Reconfigurable Vertical Profiling Framework for the Android Runtime System

TZU-HSIANG SU, HSIANG-JEN TSAI, KENG-HAO YANG, PO-CHUN CHANG, and TIEN-FU CHEN, National Chiao Tung University
YI-TING ZHAO, National Chung Cheng University

Dalvik virtual machine in the Android system creates a profiling barrier between VM-space applications and Linux user-space libraries. It is difficult for existing profiling tools on the Android system to definitively identify whether a bottleneck occurred in the application level, the Linux user-space level, or the Linux kernel level. Information barriers exist between VM-space applications and Linux native analysis tools due to runtime virtual machines' dynamic memory allocation mechanism. Furthermore, traditional vertical profiling tools targeted for Java virtual machines cannot be simply applied on the Dalvik virtual machine due to its unique design. The proposed the Reconfigurable Vertical Profiling Framework bridges the information gap and streamlines the hardware-software co-design process for the Android runtime system.

Categories and Subject Descriptors: C.4 [**Computer Systems Organizations**]: Performance of Systems—*Measurement techniques*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.4.8 [**Operating Systems**]: Performance—*Measurements, Monitors, Operational analysis*

General Terms: Design, Performance

Additional Key Words and Phrases: Profiling, vertical profiling, virtual machine profiling, nonintrusive profiling, embedded systems

ACM Reference Format:
Tzu-Hsiang Su, Hsiang-Jen Tsai, Keng-Hao Yang, Po-Chun Chang, Tien-Fu Chen, and Yi-Ting Zhao. 2014. Reconfigurable vertical profiling framework for the Android runtime systems. ACM Trans. Embedd. Comput. Syst. 13, 2s, Article 59 (January 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2544375.2544379

1. INTRODUCTION

Limited computational space and power are major design factors for the increasingly common smart mobile embedded devices. In order to maximize performance, optimizing application and system software is often the most effective and economical approach. Since how to optimize the software is not always apparent, profiling is an essential methodology for pinpointing performance bottlenecks without introducing excessive overhead which could skew measured results. In addition to optimizing software, profiling can also be used to determine the hardware requirements when a target system software has been chosen and assist the hardware-software co-design process [Shannon and Chow 2004].

This research was supported in part by the National Science Council Project NSC 100-2220-E-009-036 and also by Information and Communications Research Laboratories (ICL) of the Industrial Technology Research Institute (ITRI).
Corresponding author's (T.-F. Chen) email: tfchen@cs.nctu.edu.tw.

DOI: http://dx.doi.org/10.1145/2544375.2544379

The majority of popular mobile operation systems today are fine tuned and tied to a few selected hardware platforms in order to optimize performance. These types of specific hardware are often locked to prevent users from altering the mobile operation system, fearing it would diminish performance. Android on the other hand, is a fast evolving mobile platform that differentiates itself by delivering frequent version upgrades and supporting a variety of hardware devices. Android achieves architecture-neutrality by running applications on top of the Java inspired Dalvik runtime virtual machine [Google 2011]. These features make optimizing Android hardware and software a more challenging task, as it becomes difficult to determine whether performance bottlenecks occurred at the application level, the user-space libraries, or deeper at the Linux kernel level.

Currently, the need for performance analysis to accelerate the Android hardware-software co-design process has already produced many Java-level trace tools [Paul and Kundu 2010]. Some of these tools approximate the time spent on Linux user-space libraries by adding instrumentations to the beginning and end of each function of a test application [Chang et al. 2010]. Unfortunately, this creates large operational and runtime overheads which introduce inaccuracies to the profiling results.

In order to make up for the lack of Linux user-space and kernel-space information, Android source code provides Linux systems profiling and tracing tools to its developers along with those virtual-machine-level specific tools [Google 2010]. However, these tools are unable to provide application statistics, because runtime-interpreted Dalvik opcode segments for a DVM application are first loaded into its process' heap and then executed as the heap itself. Traditional Linux profiling and tracing tools can only see the heap being executed and can not distinguish which method is currently running.

Furthermore, since Android applications running inside Dalvik virtual machines are forked and controlled by the Zygote parent process, these Linux native performance analysis tools are unable to determine the relationship between Linux libraries and Java applications running inside Dalvik virtual machines. Thus it is difficult to retrieve useful information past the Dalvik virtual machine layer.

This article proposes a Vertical Virtual Address Remapping Integrated (VARI) profiler, a reconfigurable vertical profiling framework for the Android runtime system in which a low-overhead memory map address remapping methodology, linking virtual machine to kernel with a memory map tunnel, is devised to bridge the information barrier and provide a way for performance analysis tools to tie the usage of Linux native libraries with those Android applications that utilized them.

Furthermore, this article aims to lower vertical profiling overhead, thus minimizing probe-effect-related inaccuracies. This framework enables reconfigurable profiling at three levels: method-specific, application-specific, and service-specific profiling, which allows users to only instrument targets of interest. By combining the ability to focus on specific profiling targets, memory map remapping mechanism, and the appropriate instrumentation methodology devised to suit the unique designs of Dalvik virtual machine, this article reduces over 30% of overhead generated by traditional vertical profiling methods. The proposed reconfigurable vertical profiling framework streamlines the difficult task of identifying system bottlenecks and accelerates the Android hardware-software co-design process.

2. RELATED WORK

2.1. Opcode Instrumentation

Native code instrumentation, opcode instrumentation, and binary instrumentation are the three common approaches to altering a virtual machine for the purpose of providing further debugging information to the host analysis tools.

Opcodes are known as bytecode for Java virtual machines, because most JVM opcodes are one byte in length due to the fact that most JVMs are stack machines. Bytecode instrumentation monitors runtime-generated Java bytecodes. As the result of DVM not employing standard Java bytecodes, standard Java debugging or profiling tools do not work on Android, at least not without significant modification [Chang et al. 2010; Nicolaescu and Veidenbaum 2005]. In addition, most bytecode instrumentation methodology examines every bytecode generated at runtime, thus introducing greater overhead. Many opcode instrumentation tools rely on JVMTI.

2.1.1. JVMTI. JVMTI stands for Java Virtual Machine Tool Interface [Oracle 2002]. It was developed by Sun Microsystems, now owned by Oracle. It offers dynamic bytecode instrumentation functionality which allows users to profile for information, add breakpoints, and even make changes to the opcodes of Java methods. These JVMTI functions are made possible by adding events to the code of a method, for example, adding a call to methodEntered() at the beginning of a method. The inserted code is standard bytecodes, which JVM treats as a part of the application [Oracle 2010]. Although these additions do not modify application state or behavior, they introduce additional overhead to the original application. To sum up, JVMTI is a great Java debugging tool but perturbs the original application when utilized as the backend of a profiler.

2.2. Native-Code Instrumentation

Native-code instrumentation entails instrumenting virtual machine's native source code. Usually native-code instrumentation is event based and delivers application information to the performance analysis tool when specific virtual machine events take place [Hauswirth et al. 2004, 2005; Mousa and Krintz 2005; Mousa et al. 2010; Binder et al. 2007]. An example of native-code instrumentation is VIProf [Mousa et al. 2007].

2.2.1. VIProf. VIProf extends Oprofile to enable integrated profiling across the virtual layers of a system. The VIProf architecture modifies Oprofile to re-label samples that belong to the Java heap as a substitute file. The application's method information is collected by instrumenting Jikes RVM's method compile, recompile, and flags GC move method. VIProf is implemented on Jikes, which produces a static image in a Jikes internal format and an associated map. VIProf is able to modify the Oprofile post-processing tool to associate samples with static image produced by Jikes.

Most native-code instrumentation methodologies and Javana mentioned later are implemented on Jikes RVM [Alpern et al. 2005]. Jikes is a Research Virtual Machine, which is a JVM implemented as a Java runtime application. The issue with employing Jikes is that it is designed to be flexible. There is a MAGIC class defined in Jikes' compiler which allows users to implement machine code and interact with system memory [Alpern et al. 1999]. Barriers imposed by regular virtual machines are easier to circumvent on Jikes if functionalities like MAGIC are utilized.

2.3. Binary Instrumentation

Binary instrumentation modifies the interpreter of a virtual machine to monitor for codes of interest. When an interested native instruction occurs, the modified code utilizes special native instructions to deliver stack information relevant to the monitored instruction. These special native instructions can also be dynamically inserted to trigger profiling [Schneider et al. 2007].

2.3.1. Javana. Javana is a binary instrumentation methodology that comes with its own profiling language. It is implemented on Jikes RVM and utilizes DIOTA as its underlying profiler. Javana proposes a dynamic binary instrumentation tool that resides between the virtual machine and host operating system. This dynamic binary

Table I. Comparison of Vertical Profiling Instrumentation Methods

Profiler	Instrument method	JVM platform	Profiler
JVMTI [Chang et al. 2010]	Opcode VM agent inserts additional opcode into the original application methods	Sun/Oracle JVM	Adaptive
VIProf [Mousa et al. 2007]	Native-code VM agent saves method information to disk at JVM events. Oprofile kernel driver replaces heap execution with substitute file. Post-processing modified to read Jikes generated static image	Jikes RVM	Oprofile
Javana [Maebe et al. 2006]	Binary VM agent monitors JVM events and instruments opcode to send information to profiler	Jikes RVM	DIOTA
ProbeNOP [Inoue and Nakatani 2009]	Binary An interrupt handler that issues ProbeNOP to initiate profiling is inserted at the start of every method. ProbeNOP is a special instruction not used by the system	J2SE	Oprofile

instrumentation tool tracks all native instructions executed by the virtual machine to keep track of events, such as thread creation, thread switching, thread termination, class loading, object allocation, object relocation, method compilation, garbage collection, etc. Javana defined a set of aspect oriented instrumentation language to allow programmers to specify the events of interest, such as before or after time qualifier, memory operation to a target. The dynamic binary instrumentation tool loads the user-defined scenario and instruments at runtime. Similar to other binary instrumentation methods, tracking all native instructions in order to identify the location to instrument at runtime introduces considerable overheads [Maebe et al. 2006].

