

File-Based Sharing For Dynamically Compiled Code On Dalvik Virtual Machine

Yao-Chih Huang, Yu-Sheng Chen, Wu Yang, Jean Jyh-Jiun Shann
Department of Computer Science
National Chiao Tung University
Hsinchu City 300, Taiwan
{ychuang, yusheng, wuyang, jjshann}@cs.nctu.edu.tw

Abstract—Memory footprint is considered as an important design issue for embedded systems. Sharing dynamically compiled code among virtual machines can reduce memory footprint and recompilation overhead. On the other hand, sharing writable native code may cause security problems, due to support of native function call such as Java Native Interface (JNI). We propose a native-code sharing mechanism that ensures the security for Dalvik virtual machine (VM) on the Android platform. Dynamically generated code is saved in a file and is shared with memory mapping when other VMs need the same code. Protection is granted by controlling of file writing permissions. To improve the security, we implement a daemon process, named *Query Agent*, to control all accesses to the native code and maintain all the information of traces, which are the units of the compilation in the Dalvik VM.

We implement our code sharing mechanism on Android version 2.1 system, and experiment on an arm-based system. We get 45% code-cache size reduction and 9% performance improvement from eliminating recompilation overhead.

Keywords—Virtual Machine, Android, memory footprint, JNI, JIT Compiler, code sharing.

1

I. INTRODUCTION

Embedded systems are resource-constrained environments when compared with general-purpose computers, such as personal computers (PC). Usage of memory would affect the overall efficiency of an embedded system. So, maximizing the memory efficiency and reducing the memory footprint are important issues.

The memory footprint of an embedded system consists of two parts: the footprint of applications and that of the underlying operating system. In this paper, we would focus on memory footprint reduction of applications [1].

Sharing libraries is a common way to reduce memory footprint [2], [3]. A shared library would be loaded into memory when needed and would be dynamically shared among many processes. Today, sharing libraries is a common practice in many system designs

Dalvik virtual machine, similar to Sun Hotspot virtual machine [4], would identify *hot* methods and compiles them into native code with a dynamic compiler at run time.

Generally speaking, native code that is generated by a dynamic compiler would be stored at a memory area called *code cache*. In the Android platform, every virtual machine would allocate its own code cache. Thus, native code might be duplicated in several code cache and result in wasting of memory. On the other hand, same native code may be compiled repeatedly and result in reduced performance.

So, sharing native code in the code cache among virtual machines can reduce memory footprint. This sharing approach also avoids repeated compilation of the same code and hence improve performance.

There exist two works about code sharing in VM : shMVM and MVM [5]. Both of them share class meta-data, bytecode and dynamically compiled code across multiple virtual machines. In shMVM, each virtual machine runs in a separate OS process while MVM executes many VMs at the same time within the same OS process. Both systems were derived from the Sun Hotspot virtual machine.

In our work, we only share dynamically compiled code across multiple Dalvik VMs on Android platform. Nevertheless, sharing native code may cause security problems from malicious memory modification, due to support of JNI features in Dalvik VM. Attacker can modify shared native code through JNI without restrictions. To resolve this problem, we propose a *file-based* sharing mechanism for dynamically compiled code. Native code is saved in shared file and is shared with memory mapping when other VMs need the same native code. Protection is granted by controlling of file writing permissions. To improve the security, we implement a daemon process, named *Query Agent*, to control all accesses to the native code and maintain all the information of them.

We implement our code sharing mechanism on the Android version 2.1 system, and experiment on an arm-based experiment board (named Beagleboard). Beagleboard is low-cost, fan-less single-board computer build on the Texas Instruments OMAP3530 processor [6]. In our experiment, we obtain 45% code-cache size reduction and 9% performance improvement from eliminating repeated compilation.

