IEICE TRANS. INF. & SYST., VOL.E99–D, NO.5 MAY 2016

PAPER

Application Performance Profiling in Android Dalvik Virtual Machines

Hung-Cheng CHANG[†a)], **Kuei-Chung CHANG**[††], ***Nonmembers***, **Ying-Dar LIN**[†], ***Member***, ***and*** **Yuan-Cheng LAI**[†††], ***Nonmember***

SUMMARY Most Android applications are written in JAVA and run on a Dalvik virtual machine. For smartphone vendors and users who wish to know the performance of an application on a particular smartphone but cannot obtain the source code, we propose a new technique, Dalvik Profiler for Applications (DPA), to profile an Android application on a Dalvik virtual machine without the support of source code. Within a Dalvik virtual machine, we determine the entry and exit locations of a method, log its execution time, and analyze the log to determine the performance of the application. Our experimental results show an error ratio of less than 5% from the baseline tool Traceview which instruments source code. The results also show some interesting behaviors of applications and smartphones: the performance of some smartphones with higher hardware specifications is 1.5 times less than the phones with lower specifications. DPA is now publicly available as an open source tool.

key words: *Android, Dalvik virtual machine, profiling*

1. Introduction

Android has been one of the most important operating systems (OSs) developed for handheld devices, and more users have become concerned about the performance of their mobile devices when running an application. Thus, how to determine the performance of Android applications has become one of the major issues in their design. Most Android applications are written in JAVA language and run on a virtual machine (VM), and there are four profiling approaches to assess performance of a Java application in terms of profiling location. The first one is to instrument the application source code to acquire its performance information [1]–[3]. Second, one can embed profiling points in the virtual machine to extract the required information through JAVA bytecode analysis for VM-based applications [4]–[7]. Third, one can extract the native code in the system kernel or in the memory of the physical machine and analyze the collected information to estimate its performance [8]–[10]. Fourth, an emulator can be built to analyze the data collected from Android to profile the system [11].

Manuscript received July 15, 2015.
Manuscript revised November 27, 2015.
Manuscript publicized January 25, 2016.

[†]The authors are with the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan.
[††]The author is with the Department of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan.
[†††]The author is with the Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan.
a) E-mail: changhcs@cs.nctu.edu.tw

DOI: 10.1587/transinf.2015EDP7277

However, some approaches cannot be directly used for Android systems. First, the source code of an application usually cannot be obtained easily, which makes the source code instrumentation not feasible. Second, an Android system runs applications on a Dalvik virtual machine (DVM). A classical Java virtual machine is a stack-based VM, and a Dalvik virtual machine is a register-based VM [12], [13]: different virtual machine architectures have very different bytecode formats and processing flows, so that a VM interception analysis for a classical JAVA VM cannot be applied directly to a Dalvik VM. This means that the virtual machine interception techniques would not work well on Android systems [5]–[8]. Last, because the DVM is viewed as a single process in the kernel and each application is an exact replica of the original process, a native Android application cannot be analyzed in any detail, such as method profiling. On the other hand, although some approaches are specific to Android [4], [11], they still need the help of other tools.

An Android system is composed of several layers, as shown in Fig. 1. The Application Framework Layer provides modularized library-like functions for running applications. All Android applications are executed by a Dalvik virtual machine, and in the Linux Kernel Layer, the whole Dalvik virtual machine is regarded as a single process, so we cannot obtain detailed profiling information of the DVM in this layer. Instrumentation in a Dalvik virtual machine could be a feasible way of profiling the Android application.

In this work, we develop a novel approach to profiling, *Dalvik Profiler for Applications (DPA)*, to instrument a Dalvik virtual machine to collect profiling information at two levels, without application source code. In process-level profiling, we can estimate the performance of the target application; in method-level profiling, we can further estimate the performance distributions of all methods in the target application. Figure 1 shows the layout of *DPA*, which consists of four components in a Dalvik virtual machine and one component in the host. The method start time retriever and method end time retriever collect the entry/exit time, name, and process identity (PID) of a method. The record buffer temporarily holds the data from the retrievers and then passes the data on to the log file in some system storage (e.g. a flash memory) when the buffer is full. After profiling, the analyzer in the host can analyze the data stored and retrieve the profiling results. The remainder of this paper is organized as follows. In Sect. 2, we discuss related work in detail. The design and implementation of the proposed