2.3.2. ProbeNOP. Inoue and Nakatani [2009], utilized an unused instruction in the POWER6 architecture as their special instruction to trigger Oprofile. The ProbeNOP instruction employs a POWER6 architecture vector permute instruction unused by JVM to increment hardware performance counters and at the same time deliver context-dependent information to the hardware performance counter interrupt handler. The interrupt handler to trigger ProbeNOP is inserted into every method entry to trigger profiling. For instance, to profile lock activity, ProbeNOP can be added to all lock acquisition operations in JIT-compiled methods and the entry point of the monitor enter helper function. When lock contention takes place, ProbeNOP will provide information on the program locations that causes it by showing a high HPC count for a method's allocated address. Since ProbeNOP does not employ Oprofile, it maintains its own back-trace tree by deriving object creation profiles from the Java object headers store instruction profiles collected by hardware performance counters.

This mechanism creates relatively small overhead when compared to software solutions, such as scanning for memory locations. It also provides context-sensitive vertical profiling.

The drawback of native code instrumentation is that the target program source code needs to be manually instrumented. Additionally, duplication of efforts is required when applying to different architectures. Some architecture might not have an unutilized instruction to act as the special NOP function. These factors make binary instrumentation less ideal for the Android platform.

2.4. Comparison of Vertical Profilers

Table I compares the instrumentation methods of the just works introduced. Each work provides distinctive benefits to profiling integration on JVM. However, they all have

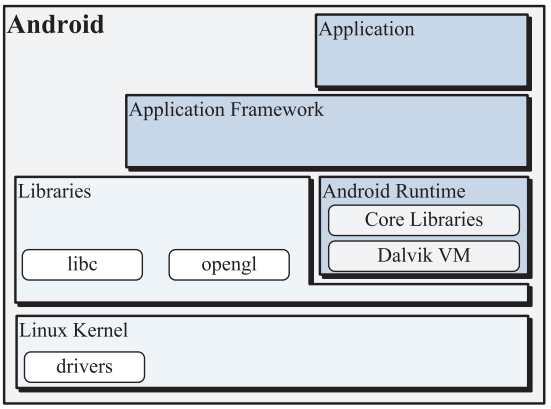


Fig. 1. Android architecture.

design aspects that are incompatible with the unique designs of DVM, such DVM being a register-based machine, rather than stack-based, like most JVM.

3. BACKGROUND AND SYSTEM DESIGN

Android presently runs on several iterations of ARM and x86. These include single-core and multicore architectures with varies memory hierarchies. Given that Android is designed to be architecture-neutral and our goal is to minimize probe effect related inaccuracies, this article adopts a modified event-based native-code instrumentation approach to suit the unique process execution and stack management of Android's Dalvik virtual machine.

3.1. Android Platform

Android platform is built on top of the Linux kernel. In the pursuit of making the system more suitable for mobile devices, many modifications are made to the system, including changes to the basic Linux system libraries, such as libc and ptrace. The basic Android framework is shown in Figure 1. Linux kernel and drivers are the foundation of Android framework and directly interact with the hardware. Android runtime is built as a part of the Linux system libraries which consists of the Dalvik Virtual Machine and other libraries that manages DVM thread creation and interprocess communication [Batyuk et al. 2009; Binder et al. 2006].

Although the Dalvik virtual machine is a modified Java virtual machine, unlike most JVM stack machines, DVM implements a register-based design. A register-based machine uses fewer instructions, but since each instruction is longer, it also has larger code size compared to the stack machines. Aiming to reduce code size, Android redesigned the packaging format to share common code and constants between class files. The resulted dex format typically produces files less than half the size of the same code in the uncompressed JAR format [Khan et al. 2009].

Application framework is a special Dalvik application which contains Android system services, such as telephony manager, activity manager, and windows manager. The rest of the applications interact with the basic systems through this application framework. Android user interfaces, such as the home screen, telephone dialer, and browser, are all Dalvik applications.

3.2. Dalvik Virtual Machine Application Initiation

The proposed vertical profiling framework will modify codes within the Dalvik virtual machine to enable sending application information to our Linux user-space profiler.

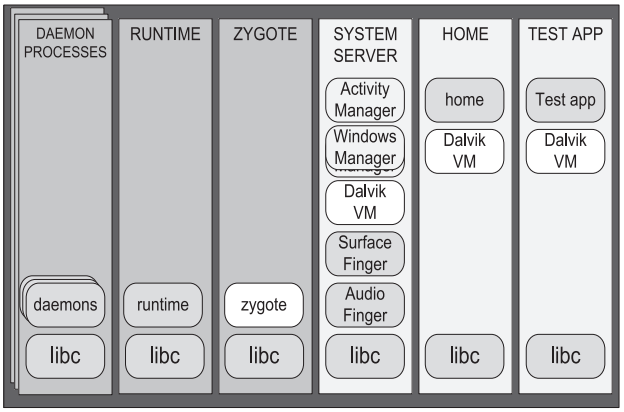


Fig. 2. Android processes [Google 2008].

During the initiation of new Android applications, a new Dalvik virtual machine is created for each, as shown in Figure 2 [Google 2008]. A process called Zygote monitors the application initiation and forks it into a new thread when it is ready. Thus, modifications for the virtual machine event-based instrumentation should focus on Zygote and the Dalvik virtual machine.

Traditional vertical profiling instrumentation methodologies track events that modify stack address of the virtual machine. These important events includes Zygote forking a new thread, Dalvik virtual machine undergoing method creation or garbage collection, new frames being pushed or popped on the thread stack, and other memory related events. A log of these changing address locations helps to correspond application information to Linux profiler samples.

3.3. Oprofile

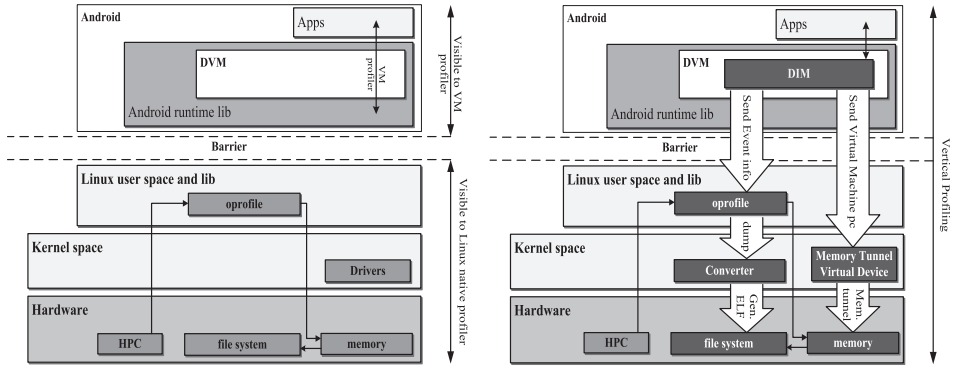
In the attempt to minimize profiling overhead, Oprofile is chosen as our Linux user-space profiler. Oprofile utilizes hardware performance counters of different architectures to sample both user-space and kernel-space usage data [Contreras and Martonosi 2005]. Using Oprofile, developers can avoid dramatically affecting accuracy by adjusting sampling frequency to minimize overhead. At the same time using hardware counters further reduces overhead [Sweeney et al. 2004; Cohen 2004].

The most recent official Oprofile release includes a JIT extension which provides a set of tools, such as opagent and opjitcov to use along with JVMTI on traditional JVMs. However, due to design differences between JVM and DVM, the extension is not functional on Android without our proposed changes to the DVM.

Furthermore, Oprofile source code included with Android is modified to adapt to the differences between Android's Linux kernel and regular Linux systems. As a result, Oprofile provided with the Android source code is older in version and stripped of newer features that are needed for the proposed communication flow. To complete our profiling work flow, the VARI profiler ported the official version of Oprofile to Android and a few other Linux components that are required to run with this version of Oprofile, such as Sed and Awk, to our test system.

3.4. Profiling Gap and System Design

The main goal of this article is to establish a bridge to communicate application information from the Dalvik virtual machine to a Linux user-space profiler without modifying target application source code.



(a) Current profiling process with an information barrier. (b) Bridging the barrier with even-based instrumentaion.

Fig. 3. Profiling flow on Android runtime. (a) Current profiling tools with the barrier. (b) The proposed method employs an event-pipe instrument module which sends method name, stack frame address, and other information to the Linux profiler.

3.4.1. Information Barrier. Figure 3(a) shows the profiling flow of those Android performance analysis tools currently available. Linux-based profiling tools are only able to gather information of the runtime virtual machine itself as an independent process. They are unable to penetrate the virtual machine and retrieve Android application method and stack address information. Therefore an information barrier is formed, where profiling disconnects take place between Android applications, Linux libraries, as well as the kernel events that the applications invoked.

3.4.2. Vertical Profiling. The proposed vertical profiling flow utilizing memory map address remapping and direct memory tunnel is shown in Figure 3(b). Event-based native-code instrumentation is achieved with the addition of the Dalvik instrumentation module, which is responsible for two different sets of communications.

First, a virtual device module is loaded in the kernel space before profiling began. When an application initiates, its DVM process opens a virtual file corresponding to the DVM's PID. During profiling, DIM delivers time-sensitive remapping memory address to kernel space through the proposed direct memory tunnel.

The Dalvik instrumentation module looks up a method's virtual memory address during virtual machine events, such as method invoke, and subsequently forwards the VMA of this method to the kernel-space virtual device file. This is the main implementation of the memory tunnel.

