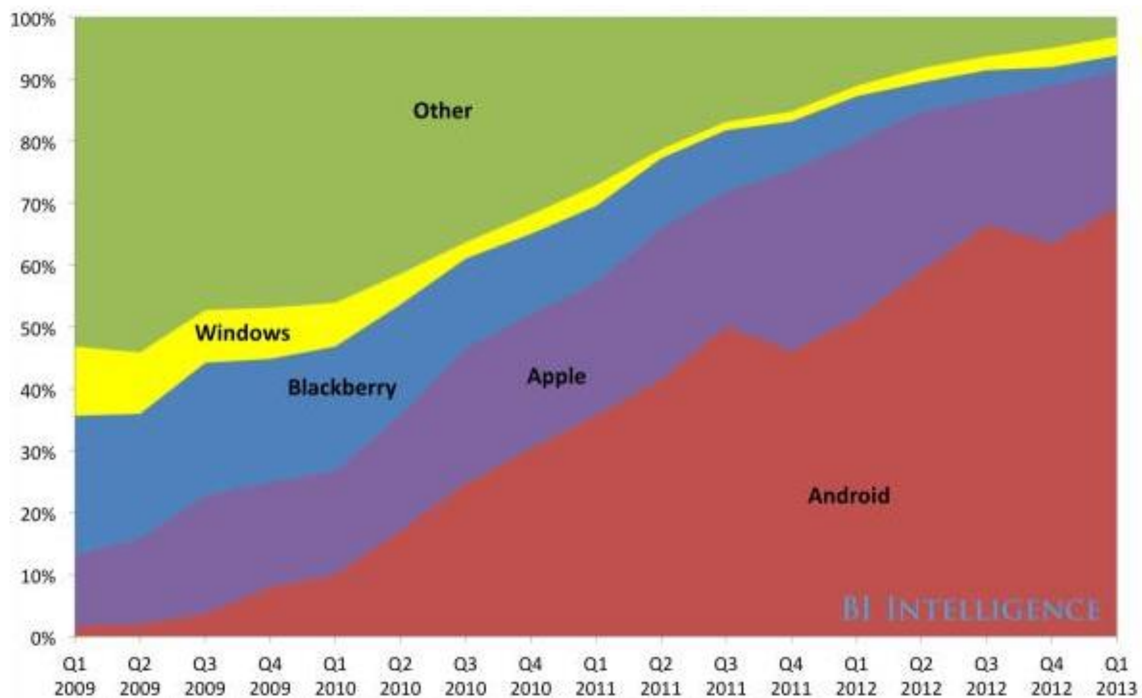

Figure 2.1: Global mobile platform market share. Source: Gartner, IDC, Strategy Analytics, BI intelligence estimates, and company filings

Android is also taking place in the global computing platform market, composed of computers, tablets, smartphones, and other devices. Latest researches by BI Intelligence (BI INTELLIGENCE, 2013) show that Android holds 53% of the global computing platform market share, taking Windows' place, which today holds 24% of the market share, as shown in Figure 2.2.

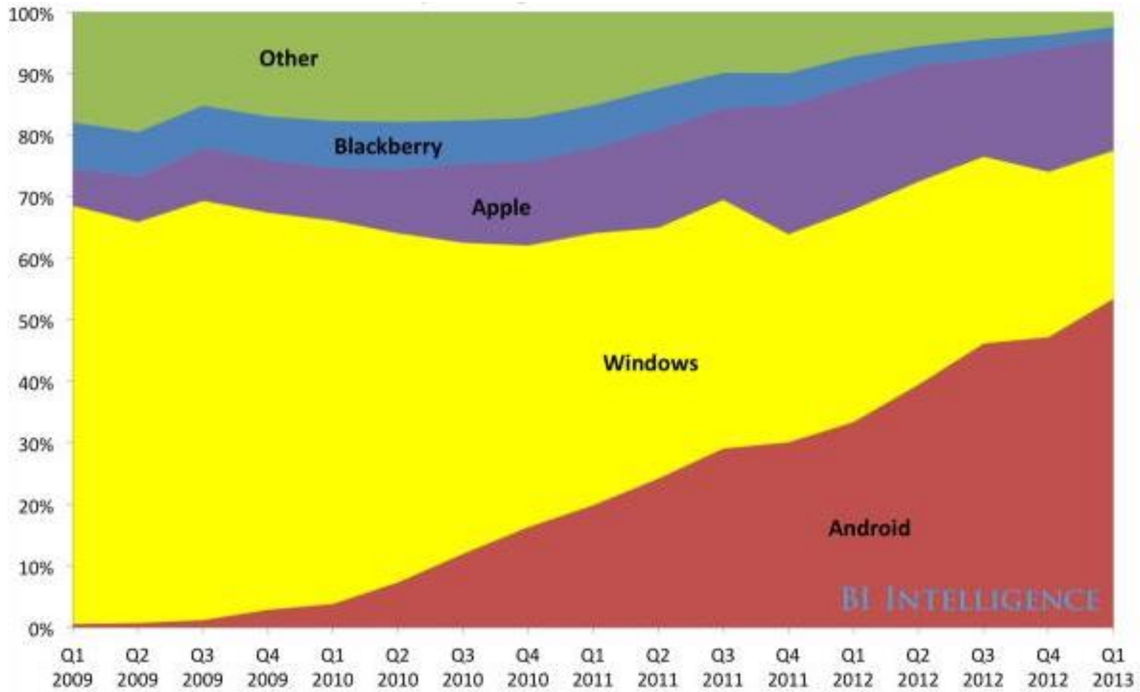

Figure 2.2: Global computing platform market share. Source: Gartner, IDC, Strategy Analytics, BI intelligence estimates, and company filings

2.2 Technical details

2.2.1 Android stack

Figure 2.3 depicts the Android stack. The Linux Kernel ① is a layer that interacts with the hardware; it also contains all essential hardware drivers. Android is based on Linux 2.6 Kernel on Android versions older than Ice Cream Sandwich and 3.x kernel for newer versions, with some architectural modifications made by Google. The libraries layer ② contains native libraries written in C or C++. These libraries are optimized mainly for CPU and GPU intensive tasks.

Google has chosen to deploy an alternative virtual machine on Android, called Dalvik Virtual Machine (DVM), instead of keeping Oracle's Java Virtual Machine (JVM). Its purpose is to provide a platform-independent programming environment that abstracts details of the underlying hardware or operating system. Therefore, application's portability is increased.

Dalvik VM, defined in the Android Runtime ③, is a register-based architecture (ANDROID-DALVIK, 2013), opposed to JVM that is a stack-based architecture. Dalvik was designed to run on low memory environments and to allow multiple instances of the VM, so every application runs on its own instance of the VM. Therefore, it provides security, isolation and effective memory management.

An advantage that a register-based architecture has over a stack-based architecture is that it is likely to have better performance. A register-based architecture needs less VM instructions to implement a high-level code, even though this comes at a cost of increased instruction size. With larger instructions, register-based architectures take more time to execute each instruction, compared to stack-based architectures. However, the product between the time per instruction and the number of executed instructions is smaller in

Figure 2.3: Android Stack. Source: Google Developers

register-based architectures than in stack-based ones, which means that a register-based architecture will take less time to execute an application (EHRINGER, 2012).

Android applications are written in Java, afterwards compiled to Oracle's Java Standard Edition bytecodes, and then these bytecodes are converted to a single Dalvik Executable (.dex) file by the "dx" tool. These Dalvik Executable bytecodes, rather than Java bytecodes, are executed on Dalvik Virtual Machine.

The Application Framework ④ consists in basic tools for applications' development. Finally, the Applications layer ⑤ is the layer where all developed applications are.

2.2.2 Native code

It is possible to develop Android applications purely in Java, in Java with parts of it using native code or purely in native code. Some reasons to use native code are the following: the interpretation overhead can be mostly avoided once the code does not execute on the Dalvik VM; and it is possible to reuse existing C or C++ code. Thus, the developer does not have to convert it to Java. Android provides two main ways of using native code in the application: by using the Java Native Interface (JNI); or by using Native Activities, which are activities written in native code. The latter was added to Android in the API level 9 (Android 2.3 Gingerbread) (ANDROID-NATIVE ACTIVITY, 2013).

2.2.3 Just-In-Time compiler

Dalvik has a Just-In-Time compiler that was added to Android 2.2 release and it is still being used and optimized in the current version of Android. This dynamic binary translation mechanism allows the application to run faster as it analyzes the code and caches the most executed parts of the translated code for further reuse, which avoids most

of the interpretation overhead of Java code. Google's experiments show an improvement of two to five times for CPU-bound code (BORNSTEIN, 2010), as shown in Figure 2.4.

In addition, the latest Android version, KitKat, allows the user to choose between two runtimes, Dalvik and ART. ART (ANDROID-ART, 2013) is an ahead-of-time compiler that has the goal to speed up applications' loading and possibly to consume less energy by performing a compilation of the intermediate code during the application's installation. Therefore, the interpretation overhead can be avoided with the cost of a longer installation time and more storage utilization to save this compiled code. As ART is an experimental runtime so far, it is not known if, in the next versions of Android, we will continue to have Dalvik runtime and JIT compiler or we will move to ART runtime.

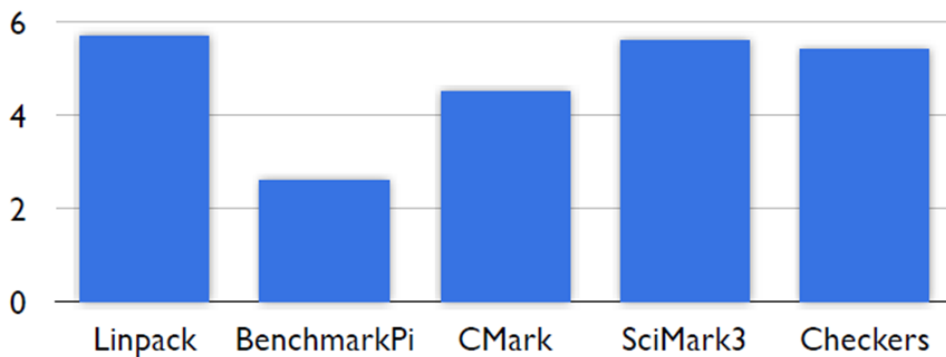

Figure 2.4: JIT speed up. Source: Google I/O 2010

3 RELATED WORK

Several attempts were made to create tools for tracing and profiling Android applications. Still, to the best of our knowledge, we are far from having a complete pack of profiling tools.

3.1 Traceview

Android Software Development Kit (SDK) provides software tools for debugging, profiling and monitoring. The Dalvik Debug Monitor Server (DDMS) that belongs to this kit, contains Traceview, which is a profiling tool that provides timeline and profile panels in a Graphical User Interface (GUI). The former panel contains the start/stop time of each thread, shown in Figure 3.1.

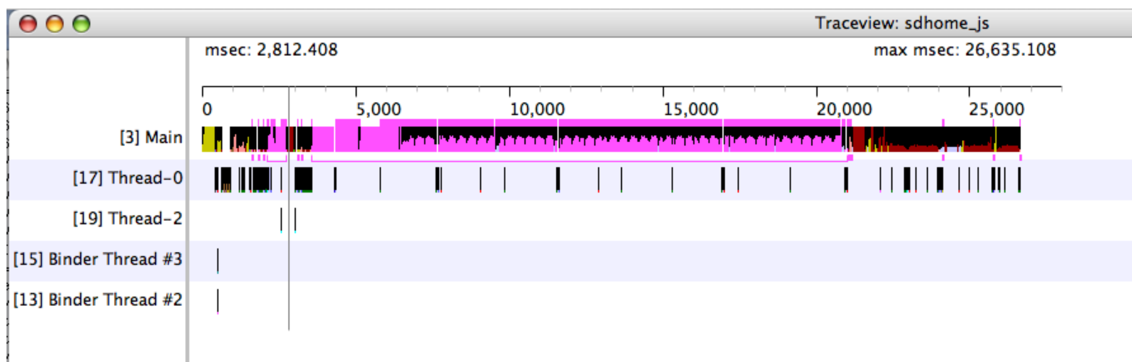

Figure 3.1: Traceview timeline panel. Source: Android Developers.

The profile panel, shown in Figure 3.2, contains a summary of each method, with the name of the method and its children methods; the inclusive and exclusive execution times and the number of calls/recursive calls of the method. The inclusive method execution time is the time spent in the method plus the time spent in any called methods, consisting in the total time that the method took to execute, from the first instruction to the last one (ANDROID-TRACEVIEW, 2013). The exclusive method execution time is the time that is spent exclusively in the method, therefore, the execution time of any child method called is not considered. This trace can be done either by changing the application's source code to call both start and stop tracing methods or by manually starting and stopping the trace after the application is already being executed.

This tool is very useful to profile an application; however, it is very limited on the amount of data that it can trace: it stores all the traced data in a buffer with a limited size, which depends on the amount of free RAM that is available on the mobile device. Therefore, if the buffer overflows, all the data profiled after the overflow will be lost. Thus, this tool does not scale with larger applications and it is not possible to fully execute

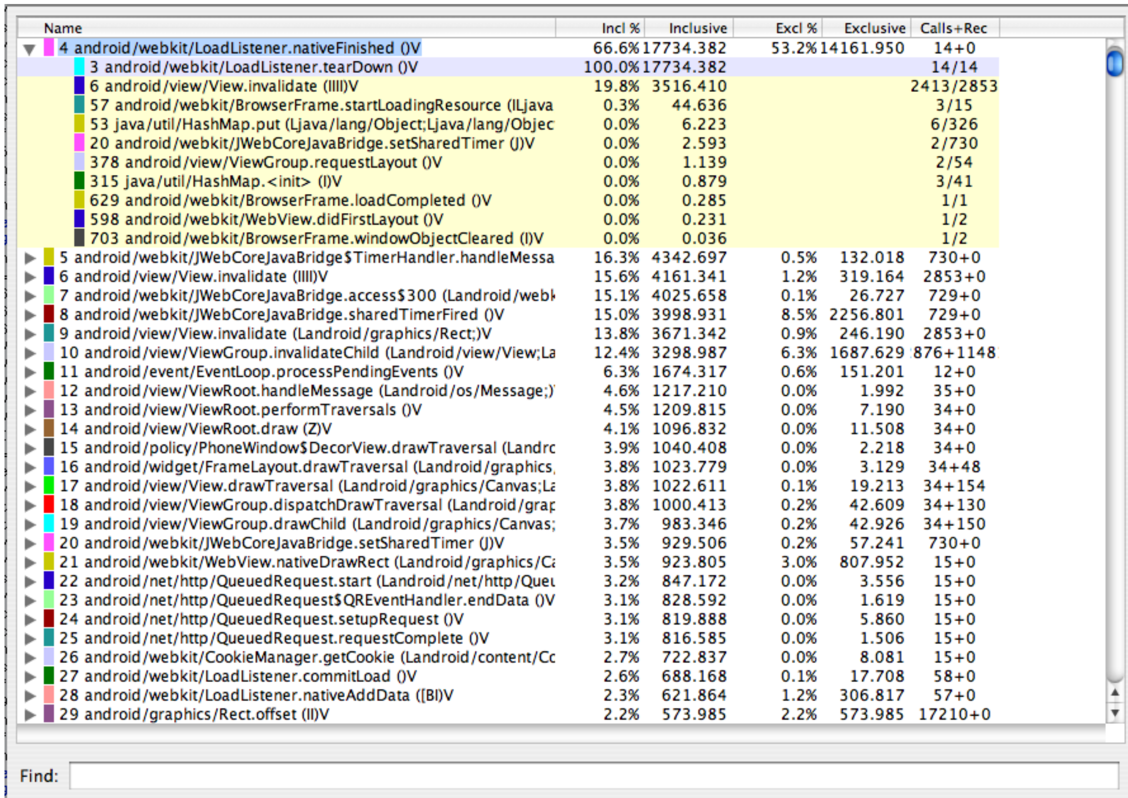

Figure 3.2: Traceview profile panel. Source: Android Developers

even small applications. Even with more than 1GB of memory available, most of the tested benchmarks (which will be further discussed later) still will make the buffer to overflow.

