AndroProf: A Profiling Tool for the Android Platform

Anderson L. Sartor¹, Ulisses B. Corrêa¹,² and Antônio C. S. Beck¹

Instituto de Informática
¹Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, Brazil
²Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense
Charqueadas, Brazil
{alsartor, ubcorrea, caco}@inf.ufrgs.br

Abstract—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, with its hard constraints, such as limited performance and power budget, a hard task to be accomplished. Therefore, a tool that provides information such as energy consumption, execution time and other statistics is mandatory when it comes to develop embedded applications. This paper presents a tool that provides the aforementioned information per application and that is able to trace both Dalvik Virtual Machine and native code. To accomplish this, we extended Android SDK’s QEMU, and we developed graphical user interfaces to process the traced data.

Keywords—Android applications; Android Emulator; QEMU; profiling tool;

I. INTRODUCTION

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 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 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 ones regarding the optimization of their applications.

To help ensure these requirements are met, the use of profiling and monitoring tools is 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 processor.

Therefore, in this work we propose a platform independent tool with the objective of providing valuable information, which today is not available, about Android applications, such as: power consumption estimation, statistics about basic blocks and instructions, and CPU cycle estimation. We have focused on Android, since it is the world’s most popular mobile platform. Android is a Linux-based mobile software platform that is mainly used in smartphones and tablets. The cut Android had in the smartphone market in 2011 was about 50% and, just one year later, that cut reached almost 70% [1].

Hence, the proposed tool can pave the way for designers to meet the aforementioned requirements (i.e. power consumption, processing capacity, and storage), so most likely the developed application will efficiently run in a mobile device.

The remaining of this work is organized as follows: Section II presents related works. Section III and IV presents the implementation and the environment setup of the proposed tool, respectively; and the related experimental results are given in Section V. Section VI, makes conclusion and we describe our future work.

II. 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.

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; the latter 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 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 [2]. The exclusive time is the time that is spent exclusively in the method, therefore, the execution time of any called method is not considered.

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 device. Therefore, if the buffer overflows, all the data profiled after the overflow will be lost. This tool does not scale with larger applications and it is not possible to fully execute even small applications. Even with more than 1GB of memory available, most of the tested benchmarks still will make the buffer 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 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 provides energy consumption information.

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 [3] 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. However, no quantitative information of this speed up is discussed in their work.

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. [4] proposed another performance analysis 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), 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.

A research conducted by Fujitsu Laboratories Ltd. [5], 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, 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. However, as the source code is not available, the possibility of extending it to have more features (e.g.: executed basic block and instruction information, power cost estimation and instruction categorization) was discarded.

Patel and other researchers proposed another QEMU modification to perform full system simulation for multicore x86 CPUs, called MARSSx86 [6]. Also, PTLsim [7], a cycle accurate x86 microprocessor simulator, was improved and ported to have integration with QEMU. 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, which limits its field of application, if one considers that ARM is the main architecture used in mobile devices (90% of the smartphones market share [8]).

All the previous works illustrate the difficulty of tracing and profiling mobile applications. In addition, a comparison among these tools and our tool is presented in TABLE I. Although we have some features in common with these previous works, it was not possible to extend any one of them to have additional features due to already mentioned reasons (i.e. source code not available or platform-specific tool). Thus, the proposed a tool, developed from scratch, is able to profile a larger range of information: executed basic blocks and instructions, power consumption estimation, etc. The tool was built on top of the QEMU emulator, which allows the development of a platform independent profiler, besides the fact that it is largely used to develop Android applications.

III. IMPLEMENTATION

Two main pillars are the basis of our 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 to create categories, with their cycle and power costs, for different Instruction Set Architectures (ISAs).

Fig. 1 shows an overview of the tool flow; components that were added are represented in green, the ones that were modified, in orange and the ones which were not modified, in blue. From Android SDK, we modified QEMU and the VNC connection mechanism (explained in details later) to trace more information about the application. Also, 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 posteriorly processed by the GUIs and saved in a database.

A. QEMU modification

We use the Android Emulator, available in the Android SDK, to execute the applications. The Emulator is based on QEMU, also included in Android SDK. 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 [9]. 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.

QEMU was modified to generate more information about the executed basic blocks, which are saved in the Basic Block (BB)
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 [12]. Also, this configuration can be modified priori execution. Our tool has the goal to be platform independent. In the current version, it supports the ARM and MIPS ISAs. We will use the former as case-study for the rest of the paper, since it is the most used in mobile devices nowadays. Therefore, in this example, the AVD is configured to use an ARM processor and other configuration options can also 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 2.3, with 256MB RAM, 512MB internal storage and 4GB SD card.

Considering the components that comprise QEMU, shown in Fig. 1, its emulation flow is demonstrated in Fig. 2. In the same way as before, green components represent the components that were added; orange, the modified; gray, the removed; and blue, the ones which were not 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 program counter (PC) is updated, the respective TB is searched: it can or cannot be already cached (i.e.: the Basic Block may be already passed through the translation process and saved for future reuse).

