

國立交通大學

資訊科學與工程研究所

碩 士 論 文

在 Dalvik 虛擬機器上對動態編譯碼的檔案
基礎式共享

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

研 究 生：黃曜志

指 導 教 授：楊 武 教 授

中 華 民 國 九 十 九 年 九 月

在 Dalvik 虛擬機器上對動態編譯碼的檔案
基礎式共享

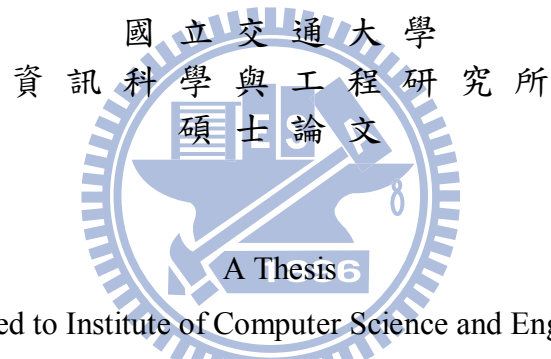
File-Based Sharing For Dynamically Compiled Code On
Dalvik Virtual Machine

研 究 生：黃曜志

Student : Yao-Chih Huang

指 導 教 授：楊 武

Advisor : Dr. Wu Yang



Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science and Engineering

September 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年九月

在 Dalvik 虛擬機器上對動態編譯碼的檔案 基礎式共享

學生：黃曜志

指導教授：楊 武 博士

國立交通大學資訊科學與工程研究所

摘 要

在嵌入式系統的環境中，記憶體的使用量被視為是一個重要的議題。共享不同虛擬機器之間的動態編譯碼，可以減少重複編譯所產生的記憶體使用量以及改善效能。另一方面，因為 Dalvik 虛擬機器提供了 Java Native Interface(JNI)，所以共享可寫入的原生碼可能會造成安全性的問題。為此，我們提出一個對於 Dalvik 虛擬機器在 Android 平台上可以確保安全性的原生碼共享機制。動態產生的原生碼會被存在檔案中，當其他的虛擬機器需要相同的原生碼時，可以透過記憶體映射的方式來分享原生碼。並且藉由控制檔案權限，來確保安全性。為了改善安全性，我們實作了一個取名為 Query agent 的 daemon process，用來控制所有原生碼的存取及保存所有原生碼相關的資訊。我們實作原生碼的共享機制在 Android 2.1 的版本上，並且在 arm 基礎的系統上做實驗。我們獲得了 45% 程式碼快取大小的縮減，並且藉由消除重複編譯的負擔來取得 9% 的效能改善。

關鍵字：Dalvik 虛擬機器、Android、記憶體使用量、JNI、Trace-Based JIT Compiler、檔案基礎分享、程式碼分享

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

Student: Yao-Chih Huang

Advisor: Dr. Wu Yang

Institute of Computer Science and Engineering

National Chiao Tung University

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 JNI. We propose a native-code sharing mechanism that ensures the security for Dalvik virtual machine 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 file-access 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 : Dalvik Virtual Machine, Android, memory footprint, JNI, Trace-Based JIT Compiler, File-Based sharing, code sharing.

誌 謝

首先感謝我的指導老師楊武教授在這兩年來的指導，協助我一步一步的完成我的論文。老師耐心以及幽默的指導方式使我在這兩年之間受益良多，讓我得以不斷的嘗試、尋找我的研究方向，並且發現思考邏輯上的盲點。也要感謝徐慰中教授、單智君教授、以及游逸平教授，在每次 meeting 的時候，都能夠給予我的研究非常中肯的建議，讓我能夠不斷的改進，不斷的修正我的研究方向。同樣也要感謝口試委員雍忠教授，以及游坤明教授，在口試時給予指教，使這篇論文更完整。

同時也要感謝裕生學長以及柏擘學長在論文上給予我莫大的幫助，在我感到無助徬徨的時候幫助我解決問題，讓我能夠走到現在，並且完成我的論文。同時也要感謝 PLASLAB 的成員，不管是已經畢業的帥維、禮君、耀崙、冠旭、有倫、阿發，還是信慶、致超、韋任以及各位學弟妹，因為有各位，才會讓這兩年的研究生生活，除了研究之外還能夠有各式各樣的酸甜苦辣。

最後要感謝我的父母以及外婆，是他們一直在背後支持我，讓我專心的做研究，並且完成我的論文。

The 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.