Another important limitation is that it can only trace methods executed by the Dalvik Virtual Machine. Therefore, native code (used through Java Native Interface (JNI) methods or Native Activities) is not considered. Tracing native code is a core feature when it comes to trace applications that have native methods (e.g.: WebKit, used by the most popular web browsers). In addition, none of the SDK tools provide power dissipation information.

3.2 Modification of DDMS

As the DDMS is written in Java and runs as an Eclipse plugin, it is not performance oriented. Thus, the tool proposed by Hyen-Ju Yoon (YOON, 2012) has the goal of speeding up the profiling process. It is done by decomposing the Traceview into a log data processing layer and a *Pretrace* program layer that create and analyze the start and end time of methods.

This modification is show in Figure 3.3. However, no quantitative information about this speed up is discussed in their work and, in the same way as Traceview, it does not provide native code information.

3.3 AndroScope

Due to before mentioned Traceview limitations (i.e. it traces just methods that are executed by the Dalvik VM, and presents poor performance in opening large amounts of data), M. Cho et al. (CHO, HWANG, *et al.*, 2012) proposed another performance analysis

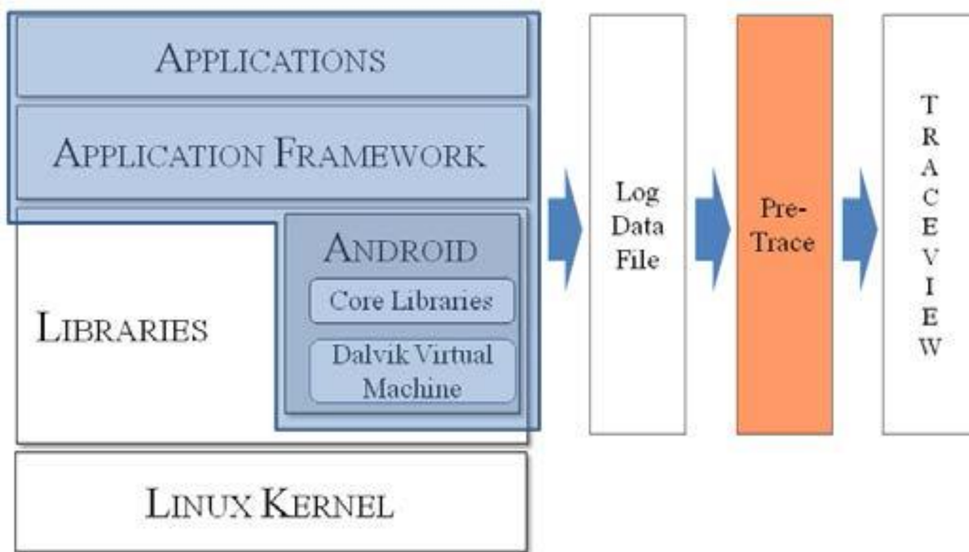

Figure 3.3: Modified method tracing. Source: (CHO, HWANG, *et al.*, 2012)

tool for the Android platform, called AndroScope. This tool was made to profile Java and native applications, Dalvik VM and Android libraries.

In this work, a low-level performance analysis through Hardware Performance Counters (HPC) is used to store counts of hardware events such as cache misses, CPU cycles and executed instructions. With this information, it is possible to obtain the Instructions Per Cycle (IPC) ratio, which can be used as a performance factor. An extended GCC compiler front-end automatically inserts instrumentation codes to obtain the trace of native libraries and to provide a runtime filtering by class, method name or signature, which allows a selective trace.

The authors also developed a graphical user interface, based on Traceview, to display the traced data. They created a new layer to process massive trace logs faster, called "tracebridge" and used Traceview just to display the post-processed data.

However, performance analysis through HPCs cannot be easily done in the Android Emulator because they are not implemented in QEMU (an open source machine emulator and virtualizer). As HPCs are architecture dependent, this implementation would have to be specific for each architecture emulated by QEMU and supported by Android (e.g.: MIPS, x86 and ARM or any future architecture that may appear), making the process of building a multi-architecture tool highly complex. Moreover, all native libraries would have to be recompiled in order to be traced. As the statistics of these counters in the Linux kernel are obtained by sampling, it may contain inaccurate measures from the applications.

3.4 CPU cycle estimation

A research conducted by Fujitsu Laboratories Ltd. (THACH, TAMIYA, *et al.*, 2012), proposed a cycle estimation methodology for an instruction-level CPU emulator. It is divided into a two-phase pipeline scheduling process. First, a static phase is conducted to obtain a rough estimation of the CPU cycle count with the purpose of reducing the instrumentation performance overhead. Then, a dynamic phase is responsible for refining the results and guaranteeing more precision. This methodology was implemented by

modifying the QEMU source-code with an estimation error in the CPU cycle count of about 10% when compared to a real CPU.

In addition, a cache simulator to model L1 and L2 caches behavior was built based on an ARM Cortex-A8 architecture. To keep track of the content of each cache line, the simulator uses virtual tag lists. By checking the tags when a memory access instruction is found, it can determined if this instruction leads to a cache hit or a cache miss. When a cache miss occurs, the cache simulator triggers the dynamic adaptation to add the cache miss penalty cycles.

However, as the source code is not available, the possibility of extending it to have more features (e.g.: counter for executed basic block and instruction information; power cost estimation; instruction categorization; identification of different applications; use a database to store the processed data for future reuse) was discarded.

3.5 **MARSSx86**

Patel and other researchers proposed another QEMU modification to perform full system simulation for multicore x86 CPUs, called MARSSx86 (MARSSX86, 2013). MARSSx86 is a cycle accurate tool for full system simulation of the x86 architecture. It performs detailed simulation of CPUs, caches and memory.

Although it is an open source tool, it currently does not support architectures other than x86. This limits its field of application, considering that ARM is the main architecture used in mobile devices (90% of the smartphones market share (TREFIS TEAM, 2013)). In the same way and with the same limitations, PTLsim (PTLSIM, 2013),

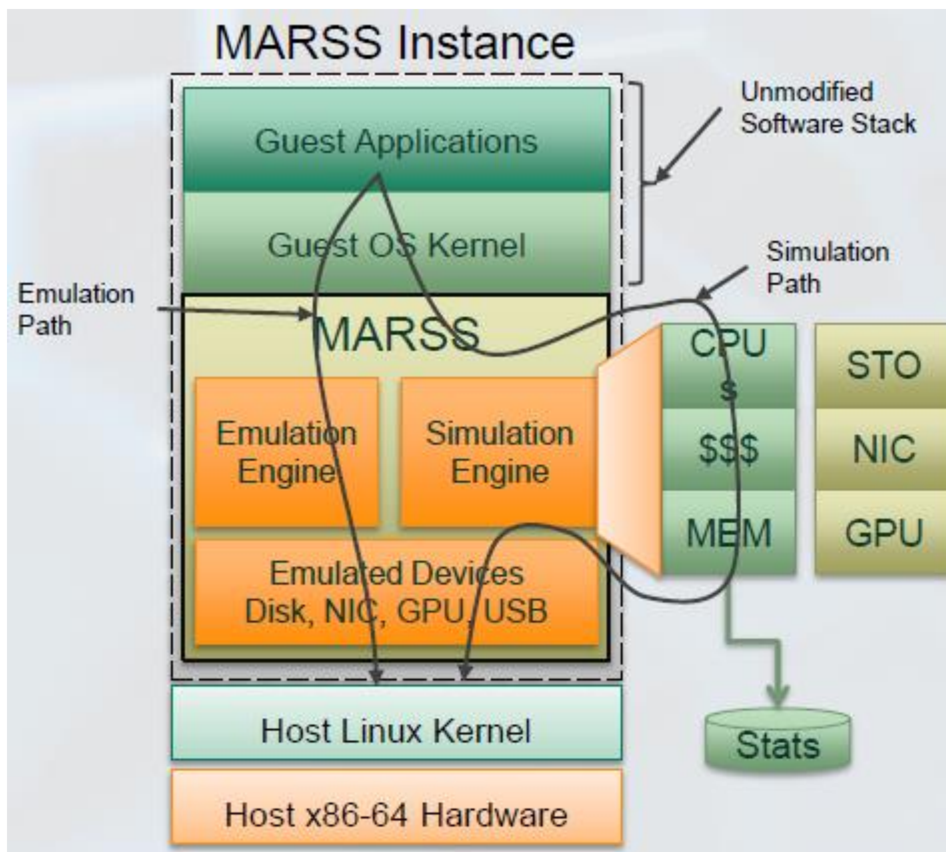

Figure 3.4: MARSSx86 overview. Source: MARSS MICRO 2012 tutorial

a cycle accurate x86 microprocessor simulator, was improved and ported to have integration with QEMU. This tool is presented in Figure 3.4.

3.6 PowerTutor

PowerTutor (POWERTUTOR, 2011) is a power estimation system that uses the model generated by PowerBooter (ZHANG, TIWANA, *et al.*, 2010) for online power estimation. PowerBooter models the most significant components regarding power dissipation in the system, which are: CPU and LCD display as well as GPS, Wi-Fi, 3G, and audio interfaces. These components' power dissipation is considered independent, which results in an error of 6,27%, according to the authors, and which suggest that a sum of these individual components is sufficient to estimate system power dissipation. The PowerBooter technique relies on knowledge of battery discharge voltage curve and access to battery voltage sensor, which is available in most smartphones. However, this information generated by PowerBooter is not publicly available.

PowerTutor was implemented for the Android platform. However, the PowerBooter models need to be created specifically to each smartphone model, as they have different power dissipation behavior. Also, each application is considered to be the only application that is running in the system, this simplification can provide an accurate estimation in this ideal scenario, but as we have several applications running at the same time, this estimation may diverge from the real power and energy consumption of the system as a whole.

In addition, to the best of our knowledge, only a few models are supported: HTC G1, HTC G2 and Nexus One smartphones; all other devices will use a generic model that may not estimate the power and energy consumption with the proper accuracy. Moreover, as PowerTutor runs concurrently with all other applications that are being executed. Therefore, it will also use the available resources in the device to run this profiling tool. In our approach, this estimation is done offline and, consequently, it does not use the available resources in the device to process the collected data.

3.7 Sesame

Sesame (DONG e ZHONG, 2011) generates and adapts energy models without any external measurement. To achieve a low overhead in the system's performance, it schedules the computation of intensive tasks to be executed when the system is idle and when it is connected to a power supply. Hence, only data collection and simple calculation are performed during system usage.

In addition, Sesame monitors the accuracy of the energy model in use and adapts it accordingly when the accuracy is below a certain threshold. This tool was developed to run on any Linux-based mobile system, which includes laptops and smartphones, for example.

Sesame uses system statistics, such as CPU timing statistics and memory usage provided by Linux. Also it uses the Advanced Configuration and Power Interface (ACPI), available on modern mobile devices, which provides platform-independent interfaces for power states of the hardware, including the processor and peripherals.

However, this approach is not able to provide this information to each application, all information is collected from the system as a whole. Which does not allow the developer

to see the actual costs of his application, once several other applications may be running concurrently in the system.

3.8 Trepn Profiler

Trepn Profiler (QUALCOMM DEVELOPERS, 2013) profiles performance and power dissipation for Qualcomm Snapdragon processors. Also, external scripts can be used in order to allow automated test environments. Moreover, the collected data can be seen in real time and it can be exported for offline analysis.

The power information is obtained through specific hardware sensors that are present in these processors. Therefore, the range of mobile devices that it reaches is very limited. This limitation forbids the evaluation of these costs in other processors and architectures, because mobile devices with different processors will present different power dissipation behavior for the same application. Figure 3.5 shows overlays of this tool in a device that is running a game.

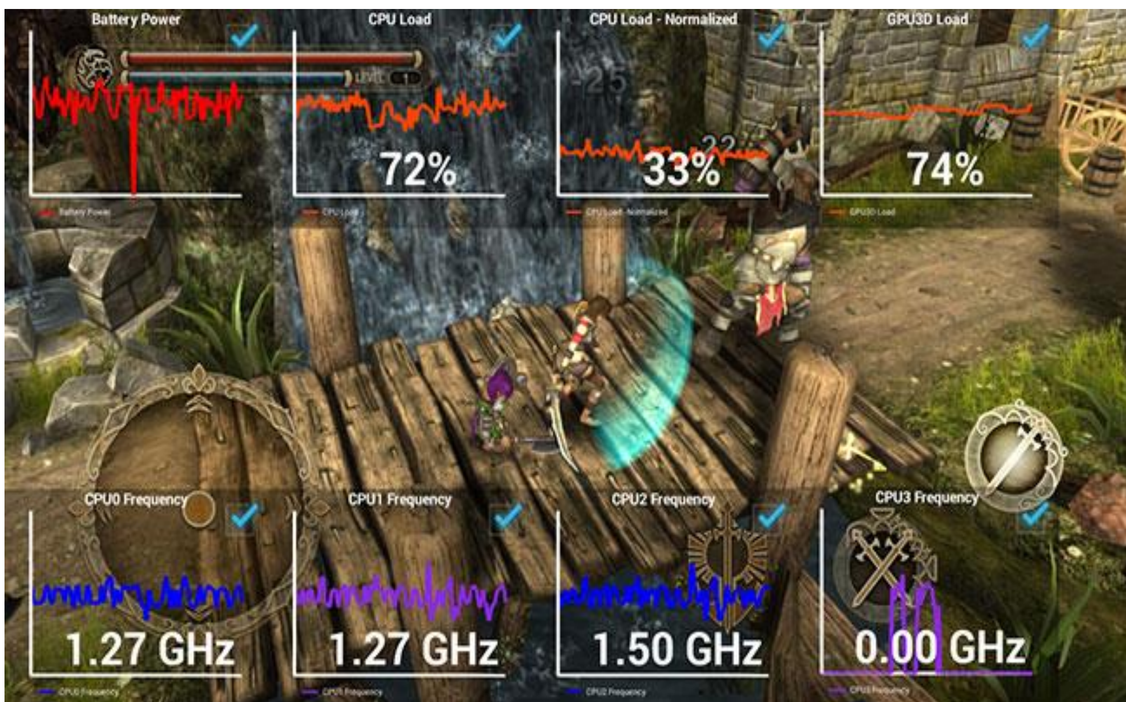

Figure 3.5: Trepn profiler example. Source: Qualcomm developers

All the previous works illustrate the difficulty of tracing and profiling mobile applications. To differentiate the proposed tool with the existent ones, a comparison among them is presented in Table 3.1. As can be observed, the proposed tool is able to profile a larger range of information: executed basic blocks and instructions, power dissipation estimation, etc.

In opposite to the existing tools, besides being capable of presenting information for each process; it also allows one to see the most costly parts of the code, separated by basic blocks. As this most costly parts cannot be separated, so far, by methods, other tools can be used to complement the trace (e.g.: Traceview or AndroScope). As it was not possible to extend any one of discussed tools to have additional features due to already mentioned reasons (i.e. source code not available or platform-specific tool), the proposed tool was developed from scratch. The tool was built on top of the Android's QEMU, which is a

QEMU version modified to run the Android emulator, due to the fact that the emulator is the official and largely used tool to develop and test Android applications.