Second, at the same time, DIM collects Android application thread information, such as, UID, PID, TGID, current stack frame, frame pointer, method name, and class name. The modified events then dump this method information in a location where our Linux user-level profiler can retrieve and correlate with sample results, thus bridging the profiling gap.

4. PROPOSED EFFICIENT VERTICAL PROFILER - VARI

4.1. Traditional Method Invoke Instrumentation Methodology

Traditionally, JVM instrumentation for vertical profiling takes place whenever an address has changed. Since typical JVM machines are stack machines, the allocation of a Java method's VMA is not known until the interpreter invokes the method. When the stack frame is full, garbage collection and other memory relocation functions also modify the method's address. Therefore, implementing vertical profiling requires

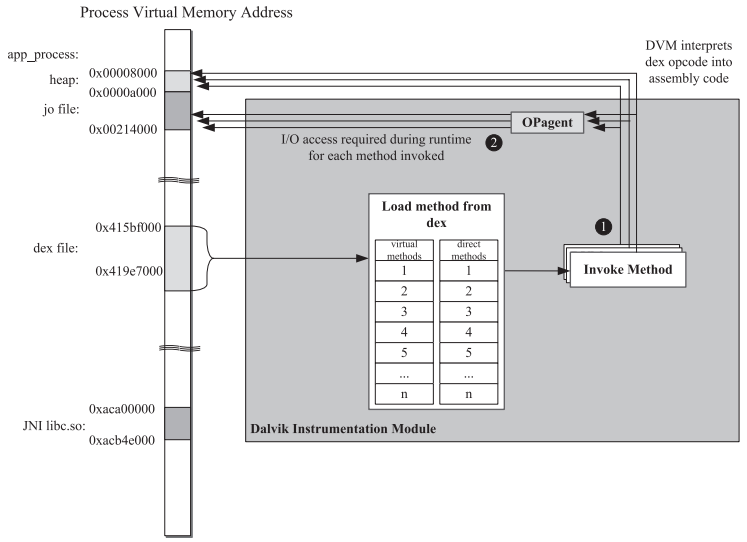


Fig. 4. Instrumenting 'method invoke' creates more runtime overhead.

instrumenting every JVM function that would alter the memory allocation, such as method compilation, method recompilation, method relocation, garbage collection, etc.

In the case of the register-based DVM, stack frames also undergo garbage collection; however, since the dex files are mapped onto the process memory map, the addresses are fixed as long as the process remains running. As a result, method VMA retrieved by DIM remains the valid method address for the duration of profiling. When DVM invokes a method, it loads a section of the mapped dex file starting from the VMA where the method's beginning opcode is located. DIM instrumentation captures this action and sends the starting VMA of the invoked method through the direct memory tunnel. To ensure DIM instruments all method invocation's final entries points, this article examined DVM opcode interpreter. The DVM interpreter contains four types of method invocations. DIM instrumentations are inserted at the DVM native code where the method invocations instructions are issued. The flow of DIM retrieving VMA for methods being invoked is show in Figure 4. The DIM component responsible for delivering VMA through the memory map tunnel is not shown in the figure.

(1) When a method is invoked, the interpreter retrieves opcodes of the method from the VMA and loads the interpreted code into the heap. For each method invoked, DIM sends method name and its VMA to a substitution symbol file through opagent. When a JNI method is invoked, DIM only sends the init method through opagent. When Dalvik-heap executes the JNI call, they are processed by Oprofile as regular Linux user-space execution.
(2) By instrumenting method invocation, every method invocation requires additional I/O operations to write the substitution symbol file.

The benefit of instrumenting method invocation is that only the information of those methods used will be saved, thus reducing the spatial overhead for the substitution symbol file and the generated substitute ELF file. For mobile devices, sample log files can use up limited storage space quickly. However, I/O operations in step 3 are costly, even if the substitution symbol file does not need to be converted into ELF format during runtime. Furthermore, the benefit of reducing spatial overhead is eliminated if most of the methods are invoked during profiling.

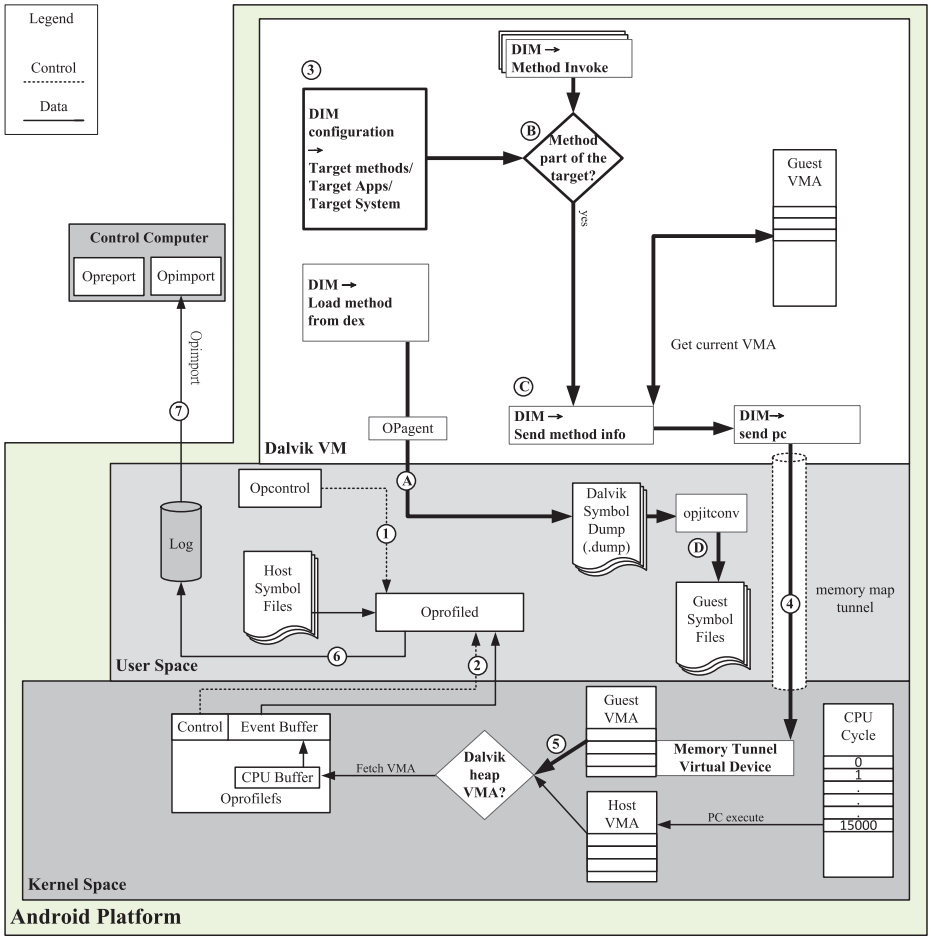


Fig. 5. Vertical profiling architecture.

4.2. VARI Profiler Architecture and Profiling Flow

Figure 5 describes the architecture of the proposed VARI Vertical Profiler. The dotted lines represent control signals. Solid lines represent dataflows. Components and flows with bold borders and fonts are modified or added for proposed vertical profiling framework. The rest are existing components of Android, Oprofile, and Oprofile JIT extension tools.

Profiler is initiated by setting up sampling frequency and assigning hardware performance counters of interest in step ①. Opcontrol then initiates the Oprofile daemon and sampling begins. Oprofile daemon will halt the system for a short period to empty and process sampled data in the event buffer when the buffer is almost full in ②.

When the DVM process begins invoking methods after initiation, DIM looks up the method's VMA, which is the VMA of the dex file in the process map plus an offset, then gathers relevant information in ③.

After retrieving method VMA, DIM sends it through the proposed memory map tunnel to a designated kernel space virtual device in ④. The virtual device file only contains a valid VMA when a method is invoked. Once Oprofile's kernel driver receives a pmnc (Performance Monitor Control) interrupt, and the modified driver determines

this sample belongs to a Dalvik method, the sample VMA is replaced with the VMA in the virtual device file in ⑤.

DIM then saves the current method name, method VMA, and size into a substitute symbol file. Once the profiling is complete, Oprofile daemon calls opjitconv which converts the saved substitute symbol files into ELF symbol files. These ELF files contain only the relevant symbol information for Oprofile to correlate samples to Android applications.

The original Oprofile JIT extension expects that Oprofile will be unable to find symbols of Java applications samples and resort to labeling them as anonymous. Due to Dalvik's process management, Oprofile actually identifies these samples as Dalvik heap or Dalvik JIT code cache execution, part of the Android shared memory process (ashmem). In step ④, DIM and Oprofile kernel driver modifications remapped Dalvik method samples with a new VMA to avoid this issue.

At this point, the ideal behavior is for Oprofile daemon to process these ELF files as it would do with regular Linux processes and saves the logs in the file system. However, these remapping VMAs actually point to application dex files on the DVM process memory map, and not the ELF symbol files. Oprofile daemon saves samples to logs in ⑥, which are retrieved along with the ELF information file in ⑦.

4.3. Memory Map Address Remapping and Substitute Symbol File

This article noticed that Dalvik VM map dex files contain application opcodes to the application's process memory map, and this address does not change for the duration of application execution. This article proposes a memory map address remapping mechanism to replace the address of Oprofile samples in kernel space with its corresponding dex file memory map location. This eliminates the need to keep track of the interpreted method address in the heap or stack as they change after VM memory operations.

DIM instrumentation at method invocation will trigger the remapping mechanism and deliver the method's address on for its corresponding dex file to a kernel space virtual device file mapped to the DVM's memory map. This communication technique goes through the profiling barrier to provide DVM application information to Oprofile; therefore it is referred to as direct memory tunnel. The Oprofile driver device is modified to monitor the pmnc (performance monitor control) interrupt and replace current sample's VMA when the memory tunnel sets the virtual device file's value. After the remapping, the rest of the flow inside the kernel space remains unchanged from the original Oprofile.