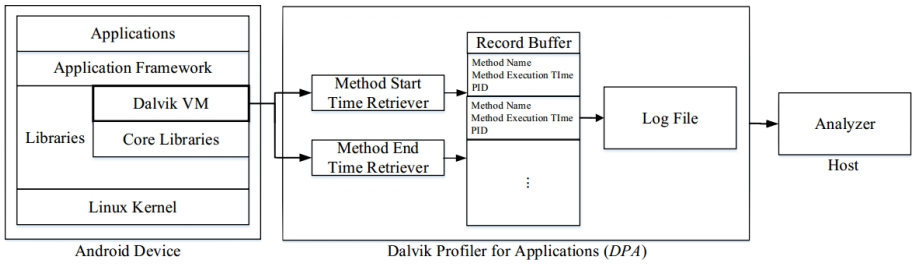


Fig. 1 Android framework with proposed *DPA*.

profiling approach in DVM are described in Sects. 3 and 4, respectively, and we evaluate the profiling system by discussing its precision and granularity in Sect. 5. Finally the conclusion and future work are given in Sect. 6.

2. Background

In terms of profiling location, there are four approaches to performance profiling of an application. The first is to intercept the application source code to acquire the performance information. Heap Profiling [1] is a tool that rewrites the application source code for profiling. Rewriting also allows more garbage collection during application running time in order to save memory space when profiling. Spoon [2] instruments an application during compiling time to profile the performance, without the need for an extra compiler or language. *Traceview* [3] is a debugging tool developed by *Google* which provides precise, detailed, and comprehensive analysis of a running Android application. However, Heap Profiling, Spoon, and *Traceview* need the support of an application source code. For example, *Traceview* needs instructions, *Debug.startMethodTracing()* and *Debug.stopMethodTracing()*, in the source code to activate a method profiling process for an application. This may be impractical for profiling because the source code usually cannot be easily obtained.

Second, one can instrument the virtual machine to obtain the required information for VM-based applications. Method-Level Phase Analysis [5] collects application workloads in JAVA virtual machines and analyzes these workloads offline to observe the method-level behavior of an application. Wave Analysis [6] profiles power consumption of JAVA applications on mobile devices. It plants a monitor across a JAVA virtual machine and the OS, and feeds the monitored results to external equipment. The wave diagram of this equipment then shows the power consumption of each application. Bytecode Instrumentation [7] dynamically instruments additional bytecode to an application in a JAVA virtual machine to obtain performance information. These three methods may not work well on Android because the architecture of a Dalvik virtual machine is essentially different from that of a classical JAVA virtual machine. AndroScope [4] instruments code throughout Android layers to collect data for whole system profiling. For method-level profiling in AndroScope, it collects data from DVM and analyzed the data to get performance information. However, the analysis process needs the aid of Traceview [3].

Third, we can intercept the native code in the kernel or in the memory of the physical machine, and analyze the collected information to estimate performance. Javana [8] and FIT [9] instrument the native code in the physical machine to profile an application. *J [10] records many application runtime events and analyzes these events to profile the application offline. As a result of the offline processing, *J avoids the overhead caused by profiling. These may all be invalid on Android because a Linux Kernel Layer regards the whole DVM as a single process, and each application is a replica of the DVM process. It is difficult to obtain detailed performance information, such as the performance of a method in an application, by means of a traditional profiling tool.

Fourth, we can build an emulator outside the Android and analyze the system. VPA [11] is such an emulator that takes data collected from the Android and some pre-built models, e.g. time model and power model, as input to profile the whole system. It can provide a comprehensive profiling result, but the additional emulator is too complex for average consumer to use.

The advantages and disadvantages of these profiling methods are listed in Table 1. Because of the constraints on profiling without source codes and the incapacity of method-level profiling in kernel, a virtual machine is an ideal location to profile Android applications. But, as noted above, traditional profiling techniques in a JAVA virtual machine may not be feasible for a Dalvik virtual machine. We thus propose *DPA* to profile Android applications in DVM. DPA is a source code which is independent of Dalvik VM specialized profiling tools. DPA obtains the information on processes and methods in a Dalvik VM directly and analyzes the information off-line to avoid profiling overhead. The only constraint is that *DPA* needs the support of the Android system because the modification of Dalvik VM is necessary. However, because Android is an open source system, we can easily implement *DPA* in the Android source code.

3. Architecture Design

Figure 2 illustrates the method interpretation process in Dalvik VM and the *DPA* profiling method. The white blocks represent the original interpretation processes of Dalvik VM, and the gray blocks represent the tasks *DPA* carries out during profiling. The entry and exit processing points of a method are first located, and then the execution time of

Table 1 Comparisons of profiling methods.