Table 3.1: Tools comparison

Features\Tools	**Our tool**	**Traceview**	**Mod. of DDMS**	**AndroScope**	**Cycle estim.**	**MARSSx86**	**PowerTutor**	**Sesame**	**Trepn Profiler**
Trace native code	X			X	?	X			
Trace basic block information: BB instructions, #calls by each process	X								
Process instruction information: #calls, categories	X								
Process a large amount of data	X		?	X	?	X			
IPC estimation				X	X	X			
Power dissipation estimation	X						X	X	X
Cycle cost information	X			X	X	X			
Faster data visualization than Traceview	-	-	X	X	-	-	-	-	-
Instruction categorization	X								
Identify and separate the data of different applications in the trace	X					?	X		
Trace method information: name, exec. time, #calls, calls hierarchy		X	X	X					
Support multiple architectures	X	X	X	?	?			X	
No need to import the data every time the program is executed: usage of a database to store data	X								

Legend:	
X	Tool has this feature
?	No information is given about this feature
-	Feature does not apply
Blank cells	Not available

4 IMPLEMENTATION

As already discussed, our tool consists in a QEMU modification to generate trace information. In addition, as it is designed to work with multiple Instruction Set Architectures (ISAs), the added modifications must be compatible with all Android officially supported platforms (i.e.: ARM, MIPS and x86). So far, ARM and MIPS architectures are fully supported whilst x86 architecture is partially supported. For x86, the PID must be obtained from the emulated device's memory, so we can identify each process and, therefore, separate the data collected for these processes. Once this additional step is done, the x86 will be fully supported.

Figure 4.1 shows an overview of the tool flow. From Android SDK, we modified QEMU to trace more information about the application. The VNC connection mechanism also passed through modifications (explained in details later) to save this trace. In addition, we developed graphical user interfaces to process the collected data and to characterize the instructions and its categories. All collected data from the emulator's execution is saved in files that will be afterwards processed by the GUIs and saved in a database.

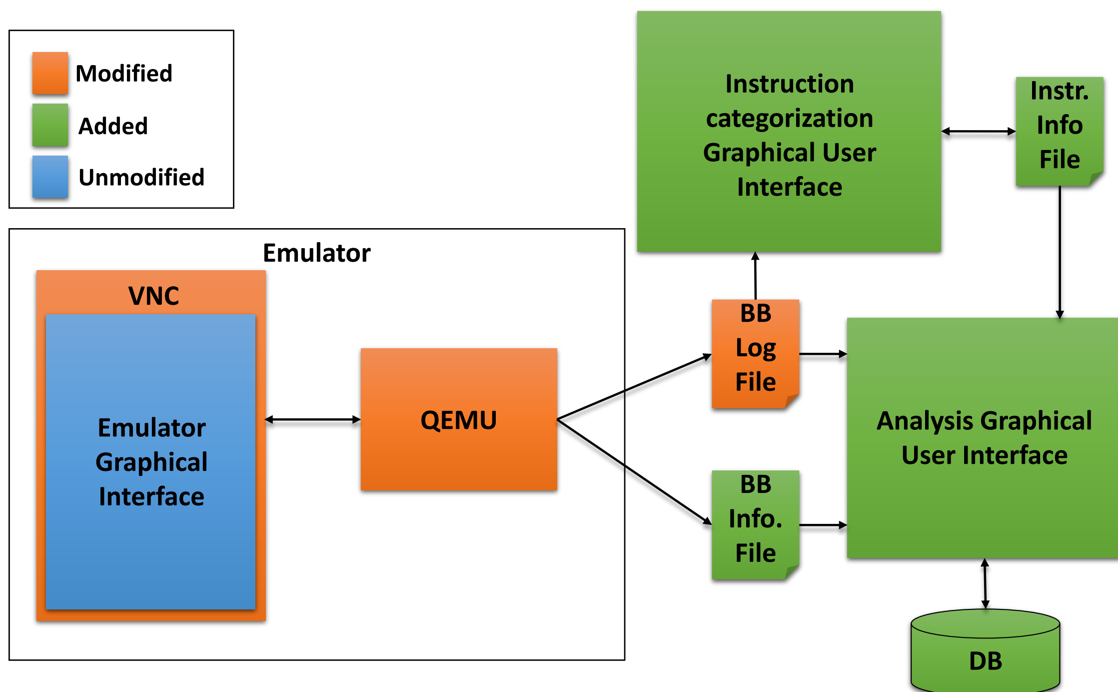

Figure 4.1: Tool overview

4.1 QEMU modification

The Android Emulator is based on QEMU and it is included in the Android SDK.

Therefore, we use the emulator to execute the applications and to generate the data that we are interested in. QEMU emulates the hardware so it is possible to run programs on a virtual hardware platform using different ISAs, such as ARM, MIPS, x86, PowerPC, SPARC and others (QEMU, 2013). This emulation is possible due to a dynamic translation mechanism. By using dynamic binary translation, it is possible to translate, typically one basic block at a time, instructions from the guest machine ISA to the host machine ISA. As expected, it is slower than execution on real hardware.

The emulator must be executed with specific QEMU options, such as the logging of all basic blocks that are translated. We modified this mechanism to insert a basic block identifier to each new basic block that is inserted in this log, before its translation. The execution count of each basic block per process is saved in a hash table. To distinguish processes, we used the context switching trace mechanism, implemented by OHA, to get the Process Identifier (PID) of the current process that is being executed. Thus, we can obtain how many times each process executed any of the basic blocks. To accomplish these features, we modified the QEMU emulation flow, as shown in Figure 4.2.

Every time there is a basic block to be executed that is not cached, QEMU translates it into a Translated Block (TB), a basic block composed of instructions implemented with the ISA of the host machine. Whenever the program counter (PC) is updated, the respective TB is searched. Whether it is not found (i.e., not cached), the basic block is

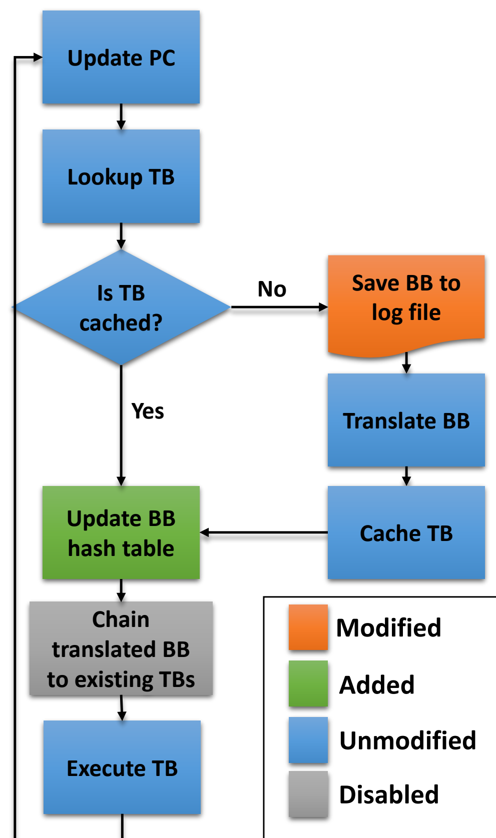

Figure 4.2: Modified QEMU emulation flow

translated and saved to the BB log file. On the other hand, if it is cached, it is loaded without the need to translate it again.

Once this step is completed, the hash table must be updated. There are two possible situations: the current process never executed the current TB (because it was just translated or because it was cached by another process) or the process already executed the current TB. In the former case, a new hash entry is created for this (PID,BB) pair. In the latter, the entry was already created, and therefore, we simply update its counter to count one more execution of the given BB.

The process of reading a TB from cache is slow. Therefore, QEMU implements a TB chaining mechanism. Every time a TB returns, QEMU tries to chain the current TB to the next TB that will be executed so it does not need to search the TBs one by one, this is done by speculating which is the next translated block to be executed after the current TB's branch. However, this mechanism had to be removed because the hash table is updated after each TB is searched or created. Therefore, we can have the original executed basic blocks and have a correct counting for each basic block and process. Finally, after the hash was updated, the TB is executed.

We modified the closing of the Virtual Network Connection (VNC) to conveniently trigger the saving of the hash table to a file so it can be posteriorly processed. VNC is a protocol that allows an operating system to be viewed and controlled remotely over the Internet. Although Android's emulator control with mouse and keyboard does not work through a VNC connection, Android Debug Bridge (ADB) is supported, and can be easily used to run and control applications. ADB is a command line tool that allows communication with an emulator instance or a connected Android device (ANDROID-ADB, 2013).

In addition, the emulator uses an Android Virtual Device (AVD) to determine the device's configuration that will be emulated. An AVD is an emulator configuration that defines software and hardware configuration, so an actual device can be modeled/emulated (ANDROID-AVD, 2013). Also, this configuration can be modified prior to execution and it can be customized to meet the device's configuration that the user considers to be the most appropriate. For instance, it is possible to emulate a device running Android 4.2.2 on an ARM processor, with 512MB RAM, 1GB internal storage and 2GB SD card; or an Android 4.0.3 on a MIPS processor, with 256MB RAM, 512MB internal storage and 4GB SD card.

4.2 Graphical User Interfaces

4.2.1 Instruction categorization GUI

A Graphical User Interface for instruction categorization ("Instruction categorization GUI" from Figure 4.1) has the goal to create an instruction information file ("Instr. Info File" from Figure 4.1) with categorized instructions in addition to cycle and the power costs for each category. These categories can be customized in order to meet the developer's requirements. For instance, BEQ, BNE and BLT instructions may be inserted in the Conditional Branch category and other categories may be created as well, such as Load, Store, Unconditional Branches and Arithmetic and Logical operations. This GUI is presented in Figure 4.3. The left table contains the instructions that does not have any category. These undefined instructions, if any, will be saved to the instruction information file with the default category, called "Undefined". The right table contains the instructions of a given category, selected by the combo box of "Selected Category".

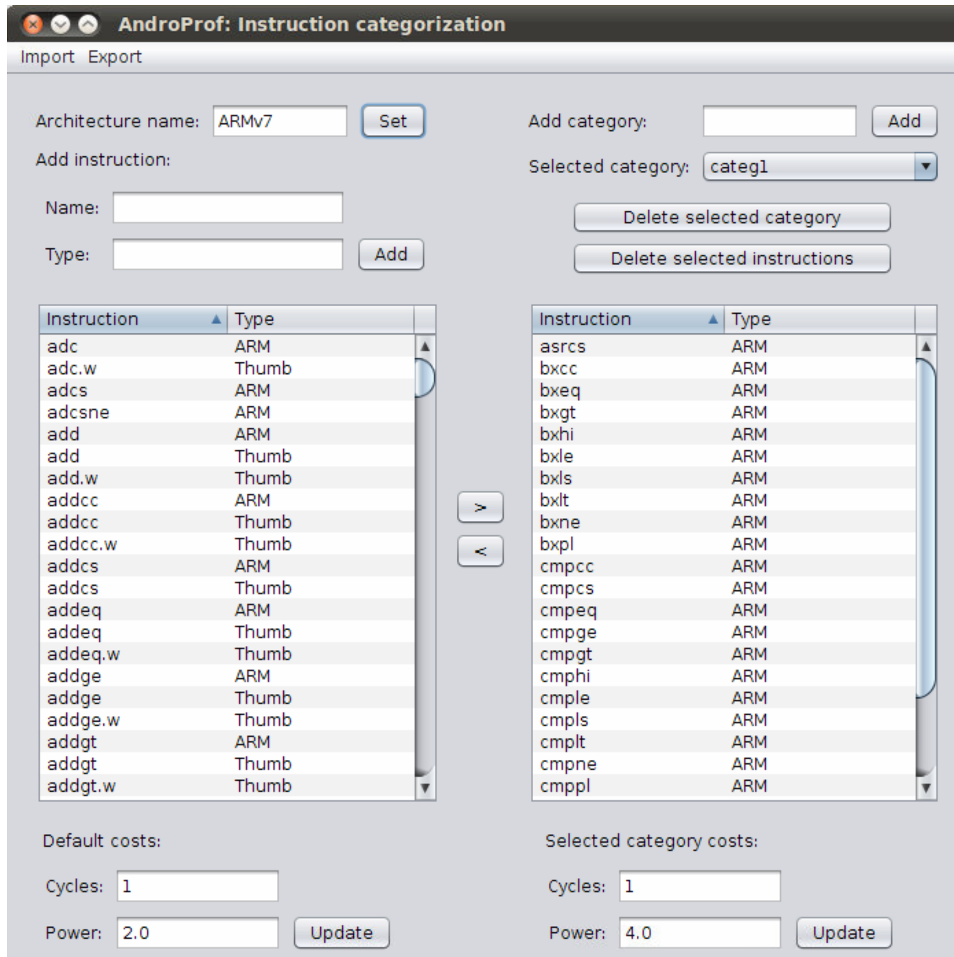

Figure 4.3: AndroProf instruction categorization GUI

Some actions that can be taken on this GUI are: create instruction information files for different ISAs, for instance: ARMv7 or MIPS; add or remove instructions manually; create categories for the instructions, specifying the average cycle cost and power dissipation of the instructions. Moreover, each architecture can have different instruction types. For instance, in an ARM architecture is possible to define either Thumb or ARM instructions. Thumb instructions are a subset of ARM instructions with reduced bit encoding size. Therefore, Thumb instructions need less memory than ARM instructions: they are 16 bits long, while ARM instructions are 32 bits long. However, not every ARM instruction has an equivalent Thumb instruction.

In addition, BB log files can be imported, so every instruction that was executed can be categorized easily; this makes the process of creating a new category characterization file much faster. Also, it is possible to import already created category characterization files, in order to edit it. The output file of this GUI is a XML file with the architecture/organization, created categories with their costs and instructions.

An example of this structure is an ARMv7 architecture, which has a conditional branch category with CPU cycle cost of 3 (ARM, 2013), power dissipation cost of 113 miliWatts (BAZZAZ, SALEHI e EJLALI, 2013). This category comprise the following instructions: BEQ (ARM), BEQ (Thumb), BNE (ARM), BLT (Thumb), BLE (ARM), BGT (Thumb), BGE (Thumb) and other conditional branch instructions.