Utilizing the direct memory tunnel to update time-sensitive application method location information as methods are invoked reduces context-switching overhead caused by implementing system calls.

4.3.1. Substitute Symbol File Generation. After remapping Oprofile's sample address, even though DVM application samples are now able to link back to their own dex file, they cannot be traced back to a method, because dex files contain Dalvik opcodes, and currently Oprofile only processes ELF format symbol files. To correlate samples to the correct Dalvik application methods, a technique is required for Oprofile to link samples to a substitute symbol file.

To resolve this issue, DIM needs to provide method name, method VMA, and method code size to an ELF format skeleton symbol file. This article utilizes opagent and opjitconv from Oprofile's JIT extension to write the substitute symbol file. When target application initiates, it creates an empty substitute symbol file and allocates it in the DVM's memory map.

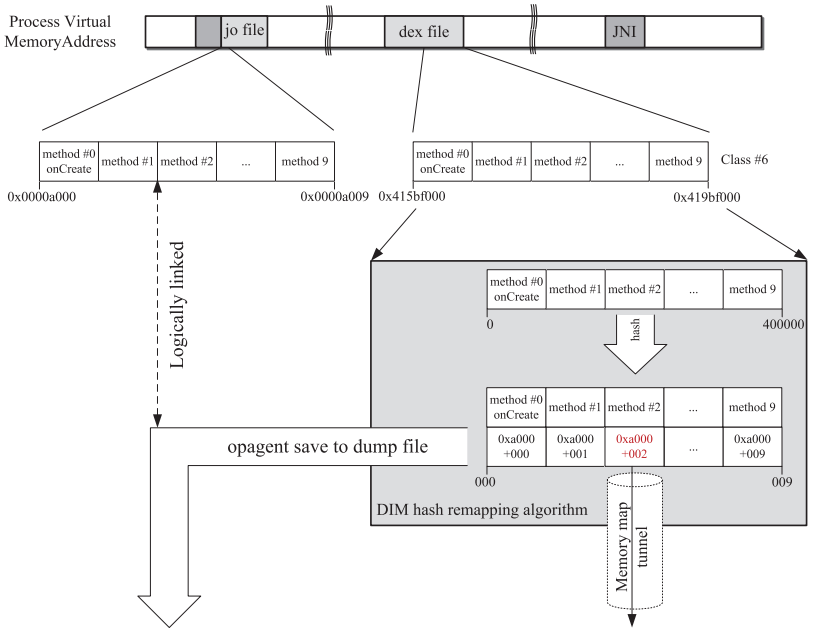


Fig. 6. Generating hash address remapping symbol file.

This file's mapped address is different from the dex files, hence before DIM sends a method's VMA through the memory map tunnel, it will need to recalculate the method's new location on the substitute symbol file location. The simplest way to do so is to shift all the address by the offset between substitute symbol file's address and the address where the dex file is mapped to. If there are more than one dex file in an application's memory map, they will also be mapped to the same substitute file with an updated offset to last entry of the file.

4.3.2. Hash Address Remapping. The issue with the previous methodology is that since the substitute ELF file contains only the method name and the address, most of the space allocated is wasted. Larger applications could cause allocated section to overflow. Figure 6 shows the remapping algorithm used to reduce memory necessary for mapping the substitution ELF file. A hash table is generated to reduce overhead of looking up the method's substituted address during runtime before sending the method VMA through the memory map tunnel. Also the substitute symbol file only records the name and address of the method.

Ideally, this substitution ELF file is generated by utilizing functionalities provided by opjitconv during runtime from the substitution symbol file provided by DIM. However, since symbol searching is carried out by Oprofile analysis tools, such as opreport and opannotate during post-processing, generating ELF files during runtime becomes unnecessary, especially when it creates extra runtime overhead. Instead, the substitution symbol file will be converted into the ELF format when profiling concludes.

By eliminating the need for runtime generated substitute file, only a dummy substitute address needs to be recorded by the hash table. This file is only logically mapped to the dex files on the process virtual memory map, since method information can be acquired by correlating the substitute address to the dex file in post-processing. The

memory space savings are shown in the following equations.

$$S_{original} = \sum_{i=1}^{N} M_i,$$

$$S_{dummy\_hash} = N,$$

$$R_{original} = (A_i - A_0) + H_0,$$

$$R_{dummy\_hash} = i + H_0,$$

where there are N methods in the target application. M_i is the code size of each method of the target application. $S_{original}$ is the memory size of the original remapping file memory map allocation. S_{dummy_hash} is the memory size of the heap table remapping file memory map allocation. $R_{original}$ is the substitution address sent through the memory tunnel for memory map address remapping without hash table. R_{dummy_hash} is the hash table substitution address sent through the memory tunnel for memory map address remapping with hash table substitution. A_0 is the first address of the first dex file on the target application memory map. A_i is the address of the current method's corresponding dex file address. H_0 is the starting address of the allocated hash remapping file. $R_{original}$ and R_{dummy_hash} need to be calculated at runtime. It is clear the hash table methodology requires less look up to calculate a method's substitute address. The maximum size (N) for a hash table substitute address file can also be determined quickly during target application's initiation.

A discovery of an instrumentation methodology specific to Dalvik virtual machine renders the dummy hash table substitution unnecessary for most methods. However, the hash remapping methodology can still be used to keep track of dynamically compiled methods generated by Android's new just-in-time compilation capabilities. Aside from Android's JIT functionalities, the aforementioned hash table remapping method can also be used to adopt VARI profiler to stack-machine-based JVMs.

5. DALVIK INSTRUMENTATION MODULE

The Dalvik instrumentation module is an agent added to DVM for the purpose of retrieving application information within the virtual machine. The DIM is aimed to be architecture-neutral; therefore, this article avoids instrumenting the assembly portion of the DVM's opcode interpreter. The applied instrumentation methodology determines the accuracy of the final profiling result. This section examines instrumentation module in DVM with the load method from dex instrumentation methodology and how to lower overhead based on DVM's distinctive features.

5.1. Load Method from dex Instrumentation

This article observed that DVM loads dex files into its process' virtual memory map during the initiation of the DVM process. Therefore, except for the JIT functionality, the dex VMA allocation of an interpreted method is known before it is invoked. When the target application initiates, DVM will store the VMA of every method from a dex file into an array. This gives the opportunity to avoid instrumenting actual runtime method invocation; instead, DIM can write the name and VMA of every method in the dex files to the substitute symbol file before the application is executed.

This methodology merges multiple runtime I/O operations in the previously mentioned method, invoking instrumentation technique into one single write operation to the substitute symbol file at the initiation of the application. Aside from a slightly slower application start time, load method from dex instrumentation does not affect

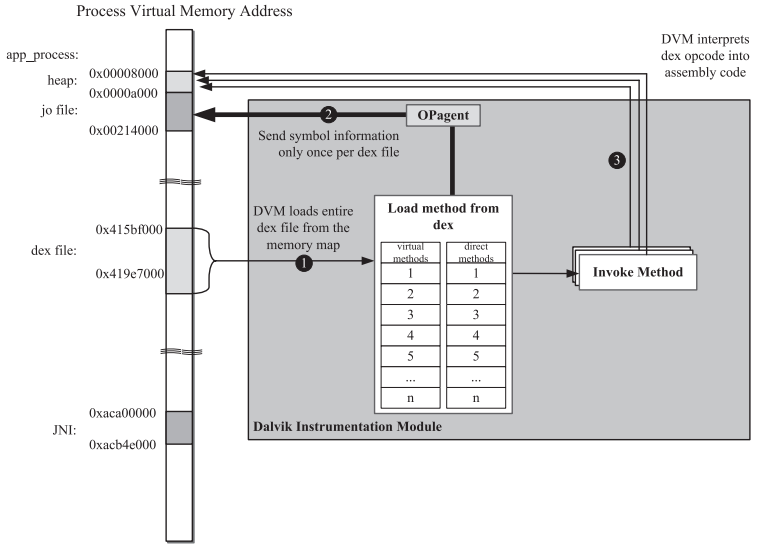


Fig. 7. Instrumenting when DVM loads the every method of a dex file.

profiling results and does not perturb the target application during runtime. Thus this methodology reduces the overhead caused vertical profiling.

Figure 7 shows the change in the DIM flow. While step 1 remains the same, before methods are invoked, in step 2, DIM sends information for every method to opagent when an application initiates. DVM doesn't begin invoking methods until it has completed loading methods into the array in step 3; therefore, this instrumentation does not impact the profiling result during runtime.

5.2. Improved VARI Profiler Architecture and Profiling Flow

Figure 8 shows the modified portion of VARI profiling flow after switching to load method from dex methodology. DIM is modified to instrument at application process initiation to loading every method from dex files and adding them to the hash address remapping table. The highlighted portion shows component changes. New components are in bold.

The following is a detailed description of modified DIM instrumentation flow.