Methods	Instrumentation or Analysis Location	Advantages	Disadvantages
Heap Profiling	Source Code	Memory space saving	Source code dependent
Spoon		Compile-time instrumentation	
Traceview		Precision	
Workload Analysis	Virtual Machine	Source code independent	Non-VM Specific
Wave Analysis			Non-VM Specific Equipment requirement
Bytecode Instrumentation			Non-VM Specific
AndroScope		Soutce code independent VM specific	Traceview dependent
Javana	Native Code	Source code independent	Incapacity of method-level profiling
FIT			
*J		Source code independent Low overhead	
VPA	Emulator	VM specific	Need of additional emulator
DPA	Virtual Machine	Source code independent VM Specific Precision Low overhead	Need of system support

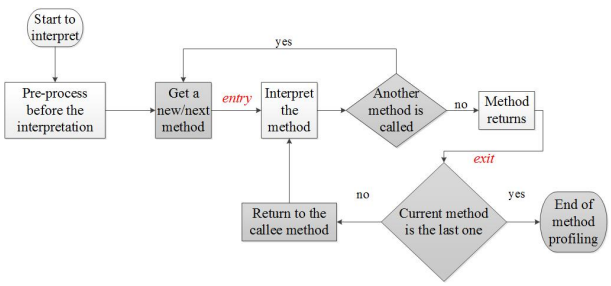


Fig. 2 Entry and exit points of method interpretation.

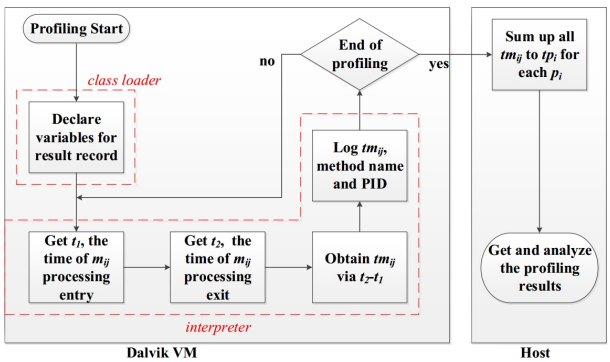


Fig. 3 Profiling design overview.

Table 2 Notations in our work.

Profiling Level	Notation	Description
	D	A device to be profiled
Process Level	P	A set of all running processes on D
	p_i	A process of P, that is, $P = \{ p_i \mid i = 1$ to n, n is the process number on $D \}$
	tp_i	The execution time of p_i
Method Level	M_i	A set of all running methods in p_i
	m_{ij}	A method of M_i, that is, $M_i = \{ m_{ij} \mid j = 1$ to k, k is the method number of $p_i \}$
	tm_{ij}	The execution time of m_{ij}

the method is obtained by subtracting the entry time from the exit time. After obtaining the execution time of each method, the execution time of the profiling process can be estimated by the total time of all methods in the process.

Table 2 defines the notations used in this work. D denotes the device to be profiled. It could be any Android device such as a smart phone or a tablet. For process-level profiling, P denotes all running processes on D, and p_i and tp_i are used to denote the individual process i in P and the execution time of process p_i, respectively. For method-level profiling, M_i denotes a set of all running methods in p_i, m_{ij} denotes method j of M_i, and tm_{ij} denotes the execution time of m_{ij}.

The objective of this work was to design a profiling approach which could measure the execution time tm_{ij} of each m_{ij}, and the execution time tp_i of each p_i for a given working device D with a set of n running processes, P.

Figure 3 shows the design concept. In *class loader* of DVM processing, many variables or data structures for profiling were declared in class loader, e.g. $t1$ and $t2$. In the *interpreter* state, we obtained the entry time $t1$ and the exit time $t2$ of method m_{ij}, and then obtained tm_{ij} by subtracting $t1$ from $t2$, after which the tm_{ij} was logged in the device's storage. Operations in the *interpreter* state were repeated until the end of profiling. Finally, we summed all the tm_{ij} in the log to get tp_i and analyzed the performance of the application on device D.