Figure 4.4 depicts the project organization of this graphical interface, that was developed following the Model-View-Controller (MVC) software architecture. In addition, Figure 4.5 shows an example of the structure of the instruction categorization

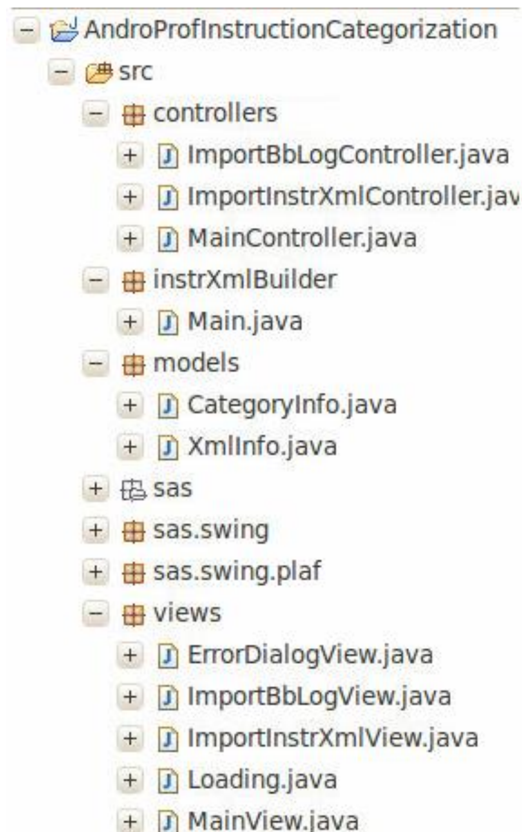

Figure 4.4: Instruction categorization GUI project

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<architectures>
    <arch name="ArchName">
        <category cycles="X" name="categ1" power="X.X">
            <instruction name="instr1" type="type1"/>
            <instruction name="instr2" type="type1"/>
            <instruction name="instr3" type="type2"/>
        </category>
        <category cycles="X" name="categ2" power="X.X">
            <instruction name="instr1" type="type1"/>
        </category>
    </arch>
</architectures>
```

Figure 4.5: Instruction categorization file XML structure

file ("Instr. Info File" from Figure 4.1), which is formatted in a XML file.

4.2.2 Analysis GUI

Finally, an analysis tool ("Analysis GUI" from Figure 4.1) with a GUI, presented in Figure 4.6, imports both created files from QEMU and the instruction categorization file and, after processing the data, it presents the analyzed data to the user for an easier

understanding of what was executed, saving all necessary data into a database. This database is necessary due to memory limitations, besides the obvious advantage of providing a way of loading previous saved architecture configurations. Some features of this GUI are:

- information about basic blocks, instructions and categories: total cycle and power costs and histograms;
- PID chart based on the total cycle or power cost of each PID. This feature allows seeing which the most costly processes that had executed are;
- performance and energy consumption estimation based on a given operation frequency and other reference data;
- import profiles (instruction characterization) for different instruction set architectures, or variations of these ISAs.

In addition, it is possible to run a set of applications automatically with the scripts that are provided. Therefore, a set of benchmarks can be executed in sequence on the emulator and their data will be automatically save. Thus, on the data analysis process, the user will

AndroProf

Import Edit Estimation Charts Benchmarks

BB Log File: /mnt/Anderson/benchmarks_SPEC/traces/sem_jit/arm/java/exec3/qemu.log

BB Info File: /mnt/Anderson/benchmarks_SPEC/traces/sem_jit/arm/java/exec3/bb_info.txt

Filters:

PID = BB ID = Filter

Benchmark	PID
com.android.spec.sparse	532
com.android.spec.fft	563
com.android.spec.mpegaudio	589

PID	BB ID	PC	Cycle Cost	Power Cost	BB Counter	Total Cycle ...	Total Power C...
66	365104	0x401cde94	15	30	64146777749	962201666235	1924403332...
66	468886	0x401d3558	18	36	29606547030	532917846540	1065835693...
66	455885	0x401d3558	18	36	21140223147	380524016646	761048033292
66	427179	0x401d3558	18	36	14089676715	253614180870	507228361740
66	365233	0x401cde94	15	30	14483982734	217259741010	434519482020
66	426729	0x401d3558	18	36	10526381118	189474860124	378949720248
66	284916	0x401d38fc	17	34	6628088539	112677505163	225355010326
66	438279	0x401d38b4	15	30	6193517720	92902765800	185805531600
66	324239	0x401d329c	11	22	3881824603	42700070633	85400141266
66	400540	0x401d38b4	15	30	2839283956	42589259340	85178518680
66	449202	0x401d3558	18	36	1916557001	34498026018	68996052036
66	448880	0x401d3558	18	36	1347650334	24257706012	48515412024
66	284915	0x401d38f0	3	6	7276823463	21830470389	43660940778
589	406217	0x4061ae68	39	78	507438972	19790119908	39580239816
66	103315	0x401d3558	18	36	864343455	15558182190	31116364380
589	406221	0x40618394	27	54	532926944	14389027488	28778054976
589	406220	0x40618420	22	44	532928181	11724419982	23448839964
601	111202	0x407daec0	7	14	1608596796	11260177572	22520355144
601	101795	0x407dd3c0	10	20	960114528	9601145280	19202290560
532	406217	0x4061ae68	39	78	246150919	9599885841	19199771682
589	250515	0x4061d8f0	13	26	712009614	9256124982	18512249964
589	467951	0x40662064	26	52	319234837	8300105762	16600211524
544	156691	0x407daf00	8	16	1024037023	8192296184	16384592368
601	447201	0x407daeec	5	10	1608575718	8042878590	16085757180
544	106577	0x407daac0	13	26	615591081	8002684053	16005368106
66	278224	0x401d38fc	17	34	454614763	7728450971	15456901942

< Previous page   Next page >

Figure 4.6: AndroProf analysis GUI

know the PID of the application of interest, and its data can be evaluated.

Figure 4.7 depicts the project of the analysis GUI, as the instruction categorization GUI, was developed with MVC software architecture. Figure 4.8 shows the Entity-Relationship (ER) diagram of the database that was implemented.

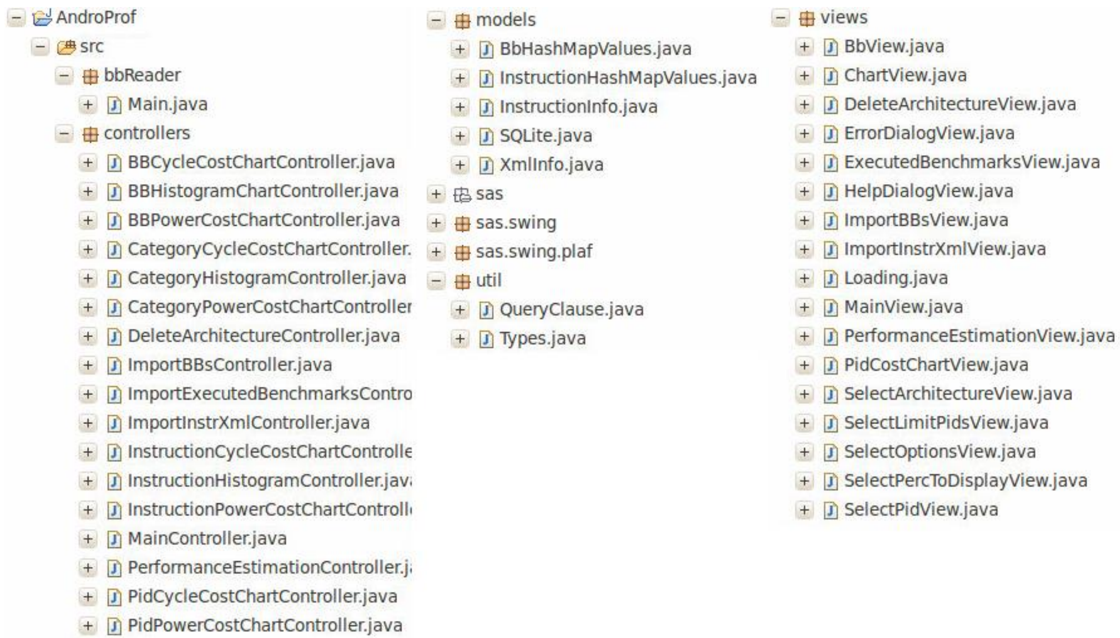

Figure 4.7: Analysis GUI project

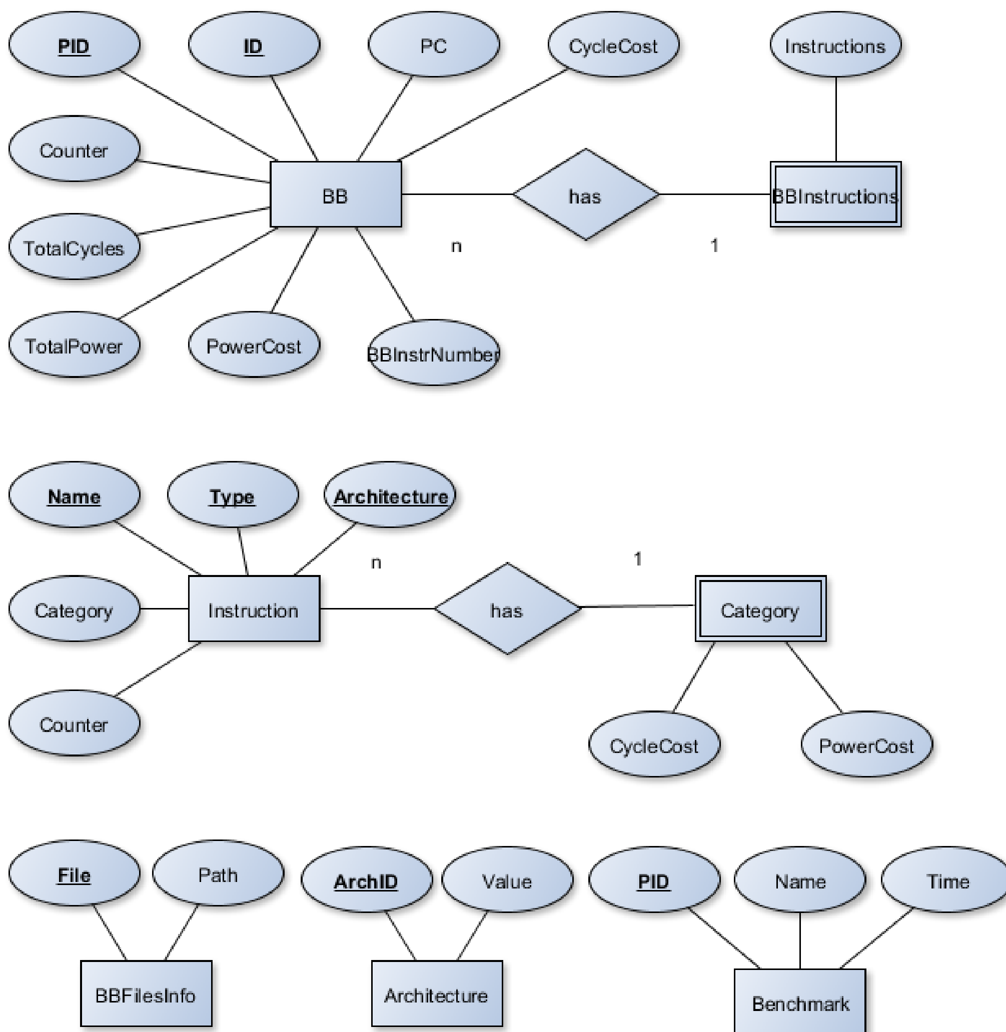

Figure 4.8: Database ER diagram

4.3 Environment setup

The tracing tool runs on the Ubuntu operating system. However, it can be executed on any other OS (e.g.: Windows and Mac OS X), as long as the Android source can be checked out from the tool's Git repository that is available in the following link: "https://bitbucket.org/AndersonSartor/androprof-release". Also, the SDK needs to be compiled (ANDROID-SOURCE, 2013).

QEMU modifications were written in the C language, since the QEMU code was also written in C. The GUIs run on any operating system that supports the Java Runtime Environment (JRE), necessary to run Java applications. We chose Java language because of its portability.

We also provide bash scripts to make the emulation process easier. For instance, SDK compilation, emulator call and automated benchmarks execution. Finally, for the database, we used SQLite3, a library that implements a serverless, transactional SQL database engine (SQLITE, 2013). This is useful because the user does not need to have a specific Database Management System (DBMS), like Oracle™ or MySQL™, in the host computer. As can be observed, all employed tools that comprise the framework were chosen so it can be as more platform independent and open-source as possible.

5 RESULTS AND CASE STUDY

To evaluate the emulation speed of the proposed tool, we compared the execution times of our implementation with original Android's QEMU. The Android Emulator's and QEMU's options used in both emulator's executions were the same. Thus, the overhead of our tool is due to the disabling of the chaining mechanism, in addition to the cost of the added profiling process. Comparing the boot times, the original emulator boots in about 50 seconds, while our modification increases the boot time to 6 minutes, on average.

The configuration of the host computer that we used to simulate the Android device was the following: Intel Core i7 860 2.80GHz, 8GB RAM, Samsung HD103SI HDD; and the AVD configuration used was: Android 4.0.3, ARM (armeabi-v7a) CPU, 512MB RAM. An Android benchmark set was created, based on the JVM SPEC 2008 benchmarks (SPEC, 2013).

Table 5.1 presents the average execution time between three executions of each benchmark. The minimum average speed down of our QEMU modification, in the tested benchmarks, was 3 times in the SQLite benchmark, while the maximum was 21 times in the MPEG Audio benchmark. We have this large variation in the speed down due to the

Table 5.1: Average execution time comparison

	Original QEMU ①	**Modified QEMU ②**	
Benchmark	**Time (s)**	**Time (s)**	**Speed down (②/①)**
SciMark FFT	89,67	1.121,33	12,51
SciMark LU	715,00	8.825,00	12,34
SciMark Monte Carlo	1.347,00	27.374,00	20,32
SciMark SOR	458,67	5.861,67	12,78
SciMark Sparse	595,00	6.820,67	11,46
Serial	1,00	16,33	16,33
Crypto AES	1.520,00	20.816,33	13,69
Crypto RSA	32,00	512,00	16,00
Crypto Sign Verify	925,00	12.712,00	13,74
Compress	1.291,00	20.471,33	15,86
MPEGAudio	765,00	16.111,33	21,06
SQLite	154,67	466,67	3,02
		Average:	14,09

benchmarks' characteristics: SQLite is a data-bound application, while MPEG Audio is CPU-bound, which impacts on the overhead depending on how intense was the use of our hash table structure. That is, CPU-bound applications intensively use the hash table, due to the increased number of executed instructions; therefore, these applications have a larger speed down on our tool when compared to data-bound applications.

The average speed down of our QEMU modification, also presented in Table 5.1, in relation to original QEMU is about 14 times. This cost is paid in order to increase the range of information that is traced from the application's execution.

To validate our tool, we compared the execution results from three different benchmarks that we developed based on JVM SPEC 2008 benchmarks (SPEC, 2013). In addition, the most exclusive time consuming methods were converted to use the JNI. This benchmark set was composed of benchmarks with the following characteristics: mathematics (SciMark Sparse and SciMark FFT) and multimedia processing (MPEG audio).

The AVD configuration used was: Android 4.0.3 with 512MB RAM for both, ARM (armeabi-v7a) and MIPS, CPUs. Also, the JIT compiler was enable in both devices.

5.1 Execution time, number of BBs and number of instructions comparison between ARM/MIPS and Java/JNI

Table 5.2 presents the average execution time between three executions, number of basic blocks and number of instructions for each Java and JNI executed benchmarks on ARM and MIPS architectures, and Figure 5.1 presents this data in a graphical form for an easier visualization. Comparing ARM and MIPS, both execution times and number of

Table 5.2. ARM and MIPS comparison

ARM			
Java	**Time (ms)**	**#BBs**	**#Instrs**
SciMark Sparse	9.126.874	6.494.177.530	57.620.647.133
SciMark FFT	1.437.846	857.892.746	6.757.657.737
MPEG Audio	19.635.264	13.455.264.830	119.524.591.950
JNI			
SciMark Sparse	1.022.425	703.986.571	6.217.487
SciMark FFT	320.962	222.462.041	1.944.832.689
MPEG Audio	44.124.298	29.272.511.931	260.106.923.761
MIPS			
Java	**Time (ms)**	**#BBs**	**#Instrs**
SciMark Sparse	9.307.618	6.634.976.525	75.351.389.129
SciMark FFT	1.397.534	946.438.842	10.719.421.833
MPEG Audio	20.245.078	14.309.245.120	162.507.503.608
JNI			
SciMark Sparse	906.160	649.293.812	7.340.895.994
SciMark FFT	398.672	229.410.566	2.300.136.199
MPEG Audio	48.343.377	33.923.478.164	385.247.616.970

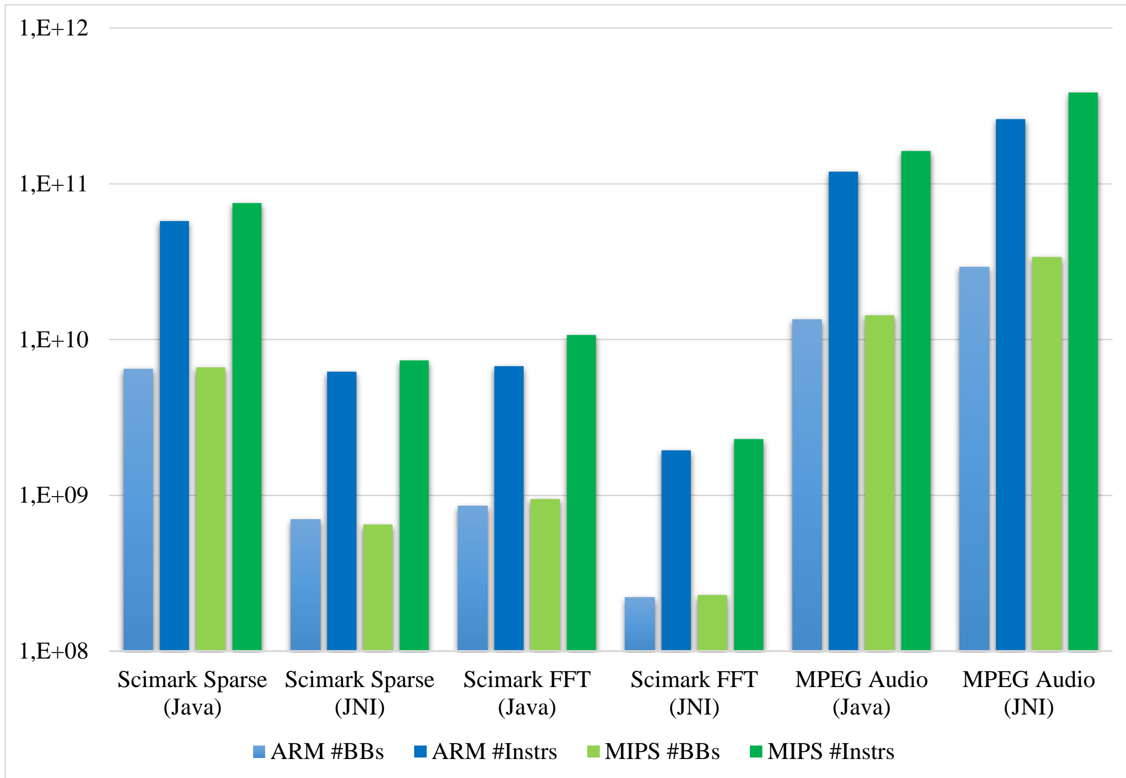

Figure 5.1: #BBs and #instructions comparison for ARM and MIPS

basic blocks were almost the same. However, MIPS executed more instructions than ARM, because its ISA comprises simpler instructions (more RISC oriented).

When one compares Java execution with JNI, it can be observed a speed up of 8,92 times when using JNI on ARM, and a speedup of 10.27 times on MIPS for the SciMark Sparse execution; and 4,48 on ARM and 3,5 on MIPS for the SciMark FFT. However, MPEG Audio benchmark presented a speed down for both processors, 2,25 on ARM and 2,39 on MIPS. Considering the average number of instructions per basic block, the average for all benchmarks was of 8 instructions/BB on ARM and 11 instructions/BB on MIPS.

The usage of JNI can be beneficial or not for a specific application, depending on several factors, such as: the overhead of accessing Java attributes and calling a Java method through a native method, the number of times that this method is called, the complexity of the method, etc. In our tests, both SciMark benchmarks had a decrease on the execution time when using JNI; these two benchmarks have one method that was executed for 80-90% of the execution time and this method performs calculation with arrays and matrices.

On the other hand, MPEG Audio had its execution time increased, this benchmark has five methods that take about 10% of the execution time each. Moreover, some of these methods are simple and called several times, which impacts on the performance as the overhead created overcomes the benefits of programing in native code.

The decision of developing some features of the application in native code depends on the type of application that is being developed. For example, many games and web browsers use native code to speed up the execution. This discussion highlights the importance of supporting the profile of native methods.

Moreover, supporting multiple platforms is extremely important for profiling tools, because in today's market we have multiple processors architectures (e.g.: ARM, MIPS and x86) and multiple processors organizations (e.g.: ARM Cortex A8, Cortex A9, Cortex A15, etc.) that need profiling tools capable of working along with all these different platforms.

5.2 Energy and performance estimation