(A) DVM loads every method from application dex files during application initiation. DIM gathers information for all methods and saves it to a substitution symbol file via opagent.
(B) At method invocation, DIM determines if the invoked method is a profiling interest. Hash address remapping is only generated to keep track of methods dynamically compiled by JIT compiler. The generated address is stored in DIM as a hash table for faster lookup. Maximum hash lookups take O(n) time. The generated remapping VMA is sent through the direct memory tunnel and to the substitute symbol file along with method name.
(C) DIM looks up remapping VMA of none JIT methods and sends it through the direct memory tunnel. For profiling flow that incorporates the hash remapping methodology, reused methods only need to look up the hash table for their original translated address. In that case, all methods will be mapped to the substitute ELF format file. Regardless of whether the current method VMA is replaced by the hash

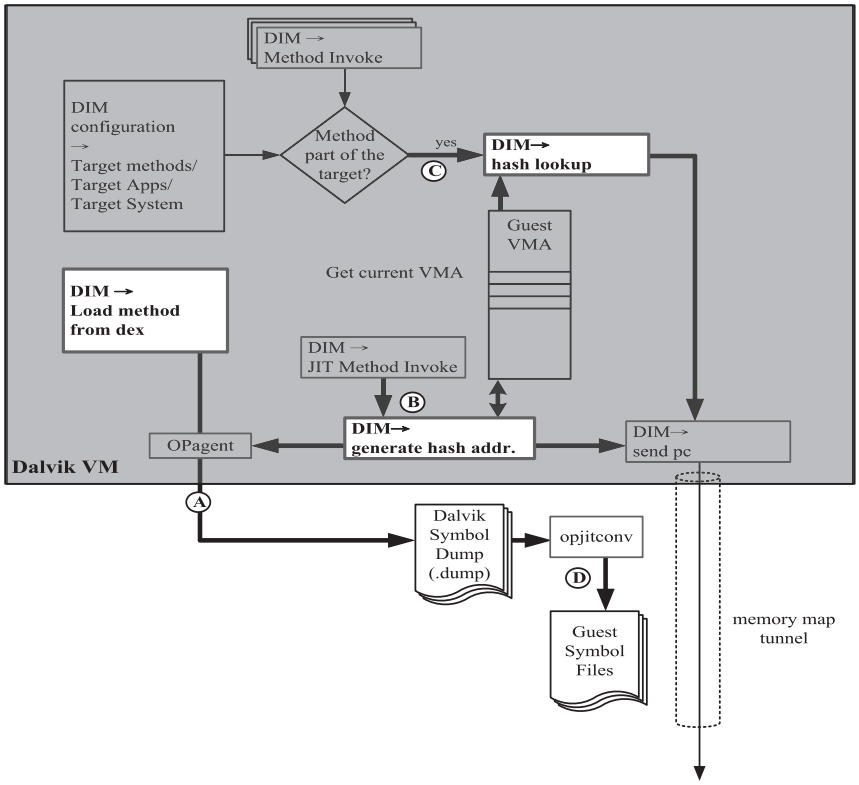


Fig. 8. Hash remapping table and load method instrumentation flow.

lookup, DIM sends it to the kernel space virtual device through the direct memory tunnel.

(D) The substitute symbol file created in step A is converted into the substitute ELF file. This substitute ELF file is the guest symbol file. To reduce overhead, this ELF file is generated post profiling. It is only logically mapped to the dex files on the process virtual memory map to ensure the memory allocation will not be used.

The hash remapping address mechanism is useful for keeping track of dynamically compiled methods from JIT. In this case, the process' virtual memory map has to allocate a section of virtual memory for the substitute symbol file to ensure address generated by the remapping algorithm will not create a conflict with other allocations. By remapping both the JIT compiled and interpreted methods into the address range of the substitute symbol file file, no renaming is required during post processing. Oprofile's post-process analysis tools will search for the symbols for DVM related symbols within the substitute symbol file file. But in comparison, renaming the substitute symbol file post profiling process is still the methodology that introduces the lowest runtime overhead.

5.3. Reconfigurable Profiling

DIM employs two techniques to achieve reconfigurable profiling. For finer granularity, DIM can be set to write information only for the specified methods from the profiling target method list to the substitute symbol file. For larger granularity, DIM can be

Table II. Testing Platform

Development Board		
Beagleboard	Rev. C4	
CPU	OMAP3530DCBB72 720MHz	
CPU Arch.	CPU: ARM V7	
POP Memory	Micron	
	2Gb NAND (256MB)	2Gb MDDR SDRAM (256MB)

configured to only profile DVM process of an application by checking the process command line, which is the equivalent of Java package name.

Hard-coding the desired application to profile will require recompilation of the entire Dalvik virtual machine every time a new application needs to be profiled. A configuration file informing DIM which application or methods to profile avoids having to recompile DVM. This configurability can be provided by modifying a text configuration file.

By combining several method-level or application-level configurations, DIM can provide a better picture of specific events, for example, combining methods from the system server process to evaluate video performance. By only writing the substitution symbol files of Android's system service process to the file system, Oprofile daemon can organize samples by system services. Services specific profiling can identify power hotspots and point to possible causes.

Such measures reduce the overhead of writing unnecessary files to the file system when the developer only wants to track down bottleneck in a specific application or a section of code.

6. RESULTS

6.1. Experimental Environment

Testing is conducted on a Beagleboard Rev. C development board running Android 2.3 Gingerbread which incorporates a JIT compiler (Table II).

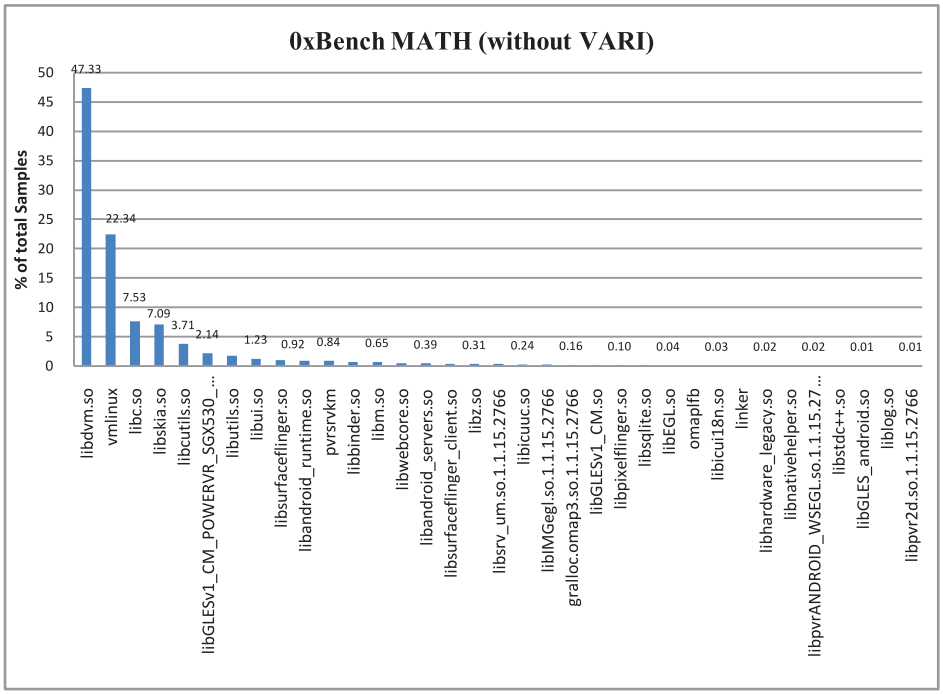
6.2. VARI Profiling Results

Figure 9 demonstrates profiling results of the unmodified Oprofile provided with the Android source code and VARI profiler, running the same 0×Bench Math benchmark. 0×Bench Math benchmark includes Java mathematic benchmarks Linpack and Scimark2 (Table III).

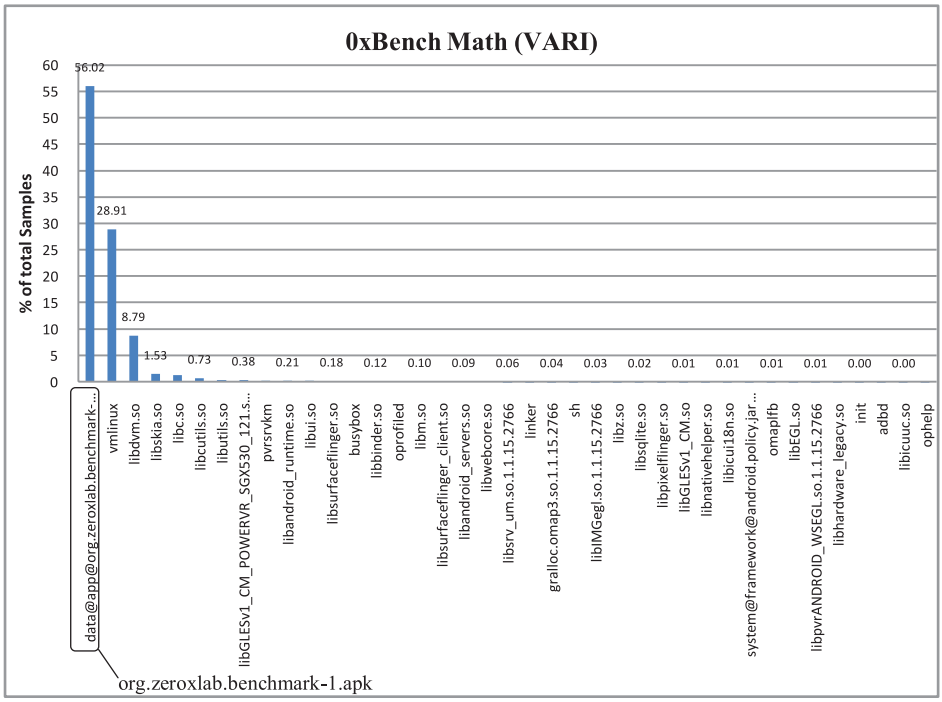
In Figure 9(a), the unmodified Oprofile's does not provide any Android application information from within the Dalvik virtual machine. Therefore it appears that the virtual machine itself (libdvm.so) takes a large portion of the execution time; however, most of the samples came from application executions and not the virtual machine itself.