Table of Contents

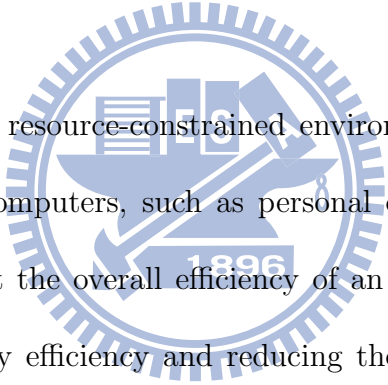
摘要.....	iii
Abstract	iv
誌謝.....	v
Table of Contents	vi
List of Figures	vii
Chapter 1 Introduction	1
Chapter 2 Background	5
2.1 Overview of Android Platform and Dalvik Virtual Machine	5
2.2 Java Native Interface	7
2.3 Trace Compilation In Dalvik VM	7
Chapter 3 Overview of File-Based Native Code Sharing Mechanism	9
3.1 Trace Tag	10
3.2 File Permission and QueryAgent	14
3.3 Workflow of the File-Based Native Code Sharing Mechanism	15
Chapter 4 Experimental Results	24
Chapter 5 Future work	28
Chapter 6 Conclusion	30
Bibliography	31

List of Figures

Figure 3.1 Architecture of File-based Sharing Mechanism for Dynamically Compiled Code	10
Figure 3.2 Dalvik VM Trace Compilation Workflow	11
Figure 3.3 Example of Trace	13
Figure 3.4 Example of Trace Tag.....	14
Figure 3.5 File-based Native Code Sharing Mechanism Workflow (Interpreter)	16
Figure 3.6: File-based Native Code Sharing Mechanism Workflow (Just-In-Time Compiler).....	17
Figure 3.7 Compilation Layout And Content Of Code Cache.....	19
Figure 3.8 The Proportion Of Trace In The system server.....	20
Figure 3.9 Memory Map Same Native Code To Different Address.	21
Figure 3.10 Hash Collision Handle.....	22
Figure 4.1 Application List	25
Figure 4.2 Code Cache Size Result.	25
Figure 4.3 Caffeinemark Result.....	26

Chapter 1

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, the shared library 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.

FX!32 is a profile-directed binary translator [6]. It has two common techniques : emulation and binary translation. The first time an x86 application runs, the emulator generates an execution profile data. After the application

exits, the background optimizer would generate the native code according to the profile data. The profile data and native code are saved on disk. This is similar to our approach, but the difference is that FX!32 can't share the native code.

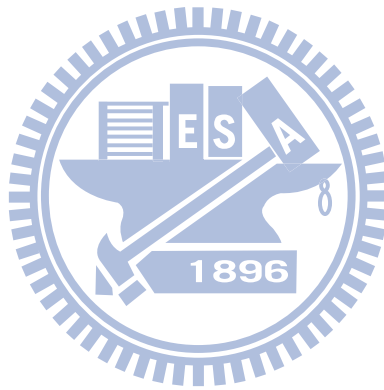
In our work, we only share dynamically compiled code across multiple Dalvik VMs on Android platform. Nevertheless, sharing native code may cause security problems from malicious memory modification, due to support of JNI features in Dalvik VM. Attacker can modify shared native code through JNI without 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 file-access 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. Note that traces are the units of compilation in the Dalvik VM.

We implement our code sharing mechanism on the Android version 2.1 system, and experiment on an arm-based experiment board (named beagle-board). Beagleboard is low-cost, fan-less single-board computer build on the Texas Instruments OMAP3530 processor. OMAP3530 uses ARM Cortex-A8 as application processor [7]. Android version 2.1 comes with a trace-based

just-in-time (JIT) compiler. In our experiment, we obtain 45% code-cache size reduction and 9% performance improvement from eliminating repeated compilation.

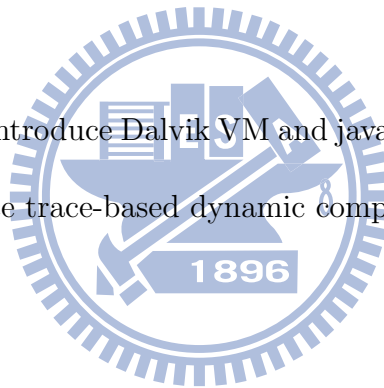
The rest of this paper is organized as follows : The next section gives a brief background about the Android platform, JNI and trace compilation. Section 3 presents our file-based sharing mechanism architecture. Section 4 gives an experimental result. Section 5 is the conclusion.



Chapter 2

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.



2.1 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 ker-

nel version 2.6. It takes Apache Harmony Java class library as core library. 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. Many classes are covered in a single dex file. Duplicate strings and superfluous constants used in multiple class would be contained only once in dex file to save storage.

Dalvik VM is not similar to Java VM and most virtual machine we know is stack-based machine. Dalvik VM is register-based machine. It can help product better performance than stack-based machine. For example, register-based machine can eliminate redundant memory operation like push and pop.

Dalvik VM implements simple garbage collection mechanism (mark-sweep). It can't translate the class to another class type arbitrary. 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 [8, 9, 10, 11].

2.2 Java Native Interface

The Java native interface (JNI) is a powerful feature of the Java language [12]. 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 [13, 14]. 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.