To estimate the energy and performance, we obtained data on processor power dissipation and Cycles per Instruction (CPI) from the Excel spreadsheet data embedded in the "A Detailed Analysis of Contemporary ARM and x86 Architectures" Technical Report (BLEM, MENON e SANKARALINGAM, 2013). As the Technical Report does not provide power dissipation per instruction or category, we did not use this feature, even though it is available in AndroProf (we have considered that all instructions had the same power dissipation).

Table 5.3 presents the data obtained from the arithmetic average of all executed benchmarks from the Excel file embedded in technical report. With this information, we can estimate the main memory access rate and also consider the power dissipation given by these accesses.

The main memory access rate is of about 1% for both Cortex A8 and Cortex A9. Also, we considered the processors' frequency to be 1GHz for the Cortex A8 and 2GHz for the Cortex A9 (ARM-CORTEX-A, 2013).

Main memory read/write power dissipations were obtained from CACTI 6.5 (CACTI, 2013), with the following configuration: 512MB, 8 banks, block size of 64 bytes and 45nm technology; which results in an access time of 8,26ns and 2,66nJ/2,56nJ of read/write energy, respectively. We will consider the average of these two values as the

Table 5.3: ARM Power and cycles information

ARM Cortex A8	
Processor power dissipation (W)	0,89
CPI	13,03
Number of Instructions	39.171.751.755
Number of cache misses	521.966.483
ARM Cortex A9	
Processor power dissipation (W)	1,25
CPI	2,37
Number of Instructions	39.372.723.114
Number of cache misses	451.238.875

Operation frequency (GHz): 1
Processor power dissipation (W): 0.89
CPI: 13.03
Base number of instructions: 39171751755
Base number of cache misses: 521966483
Memory access memory consumption (nJ): 2.61
Results
Time estimation (s): 739.45535128975
Estimated power dissipation (W): 0.8926691053428465
Estimated energy consumption (J): 660.0889468768014
Calculate

Figure 5.2: Time and power estimation

energy consumption of each memory access (2,61nJ), disregarding if it was a memory read or write, to simplify calculation.

With this information in hand we can estimate the number of cycles and the power dissipated with the help of the GUI from Figure 5.2, which uses the following formulas:

$$EstimCycles = CPI * \#Instructions \quad (1)$$

$$\text{ExecTime} = \frac{\text{EstimCycles}}{\text{ProcFrequency}} \quad (2)$$

$$MemAcPwDis = \frac{MAcEnCons * MAcRate * \#Inst}{ExecTime} \quad (3)$$

$$EstimPwDis = ProcPwDis + MemAcPwDis \quad (4)$$

Table 5.4: ARM Cortex A8 and A9 cycles comparison

	ARM Cortex A8	**ARM Cortex A9**
Java	**Estimated cycles**	**Estimated cycles**
SciMark Sparse	750.989.080.031	136.515.747.939
Scimark FFT	88.074.803.379	16.010.349.521
MPEG Audio	1.557.803.804.988	283.179.552.454
JNI		
SciMark Sparse	81.034.581.835	14.730.569.115
SciMark FFT	25.347.652.007	4.607.728.347
MPEG Audio	3.390.060.145.184	616.249.435.025
Average	982.218.344.570,69	178.548.897.066,97

Table 5.5. ARM Cortex A8 and A9 energy consumption comparison

	ARM Cortex A8	**ARM Cortex A9**
Java	**Estim. energy(J)**	**Estim. energy(J)**
SciMark Sparse	670,39	87,05
Scimark FFT	78,62	10,21
MPEG Audio	1.390,61	180,57
JNI		
SciMark Sparse	72,34	9,39
SciMark FFT	22,63	2,94
MPEG Audio	3.026,22	392,95
Average	876,8	113,85

Table 5.4 presents the average of the estimated cycles for both Cortex A8 and Cortex A9 based on the information of the technical report and the data we traced from our applications. Comparing the average estimated cycles for our benchmarks in both Cortex A8 and Cortex A9, we have a 5,5 A8/A9 ratio due to the difference between both processors CPI.

CPI and number of cache misses vary depending on the application, therefore the power dissipation varies too. However, as we considered the CPI and the cache misses to be the same to all of our benchmarks, they all presented the same power dissipation. Calculating the power dissipation by (4), we have a 0,8927W power dissipation for Cortex A8 and 1,2753W for Cortex A9 for each executed benchmark, which results in an A8/A9 power dissipation ratio of approximately 0,7.

Table 5.5 presents the A8 and A9 energy consumption comparison, which gives us an A8/A9 average energy consumption ratio of 7,7. This shows how the power dissipation and energy consumption can vary depending on the organization of the same processor architecture, feature that is also available on our tool.

5.3 QEMU modification impact for Java and JNI applications on ARM and MIPS architectures

Table 5.6 presents the comparison between the average execution time in the unmodified version of the emulator and our modified version. The speed down for ARM architecture varied between 11,30 and 26,02, and the average was 18,30. For MIPS architecture, the speed down varied between 1,71 and 9,98, with an average of 5,51. This variation in the speed down, as aforementioned, is due to the disabling of the chaining mechanism and how the application uses the hash table: more accesses and more insertions in the hash table will imply on a larger overhead on the execution time. Comparing the speed down between each Java/JNI application, it stayed relatively the same for both ARM and for MPEG Audio on MIPS. However, SciMark Sparse and FFT presented a difference of 3 times in the speed down of the JNI version when compared with the Java version. This difference may be due the huge amount (GB scale) of information that original MIPS version saves to QEMU log file, increasing the execution time depending on how much information is saved besides the basic blocks. In addition, this huge amount of information that is saved on original MIPS version affects its

Table 5.6. ARM and MIPS QEMU modification overhead

	ARM			MIPS		
QEMU	**Original**	**Modified**		**Original**	**Modified**	
Java	**Average time (ms) ①**	**Average time (ms) ②**	**Speed down (②/①)**	**Average time (ms) ③**	**Average time (ms) ④**	**Speed down (④/③)**
SciMark Sparse	595.000	9.126.874	15,34	932.383	9.307.618	9,98
SciMark FFT	89.667	1.437.846	16,04	318.407	1.397.534	4,39
MPEG Audio	765.000	19.635.264	25,67	2.722.453	20.245.078	7,44
JNI						
SciMark Sparse	66.237	1.022.425	15,44	274.228	906.16	3,30
SciMark FFT	28.398	320.962	11,30	233.453	398.672	1,71
MPEG Audio	1.695.486	44.124.298	26,02	7.765.469	48.343.377	6,23
		Average:	18,30		**Average:**	5,51

performance, as we removed most of this additional information that was saved because it would not be useful now, the speed down from MIPS was lower than on ARM.

5.4 JIT compiler evaluation

To evaluate the JIT compiler, we disabled it in the emulator and obtained the applications' data for the same set of benchmarks. In the emulator, a new system image needs to be created in order to make this change in the system propriety permanent; otherwise, it will load the default system image in the next reboot, which loads the configuration that has the JIT enabled.

The results are presented in Table 5.7, comparing it to Table 5.2, it can be noticed that there was no real speed down in the applications, it stayed almost the same (20% variation, which is usual for the Android Emulator). Therefore, it can be implied that or the applications are not suitable for JIT optimization at all or the JIT compilation is not fully supported by the emulator, which may not consider this modification in its system's properties. To test the second hypothesis, we did the same test in both MIPS and x86 emulator, and no relevant speed down was achieved either. Then, the same applications were executed in a Motorola Atrix MB860, to test the first hypothesis; its results are

Table 5.7: Execution time on the emulator (ARM) with JIT disabled

ARM			
Java	**Time (ms)**	**#BBs**	**#Instrs**
SciMark Sparse	9.309.768	6.548.418.024	58.126.721.664
SciMark FFT	1.378.704	961.762.098	8.513.355.694
MPEG Audio	20.666.729	14.264.664.658	126.753.132.018
JNI			
SciMark Sparse	959.598	647.920.652	5.728.513.790
SciMark FFT	330.707	226.595.211	1.984.727.422
MPEG Audio	46.721.163	31.488.794.368	279.769.953.812

presented in Table 5.8. Therefore, it can be implied that the emulator does not support, so far, the execution of applications with JIT either enabled or disabled.

The results in Table 5.8 were obtained from executing the benchmarks on a Motorola Atrix with an unofficial ROM image of Android 4.0.3, this Android version was chosen in order to be the same version that was being used in the emulator. In addition, the results represent the behavior expected from the applications. That is, for Java applications, a speed up from two to almost four times can be achieved when the JIT compiler is enabled. In addition, when disabling the JIT, JNI applications do not present a large speed down as Java applications do because, as the most costly methods are implemented in native code, it does not execute on the DVM, consequently it does not get affected by the JIT compiler.

Table 5.8: JIT speed up - Motorola Atrix

Atrix	**With JIT**	**Without JIT**	**Speed up**
Java	**Time (ms) ①**	**Time (ms) ②**	**②/①**
Scimark Sparse	27.940,33	70.512,67	2,52
Scimark FFT	8.851,00	18.759,00	2,12
MPEG Audio	41.942,33	152.220,00	3,63
JNI			
Scimark Sparse	12.954,33	13.309,00	1,03
Scimark FFT	6.687,33	8.240,00	1,23
MPEG Audio	133.411,67	203.889,67	1,53

6 CONCLUSION AND FUTURE WORK

There are only a few options of tracing and profiling tools available to Android developers, and all of these options have limitations. Due to the difficulty of getting useful data of Android applications' execution, in this work we presented a tool to generate more relevant information (e.g. power dissipation estimation and information about the basic blocks) for an easier data analysis at earlier designs stages. As future work, we will create default profiles, with power and cycles costs of the main instruction categories, for multiple processors architectures and organizations. Therefore, one can use the tool to estimate power and cycles with more accuracy for multiple platforms, helping the developer to make project choices based on the behavior of the application that is being developed and also by comparing this behavior with other applications.

We will also modify the Android Emulator to trigger the saving of the BB data. Therefore, the VNC connection will no longer be needed. By doing this, mouse and keyboard will be supported to control the emulator and the communication with our tool will be done directly through the emulator. In addition, the x86 architecture will be fully compatible and method information will be generated. Moreover, a cache simulator will be implemented in order to have a more accurate cycles and power estimation.

Our tool presents a large speed down, depending on the application's characteristics, because we added a new layer to the QEMU execution flow and the chaining mechanism was disabled. Therefore, we will also study means to trace the applications with chaining enabled and to use a better-optimized hash. In addition, the tool will provide trace method information and cache events simulation.

REFERENCES

ANDROID-ADB, 2013. **Android Debug Bridge**. Available at: <http://developer.android.com/tools/help/adb.html>. Last access: July 2013.

ANDROID-ART, 2013. **Introducing ART**. Available at: <https://source.android.com/devices/tech/dalvik/art.html>. Last access: November 2013.

ANDROID-AVD, 2013. **Managing Virtual Devices**. Available at: <http://developer.android.com/tools/devices/index.html>. Last access: May 2013.

ANDROID-DALVIK, 2013. **Bytecode for the Dalvik VM**. Available at: <http://source.android.com/tech/dalvik/dalvik-bytecode.html>. Last access: May 2013.

ANDROID-NATIVE ACTIVITY, 2013. **Native Activity**. Available at: <http://developer.android.com/reference/android/app/NativeActivity.html>. Last access: May 2013.

ANDROID-PROJECT, 2013. **Android Open Source Project**. Available at: <http://source.android.com/>. Last access: May 2013.

ANDROID-SOURCE, 2013. **Android Developers - Source**. Available at: <http://source.android.com/source/developing.html>. Last access: June 2013.

ANDROID-TRACEVIEW, 2013. **Profiling with Traceview and dmtracedump**. Available at: <http://developer.android.com/tools/debugging/debugging-tracing.html>. Last access: May 2013.

ARM, 2013. **Instructions cycle count**. Available at: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0214b/CHDGHAAG.html>. Last access: September 2013.

ARM-CORTEX-A, 2013. **Cortex-A Series**. Available at: <http://www.arm.com/products/processors/cortex-a/index.php.>. Last access: September 2013.

BAZZAZ; SALEHI, M.; EJLALI, A. An Accurate Instruction-Level Energy Estimation Model and Tool for Embedded Systems. **Instrumentation and Measurement, IEEE Transactions on**, v. 62, n. 7, p. 1927-1934, 2013. Available at: <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6478810>.

BI INTELLIGENCE, 2013. Available at: <https://intelligence.businessinsider.com/welcome>. Last access: May 2013.