Each method was invoked with a *method call* and terminated with a return statement. To estimate the execution time of a method, we used the get-current-time function, *clock_gettime()* defined in *sys/time.h*, to obtain the time before the *method call* and after the *return* statement. We obtained the execution time of a method by subtracting the time of these two points. However, the technique cannot be applied to process-level profiling. As defined by *Google* [12], an application is mapped onto a process when being executed, and a process can also be viewed as a main activity. The *Activity Life Cycle* shows that there is no state defining the exit of a process. A process can only be termi-

nated by a *kill system call* or destroyed by the system. So we summed the time of all methods of a process to obtain the process execution time.

A *clock_gettime()* performs a system call in the OS, which imposes execution overhead. When method calls were nested or recursive, the execution time calculated by this technique consequently incurred significant overhead. In order to mitigate this problem, we recorded the calling times of *clock_gettime()* when profiling, and deducted the overhead offline. On the other hand, writing the record to the storage at each method *return* would also generate overhead, because the storage I/O also performs a system call. Memory space was allocated as a buffer to retain the profiling data temporarily. To reduce the I/O overhead, the profiling data were written to storage only when the buffer was full.

4. Implmentation

The DVM reads a *.dex* file (Dalvik Executable File) as input and maps the whole file to the memory. Then the mapped file is parsed and the data in the file are loaded into a *DexFile* structure. The most important variable of *DexFile* is *pClassDefs*, which is a pointer to indicate the position of the first class in the mapped file.

After file mapping, the DVM will start to load classes. While loading a class, the DVM looks up a hashing table in advance to check if this class exists. If the class exists, the data of the class is loaded into a *ClassObject* structure. The *ClassObject* structure contains a lot of information about the class, including the address of each method. After the preprocessing, the application is sent to the interpreter.

The interpreter translates the bytecode into executable native code. Because the efficiency of translation dominates the performance of the application execution, *Google* rewrites the interpreter in assembly. However, each portion of the assembly code only fits a certain target machine architecture, ARM or x86. In case of an absence of new architecture support or debugging issues, *Google* still retains the original code written in *C*. These two types of code lead to different application execution modes. The mode written in assembly is platform specific for performance requirements, while the mode written in *C* is for cross-platform requirementsor debugging. Because the relative performance of each application is not affected by the execution mode, and the adjustment or configuation is easily carried out in *C*, we implemented our design by modifying the mode written in *C*. In file *Interp.cpp*, the variable *stdInterp* determines what kind of interpreter should be used, and we set *stdInterp* to be *dvmInterpretPortable* to ask the interpreter to run in portable mode.

To implement method profiling, we needed to set some variables in the *Method* structure, which is is declared in file *Object.h*, and it defines variables to record the information of a method, such as the method name, the bytecode instruction count, and the address of the first bytecode instruction. We appended two variables of *timeval* data type (defined in */sys/time.h*), ***t1*** and ***t2***, to the Method structure, which were used to record the entry and exit time of a method during profiling.

The bytecode translation in the interpreter was implemented in the function *dvmInterpretPortable()* which was divided into several regions by brackets. Each region manages a state of method execution. The region *GOTO_TARGET(invoke method)* manages some preprocessing before method execution, such as variables initializing, and we obtained the entry time ***t1*** by the API *clock_gettime()* in this region. The region *GOTO_TARGET(returnFromMethod)* manages the return of a method, and we obtained another time ***t2*** by the API *clock_gettime()* in this region. The subtraction of ***t2*** and ***t1*** gives the method exection time exactly.

After each method was profiled, the records were logged to a file. Because the logging process (I/O operations) also needs system calls that may cause overhead, we attached a buffer in *dvmInterpretPortable()* to enable the records to be written to the file in batch mode to reduce the I/O overhead. Little system memory space was allocated for the buffer. Each entry of the buffer was implemented in a *C* structure with three values: the method name, the execution time of the method, and the process identity (PID) of the method. The method name and execution time can be easily obtained from the *Method* structure. The PID was recorded in the variable *systemTid*, which was declared in *Thread* structure. The current instance of thread can be obtained from the argument self in *dvmInterpretPortable()*. A process name was recorded in */proc/PID/cmdline* when the process starts. This recording work was done in Application Framework layer and the process name was not passed to Dalvik, so */proc/PID/cmdline* had to be accessed for the process name.