The rest of this paper is organized as follows : The next section gives a brief background about the Android platform, JNI and trace compilation. Section 3 presents our file-based sharing mechanism architecture. Section 4 gives

¹The work reported in this paper is partially supported by National Science Council, Taiwan, Republic of China, under grants NSC 96-2628-E-009-014-MY3, NSC 98-2220-E-009-050, NSC 98-2220-E-009-051, and 99-2219-E009-013 and a grant from Sun Microsystems OpenSparc Project.

an experimental result. Section 5 is the conclusion.

II. BACKGROUND

In this section, we will introduce Dalvik VM and Java Native Interface. In addition, we also introduce trace-based dynamic compilation which is adopted Dalvik VM.

A. Overview Android Platform And Dalvik Virtual Machine

Android is a very popular platform in mobile device market. The Android platform is the product of the Open Handset Alliance (OHA), a group of organizations collaborating to build a better mobile phone. The group is led by Google and the other companies in 2007.

Android is a complete operating environment based on the Linux kernel version 2.6. Android applications is written in the Java programming language, but Android applications can't run within Java virtual machine directly. They must execute through Dalvik VM. Dalvik VM has its own instruction set whose name is the dalvik bytecode (dex code). Every Android application has to be translated to the dex code, from java bytecode. There is a build-in tool called dx is used to translate Java class files (.class) to dex file.

With the Dalvik VM, is executed within each android application within an OS process. Thus, several Dalvik VMs will be executed simultaneously on the Android platform [7], [8], [9], [10].

B. Java Native Interface

The Java native interface (JNI) is a powerful feature of the Java language [11]. Java application could use native code written in other programming languages such as C or C++, just like code written in the Java programming language.

Before Android version 2.1, Dalvik VM only have interpreter. To improve performance of applications, Dalvik VM offers JNI feature. Application can access native library through JNI.

However, JNI may violate Java's safety feature [12], [13]. The most obvious part of the security problem come from inherently unsafe C code that can read/write memory address arbitrary. Especially, if we share writable native code across multiple virtual machines, someone with bad intentions may crash the system by modifying shared native code through JNI. Thus, we propose a file-based code cache to prevent this problem by setting of a shared file-access permissions.

C. Trace Compilation In Dalvik VM

Traditional Just-In-Time (JIT) compilers in VMs, such as Sun's Hotspot VM, are method-based, which takes individual methods as compilation units. It would detect *hot* methods and converts them to native code. However, not the entire method is worth to compile. Although a hot method often contains performance-critical parts, such as loops, but it usually also includes some slow paths and non-loop code [14], [15], [16].

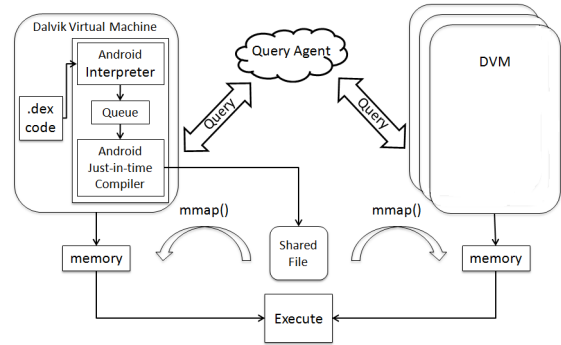


Figure 1. Architecture of the File-based Sharing Mechanism for Dynamically Compiled Code

Trace-based compilation takes a finer granularity of translation and a different hot-code detection approach. It would gather run-time statistics of the interpreted bytecode to determine hot traces.

In Dalvik VM, the trace compiler looks for chunks of bytecode instructions that are executed frequently by interpreter. It then converts these hot chunks to straight-line code and store it in the code cache. It has tight integration with the interpreter and only compiles small chunks that are important in each application [17], [18].

III. OVERVIEW FILE-BASED NATIVE CODE SHARING MECHANISM

To reduce memory footprint, native code generated by the JIT compiler is made sharable. On the other hand, this approach may raise security issues. An attacker could modify shared native code through JNI without any restriction. In order to solve this security problem, we propose a file-based sharing mechanism for dynamically compiled code.