2.3 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,

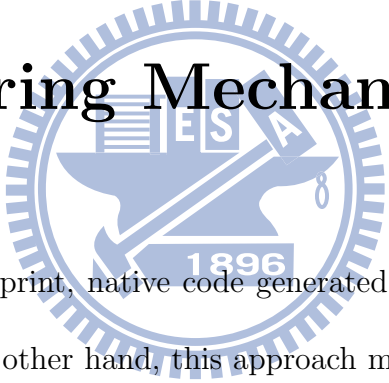
not the entire method is worth to compile. Although a hot method often contains performance-critical parts, such as loops, but it usually also includes some slow paths and non-loop code [15, 16, 17]. Nevertheless, method-based compiler always analyze and compile entire method, even it only has some worth compiling parts.

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. When certain taken of backward branch occur frequently, it means that might have a loop and a loop header would be identified. Once it identifies such loop header, it would start to follow execution of the interpreter to record the sequence of executed bytecode instructions. The record would terminate when the interpreter return to the loop start point eventually since it begin with the loop start point. The resulting sequence of bytecode instructions is what we called a trace. In other words, traces represent a single iteration through a loop, and trace guarantees be the hot code in the entire program.

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 [18, 19].

Chapter 3

Overview of 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 3.1 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.

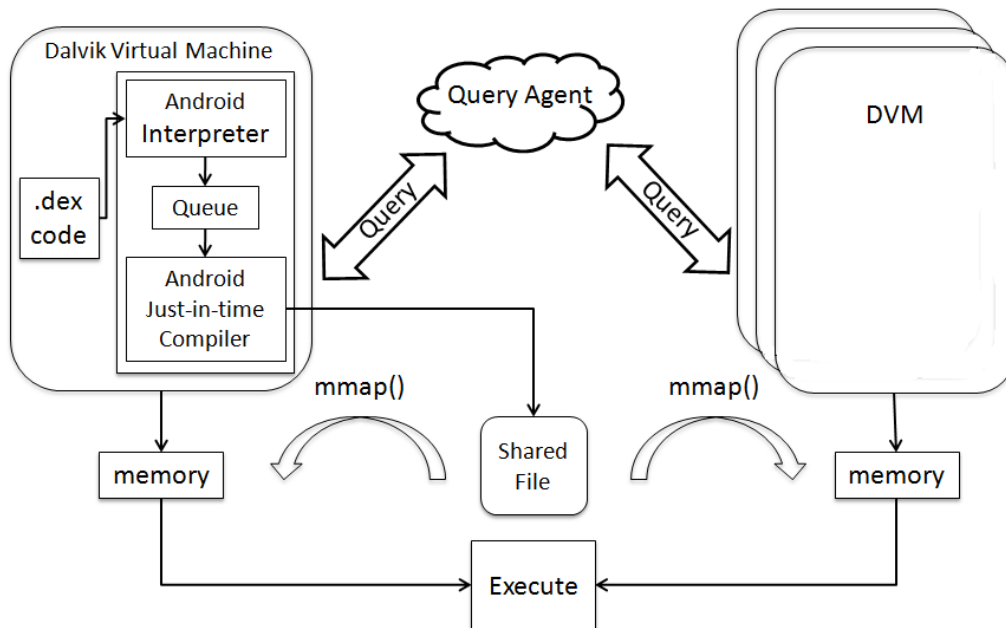


Figure 3.1: Architecture of File-based Sharing Mechanism for Dynamically Compiled Code.

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.

3.1 Trace Tag

To search for a particular trace, we need a unique tag to identify each trace.