The value of method execution time may be inaccurate when method calls are recursive or nested. Figure 4 shows a simplified example demonstrated in *C*, in which method M2 was invoked in method M1. In this technique, we used two *clock_gettime()* APIs to obtain the start and end time of M2; the execution time of M1 we profiled included the time these two *clock_gettime()* calls consumed. If more methods are invoked in M2, the execution time of M1 profiled would be more inaccurate. To solve this problem, we declared a variable, *level*, in *dvmInterpretPortable()* to record the invoking level of the method. The *level* is increased by 1 when a method is invoked, and the *level* is decreased by 1 when a method returns. We can use this variable to count

```
void M0()
{
    clock_gettime(t1);
    M1();
    clock_gettime(t2);
}
```

```
void M1()
{
    clock_gettime(t1);
    M2();
    clock_gettime(t2);
}
```

Fig. 4 The profiling of nested method calls.

the number of *clock_gettime()* calls in a profiled method and subtract the time these *clock_gettime()* calls consumed from the time profiled to get an accurate result. For the execution time of each *clock_gettime()*, we imitated the way *Traceview* operates: an inline function is declared and composed of several *clock_gettime()* function calls, and the execution time of *clock_gettime()* is determined by dividing the inline function execution time by the number of *clock_gettime()* function calls it contains.

As *Google* defined [12], there is no certain state for exit in *Activity Life Cycle*. A process can be viewed as a main activity, so the similar profiling technique we used in method profiling is not suitable for process profiling. Instead, we summed the execution time of methods directly invoked by the main activity. Because we inserted the variable *level* to record the method level, we simply summed up the execution times of methods with *level* 1 to obtain the process execution time.

5. Evaluation

In this section, we verify the accuracy of *DPA* on a development board and use this technique to profile popular applications on smartphones. The verification was confirmed by comparing the results of *DPA* with the results of the baseline tools, *Traceview* and *RMP*. In the end of this section, we analyze the execution properties of some popular applications after *DPA* profiling without any support of application source code to help users to realize applications they may use and choose the suitable ones from them.

Lin et al. have run a variety of methods on both DVM and the Library Layers in Android, and compared their performances [13]. We ran these methods on BeagleBoard [14] and profiled them with *DPA*. As mentioned above, the function *clock_gettime()* causes extra execution time that makes the profiling imprecise. Both *Traceview* and *RMP* can dynamically control the number of *clock_gettime()* in the application source code, so they can precisely profile an application. Thus we took *Traceview* and *RMP* as the baseline tools for *DPA* verification. Let the profiled time by *DPA* be t_{DPA}, the profiled time by *Traceview* be $t_{Tra.}$, and the profiled time of *RMP* be t_{RMP}. We defined the error rate of method-level profiling Err_m and the error rate of process-level profiling Err_p as

$$Err_m = \frac{t_{DPA} - t_{Tra.}}{t_{Tra.}} \quad (1)$$

and

$$Err_p = \frac{t_{DPA} - t_{RMP}}{t_{RMP}} \quad (2)$$

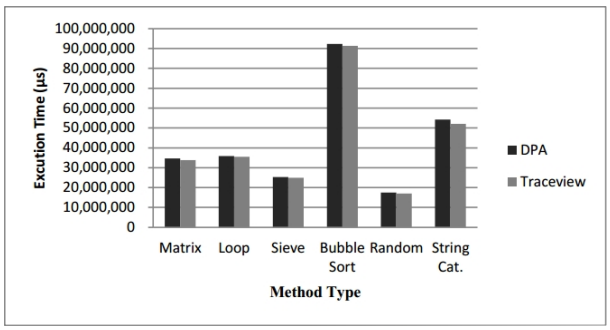
respectively. Table 3 is the specification of BeagleBoard and Fig. 5 (a) shows that the error rate of each test case was $< 5\%$. However, if we do not apply buffer and overhead deducting strategy in *DPA*, the error rates of *Random* and *String Concatenation (String Cat.)* increase to 12.58% and 17.66% respectively, because these two methods are called recursively. On the other hand, profiling without buffer and system call, overhead deduction had nearly no effect on the other four methods. Figure 5 (b) shows the profiling results of the put together methods; the "All" means all six methods tested in method-level profiling. The error rate was $< 5\%$ if the methods inside the process were not nested. Even for processes with nested methods, the error rate was just slightly more than 5%.

However, when the code of an application became complex, *DPA* encountered a fatal miss. The error rate was as large as 30%. It was because *DPA* only profiled the execution time of methods inside the process, ignoring the system operations such as memory allocation. To solve this problem, in the future we will extend *DPA* out of the interpreter in the Dalvik VM and get the *Activity Life Cycle* information to clearly define the start/end point of process-level profiling. Furthermore, with the aid of kernel modification, we can calculate the time caused by system operations to improve the precision of process-level profiling.