BLEM, E.; MENON, J.; SANKARALINGAM, K. A detailed Analysis of the Contemporary ARM and x86 Architectures. **Report, UW-Madison Technical**, 2013.

BORNSTEIN, 2010. Dalvik JIT. **Android Developers Blog**. Available at: <http://android-developers.blogspot.com.br/2010/05/dalvik-jit.html>. Last access: May 2013.

CACTI, 2013. Available at: <http://www.cs.utah.edu/~rajeev/cacti6/>. Last access: September 2013.

CHO, M. et al. **AndroScope for detailed performance study of the android platform and its applications**. Consumer Electronics (ICCE), 2012 IEEE International Conference on. [S.l.]: [s.n.]. 2012. p. 408-409.

DONG, M.; ZHONG, L. **Self-constructive high-rate system energy modeling for battery-powered mobile systems**. MobiSys '11 Proceedings of the 9th international conference on Mobile systems, applications, and services. [S.l.]: [s.n.]. 2011. p. 335-348.

EHRINGER, D. **The Dalvik Virtual Machine Architecture**, March 2012.

IDC. **Android and iOS Combine for 91.1% of the Worldwide Smartphone OS Market in 4Q12 and 87.6% for the Year**, 2013. Available at: <http://www.idc.com/getdoc.jsp?containerId=prUS23946013>. Last access: May 2013.

MARSSX86, 2013. **Micro-ARchitectural and System Simulator for x86-based Systems**. Available at: <http://marss86.org/~marss86/index.php/Home>. Last access: June 2013.

OHA, 2013. Open Handset Alliance Members. Available at: <http://www.openhandsetalliance.com/oha_members.html>. Last access: May 2013.

POWERTUTOR, 2011. Available at: <http://powertutor.org/>. Last access: November 2013.

PTLSIM, 2013. **PTLsim x86-64 Cycle Accurate Processor Simulation Design Infrastructure**. Available at: <http://www.ptlsim.org/>. Last access: June 2013.

QEMU, 2013. **QEMU - open source procesor emulator**. Available at: <http://wiki.qemu.org/Main_Page>. Last access: May 2013.

QUALCOMM DEVELOPERS, 2013. **Trepn Profiler**. Available at: <https://developer.qualcomm.com/mobile-development/performance-tools/trepn-profiler>. Last access: November 2013.

SPEC, 2013. **SPEC Benchmark Downloads**. Available at: <http://www.spec.org/download.html>. Last access: June 2013.

SQLITE, 2013. Available at: <http://www.sqlite.org/>. Last access: May 2013.

THACH, D. et al. **Fast cycle estimation methodology for instruction-level emulator**. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012. [S.l.]: [s.n.]. 2012. p. 248-251.

TREFIS TEAM, 2013. **Can Intel Challenge ARM's Mobile Dominance?** Available at: <http://www.forbes.com/sites/greatspeculations/2012/11/09/can-intel-challenge-arms-mobile-dominance/2/>. Last access: July 2013.

YOON, H.-J. **A Study on the Performance of Android Platform**, Internation Journal on Computer Science and Engineering (IJCSE), 4, n. 4, April 2012. 532-537.

ZHANG, L. et al. **Accurate online power estimation and automatic battery behavior based power model generation for smartphones**. Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on. [S.l.]: [s.n.]. 2010. p. 105-114.

APPENDIX A: JAVA APPLICATIONS EXECUTION TIME

Table A.1: Java applications execution time on original QEMU (ARM)

Original QEMU	Exec. 1		Exec. 2		Exec. 3	
Java	**PID**	**Time (s)**	**PID**	**Time (s)**	**PID**	**Time (s)**
1) scimark fft	1181	1200	589	964	1074	1200
2) scimark lu	1223	8113	644	8393	663	9969
3) scimark monte carlo	1374	29708	690	29278	608	23136
4) scimark sor	1269	5462	748	6323	982	5800
5) scimark sparse	1313	8076	984	6236	1027	6150
6) serial	1436	16	862	19	1117	14
7) crypto aes	1476	18672	1084	20732	851	23045
8) crypto rsa	562	409	902	578	1158	549
9) crypto signverify	602	12009	1032	11694	929	14433
10) compress	651	20770	795	21941	710	18703
11) mpegaudio	706	18289	1156	15039	780	15006
12) sqlite	770	468	943	468	1198	464

Table A.2: Java applications execution time on modified QEMU (ARM)

Modified QEMU	Exec. 1		Exec. 2		Exec. 3	
Java	**PID**	**Time (s)**	**PID**	**Time (s)**	**PID**	**Time (s)**
1) scimark fft	546	90	465	90	513	89
2) scimark lu	586	702	513	703	554	740
3) scimark monte carlo	625	1323	552	1333	594	1385
4) scimark sor	666	467	594	452	636	457
5) scimark sparse	705	628	634	589	675	568
6) serial	745	1	685	1	716	1
7) crypto aes	784	1523	724	1500	756	1537
8) crypto rsa	826	34	766	31	797	31
9) crypto signverify	865	959	805	901	837	915
10) compress	467	1263	847	1298	877	1312
11) mpegaudio	546	774	887	741	921	780
12) sqlite	507	157	962	153	963	154

APPENDIX B: JAVA AND JNI APPLICATIONS TIME, #BBS, #INSTRUCTIONS ON ARM AND MIPS

Table B.1: Java/JNI applications on ARM (execution 1)

ARM	Exec 1			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	524	9.952.442	7.175.033.543	63.649.663.756
Scimark FFT	550	1.491.727	1.050.006.489	9.293.766.960
MPEG Audio	633	20.132.107	13.969.674.398	124.129.261.344
JNI				
Scimark Sparse	535	1.200.588	831.938.143	7.348.385.437
Scimark FFT	546	331.967	243.906.208	2.135.079.624
MPEG Audio	590	49.579.028	34.073.346.887	302.755.747.196

Table B.2: Java/JNI applications on ARM (execution 2)

ARM	Exec 2			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	541	8.800.688	6.392.049.364	56.750.218.825
Scimark FFT	592	1.411.419	989.290.087	8.753.545.538
MPEG Audio	612	16.717.209	10.532.954.021	93.599.318.742
JNI				
Scimark Sparse	552	907.459	613.819.947	5.420.042.394
Scimark FFT	579	327.335	223.512.232	1.954.384.697
MPEG Audio	608	44.896.131	29.792.921.446	264.716.639.591

Table B.3: Java/JNI applications on ARM (execution 3)

ARM	Exec 3			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	543	8.627.491	5.915.449.683	52.462.058.818
Scimark FFT	634	1.410.392	534.381.662	2.225.660.712
MPEG Audio	617	22.056.475	15.863.166.071	140.845.195.764
JNI				
Scimark Sparse	542	959.228	666.201.623	5.884.033.981

Scimark FFT	569	303.583	199.967.684	1.745.033.746
MPEG Audio	598	37.897.735	23.951.267.460	212.848.384.497

Table B.4: Java/JNI applications on MIPS (execution 1)

MIPS	**Exec 1**			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	604	10.487.097	7.259.923.444	82.479.997.535
Scimark FFT	662	1.481.143	1.025.728.420	11.627.696.468
MPEG Audio	680	20.296.199	13.802.013.951	156.756.864.319
JNI				
Scimark Sparse	744	931.673	677.456.878	7.659.968.520
Scimark FFT	600	444.305	155.087.887	911.216.558
MPEG Audio	799	50.039.346	35.017.503.392	397.617.642.551

Table B.5: Java/JNI applications on MIPS (execution 2)

MIPS	**Exec 2**			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	600	7.712.524	5.669.756.824	64.391.879.180
Scimark FFT	704	1.501.045	1.049.075.048	11.889.875.728
MPEG Audio	673	19.000.087	13.878.762.485	157.637.243.834
JNI				
Scimark Sparse	757	953.706	658.027.071	7.433.314.612
Scimark FFT	785	391.464	267.990.852	3.010.294.383
MPEG Audio	814	48.327.677	33.464.711.742	380.044.701.743

Table B.6: Java/JNI applications on MIPS (execution 3)

MIPS	**Exec 3**			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	581	9.723.233	6.975.249.308	79.182.290.671
Scimark FFT	746	1.210.414	764.513.059	8.640.693.302
MPEG Audio	655	21.438.947	15.246.958.924	173.128.402.672
JNI				
Scimark Sparse	578	833.101	612.397.487	6.929.404.851
Scimark FFT	605	360.247	265.152.959	2.978.897.657
MPEG Audio	633	46.663.107	33.288.219.359	378.080.506.615

APPENDIX C: JAVA AND JNI APPLICATIONS TIME, #BBS, #INSTRUCTIONS ON ARM EMULATOR WITH JIT DISABLED

Table C.1: Java/JNI applications on ARM without JIT (execution 1)

ARM without JIT	Exec 1			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	536	9.313.152	6.572.364.308	58.360.163.878
Scimark FFT	567	1.396.684	981.508.892	8.687.760.472
MPEG Audio	593	20.106.173	13.716.040.321	121.787.684.906
JNI				
Scimark Sparse	515	1.001.531	698.239.912	6.178.322.405
Scimark FFT	540	359.485	255.732.490	2.241.238.529
MPEG Audio	567	49.740.040	34.189.793.798	303.777.013.368

Table C.2: Java/JNI applications on ARM without JIT (execution 2)

ARM without JIT	Exec 2			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	526	8.799.521	6.381.261.714	56.618.080.751
Scimark FFT	557	1.423.476	1.011.203.401	8.950.055.699
MPEG Audio	583	21.624.236	15.322.115.509	136.127.668.597
JNI				
Scimark Sparse	517	761.299	462.299.765	4.078.783.875
Scimark FFT	545	270.415	169.363.627	1.481.465.926
MPEG Audio	571	42.723.867	28.100.681.862	249.671.069.177

Table C.3: Java/JNI applications on ARM without JIT (execution 3)

ARM without JIT	Exec 3			
Java	**PID**	**Time (ms)**	**#BBs**	**#Instrs**
Scimark Sparse	532	9.816.631	6.691.628.049	59.401.920.363
Scimark FFT	563	1.315.951	892.574.002	7.902.250.912
MPEG Audio	589	20.269.778	13.755.838.144	122.344.042.552

JNI				
Scimark Sparse	526	1.115.965	783.222.280	6.928.435.089
Scimark FFT	557	362.221	254.689.516	2.231.477.812
MPEG Audio	583	47.699.582	32.175.907.445	285.861.778.892

APPENDIX D: JAVA AND JNI APPLICATIONS ON MOTOROLA ATRIX WITH AND WITHOUT JIT

Table D.1: Java/JNI applications on Motorola Atrix with JIT

Motorola Atrix			
With JIT	**Exec 1**	**Exec 2**	**Exec 3**
Java	**Time (ms)**	**Time (ms)**	**Time (ms)**
Scimark Sparse	27.948	27.838	28.035
Scimark FFT	9.397	8.161	8.995
MPEG Audio	42.344	41.559	41.924
JNI			
Scimark Sparse	12.618	13.417	12.828
Scimark FFT	6.681	6.170	7.211
MPEG Audio	133.559	133.268	133.408

Table D.2: Java/JNI application on Motorola Atrix without JIT

Motorola Atrix			
Without JIT	**Exec 1**	**Exec 2**	**Exec 3**
Java	**Time (ms)**	**Time (ms)**	**Time (ms)**
Scimark Sparse	70.661	70.411	70.466
Scimark FFT	18.319	19.340	18.618
MPEG Audio	152.208	152.325	152.127
JNI			
Scimark Sparse	13.543	13.071	13.313
Scimark FFT	8.847	7.494	8.379
MPEG Audio	203.593	203.480	204.596

APPENDIX E: PROJECT DESCRIPTION (TG1)

AndroProf: A Profiling tool for the Android platform

Anderson Luiz Sartor

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

Porto Alegre – RS – Brazil

andersonsartor@gmail.com

Abstract. *Tracing and profiling tools for mobile development are very limited in which and how much information they can trace or profile. They are also scarce when compared to general-purpose development tools. Due to this fact, many projects are not carried on because of the difficulty to get the desired information about the applications. For developers, good profiling information usually means that they will develop better applications. To help solving this issue, this research proposes a tool that will provide a large range of information, such as: executed basic blocks statistics, power consumption estimation and CPU cycle count estimation. In order to validate the tool, we propose a study based on a series of Android benchmarks. Each benchmark will have three different implementations: purely in Java, Java using Java Native Interface (JNI) and Java with Native Activities. In addition, each implementation of each benchmark will be tested with and without the Just-In-Time (JIT) compiler, so we will be able to measure the impact of these different ways of execution on performance and power consumption.*

1. Introduction

Mobile technology has changed the world at an unprecedented speed. As this change goes by, more people have been using smartphones and tablets to communicate, to work, and to stay connected to the world. Companies have been developing both hardware and software to meet this increasing and more demanding market.

We have focused our research on Android, a Linux-based mobile software platform that is mainly used in smartphones and tablets, since it is the world's most popular mobile platform. The cut Android had in the smartphone market in 2011 was about 50% and, just one year later, that cut reached almost 70% [1], mainly because the decrease in sales of smartphones with Symbian or BlackBerry operating systems.

Mobile devices have hard constraints. Physical resources like storage, processing capacity and power supply are critical for these systems. For instance, mobile systems have only a few Gigabytes of storage, so applications must be compiled considering that.

Moreover, a number of applications must run concurrently in an environment that is not optimized for performance, but rather for power consumption. The power consumption must be kept as low as possible to maintain an acceptable battery lifetime. Therefore, mobile systems' developers must think differently from general-purpose developers regarding the optimization of their applications.

To help ensure these requirements are met, the use of profiling and monitoring tools are extremely important. With these tools, it is possible to guarantee that the application will not consume excessive power, will not use more memory than is strictly necessary with useless or not optimized data and will not overuse the device's processing capacity. The proposed tool has the objective to provide valuable information, which today is not available, about Android applications. Hence, these requirements are likely to be achieved and consequently the application is expected to successfully run in a mobile device.