The VARI profiler result running the same benchmark is shown in Figure 9(b). It reveals that the actual Dalvik VM itself only takes up less than 9% of the execution time. The system spends most of time, 56%, executing the Android benchmark application data@app@org.zeroxlab.benchmark-1.apk@classes.dex. Bottlenecks of an application could exist in the application itself or certain system library. If this were an underperforming application, it would indicate that the source of bottleneck is likely to exist in the application instead of the system's native libraries.

Multimedia applications depend on Linux native libraries for image processing. Figure 10 shows the profiling result of running the Newton's Cradle application, which is a two-dimensional graphic-intensive application. In this case, graphics drivers and libraries, such as Android's two-dimensional graphic library libskia, take up the most execution time after the Linux kernel. The VARI profiling result reflects the



(a) Original Oprofile result without VARI profiler.



(b) Same benchmark result with VARI profiler.

Fig. 9. Profiling results of 0xBench Math benchmarks: (a) Original Oprofile provided by Android source code, only Linux level information is visible. (b) The proposed VARI profiler can identify Android application samples.

Table III. Experimental Environment

Experimental Environment		
Android	2.3 Gingerbread	
Kernel	2.6.29-00261	
Compiler	gcc version 4.4.0	
Oprofile	0.9.6	
Benchmarks	Newton's Cradle [Hyndman 2011]	
	0×Bench Math [jserv 2010]	Linpack [Dongarra et al.][Dongarra et al.]
		Scimark2 [Pozo and Miller 2004]
	0×Bench VM [jserv 2010]	

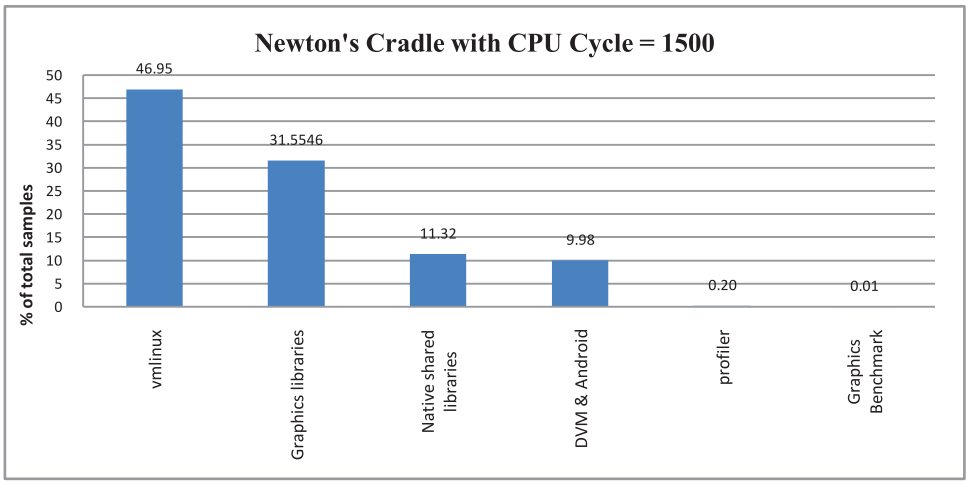


Fig. 10. Example of invoked Linux graphics libraries and native libraries taking up more execution time than the Android application itself.

characteristics of the Android application. In comparison with the graphics library, the actual Newton's cradle application workload is less significant.

If this is an underperforming application, there are two targets for optimization. First, the lack of performance could be due to not using the optimal native library functions for the task, and performance could be improved by better utilizing Linux native libraries. Second, the performance might improve by optimizing the Linux native library or the Linux device driver. System software engineers can use VARI profiler to investigate whether the performance of a native library or driver is affected running with applications inside the Dalvik virtual machine.

The last example is when the Dalvik virtual machine is actually consuming most of the computing time. Figure 11 is the VARI profiler result of 0xBench's virtual machine benchmark, which performs garbage collection tests. The result shows that libdvm.so and other Android services are using up close to 70% of the CPU time. These types of results could mean performance is affected because the application is causing too many GC.

Reducing resources used or cleaning up allocations in the application can minimize numbers of GC; for instance, scaling back frame rate for graphic-intensive games can decrease virtual machine workload. System software engineers can also use the VARI profiler to identify whether the Dalvik virtual machine itself requires optimization, such as adjusting heap frame size or improving opcode translation for the target architecture.

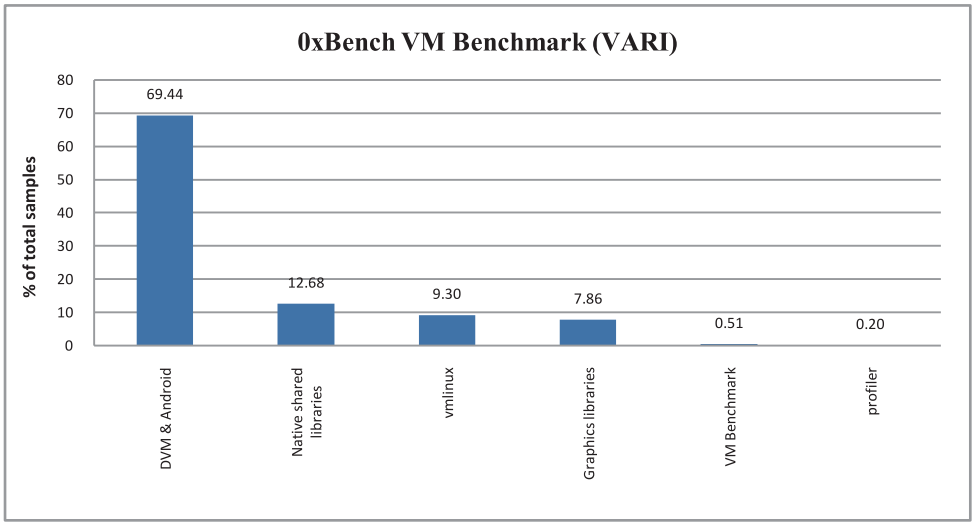


Fig. 11. VARI profiling result of 0×Lab Benchmark with overhead in DVM itself.

Table IV. Sample DIM Instrumented Results Corresponding to Memory Map

PC	VMA	Method Name	Class Description	Source File Name	Line#
6	0 × 42729240	run	Ljava/lang/Runnable;	ViewRoot.java	2044
636	0 × 43966df0	atan	Ljava/lang/Math;	BallsView.java	616
725	0 × 254608	native_drawBitmap	Landroid/graphics/Canvas;	Canvas.java	−2

(a) VMA can trace back to the line number of the source Java file.

Start Addr.	End Addr.	Mod	Nod.	Mapped File Name
0000a000	002bd000	rwxp	0	[heap]
4245b000	42b01000	r-xp	293	/data/dalvik-cache/system@framework@framework.jar@classes.dex
43964000	4396b000	r-xp	723	/data/dalvik-cache/data@app@com.geekyouup.android.newton-1.apk @classes.dex

(b) VARI profiler and VMA remapping correlates heap exec. to respective dex files.

The data generated by vertical profiling helps developers to spot possible bottlenecks quickly. If a suspected Linux library is compiled with debugging symbol information, the profiling result could be matched to each line of the source code. With this feature, developers could determine if a Linux library can be improved to gain better performance.

6.3. VARI Mapping Results

6.3.1. DIM and Symbol Correlation. The next table illustrates how results from the Dalvik instrumentation module can be correlated to symbol files mapped to the process virtual memory map. Table IV(a) demonstrates that DIM is able to retrieve information inside the virtual machine, such as a method's program counter, method name, Java class description, the name of its source Java file, and the line number of the method in the source file. This is achieved by matching the process memory map for the test application, as shown in Table IV(b).

Three virtual machine methods are used as examples. The first method, named run, has a VMA value of 0 × 42729240, which falls into the range of process map address belonging to the Android framework dex file. The VMA of the second method `atan` falls

```
603  for(int p=0;p<numPointers;p++)
604  {
605      if(mObjectTouched && mObjectsTouchedIds[p]!=-1)
606      {
607          mBallVelocity[mObjectsTouchedIds[p]]=0;
608
609          //translate the roational origin
610          float transX = reflectGetXForPointer(event, p)-mCenterOfRotationX[mObjectsTouchedIds[p]];
611          float transY = reflectGetYForPointer(event, p)-mCenterOfRotationY;
612
613          if(!isOrientNormal) {transX=-transX; transY=-transY;}
614
615          //figure out the angle the ball is now at
616          if(transY>0) mBallAngle[mObjectsTouchedIds[p]] = ((float) (-Math.atan(transX/transY))+PI_F);
617          else mBallAngle[mObjectsTouchedIds[p]] = (float) -Math.atan((transX/transY));
618      }
619  }
```

Fig. 12. BallsView.java source code for Newton's cradle.

Table V. Call Graph Result

samples	%	image name	symbol name	Code
474	25.9302	app_process	/system/bin/app_process	
→474	25.9302	app_process	/system/bin/app_process	[self]
193	18.7016	vmlinux	/sdcard/vmlinux	
→193	100.000	vmlinux	/sdcard/vmlinux	[self]
113	10.9496	org.test.app	org.test.app	
→45	38.823	org.test.app	org.test.app	org.test.app/.Graphics
→68	60.000	libGLES_android.so	libGLES_android.so	[self]

$ opreport -t 10 -l ../samples/.
CPU: ARM V7 PMU, speed 0 MHz (estimated).
Counted CPU_CYCLES events (clock cycles counter) with a unit mask of 0 × 00 (No unit mask) count 15,000.

into the test application's dex address range, and this function can be found in the BallsView.java application source file, as show in Figure 12.

The third method native_drawBitmap is a graphics library function of the Android system. Since the VARI profiler is only instrumenting, its VMA was not replaced by a memory map tunnel and therefore falls into the heap section of the process map. Consequently, Oprofile will be unable to correlate the samples of this function back to its Java method and source location.