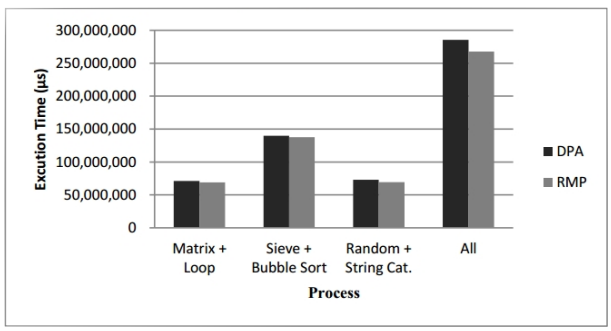
Tables 4 and 5 show the specifications of the two phones and the version of each application respectively. Figure 6 shows three methods which took the most execution

Table 3 The specification of BeagleBoard.

Platform	BeagleBoard-xM
CPU	OMAP3530 720 MHz, ARM Cortex-A8 core
Memory	**256 MB LPDDR RAM**
O.S.	Android 2.3
Storage	1GB SD card



(a) The method-level profiling results of and *Traceview*.



(b) The process-level profiling results of *DPA* and *RMP*.

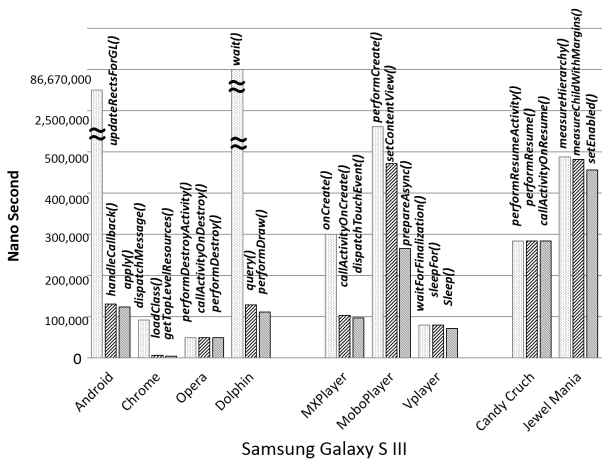
Fig. 5 The verification results.

Table 5 Applications under test.

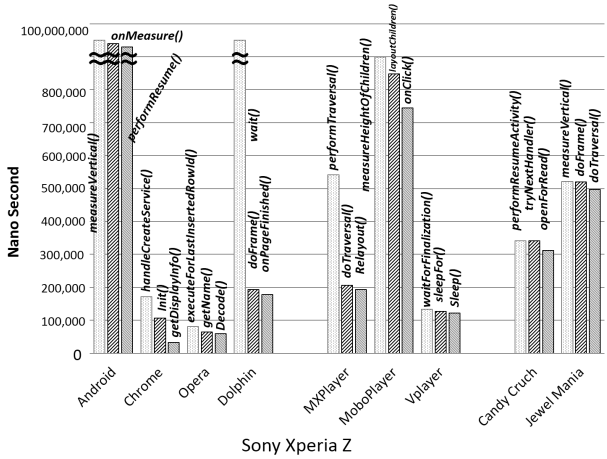
Type	Brower				Video Player			Game	
Applications	Android Built-in	Chrome	Opera	Dolphin	MX Player	Mobo Player	VPlayer	Candy Crush	Jewel Mania
Version	4.3	31.0.165 0.59	16.0.1212. 65583	10.2.2	1.7.26	1.3.27 5	3.2.6	1.30.1	1.1.9.9

Table 4 The specification of tested smart phones.

Phone	Sony Xperia Z	Samsung Galaxy S III
CPU	Qualcomm APQ8064 1.5GHz	Samsung Exynos 4412 1.4GHz
Memory	2GB	1GB
O.S.	Andoid 4.3	Android 4.3
Storage	2GB	2GB
Source Code	AOSP	Cyanogenmod



(a) Method execution on Samsung Galaxy S III.



(b) Method execution time on Sony Xperia Z.

Fig. 6 Execution time of hotspot methods on tested phones.

time in each application on each of the two phones. Although the hotspot methods in the profiled applications were different on Samsung Galaxy S III (Fig. 6 (a)) and Sony Xperia Z (Fig. 6 (b)), the trends of the exectuion time of the two phones were approximately the same except for the Android built-in browser.