Figure 1 shows the architecture of our code sharing mechanism. Each Dalvik VM (DVM) can save native code into a shared file after translating hot traces in dex code to native code. Other Dalvik VMs can obtain (the entry point of) the native code of the trace it needs by asking the query agent. DVM executes the shared code from a memory-mapped shared file.

In this section, we introduce *trace tags*, which are used to identify different traces, and the file-based sharing mechanism. Moreover, we implement a daemon process, named *Query Agent*, to enforce security by controlling the file-write permissions and compiled code write address.

A. Trace Tag

To search for a particular trace, we need a unique tag to identify each trace. Consider Figure 2. In Dalvik VM, the interpreter would record the address of an instruction that is a potential trace header. (A potential header is the target of a backward branch instruction.) The counter of this potential

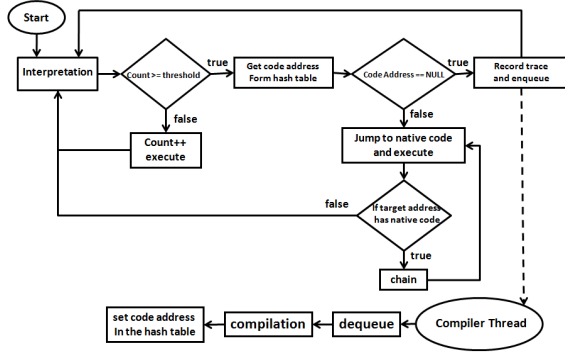


Figure 2. Dalvik VM Trace Compilation Workflow

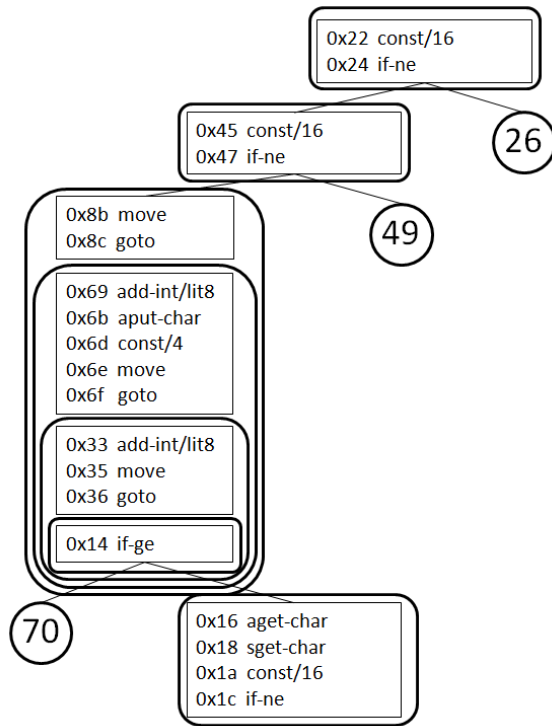


Figure 3. Example of a Trace.

header will be incremented each time it is executed. When the counter reaches a predefined threshold, the interpreter would begin to record a trace that starts from the potential trace header and ends at a control-flow instruction, such as *if*. JIT would be notified after the interpreter enqueues the trace. JIT translates the trace into native code. Finally, it would update the hash table according to the trace header's address.

Figure 3 is an example of trace construction in Dalvik VM for the *fixSlashes* method in the *File* class. The number in a circle is the offset of the first instruction of a trace (not shown here) in the *fixSlashes* method. A round rectangle represents a trace. A trace may contain one or more basic

```
TRACEINFO (2): 0x4177b594 Ljava/io/File;fixSlashes
Compiler: Building trace for fixSlashes, offset 0x16
0x0049 aget-char
0x0065 sget-char
0x0013 const/16
0x0033 if-ne
```

Trace Tag : **Ljava/io/File;fixSlashes:49:65:13:33,0x16**
 After Hash function : **3183876070, 0x16**

Figure 4. Example of Trace Tag.

blocks. As Figure 3 shows, every trace ends at a control-flow instruction, and different traces would start from different starting addresses even though they may contains other traces.