The remaining of this work is organized as follows: Section 2 introduces the Android platform. Section 3 presents related works. Section 4 and 5 presents the project and the implementation methodology of the proposed tool, respectively. Section 6 presents the case study that will be made. Section 7 presents a chronogram regarding the next steps of this research. Finally, Section 8 makes conclusion.

2. Android - Overview

Android is an open-source software stack for mobile devices [2], developed by OHA (Open Handset Alliance) consortium, which is composed of mobile operators, handset manufacturers, semiconductor companies, software companies and commercialization companies, such as: Google, ARM, Intel, Samsung, HTC, Motorola, Qualcomm, NVIDIA and many others [3].

2.1. Market share

Figure 1 shows the growing popularity of Android in the market. This growth in mobile platforms is impressive: in these days, it holds 70% of the market share, followed by iOS, with about 20%.

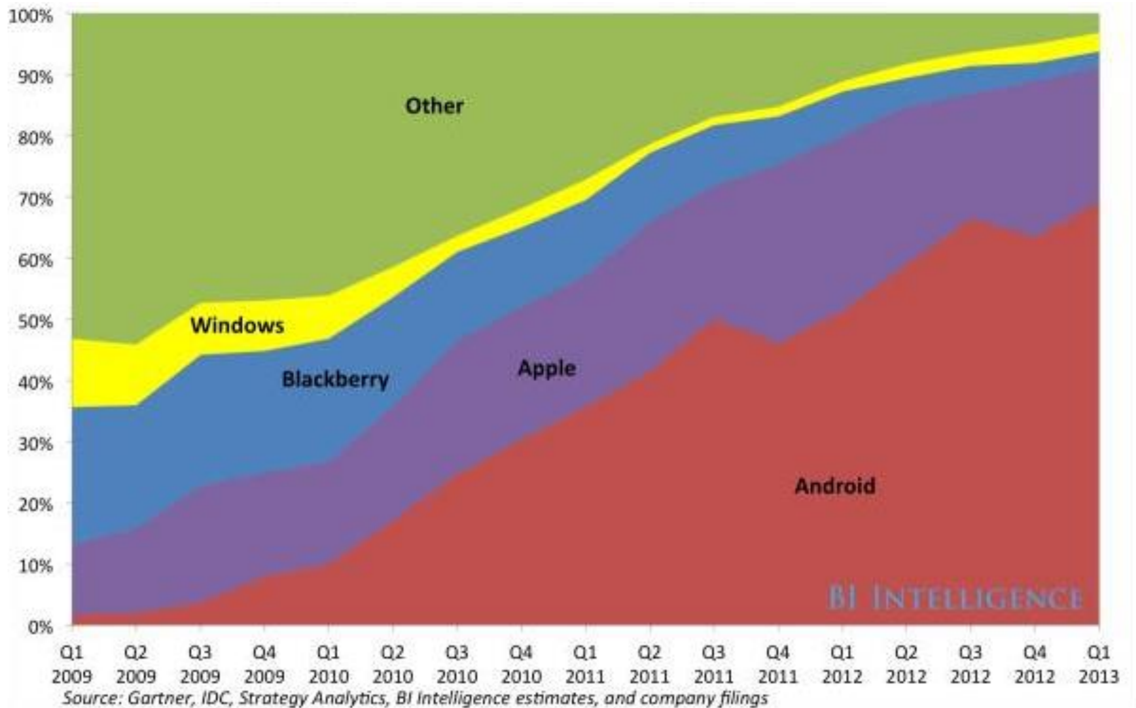

Figure 16. Global Mobile Platform Market Share

Android is also taking place in the global computing platform market, composed of computers, tablets, smartphones, and other devices. Latest researches by BI Intelligence [4] show that Android holds 53% of the global computing platform market share, taking Windows place, which today holds 24% of the market share, as shown in Figure 2.

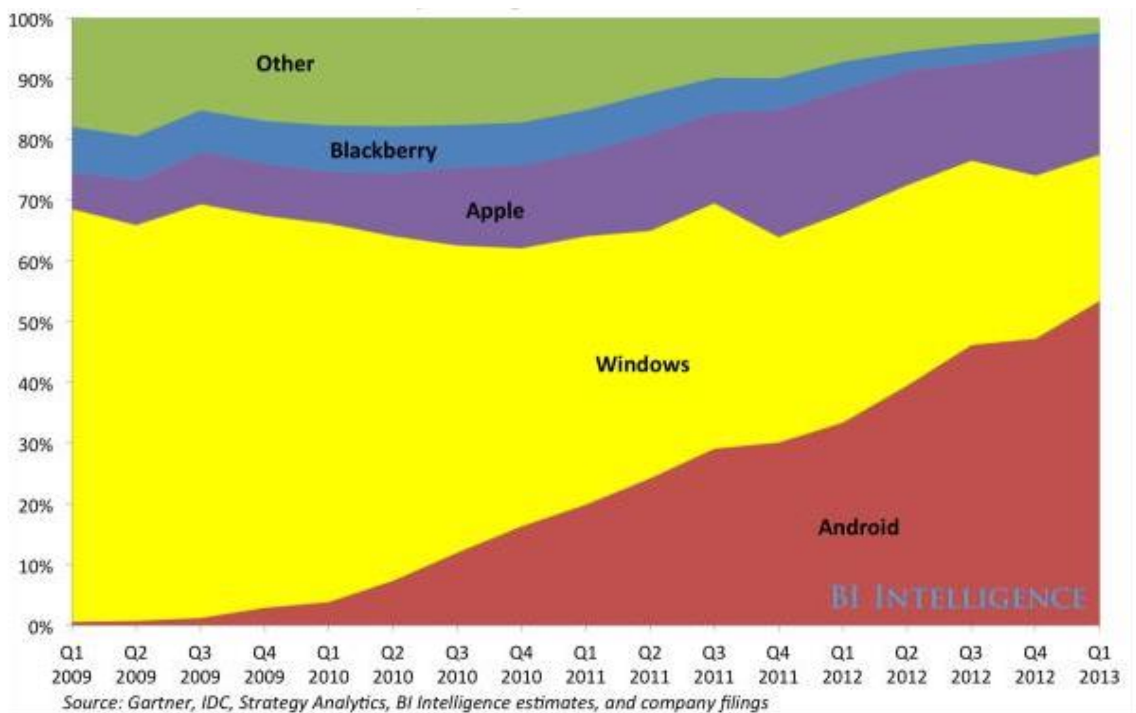

Figure 17. Global Computing Platform Market Share

2.2. Technical details

Figure 3 depicts the Android stack. The Linux Kernel ① is a layer that interacts with the hardware; it also contains all essential hardware drivers. Android is based on Linux 2.6 Kernel with some architectural modifications made by Google. The libraries layer ② contains native libraries written in C or C++. These libraries are optimized mainly for CPU and GPU intensive tasks.

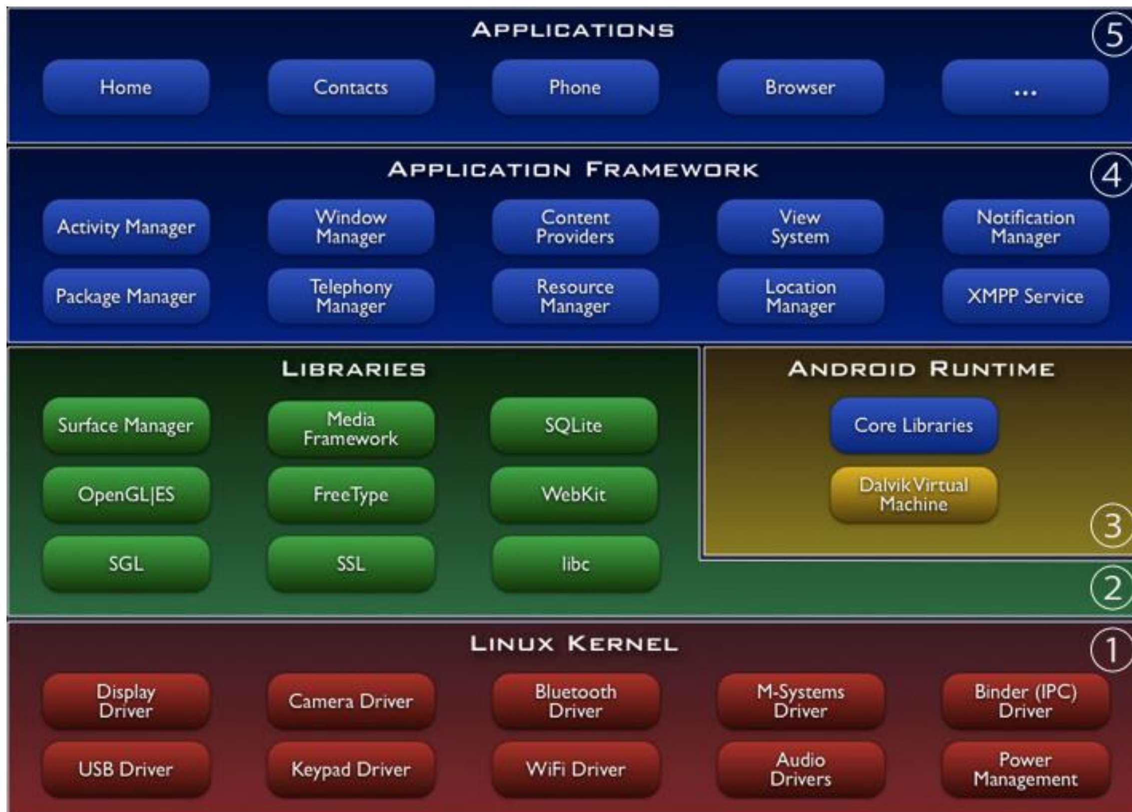

Figure 18. Android Stack. Source: Google Developers

Google has chosen to deploy an alternative virtual machine on Android, called Dalvik Virtual Machine (DVM), instead of keeping Oracle's Java Virtual Machine (JVM). A virtual machine runs an application inside an operating system. Its purpose is to provide a platform-independent programming environment that abstracts details of the underlying hardware or operating system [5]. Therefore, application's portability is increased. For instance, if one considers that a virtual machine supports ARM and Intel x86 processors, once compiled, a given application could be executed on these processors without the need to recompile.

Dalvik VM, defined in the Android Runtime ③, is a register-based architecture [6], opposed to JVM that is a stack-based architecture. Dalvik was designed to run on low memory environments and to allow multiple instances of the VM, so every application runs on its own instance of the VM. Therefore, it provides security, isolation and effective memory management.

An advantage that a register-based architecture has over a stack-based architecture is that it is likely to have better performance. A register-based architecture needs less VM instructions to implement a high level code, even though this comes at a cost of increased instruction size. With larger instructions, register-based architectures take more time to execute each instruction, compared to stack-based architectures. However, the product

between the time per instruction and the number of executed instructions is smaller in register-based architectures than in stack-based ones, which means that a register-based architecture will take less time to execute an application [7].

Android applications are written in Java, afterwards compiled to Oracle's Java Standard Edition bytecodes, then these bytecodes are converted to a single Dalvik Executable (.dex) file by the "dx" tool. These Dalvik Executable bytecodes, rather than Java bytecodes, are executed on Dalvik Virtual Machine.

The Application Framework ④ consists in basic tools for applications' development. Finally, the Applications layer ⑤ is the layer where all developed applications are.

It is possible to develop Android applications purely in Java or parts of it using native code. Some reasons to use native code are the following: the interpretation overhead can be mostly avoided once the code does not execute on the Dalvik VM; and it is possible to reuse existing C or C++ code. Thus, the developer does not have to convert it to Java. Android provides two main ways of using native code in the application: by using the Java Native Interface (JNI); or by using Native Activities. The latter was added to Android in the API level 9 (Android 2.3 Gingerbread) [8].

Moreover, Dalvik has a Just-In-Time compiler that was added to Android 2.2 release and it is still being used in the current version of Android. This mechanism allows the application to run faster as it analyzes the code and caches the most executed parts of the translated code for further reuse, which avoids most of the interpretation overhead. Google's experiments show an improvement of two to five times for CPU-bound code [9].

3. Related Work

Several attempts were made to create tools for tracing and profiling Android applications. Still, to the best of our knowledge, we are far from having a complete pack of tools for profiling these applications. The most relevant tools are described in the next subtopics.

3.1. Traceview

Android Software Development Kit (SDK) provides software tools for debugging, profiling and monitoring. For example, the Dalvik Debug Monitor Server (DDMS) contains Traceview, which is a profiling tool that provides timeline profile panels. The former panel contains the start/stop time of each thread; the latter contains a summary of each method, with the name of the method and children methods, the inclusive and exclusive execution times and the number of calls/recursive calls of the method. The exclusive time is the time that is spent exclusively in the method, therefore, the execution time of any called method will not be considered. The inclusive time is the time spent in the method plus the time spent in any called methods, consisting in the total time that the method took to execute, from the first instruction to the last one [10] .

This tool is very useful to profile an application; however, it is very limited on the amount of data that it can trace: it stores all the traced data in a buffer with a limited size. Therefore, when the trace execution ends, it saves only the amount of trace data that were in the buffer, losing the remaining of it. This tool does not scale with larger applications and it is not possible to run a full execution trace even of small applications. The buffer size depends on the amount of free RAM that is available on the device. However, even with more than 1GB available, most of the tested benchmarks still will make the buffer overflow. Another important limitation is that it can only trace methods executed by

Dalvik Virtual Machine. Therefore, native code (JNI methods or Native Activities) is not considered. Tracing native code is a core feature when it comes to trace applications that have native methods.

3.2. Modification of DDMS

As the DDMS is written in Java and runs as an Eclipse plugin, it is slow by nature. Therefore, the tool proposed by Hyen-Ju Yoon [11] has the goal of speeding up the profiling speed. It is done by decomposing the Traceview into a log data processing layer and a Pretrace program layer that create and analyze the start and end time of methods. However, no quantitative information of this speed up is given at all. Moreover, as already explained, DDMS focuses on Dalvik VM only, so native library and Linux kernel cannot be observed in detail.

3.3. AndroScope

Due to before mentioned Traceview limitations (i.e it traces just methods that are created by the Dalvik VM, and presents poor performance in opening large amounts of data), M. Cho et al. [12], proposed a performance analysis tool for the Android platform. This tool was made to support Java and native applications, Dalvik VM and Android libraries.

A low-level performance analysis through Hardware Performance Counters (HPC) is used to store counts of hardware events such as cache misses, CPU cycles and executed instructions. With this information, it is possible to obtain the IPC, as a performance factor. An extended GCC compiler front-end automatically inserts instrumentation codes to obtain the trace of native libraries and to provide a runtime filtering by class, method name or signature, which allows a selective trace.

M. Cho et al. also developed a graphical user interface, based on Traceview, to display the traced data. They created a new layer to process mass trace logs faster, called "tracebridge" and used Traceview just to display the post-processed data.

This performance analysis through HPCs cannot be done in the Android Emulator because HPCs are not implemented in QEMU, an open source machine emulator and virtualizer that will be the basis for part of the tool we are proposing. Therefore, to use functionalities that HPC provides, we would have to implement it or use alternative ways to get the same results. We will get executed instructions with QEMU's basic block log mechanism and CPU cycles estimation by importing a category characterization file. However, initially our tool will not be able to get cache misses events information because we do not have temporal information of the execution.

3.4. CPU cycle count estimation

A research conducted by Fujitsu Laboratories Ltd. [13], proposed a cycle estimation methodology for an instruction-level CPU emulator. It is divided into a two-phase pipeline scheduling process. First, a static phase is conducted to obtain a rough estimation of the CPU cycle count with the purpose of reducing the instrumentation performance cost. Then, the dynamic phase is responsible for refining the results and guaranteeing more precision. This methodology was implemented by modifying the QEMU source-code with an error in the CPU cycle count of about 10% when comparing to a real CPU. However, as the source code is not available, the possibility of extending it to have more features (e.g.: such as: executed basic block and instruction information, power cost estimation and instruction categorization) was discarded.

All the previous works illustrate the difficulty of tracing and profiling mobile applications. In addition to CPU cycle estimation, this work proposes an approach that will provide information about the executed basic blocks and power consumption.

A comparison between these tools with our tool is presented on Table 1:

Table 10. Tools comparison

Features\Tools	Our tool	Trace view	Mod. of DDMS	Andro Scope	Cycle estim.
Trace native code	X			X	?
Trace basic block information: BB instructions, #calls by each process	X				
Process instruction information: #calls, categories	X				
Process a large amount of data	X		?	X	?
IPC estimation				X	X
Power cost information	X				
Cycle cost information	X			X	X
Faster data visualization than Traceview	-	-	X	X	-
Instruction categorization	X				
Identify and separate the data of different applications in the trace	X				
Trace method information: name, exec. time, #calls, calls hierarchy		X	X	X	
Support different architectures	X	X	X	?	?
No need to import the data every time the program is executed: usage of a database to store data	X				

Legend:

X	Tool has this feature
?	No information is given about this feature
-	Feature does not apply
Blank cells	Not Available

4. Project

To accomplish this, two main pillars will be the basis of the tool:

- A QEMU modification to generate trace information about the executed basic blocks;
- A graphical user interface to import and analyze the collected data and a second graphical user interface that will allow the user to define different Instruction Set Architectures (ISAs) and set categories' cycle and power costs;

We are going to use the Android Emulator, available in the Android SDK, to execute the applications. The Emulator is based on QEMU, also included in Android SDK. Our tool will not require a specific Android Virtual Device (AVD). AVD is an emulator configuration that defines software and hardware configuration, so an actual device can be modeled/emulated [14]. For instance, it is possible to emulate a device

running Android 4.2.2, on an ARM processor, with 512MB RAM, 1GB internal storage and 2GB SD card; or an Android 2.3 running on an Intel Atom, with 256MB RAM, 512MB internal storage and 4GB SD card. In our tool, the user will be free to choose the configuration that is considered the most appropriate.

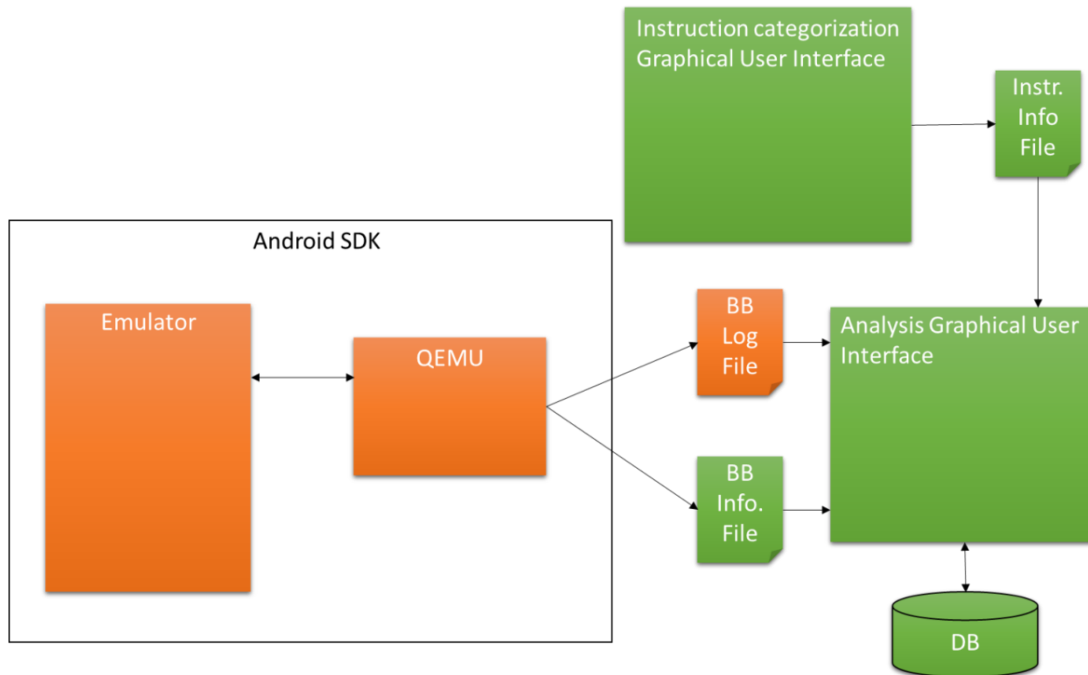

Figure 19. Tool overview