In browser profiling, Opera and Chrome perform better

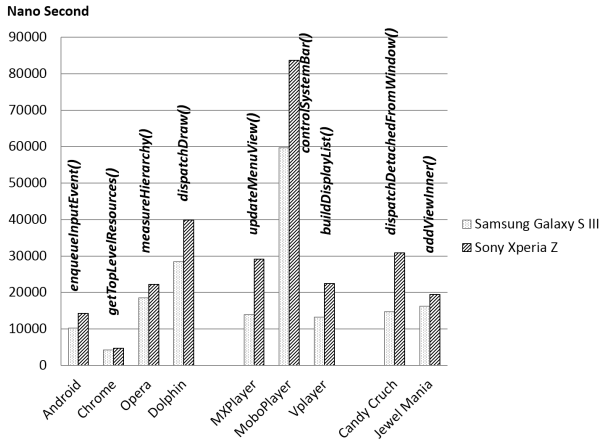


Fig. 7 Method execution time comparison.

because most methods inside them have shorter execution time. Dolphin has a relatively low performance because the method *wait()* has an extremely long execution time. We suspect that Dolphin supplies comprehensive gesture input functions and takes a long time to wait for and react to the gesture input. On the other hand, Android built-in browsers on different phones have different methods of execution behavior. Although both of them are tagged *version 4.3* in the system information, they are distinctly different in size. The browsers might not be the same, and we surmise that the browser in Samsung smartphones might be optimized.

In video player profiling, VPlayer has smoother playing and reacts faster to users. From the profiling results we found that the methods *sleepFor()* and *sleep()* of VPlayer are two of the three hotspot methods, and we suspect that VPlayer offloads the video decoding to hardware so that it can "sleep" and thus generate better performance. The other two players have a lower performance than VPlayer. This might be because this version of the two players does not support hardware decoding.

In game profiling, we profiled Candy Crush and Jewel Mania because they have analogous playing operations. We found that Jewel Mania has better display quality but longer loading time and a less smooth reaction to users. Methods in Jewel Mania need more execution time to attain display quality and load game data.

Finally, we selected one method from these tested applications and compared the execution time on Samsung S III and Sony Xperia Z. Figure 7 shows that the execution time of methods on Sony Xperia Z is about 1.1–1.5 times slower than that of methods on Samsung S III. Versions of Android in Samsung S III and Sony Xperia Z are both 4.3, but the Android in Samsung S III is optimized by

Cyanogenmod [17] and that in Sony Xperia Z is native from AOSP [16]. This gives Samsung S III a better performance even though Sony Xperia Z has better hardware specifications.

6. Conclusions and Future Work

In this work, we designed *DPA* to profile applications in Dalvik virtual machine without the source code of applications. We measured the execution time of methods and processes in an application, and found the following.

1. *DPA* has good profiling accuracy: from the baseline tool *Traceview*, *DPA* had an error rate of 5% on method-level profiling and slightly more than 5% on process-level profiling.
2. The result of *DPA* reflects the execution time of methods inside the application: smoother applications have shorter method execution time; if an application provides greater display quality or fancier functions, performance may decrease and the methods in the application may also take longer to execute.
3. Better hardware specifications may still have lower performance: we also found that even with better hardware specifications and the same Android version, performance may still be low because the Android itself is not optimized. We found that performance disparity was 1.1–1.5 times.

The execution time we measured was the turn around time, not the real execution time of an application. If the system context switches frequently, the result of *DPA* may be inaccurate. The process profiling may also encounter a fatal miss when the code of an application is complex or large, because we only considered the execution time of methods inside the process and ignored the system operations such as memory allocation. On the other hand, *DPA* cannot profile the method implemented in native code, because the execution of native code is not processed in the DVM. It needs another strategy for profile. Future work will address these problems so as to render *DPA* more accurate. *DPA* is now publicly available as an open source tool [18].