Consider Figure 3.2. In Dalvik VM, the interpreter would record the ad-

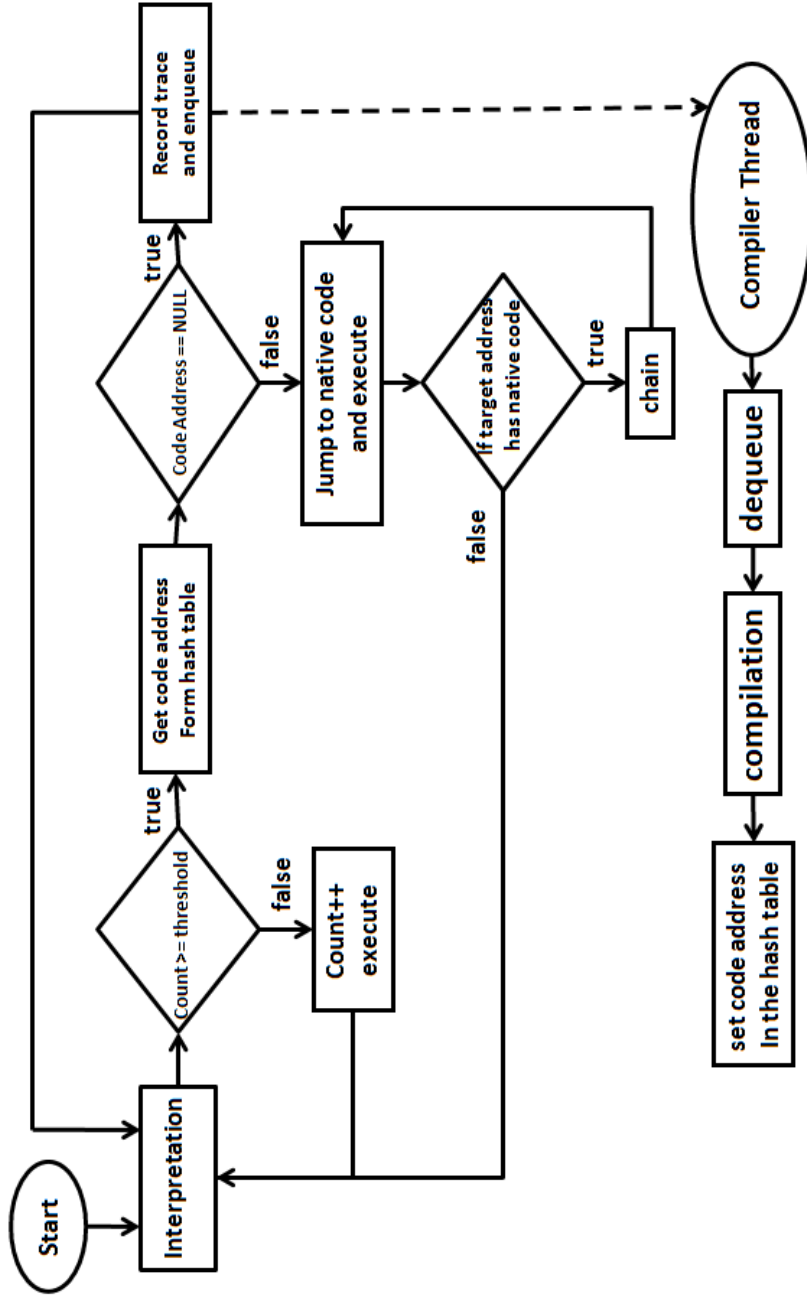


Figure 3.2: Dalvik VM Trace Compilation Workflow.

dress 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 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 3.3 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 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 contain other traces.

In other words, the original Dalvik VM identifies a trace by its 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