Whether the Translated Block (TB) is not cached, the original basic block is saved into a log file (enabled by the option \texttt{-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 included a basic block identifier, so each basic block is unique. This identifier is used, in addition to the Process ID (PID), to keep track of how many times each process executed each 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 our hash table is created using the PID and the Basic Block ID as key. This entry stores 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 is updated if the process already executed the given TB, or a new entry is 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.

| Features\Tools | Our tool | Traceview \[2\] | Mod. of DDMS \[3\] | AndroScope \[4\] | Cycle estim. \[5\] | MARSS \[6\] |
|----------------|----------|-----------------|-----------------|----------------|-----------------|----------------|
| Trace native code | X | X | ? | X | X |
| Trace basic block information: BB instructions, #calls by each process | X | X | ? | X | X | X |
| Process instruction information: #calls, categories | X | X | X | X | X | X |
| Process a large amount of data | X | ? | X | ? | X | X |
| IPC estimation | X | X | X | X | X | X |
| Power cost information | X | X | X | X | X | X |
| Cycle cost information | X | X | X | X | X | X |
| Faster data visualization than Trace view | X | X | X | X | X | X |
| Instruction categorization | X | X | X | X | X | X |
| Identify and separate the data of different applications in the trace | X | X | X | X | X | X |
| Trace method information: name, exec. time, #calls, calls hierarchy | X | X | X | X | X | X |
| Support different architectures | X | X | X | X | X | X |
| No need to import the data every time the program is executed: usage of a database to store data | X | X | X | X | X | X |

Legend:
- Tool has this feature
- No information is given about this feature
- Feature does not apply
- Not Available
However, we removed 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.

B. Graphical User Interfaces

A Graphical User Interface for instruction categorization (“Instruction categorization GUI” from Fig. 1) has the goal to create an instruction information file (“Instr. Info File” from Fig. 1) with categorized instructions in addition to cycle and the power costs for each category. This GUI is presented in Fig. 3, the left table contains the instructions that does not have any category, this 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 “Selected category”.

Some features of this GUI are: create instruction information files for different ARM organizations, for instance: ARMv6 or ARMv7; add or remove instructions manually; create categories for the instructions, specifying the average cycle cost and power consumption 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 cycle cost of 1, power consumption cost of 2 miliWatts. 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.

Finally, an analysis tool (“Analysis GUI” from Fig. 1) with a GUI, presented in Fig. 4, 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

Fig. 1. Tool overview

Fig. 2. Modified QEMU Emulation Flow

Fig. 3. AndroProf instruction categorization GUI.
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 estimation based on a given operation frequency;
- import profiles (instruction characterization) for different ARM instruction set architectures.

IV. 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 Android’s Git repository. Also, Repo, a repository management tool complementary to Git [13], needs to be supported and the SDK needs to be compiled.

QEMU modifications were written in C language, since the QEMU code was also written in C. The GUIs run on any operating system that supports 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 the emulator’s execution. For instance, scripts for SDK compilation and emulator call. Finally, for the database, we used SQLite3, a library that implements a serverless, transactional SQL database engine [14]. It 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 as possible.

V. RESULTS

To evaluate the emulation speed of the proposed tool, we compared the execution times of our implementation with original Android’s QEMU. The latter comparison has the goal to contrast the execution time between an emulated device and a real device. 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 [15].

TABLE II 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 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.
TABLE II. AVERAGE EXECUTION TIME COMPARISON

| Benchmark        | Original QEMU ① | Modified QEMU ② | 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

The average speed down of our QEMU modification, also presented in TABLE II, 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.

VI. CONCLUSIONS AND FUTURE WORK

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 proposed a tool to generate more information and to process this information (e.g. power consumption estimation and information about the basic blocks) for an easier data analysis.

As future work, our goal is to modify QEMU to also support different architectures. 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.

Our tool presents a large speed down, depending on the application’s characteristics, also because the chaining mechanism was disabled. Therefore, we will also study means to trace the applications with chaining enabled. In addition, the tool will provide trace method information and cache events simulation.

REFERENCES

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

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

[3] H.-I. Yoon, "A Study on the Performance of Android Platform," International Journal on Computer Science and Engineering (IJKSE), vol. 4, no. 4. pp. 532-537, April 2012.

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

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

[6] "MARSx86 - Micro-ARChitectural and System Simulator for x86-based Systems," [Online]. Available: http://mass86.org/~mass86/index.php/Home. [Accessed June 2013].

[7] "PTLsim x86-64 Cycle Accurate Processor Simulation Design Infrastructure," [Online]. Available: http://www.ptlsim.org. [Accessed June 2013].

[8] Trefis team, “Can Intel Challenge ARM's Mobile Dominance?,” [Online]. Available: http://www.forbes.com/sites/greatspeculations/2012/11/09/can-intel-challenge-arms-mobile-dominance/2/. [Accesso em July 2013].

[9] QEMU, "QEMU - open source processor emulator," [Online]. Available: http://wiki.qemu.org/Main_Page. [Accessed May 2013].

[10] Ubuntu, “VNC,” [Online]. Available: https://help.ubuntu.com/community/VNC. [Accessed June 2013].

[11] “Android Debug Bridge,” Developers Android, [Online]. Available: http://developer.android.com/tools/help/adb.html. [Acesso em July 2013].

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

[13] Android, “Android Developers,” Google, [Online]. Available: http://source.android.com/source/developing.html. [Accessed June 2013].

[14] SQLite, [Online]. Available: http://www.sqlite.org/. [Accessed May 2013].

[15] SPEC, “SPEC Benchmark Downloads,” [Online]. Available: http://www.spec.org/download.html. [Accessed June 2013].