Figure 4 shows an overview of the tool flow; components that will be added are represented in green and the ones that will be modified, in orange. QEMU will be modified to generate more information about the basic blocks, which will be saved in the BB info file, and to include an ID for each basic block in the BB log. The Emulator will be modified to trigger the BB data collected from QEMU.

A Graphical User Interface for instruction categorization will be made to assign categories for instructions; these categories will have both cycle and power costs. Finally, an analysis tool with a Graphical User Interface will import both created files from QEMU and the instruction categorization file and, after processing the data, it will present the analyzed data to the user for an easier understanding of what was executed, saving all necessary data into a database. The database is necessary because of memory limitations, besides the obvious advantage of providing a way of loading previous saved configurations.

4.1. QEMU modification

QEMU [15] emulates the hardware so it is possible to run programs on a virtual hardware platform using different ISAs, such as ARM, MIPS, x86, PowerPC, SPARC and others. This emulation is possible due to a dynamic translation mechanism. As expected, it is slower than execution on real hardware. By using dynamic binary translation, it is possible to translate, typically one basic block at a time, instructions from one ISA to another.

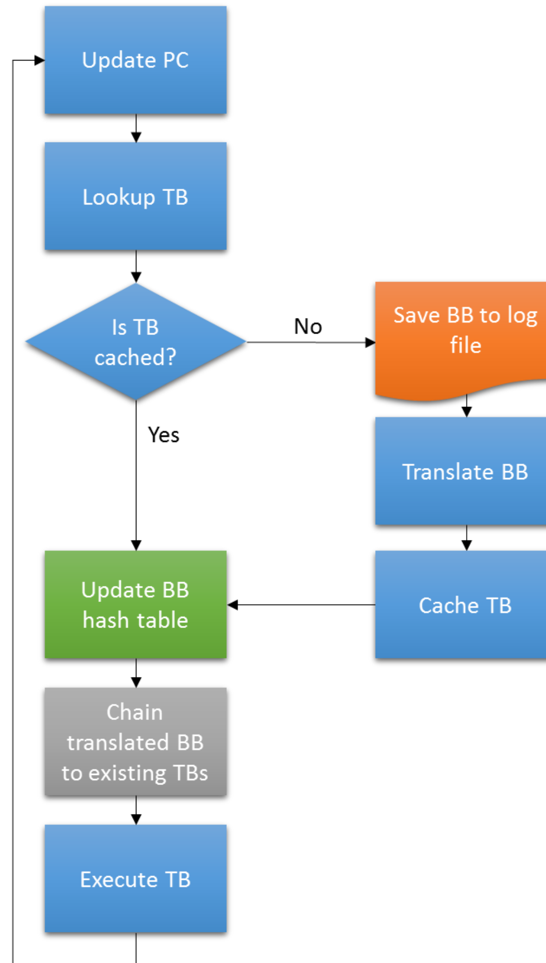

Figure 20. Modified QEMU emulation flow

Considering the components that comprise QEMU, shown in Figure 4, its emulation flow is demonstrated in Figure 5. In the same way as before, green components represent the components that will be added; orange, the modified; gray, the removed; and blue, the ones which will not be modified.

Every time there is a new Basic Block to be executed, QEMU translates it into a Translated Block (TB), a Basic Block composed of instructions implemented with the ISA of the host machine. Whenever the PC is updated, the respective TB is searched. The TB can or cannot be cached.

Whether the Translated Block (TB) is not cached, the original basic block is saved into a log file (enabled by the option –d in_asm on QEMU command line interface) and the TB is cached so it is not necessary to translate it again whenever this same BB is found. A modification in the log saving process will include a basic block identifier, so each basic block will be unique. This identifier will be used, in addition to the Process ID (PID), to keep track of how many times each process executed this basic block. Therefore, we do not need to store the whole basic block in the structure responsible for counting the executed basic blocks, only its identifier. In addition, an entry in a hash table will be created using the PID and the Basic Block ID as the key. This entry will store how many times the BB was called, in addition to the PC address from the first instruction of each BB.

On the other hand, if the TB is cached, the entry in the hash table will be updated if the process already executed the given TB, or a new entry will be created if it is the first time the BB is executed by the given process. After finding or creating the TB and updating the hash table, QEMU executes it.

The process of reading a TB from cache is slow. Therefore, QEMU implements a TB chaining mechanism. Every time a TB returns, QEMU tries to chain this TB to the next TB that will be executed so it does not need to search the TBs one by one. However, we will remove this mechanism because our hash table is updated after each TB is searched or created. Therefore, we can have the original executed translated blocks and have a correct counting for each basic block and process.

Nowadays, some smartphones and tablets are coming with other architectures besides ARM; for instance: Intel x86. This percentage of architectures other than ARM tends to increase once more processors' manufacturers start to reveal interest in this area. Hence, the goal is also to modify QEMU to support different architectures and provide this information not only for ARM processors.

4.2. Graphical user Interfaces

We propose two GUIs (Graphical user Interfaces) that will help in the data analysis process: an instruction categorization GUI and an analysis GUI.

The former GUI has the goal to create an instruction information file ("Instr. Info File" from Figure 4) with categorized instructions with data on cycle and the power costs for each category. Some features of this GUI will be:

- Create instruction information files for different processors' architectures and organizations;
- Import either instructions from a BB log file or instructions, with their categories and costs, from a previously created file of this GUI. It also will be possible to add instructions manually;
- Create categories for the instructions, specifying the average cycle cost and power consumption of the instructions on this new category;
- Each instruction will have a type. For instance, in an ARM architecture it will be possible to define either Thumb or ARM instructions. Thumb instructions are a subset of ARM instructions with reduced bit encoding size. Therefore, Thumb instructions need less memory than ARM instructions: they are 16 bits long, while ARM instructions are 32 bits long. However, not every ARM instruction has an equivalent Thumb instruction;
- Export these categorized instructions with their respective costs;

An example of this structure is given as follows: for an ARMv7 architecture, we can create a conditional branch category with cycle cost of 1, power consumption cost of 2 watts, which comprise the following instructions: BEQ (ARM), BEQ (Thumb), BNE (ARM), BLT (Thumb), BLE (ARM), BGT (Thumb), BGE (Thumb) and other conditional branch instructions; and other categories.

We also propose a second GUI to process the data generated by the modified QEMU and organize it in a more legible way. Some features that will be available are:

- Information about the basic blocks: total cycle cost per basic block. This allows the user to know the most costly basic blocks in terms of cycles and compare the behavior of the basic blocks of a specific process;

- PID chart based on the total cycle cost of each PID. This feature will allow seeing which the most costly processes that had executed are;
- Performance estimation based on a given operation frequency;
- Instructions and basic blocks histogram, which will allow seeing which were the most executed instructions and basic blocks, respectively;
- A chart that shows the total cycle cost by instruction;
- Instruction categories histogram and total cycle cost chart;
- Import profiles (instruction characterization) for different QEMU supported instruction set architectures;
- Power consumption estimation, based on the information provided by the user when configuring the instruction categories. This is a useful feature, once the mobile devices have limitations in relation to battery capacity, so the development of an application that consumes the lowest power as possible is extremely important;

5. Implementation methodology

The tracing tool is proposed to run on Ubuntu operating system; however, nothing prevents it from running on another OS, as long as the Android source can be checked out from the Android's Git repository, an open-source version-control system designed to handle large projects distributed over multiple repositories. Also, Repo, a repository management tool complementary to Git [16] needs to be supported and the SDK needs to be compiled.

The GUI runs on any operating system that supports Java Runtime Environment (JRE), necessary to run Java applications.

The following programming languages and libraries will be used:

- For the execution of the emulator, bash scripts to make the emulation process easier will be provided;
- QEMU modification: C language, since the QEMU code was written in C, the modifications and the new modules that will be added will be written in C as well;
- GUI: Java language, because of its portability. Any operating system that supports Java Virtual Machine (JVM) will be able to run the tools' GUIs;
- BD: SQLite3, a library that implements a serverless, transactional SQL database engine [17]. Useful because the user does not have to get a specific Database Management System (DBMS), like Oracle or MySQL, in the host computer.

In relation to the development environment:

- QEMU modification: gedit (standard Ubuntu's text editor) and GCC, to maintain the SDK compilation, which compiles both QEMU and Android Emulator;
- GUI: Eclipse IDE with WindowBuilder plugin, a plugin that helps on the creation of graphical windows.

6. Case Study

Considering the overhead caused by the use of a virtual machine, we propose to convert some SPEC benchmarks to both Java Native Interface (JNI) and Native Activities. This conversion will take the methods that are executed most of the time and convert it to a native C method that will be called by the Java side. Therefore, the execution of the method will not be made by Dalvik VM. By doing this, a low virtual machine overhead is expected in the execution of these native methods.

Dalvik JIT mechanism was implemented to speed up the execution of applications once the cached code will not have to be interpreted again on its execution. Considering this, we propose six simulations for each benchmark: Java applications with and without JIT, JNI applications with and without JIT and Native Activities applications with and without JIT.

With the developed tool, it will be possible to have more information of how the Dalvik VM and the JIT mechanism affects the execution, in addition to the time measures to compare performance.

7. Chronogram

A chronogram of the activities that will be developed in this work is organized in Table 2.

Table 11. Activities chronogram

	Jul.	Ago.	Sep.	Oct.	Nov.	Dec.
QEMU modification	X	X	X			
Instruction categorization GUI		X				
Processing GUI		X	X	X		
Case study			X	X	X	X

8. Conclusion

Regarding the available tracing and profiling tools that are available to Android developers, there are only a few options, and all of these options have limitations. Due to the difficulty of getting useful data of Android applications' execution, we propose a tool to generate more information and to process this information so it will be easier for the user to analyze the data. In addition, the tool has the goal to be able to process a larger amount of data than some other profiling tools, like Traceview.

In relation to the case study, it will be possible to validate the tool as well as make a study with a series of comparisons between different kinds of applications, with different implementations and running with different optimization mechanisms.

References

[1] IDC (2013) "Android and iOS Combine for 91.1% of the Worldwide Smartphone OS Market in 4Q12 and 87.6% for the Year", http://www.idc.com/getdoc.jsp?containerId=prUS23946013, May 2013.

[2] Android "Android Open Source Project", http://source.android.com/, May 2013.

[3] OHA "Open Handset Alliance Members", http://www.openhandsetalliance.com/oha_members.html, May 2013.

[4] BI Intelligence, https://intelligence.businessinsider.com/welcome, May 2013.

[5] "Virtual machine", http://en.wikipedia.org/wiki/Virtual_machine, June 2013.

[6] Android "Bytecode for the Dalvik VM", http://source.android.com/tech/dalvik/dalvik-bytecode.html, May 2013.

[7] Ehringer, D. "The Dalvik Virtual Machine Architecture" March 2012.

[8] Android Developers "NativeActivity", http://developer.android.com/reference/android/app/NativeActivity.html, May 2013.

[9] Bornstein, D. "Dalvik JIT", http://android-developers.blogspot.com.br/2010/05/dalvik-jit.html, May 2013.

[10] Android "Profiling with Traceview and dmtracedump", http://developer.android.com/tools/debugging/debugging-tracing.html, May 2013.

[11] Yoon, H.-J. "A Study on the Performance of Android Platform" Internation Journal on Computer Science and Engineering (IJCSE), vol. 4, no. 4, pp. 532-537, April 2012.

[12] Cho, M., Hwang, S. J., Lee, H. J., Kim, M. and Kim, S. W. "AndroScope for Detailed Performance Study of the Android" IEEE International Conference on Consumer Electronics (ICCE), pp. 408-409, 2012.

[13] Thach, D., Tamiya, Y., Kuwamura, S. and Ike A. "Fast Cycle Estimation Methodology for Instruction-Level Emulator" 2012.

[14] Android "Managing Virtual Devices", http://developer.android.com/tools/devices/index.html, May 2013.

[15] QEMU "QEMU - open source procesor emulator", http://wiki.qemu.org/Main_Page, May 2013.

[16] Android "Android Developers", http://source.android.com/source/developing.html, June 2013.

[17] SQLite, http://www.sqlite.org/, May 2013.