In other words, the original Dalvik VM identifies a trace by it's trace header address. However, a trace header's addresses may be different in the different Dalvik VMs. Thus, we must design a new tag that uniquely identifies a trace across all different virtual machines.

In this work, We use the combination of the string that represents the signature of the method which the trace belongs to, every opcode in this trace, and the offset of the trace header in the method as the tag of a trace. Consider the simple trace in Figure 4. This trace belongs to the *fixSlashes* method and the trace header's offset in the method head is 0x16. Then we use the pair *Ljava/io/File;fixSlashes:49:65:13:33* and *0x16* as the tag of this trace. Nevertheless, in order to reduce memory usage and the number of string comparison operations, we will hash the signature of the method and opcodes of this trace into a number. So, we take the pair *3183876070* and *0x16* as the tag in our system.

B. File Permission and Query Agent

To avoid malicious and illegal memory modification, we protect the shared native code with file-access permissions. The writing permission is enabled only before storing native code. Any file system allowing control writing permission can be used in our design.

To improve the security, we create a daemon program *Query Agent*. Query Agent maintains a global hash table that contains information about all traces, such as offset of native code in the shared file, trace tags, and the instruction set that is used. The writing permissions of stored files are also controlled by Query Agent.

When a Dalvik VM compiles a piece of code, it asks the Query Agent to allocate a starting address for the generated native code. A Dalvik VM also asks the Query Agent for the offset of the native code that are compiled by other VMs. With this approach, we not only control write accesses to the shared file but also reduce redundant trace compilations.

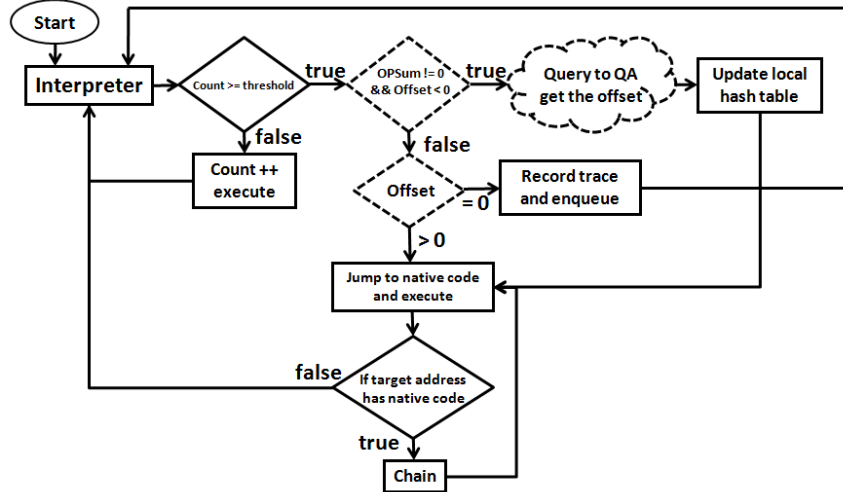


Figure 5. Work-flow of the File-based Native-Code Sharing Mechanism (Interpreter).

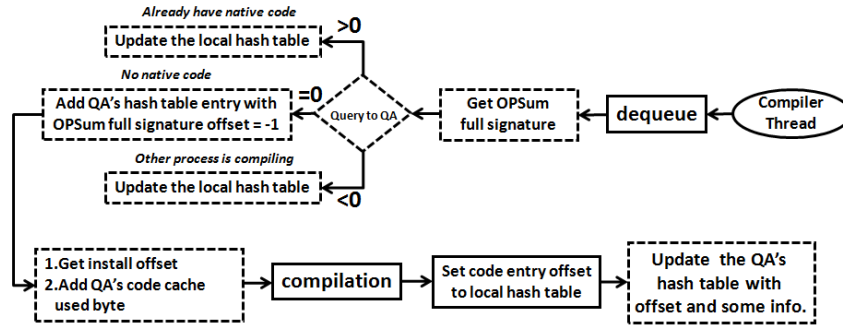


Figure 6. Work-flow of the File-based Native-Code Sharing Mechanism (Just-In-Time Compiler).