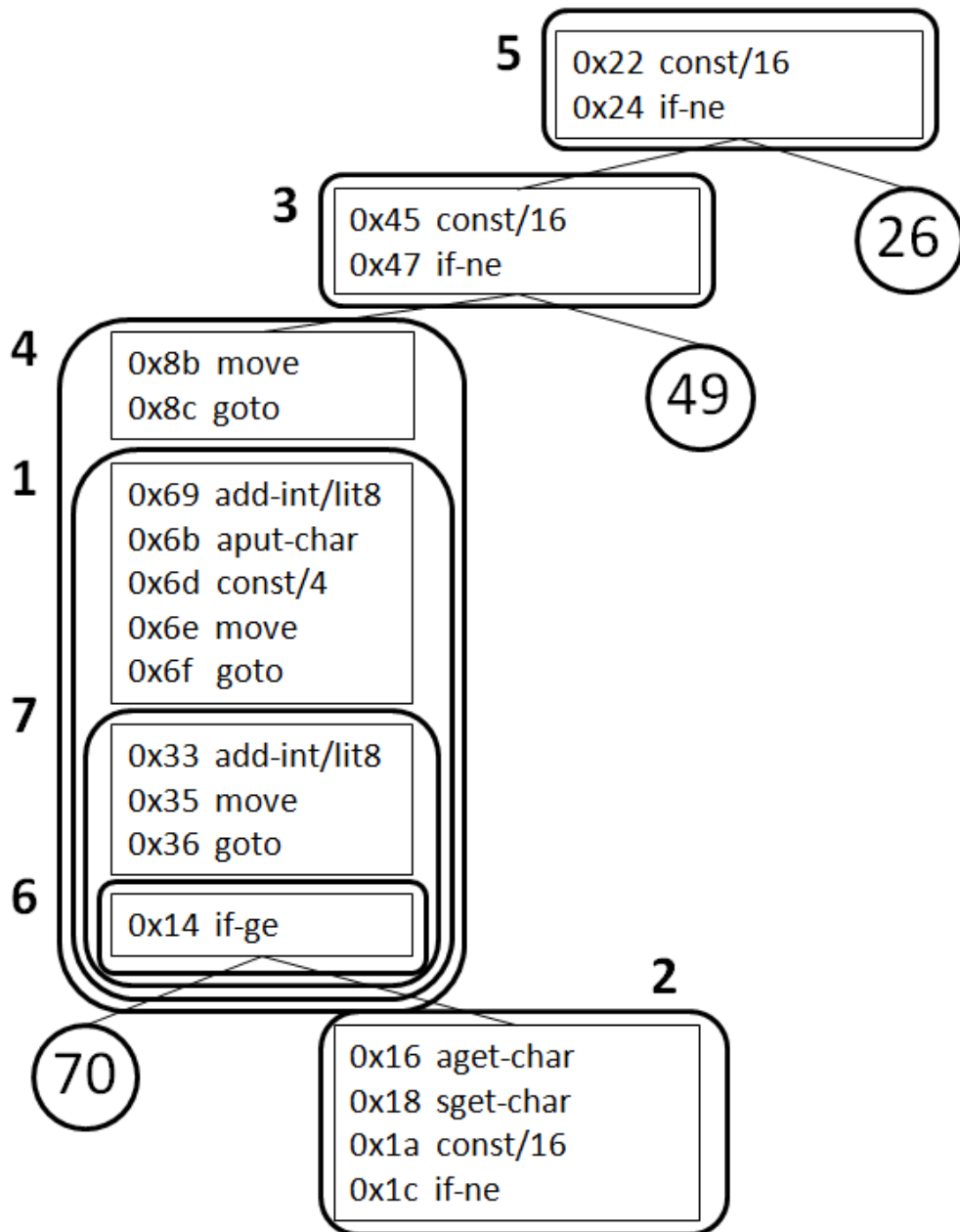


Figure 3.3: Example of Trace.

```
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 3.4: Example of Trace Tag.

trace, and the offset of the trace header in the method as the tag of a trace. Consider the simple trace in Figure 3.4. 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 first part into a number. So, we take the pair *3183876070* and *0x16* as the tag in our system.

3.2 File Permission and Query Agent

To avoid malicious and illegal memory modification, we protect the shared native code with file-access permissions. The write permission is enabled only before storing native code.

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 file-access permissions are 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 accesses to the shared file but also reduce redundant trace compilations.

3.3 Workflow of the File-Based Native-Code Sharing Mechanism

Figure 3.5 and Figure 3.6 show the workflow 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

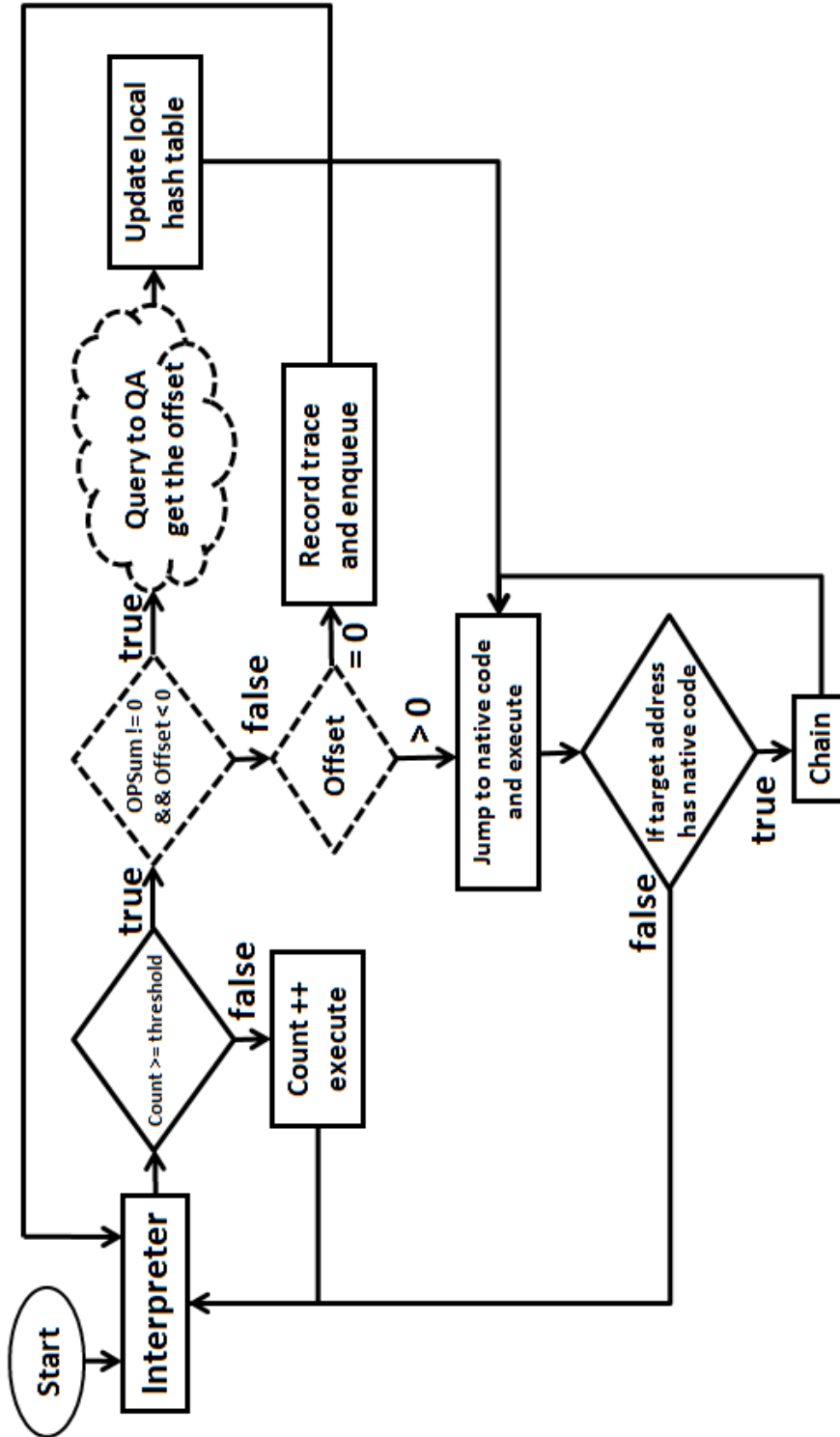


Figure 3.5: File-based Native Code Sharing Mechanism Workflow (Interpreter).

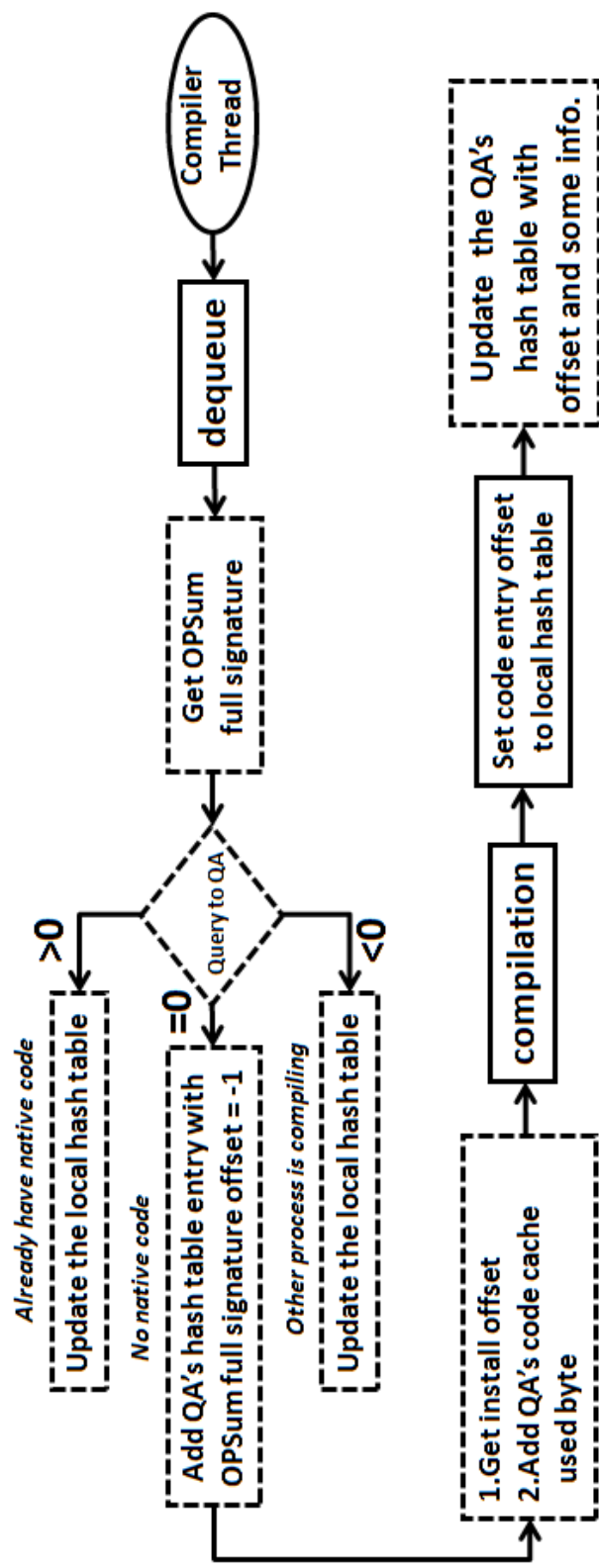


Figure 3.6: File-based Native Code Sharing Mechanism Workflow (Just-In-Time Compiler).