6.3.2. VARI Call-Graph Results. The vertical profiler generates call graphs and other features provided by Oprofile. Table V is a section of call-graph information generated by opreport after a profiling session. This particular call graph shows that after Kubench benchmark finishes one round of testing, it tries to use another Linux system library.

Graphical representations of call graph table can be generated by dot or the VARI profiler eclipse plug-in. Utilizing results like Table V and Figure 13 enables developers to discern relations between Linux user-space libraries and Android applications right away. Bottlenecks of an application could exist in the application itself or certain system library. The data generated by vertical profiling helps developers to spot possible bottlenecks quickly.

6.4. DIM Overhead

The following is overhead statistics of the proposed VARI profiler. Sending VMA through the memory tunnel and writing method information to the substitution symbol file are the two most significant DIM events. It is essential to implement these two

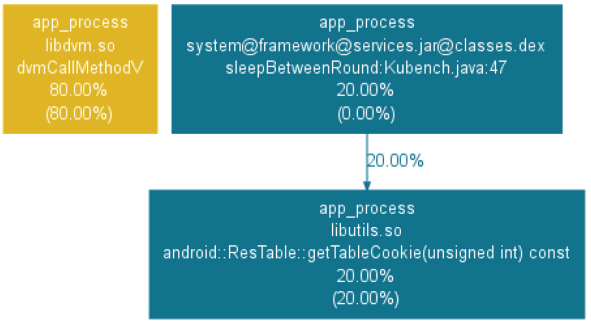


Fig. 13. Graphic representation of the call graph using dot.

functions appropriately, as they greatly affect profiling performance and accuracy. This section examines whether the proposed methodology is efficient.

The analysis is done by running these methodologies through four benchmarks. Newton's cradle is a two-dimension graphics simulation of the conservation of momentum and energy. User controls the application through drag and release motions. 0×Benchmark's math test consists of Linpack and Scimark2. 0×Benchmark's VM test consists of a garbage collection test that creates a large binary tree and a large array of doubles to test GC performance. 0×Benchmark's 3D benchmark includes popular OpenGL benchmarks, such as OpenGL Cube, OpenGL Blending, OpenGL Fog, and Flying Teapot.

6.4.1. DIM Log File Size Overhead. Enabling vertical profiling unavoidably increases profiling result file size by introducing Java application information to original Linux level samples. If the DIM methodology is applied to Linux tracing tools which generate GB sized result files within a short execution time, it is also undesirable to add large virtual machine instrumentation logs to the limited system resources and storage space. These additional file system accesses to provide vertical profiling capability should be limited to avoid perturbing the target application.

Traditional vertical profiling methodologies require recording every time a method is invoked to correlate method with the currently assigned heap address. If a method is invoked more than once, the profiler needs to record duplicates of the method's information. The longer the profiling process goes on, the more excess logs need to be written.

VARI profiler's DIM methodology avoids writing to the file system during profiling runtime to minimize runtime overhead. It does so by exporting every method's information during the application initiation. This means if a large target application invokes only a few methods during profiling, the unused method information is still written to file.

The result file size overhead is shown in Figure 14. Newton's cradle has a relatively small code size of 25.9KB. Thus, of the four benchmarks, it generates the smallest VARI profiler log. The other three benchmarks are part of the same application with a code size of 236.6KB. Therefore they have similar VARI profiler log sizes. The small variations in log sizes arise from extra information included for the different Android core libraries that each benchmark called. The largest log size of the three is the three-dimensional graphics benchmark with 276KB. It is only 48KB larger than the virtual machine benchmark, with the smallest log file of the three. Log size generated by DIM does not increase with application execution time.

On the other hand, the traditional vertical profiling methodology incurs 166KB more log file size overhead than VARI profiler for the Newton's cradle application, because despite its small code size, during experiments, Newton's cradle has a long application execution time. Of the three 0×Bench benchmarks, the virtual machine benchmark

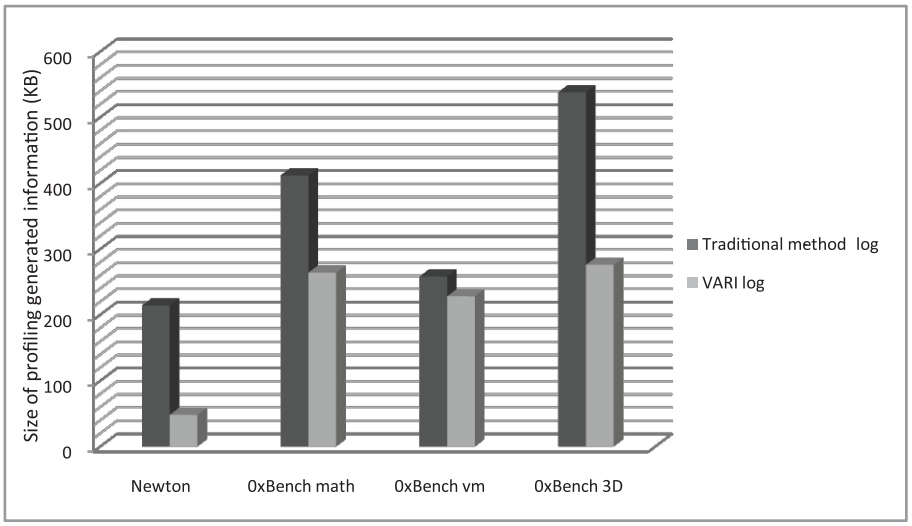


Fig. 14. Vertical profiler sample information size overhead.

has the shortest execution time, and as a result, also has the smallest traditional vertical profiling log size of the three. The three-dimension graphics benchmark has the longest execution time; therefore, it has the largest log file overhead, consuming a total of 538KB additional space.

6.4.2. DIM Runtime Overhead. Sending VMA through the memory tunnel and writing method information to the substitution symbol file are the two most significant DIM events. It is essential to implement these two functions appropriately, as they greatly affect profiling performance and accuracy. This section examines whether the proposed methodology is efficient. The runtime overhead experiment uses three benchmarks, Newton's Cradle, 0×Bench Math benchmark, and 0×Bench virtual machine's benchmark to demonstrate the efficiency of each methodology. Target applications' execution time running on the unmodified Android Dalvik machine using original Oprofile from the Android source code is used as control statistics.

VARI profiler utilizes Oprofile together with memory map address remapping, direct memory tunnel, and the instrumentation methodology proposed by this article. Figure 15 compares the total profiling time in seconds among three different methods for Dalvik virtual machine instrumentation. The DIM overhead statistics are measured with performance counter cycle count and then converted to seconds with CPU clock frequency.

Figure 16 shows the overhead analysis of the additional runtime in percentage over the original Oprofile execution time. The y-axis is in logarithmic scale due to a huge difference between overhead created by different methodologies. The x-axis shows DIM instrumentation methodology overhead with each benchmark.

Vertical profiling methodology, such as instrumenting every 'method invoke', involves writing method information to a log file when the method is called during runtime in order to keep track of the allocated heap address. It is the traditional approach to instrument JVM. Since it writes to file frequently during runtime, the overhead is much large in comparison to instrumenting 'load method from dex', which only writes to file once when the test application initiates.

The extra overhead is generated by context switches to access I/O and I/O latencies. An addition of at least eight seconds is added to the original profiling time for the

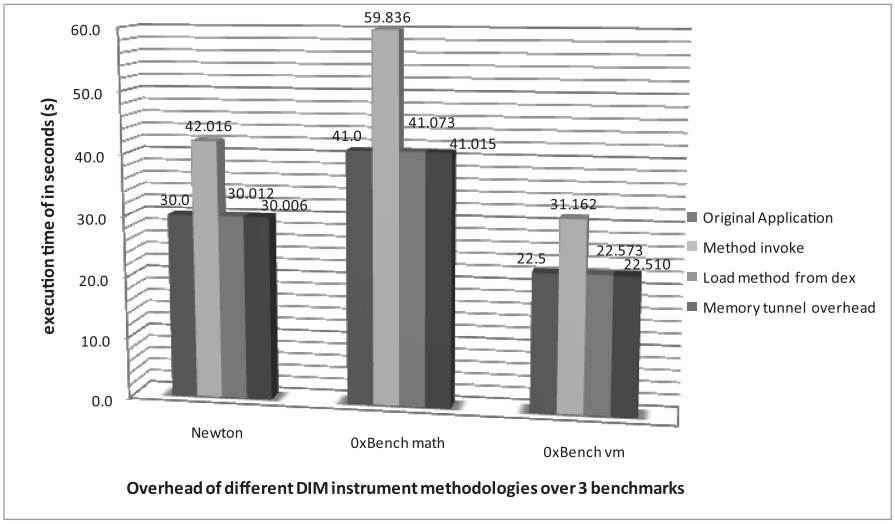


Fig. 15. Total profiling time of different DIM instrument methodologies.

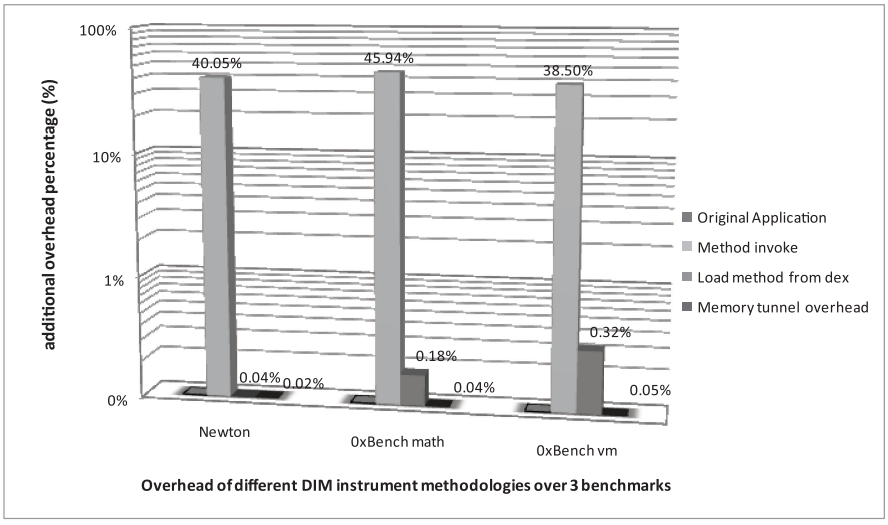


Fig. 16. Overhead of different DIM instrumentation methodologies.

method invoke tactic, which introduces an overhead of at least 35%, where as 'load method from dex' instrumentation only asserts at most an additional 0.08 seconds, an overhead of 0.32%.