C. Work-flow of the File-Based Native-Code Sharing Mechanism

Figure 5 and Figure 6 show the work-flow of the file-based native-code sharing mechanism. Before each compilation, the compiler thread asks Query Agent to check whether it already has the native code of the trace. According to Query Agent's reply, there are three cases:

- i. There is compiled native code for the trace we query. In this case, we do not need to compile the trace. Instead, the hash table of this Dalvik VM records related trace information. In this case, Query Agent returns the native code's entry offset.
- ii. Some other VM is compiling the same trace currently. In this case, we do not need to compile the trace. Instead, in the local hash table, the offset of the trace is -1 temporarily. Later, the actual offset of the trace will be recorded in the local hash table.
- iii. This trace has not been compiled before. In this case, JIT asks the Query Agent to allocate an installation offset and compile the trace. Dalvik VM records in

the hash table and the global hash table information about the trace, including the trace tag, header size, instruction set and installed offset of native code after compilation.

Since in the Android system, several Dalvik VMs may run concurrently, synchronization among multiple VMs must be considered. Sometimes, several VMs may happen to request to compile the same trace. The first requester will compile the trace.

In addition to compiled native code, the shared file also includes some VM helpers (called *template code*). As Figure 7 shows, the template code is placed on the top of the code cache so that it can be invoked with a single instruction. The trace compiler will generate a compilation layout for each trace. A compilation layout contains native code body, chaining cells, trace description and the literal pool. Since every trace ends at a control-flow instruction, it must jump to some point. The chaining cell data structure stores the target of the last branch instruction in a trace.

Currently, we use a single shared file to store native code of all traces, because it is simpler to manage. Multiple shared

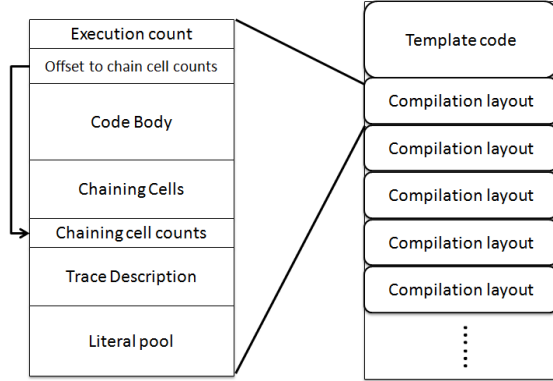


Figure 7. Contents of the Code Cache and a Compilation Layout of Dalvik VM.

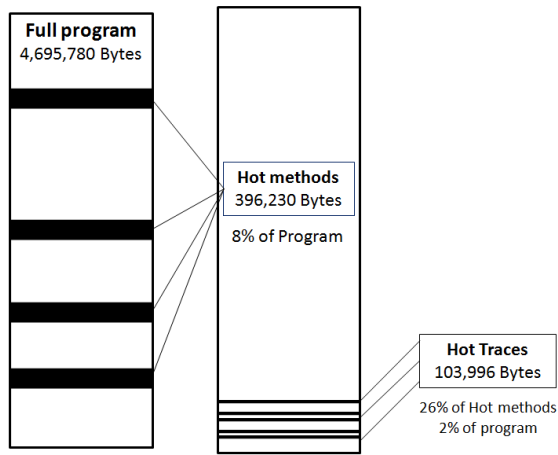


Figure 8. The Portion of Traces in the system_server.

files may cause external fragmentation due to the memory mapping requirement.