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 3.7 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

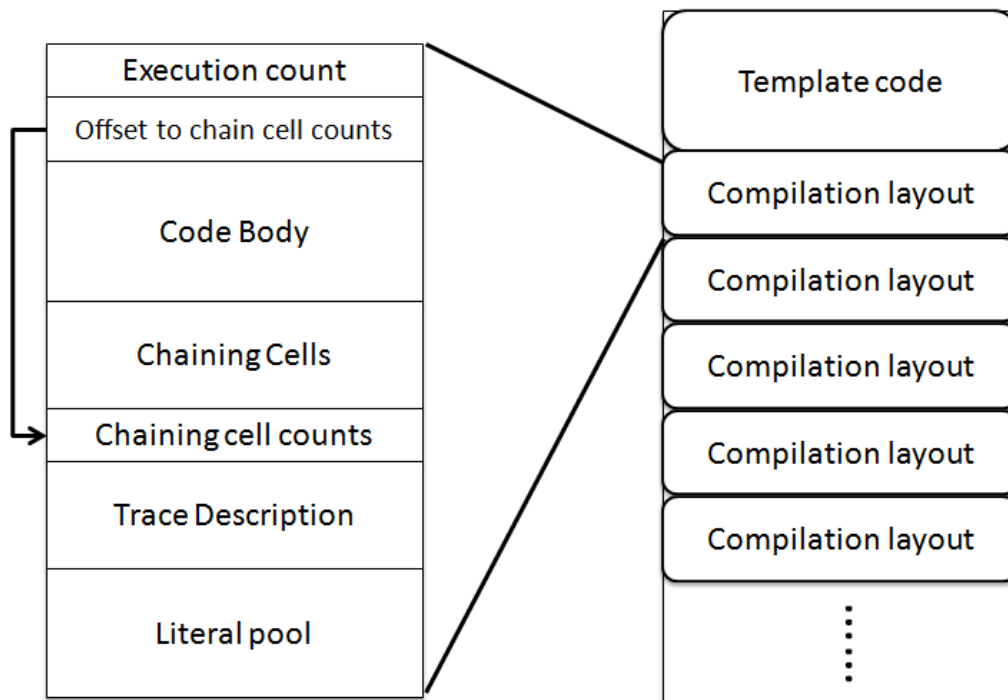


Figure 3.7: Compilation Layout And Content Of Code Cache.

description and the literal pool. Since every trace ends at a control-flow instruction, it must jump to some point. The chaining cell 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 files may cause internal 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

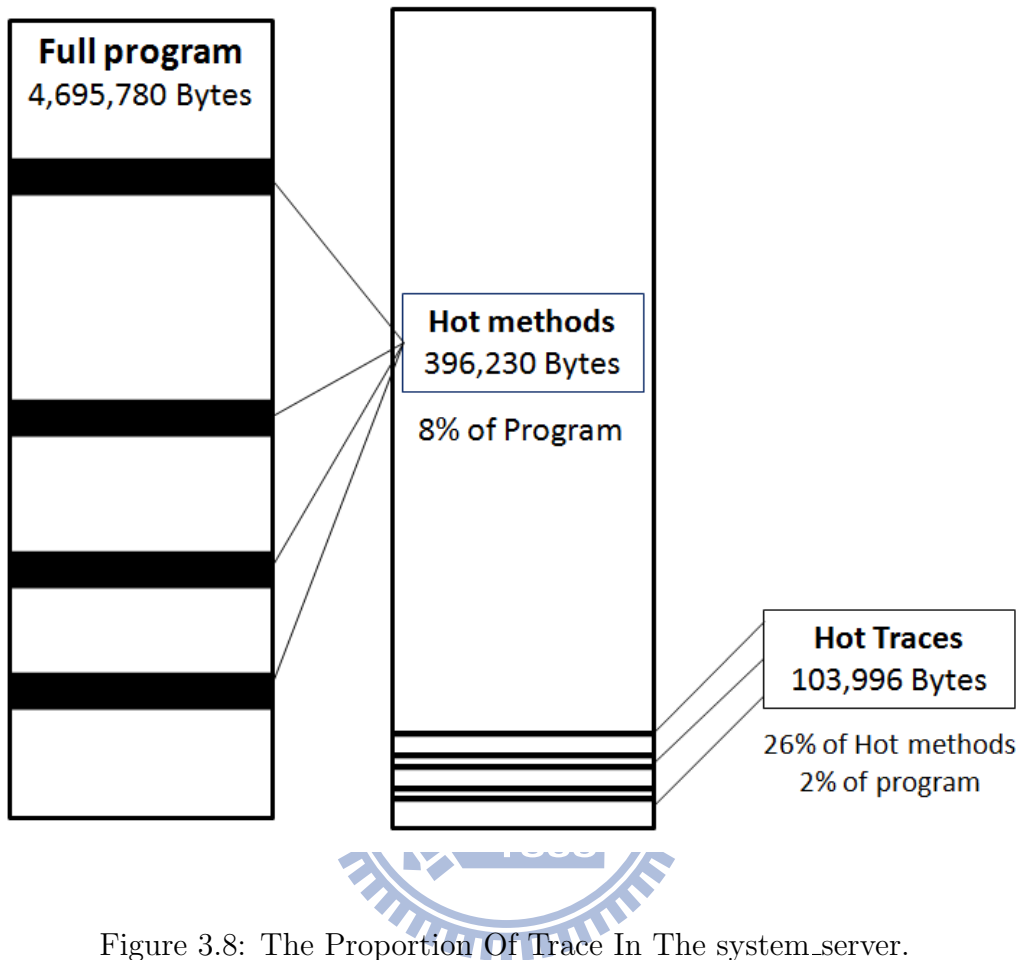


Figure 3.8: The Proportion Of Trace In The `system_server`.

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 3.8 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

Before unmap : codeCache : 0x41b97000	After unmap : codeCache 0x41b98000
0x41b9745f: ldr r0, [r5, #12]	0x41b9845f: ldr r0, [r5, #12]
0x41b97461: movs r1, #28	0x41b98461: movs r1, #28
0x41b97463: ldr r2, [r5, #16]	0x41b98463: ldr r2, [r5, #16]
0x41b97465: cmp r0, #0	0x41b98465: cmp r0, #0
0x41b97467: beq.n 0x41b9747a	0x41b98467: beq.n 0x41b9847a
0x41b97469: str r2, [r0, r1]	0x41b98469: str r2, [r0, r1]
0x41b9746b: ldr r3, [r5, #12]	0x41b9846b: ldr r3, [r5, #12]
0x41b9746d: movs r0, #24	0x41b9846d: movs r0, #24