Figure 16 is in logarithmic scale due to the huge difference between overhead created by different methodologies.

The memory map tunnel is the foundation of this article, as it makes memory remapping technique possible; furthermore, it generates less than 0.02 seconds of additional overhead during the entire profiling process when compared to the original Oprofile flow. Vertical profilers that do not utilize a memory map tunnel have to record the same information to a file. If memory tunnel is removed from the VARI profiler, given that

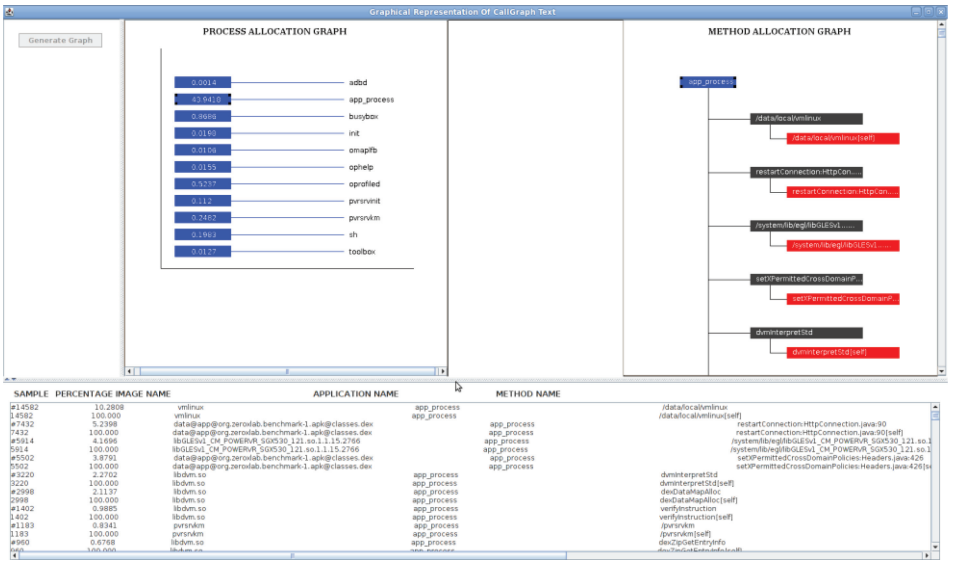


Fig. 17. VARI profiler Eclipse plug-in.

the remapping VMA is sent through the memory map tunnel at every method invoke, the overhead of writing them to a file would be similar to the overhead for the method invoke tactic. This demonstrates the efficiency of the memory map tunnel. It greatly reduces overhead by avoiding context switches and I/O latencies during runtime.

7. CONCLUSIONS

The goal of this article is to establish a bridge to communicate application information from Dalvik virtual machine to a Linux user-space profiler. Moreover, proposed vertical profiling methodology, such as memory map address remapping methodology and communication tunnel, are not bound to Oprofile, and can also be adapted by other Linux user-space trace tools, such as strace or LTTng.

Furthermore, the techniques used in the Dalvik instrumentation module, such as the hash remapping address, can be applied on other Java virtual machines to achieve register-based machine-styled profiling on stack machines.

Currently, even though the VARI profiler provides some source-line-level information, this functionality is not integrated with native tools found in Oprofile. There are several ways to approach this issue, such as converting dex file's DWARF3 inspired debug_info_item [45] into an ELF symbol file, or modifying oprofile's post-processing tools to utilize dex disassembler to treat dex files as a symbol file.

As it is, by bridging the profiling gap between the Dalvik virtual machine and Linux user and kernel space, the VARI profiler achieves vertical profiling on Android mobile systems. This ability enables users to identify Java-level bottlenecks and trace back to the point of origin in the Linux system library while introducing less than 1% runtime overhead.

VARI and DIM enables Oprofile to provide integrated virtual machine application analysis. Eclipse is the recommended integrated development environment (IDE) for Android. By incorporating graphical analysis and side-by-side source-line annotation from the VARI profiler into eclipse IDE, as shown in Figure 17, developers can analyze application bottlenecks intuitively.

ACKNOWLEDGMENTS

Many thanks to Professor Tien-Fu Chen for his guidance and to our lab mates at the National Chiao Tung University for their assistance and inputs. Special thanks to Po-Chun Chang for advising to understand the issue and not just trial and error.

REFERENCES

B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Syst. J. 44*, 2, 399–417.

B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Shepherd, and M. Mergen. 1999. Implementing jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 314–324.

L. Batyuk, A.-D. Schmidt, H.-G. Schmidt, A. Camtepe, and S. Albayrak. 2009. Developing and benchmarking native Linux applications on Android. In *MobileWireless Middleware, Operating Systems, and Applications*. J.-M. Bonnin, C. Giannelli, and T. Magedanz, Eds., Springer Berlin, 381–392.

W. Binder, J. Hulaas, and P. Moret 2006. A quantitative evaluation of the contribution of native code to Java workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 201–209.

W. Binder, J. Hulaas, and P. Moret. 2007. Advanced Java bytecode instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. ACM Press, 135.

C.-W. Chang, C.-Y. Lin, C.-T. King, Y.-F. Chung, and S.-Y. Tseng. 2010. Implementation of JVM tool interface on Dalvik virtual machine. In *Proceedings of the International Symposium on VLSI Design Automation and Test (VLSI-DAT)*. IEEE, 143–146.

W. Cohen. 2004. Tuning programs with Oprofile. *Wide Open Mag. 53*.

G. Contreras and M. Martonosi. 2005. Power prediction for Intel XScale® processors using performance monitoring unit events. In *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM Press, 221.

J. Dongarra, R. Wade, and P. McMahan. Linpack Benchmark – Java Version. http://www.netlib.org/benchmark/linpackjava/.

Google. 2011. Dalvik - Code and documentation from Android's VM team - Google Project Hosting. http://code.google.com/p/dalvik/.

Google. 2010. Using DDMS | Android Developers. http://developer.android.com/guide/developing/debugging/ddms.html.

Google. 2008. Anatomy & Physiology of an Android - 2008 Google I/O Session Videos and Slides. http://sites.google.com/site/io/anatomy–physiology-of-an-android.

M. Hauswirth, A. Diwan, P. F. Sweeney, and M. C. Mozer. 2005. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM Press, 281.

M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. 2004. Vertical profiling: Understanding the behavior of object-priented applications. *ACM SIGPLAN Not. 39*, 251–269.

R. Hyndman. 2011. Newtonscradle - Android app to model the physics of Newton's Cradle - Google Project Hosting. http://code.google.com/p/newtonscradle/.

H. Inoue, and T. Nakatani. 2009. How a Java VM can get more from a hardware performance monitor. *ACM SIGPLAN Not. 44*, 137–154.

JSERV. 2010. 0×bench - Comprehensive Benchmark Suite for Android - Google Project Hosting. http://code.google.com/p/0xbench/.

S. Khan, S. Khan, S. H. K. Banuri, M. Nauman, and M. Alam. 2009. Analysis of Dalvik virtual machine and class path library. Tech. rep. Security Engineering Research Group, Institute of Management Sciences, Peshawar, Pakistan.

J. Maebe, D. Buytaert, L. Eeckhout, and K. De Bosschere. 2006. Javana: A system for building customized Java program analysis tools. *ACM SIGPLAN Not. 41*, 10, 153–168.

H. Mousa, and C. Krintz. 2005. HPS: Hybrid profiling support. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE, 38–47.

H. Mousa, K. Doshi, T. Sherwood, and E. Ould-Ahmed-Vall. 2010. VrtProf: Vertical profiling for system virtualization. In *Proceedings of the 43rd Hawaii International Conference on System Sciences (HICSS)*. IEEE, 1–10.

H. Mousa, C. Krintz, L. Youseff, and R. Wolski. 2007. VIProf: Vertically integrated full-system performance profiler. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium. (IPDPS'07)*. IEEE, 1–6.

D. Nicolaescu and A. Veidenbaum. 2005. Understanding and comparing the performance of optimized JVMs. In *Proceedings of the Conference on Innovative Architecture for Future Generation High-Performance Processors and Systems*. IEEE.

Oracle. 2002. JVM(TM) Tool Interface 1.0.38. JVM Tool Interface. http://download.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html.

Oracle. 2010. Java SE - Java Platform Debugger Architecture Home. http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

K. Paul and T. K. Kundu. 2010. Android on mobile devices: An energy perspective. In *Proceedings of the IEEE 10th International Conference on Computer and Information Technology (CIT)*. 2421–2426.

R. Pozo and B. Miller. 2004. Java SciMark 2.0. http://math.nist.gov/scimark2/.

F. T. Schneider, M. Payer, and T. R. Gross. 2007. Online optimizations driven by hardware performance monitoring. *ACM SIGPLAN Not. 42,* 6, 373–382.

L. Shannon and P. Chow. 2004. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. ACM, 190–199.

P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. 2004. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium*. Vol. 3, USENIX Association, 5.

Received January 2012; accepted January 2013