Our approach is similar to the Ahead-Of-Time (AOT) compiler. Actually, this design can support both an AOT compiler and a JIT compiler. We choose to use a JIT compiler to implement our mechanism in this work. In addition, Android version 2.1 already has a JIT compiler, which can help us to implement our design. On the other hand, the JIT compiler just keeps the hot portion of the class library in the code cache, and it guarantees all the code in the code cache is frequently used. Figure 8 presents the proportion of traces in the *system_server*. As the figure shows, the total size of all traces is just 2% of the whole program. Although the JIT compiler compiles the traces (into native code) at run time, but the cost is negligible in the long term. If we choose AOT compiler, we have to use shared native code through JNI. It would cause a lot of overhead.

In general, we have to consider the relocation issues of native code after mapping. However, we found the code

generated by the Dalvik JIT compiler is already position independent. So, we do not need to handle relocation issues.

IV. EXPERIMENTAL RESULTS

We merge our mechanism into Oxdroid, which is the Android version ported to Beagleboard by Oxlab company [19]. Performance and memory usage are evaluated for our design.

Application Name	Description
rowboat	a set of performance test applications for the Android Platform
Benchmark	Tool measure phone performance
MultiSearch	Search simultaneously on all selected engines
Frozen Bubble	Frozen Bubble game
Bs09lite	Baseball game
Replica Island	Replica Island game
Tossit	Tossit game
Air Attack	Air attack game
Jewels 1.6	Jewels game v1.6
SoccerUnleashed	Soccer game

Figure 9. List of Applications.

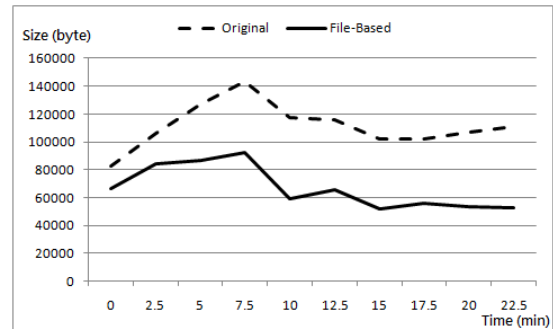


Figure 10. Total Code Cache Size During Execution Time.

Ten applications are chosen from the Android market to evaluate memory saving of our design. They consist of two performance measurement tools, one data searching utility, and seven entertainment games. Detail Information about these applications is shown in Figure 9.

All programs are executed automatically by using SIKULI script tool [20]. SIKULI will start one application every 2.5 minutes. They are executed in the order they are placed in Figure 9. Some of them would be terminated by the Android operating system when memory is not enough to execute the next application. In our current system, at most four applications can be run simultaneously. Figure 10 shows change of total code cache size during execution of the experiment. As this figure shows, at each point of time, measured code cache size under our mechanism is smaller than under the original design. There are two reasons: (1)

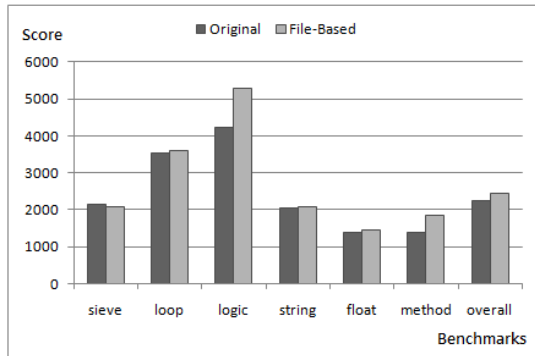


Figure 11. CaffeineMark Results.

some applications can utilize the native code generated by previous applications since they use the same system libraries, and (2) we reused the template code among all instances of Dalvik VM. In summary, code cache size is reduced by 45% in this experiment.

For performance evaluation, we use the CaffeineMark benchmark to evaluate performance. We execute CaffeineMark ten times to calculate the average score. 9% improvement on the overall score is obtained from the reduction of recompilations. Except the first iteration, all other iterations can benefit from reusing the native code generated. As Figure 11 shows, *logic* got the best performance improvement, because *logic* constructs the most traces in the CaffeineMark benchmark. *logic* has more opportunity to eliminate repeated compilations than other benchmarks. Thus it can get the highest reductions. However, *sieve* presents a little decrease because it does not construct enough traces to amortize the overhead of our design.