References

[1] R. Shaham, E.K. Kolodner, and M. Sagiv, "Heap profiling for space-efficient Java," ACM SIGPLAN Notices, vol.36, no.5, pp.104–113, 2001.
[2] R. Pawlak, "Spoon: Compile-time Annotation Processing for Middleware," IEEE Distributed Systems Online, vol.7, no.11, p.1, Nov. 2006.
[3] *Traceview*, http://developer.android.com/tools/debugging/debugging-tracing.html
[4] M. Cho, H.J. Lee, M. Kim, and S.W. Kim, "AndroScope: An Insightful Performance Analyzer for All Software Layers of the Android-Based Systems," ETRI Journal, vol.35, no.2, pp.259–269, April 2013.
[5] A. Georges, D. Buytaert, L. Eeckhout, and K.D. Bosschere, "Method-Level Phase Behavior in Java Workloads," ACM SIGPLAN Notices, vol.39, no.10, pp.270–287, 2004.
[6] L.-W. Liu, A Method-Level Energy Profiling Tool for Java Applications on Mobile Devices, Master Thesis, National Tsing Hua University, Taiwan, 2010.
[7] M. Dmiuiev, "Selective Profiling of Java Applications Using Dynamic Bytecode Instrumentation," Performance Analysis of Systems and Software, pp.141–150, 2004.
[8] J. Maebe, D. Buytaert, L. Eeckhout, and K.D. Bosschere, "Javana: A System for Building Customized Java Program Analysis Tools," ACM SIGPLAN Notices, vol.41, no.10, pp.153–168, 2006.
[9] B.D. Bus, D. Chanet, B.D. Sutter, L.V. Put, and K.D. Bosschere, "The Design and Implementation of FIT: a Flexible Instrumentation Toolkit," ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp.29–34, 2004.
[10] B. Dufour, L. Hendren, and C. Verbrugge, "*J: A Tool for Dynamic Analysis of Java Programs," Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp.306–307, 2003.
[11] C.-H. Tu, H.-H. Hsu, J.-H. Chen, C.-H. Chen, S.-H. Hung, "Performance and power profiling for emulated android systems," ACM Transactions on Design Automation of Electronic Systems, vol.19, no.2, pp.1–25, March 2014.
[12] Security Engineering Research Group, Analysis of Dalvik virtual machine and class path library, Institute of Management Sciences, Peshawar, Pakistan, 2009.
[13] C.-M. Lin, J.-H. Lin, C.-R. Dow, C.-M. Wen, "Benchmark Dalvik and Native Code for Android System," Innovations in Bio-inspired Computing and Applications, pp.320–323, 2011.
[14] BeagleBoard, http://beagleboard.org/
[15] Y.-D. Lin, K.-C. Chang, Y.-C. Lai, and Y.-S. Lai, "Reconfigurable Multi-Resolution Performance Profiling in Android Applications," IEICE Transactions on Information and Systems, vol.E96-D, no.9, pp.2039–2046, 2013.
[16] Android Open Source Project, (AOSP), https://source.android.com/
[17] Cyanogenmod, http://www.cyanogenmod.org/
[18] Dalvik Profiler for Applications (*DPA*), http://sourceforge.net/projects/dpa/files/latest/download

Hung-Cheng Chang received the M.S. degree in computer science and information engineering from National Taiwan University of Science and Technology in 2010. He is now a Ph.D. student in computer science from National Chiao Tung University. His research interests include the flash memory, virtual machines, and embedded operating systems.

Kuei-Chung Chang received the Ph.D. degree in computer science from National Chung Cheng University in 2008. He is Associate Professor of Department of Information Engineering and Computer Science at Feng Chia University in Taiwan. His research interests include system-on-chip, network-on-chip, embedded system, and multi-core system.

Ying-Dar Lin is a Distinguished Professor of computer science at National Chiao Tung University (NCTU), Taiwan. He received his Ph.D. in computer science from the University of California at Los Angeles (UCLA) in 1993. He was a visiting scholar at Cisco Systems in San Jose, California, during 2007–2008. Since 2002, he has been the founder and director of Network Benchmarking Lab (NBL, www.nbl.org.tw), which reviews network products with real traffic and has been an approved test lab of the Open Networking Foundation (ONF) since July 2014. He also cofounded L7 Networks Inc. in 2002, which was later acquired by D-Link Corp. His research interests include network security and wireless communications. His work on multihop cellular was the first along this line, and has been cited over 650 times and standardized into IEEE 802.11s, IEEE 802.15.5, IEEE 802.16j, and 3GPP LTE-Advanced. He is also an IEEE Fellow (class of 2013), IEEE Distinguished Lecturer (2014–2015), and an ONF Research Associate. He currently serves on the editorial boards of several IEEE journals and magazines. He published a textbook, Computer Networks: An Open Source Approach (www.mhhe.com/lin), with Ren-Hung Hwang and Fred Baker (McGraw-Hill, 2011).

Yuan-Cheng Lai received the Ph.D. degree in computer science from National Chiao Tung University in 1997. He joined the faculty of the Department of Information Management at National Taiwan University of Science and Technology in 2001 and has been a professor since 2008. His research interests include wireless networks, network performance evaluation, network security, and content networking.