V. CONCLUSION

In this paper, we present a native-code sharing mechanism. We share native code generated by the Just-In-Time compiler at runtime across multiple virtual machines. To avoid malicious and illegal memory modification, we propose file-based code cache to share native code by memory mapping and protect native code by controlling file-access permissions. To enforce file permissions and avoid repeated compilation of the same trace, we implement a daemon process, named Query Agent. At last, we get 45% code cache size reduction from native-code sharing and 9% performance improvement from avoiding of repeated recompilations of the same code.

REFERENCES

- [1] Koen De Bosschere, *Memory footprint reduction for embedded systems*, In Proceedings of the 11th international workshop on Software & compilers for embedded systems, pp 31-31, Munich, Germany, 2008.
- [2] D. M. D. M. Beazley, B. D. Ward, and I. R. Cooke, *The Inside Story on Shared Libraries and Dynamic Loading*, IEEE Computing in Science & Engineering Vol. 3, Issue 5, pp 90-97, September/October 2001.
- [3] Jaesoo Lee, Jiyong Park, Seongsoo Hong, *Memory Footprint Reduction with Quasi-Static Shared Libraries in MMU-less Embedded Systems*, In Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 24-36, April 04-07, 2006.
- [4] Sun Microsystems Inc, *The Java HotSpot Virtual Machine Technical White Paper*, 2001
- [5] Grzegorz Czajkowski, Laurent Daynes, Nathaniel Nystrom, *Code Sharing among Virtual Machines*, In Proceedings of the 16th European Conference on Object-Oriented Programming, pp. 155-177, Malaga, Spain, June 2002.
- [6] Beagleboard.org, "Beagleboard.org - about". [online]. Available: <http://beagleboard.org/about>.
- [7] Google Inc, "Android.com." [online]. Available: <http://www.android.com/>.
- [8] IBM Inc, "Introduction to Android development." [online]. Available: <http://www.ibm.com/developerworks/opensource/library/os-android-devel/>.
- [9] Google Inc, "Dalvik Porting Guide." [online]. Available: <http://android.git.kernel.org/>.
- [10] Wiki, "Dalvik(software)." [online]. Available: [http://en.wikipedia.org/wiki/Dalvik_\(software\)](http://en.wikipedia.org/wiki/Dalvik_(software)).
- [11] Sheng Liang, *Java Native Interface: Programmer's Guide and Specification*, Addison-Wesley, ISBN:0201325772, June 20, 1999
- [12] Gang Tan, Srimat Chakradhar, Raghunathan Srivaths, Ravi Daniel Wang, *Safe Java Native Interface*, In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering, pp 97-106, March, 2006
- [13] Michael Furr, Jeffrey S. Foster, *Checking type safety of foreign function calls*, In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp 62-72, Chicago, IL, USA, 2005
- [14] Christian W. Probst, Andreas Gal, Michael Franz, *HotpathVM: An effective JIT compiler for resource-constrained devices*, International Conference on Virtual Execution Environments, pp.144-153, Ottawa, Ontario, Canada, 2006, ACM Press.
- [15] A. Gal and M. Franz. *Incremental dynamic code generation with trace trees*, Technical Report ICS-TR-06-16, University of California, Irvine, Nov. 2006.
- [16] Andreas Gal, "Tracing The Web." [online]. Available: <http://andreasgal.wordpress.com/>.
- [17] Dalvik Virtual Machine, "Dalvik Virtual Machine." [online]. Available: <http://www.dalvikvm.com/>.
- [18] Google Inc, "Google I/O 2010." [online]. Available: <http://code.google.com/intl/zh-TW/events/io/2010/>.
- [19] Oxlab, "Oxdroid project." [online]. Available: <http://code.google.com/p/Oxdroid/>.
- [20] Massachusetts Institute of Technology, "Project SIKULI." [online]. Available: <http://groups.csail.mit.edu/uid/sikuli/>.