0x41b9746f: ldr r3, [r3, r0]	0x41b9846f: ldr r3, [r3, r0]
0x41b97471: ldr r0, [r5, #8]	0x41b98471: ldr r0, [r5, #8]
0x41b97473: cmp r3, r0	0x41b98473: cmp r3, r0
0x41b97475: str r3, [r5, #0]	0x41b98475: str r3, [r5, #0]
0x41b97477: beq.n 0x41b97480	0x41b98477: beq.n 0x41b98480

Figure 3.9: Memory Map Same Native Code To Different Address

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. We map same native code to different address, as figure 3.9 shown, every address of branch target with the mapping address changed. Actually, the address of branch target would be computed at runtime. So, we think the code generated by Dalvik JIT compiler is position independent. Thus, we do not need to handle relocation issues.

With the execution of system, code cache would fill up eventually. In the Android version 2.1, Dalvik VM would terminate the compile request until

VM restart. But in the Android version 2.2 (froyo), Dalvik VM would flush whole code cache. Our work was implemented at Android version 2.1, thus we follow the version 2.1's approach.

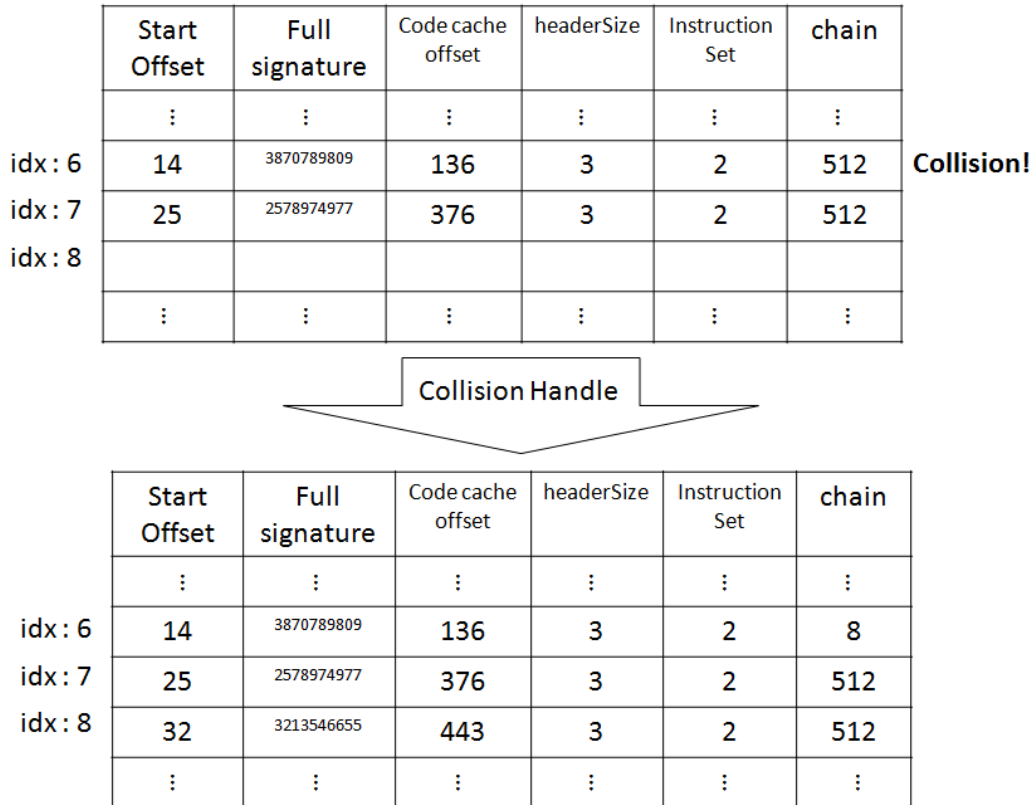


Figure 3.10: Hash Collision Handle

To maintain the information of native code, we use many hash tables, but hash table is hard to avoid collision. We would find the first empty entry after collision entry when collision happened and record the first empty entry number. Figure 3.10 is an example of collision handle. When collision happened at index 6, we will find the first empty entry at index 8, and record

index 8 at index 6's chain field.



Chapter 4

Experimental Results

We merge our mechanism into Oxdroid, which is the Android version ported to Beagleboard by Oxlabs [20]. Performance and memory usage are evaluated for our design.

Ten applications are chosen from the Android market and executed automatically using SIKULI scripts [21]. The ten applications consist of two benchmarks to test CPU performance on the Android platform, one application for searching data from many search engines, and seven interesting games. Information about applications is shown in Figure 4.1.

We start one application every 2.5 minutes. Some applications would be terminated by the Activity Manager in Android when memory space is not enough to execute the next application. In our experiment, at most four applications run simultaneously in the system. Figure 4.2 shows the change

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 4.1: Application List.

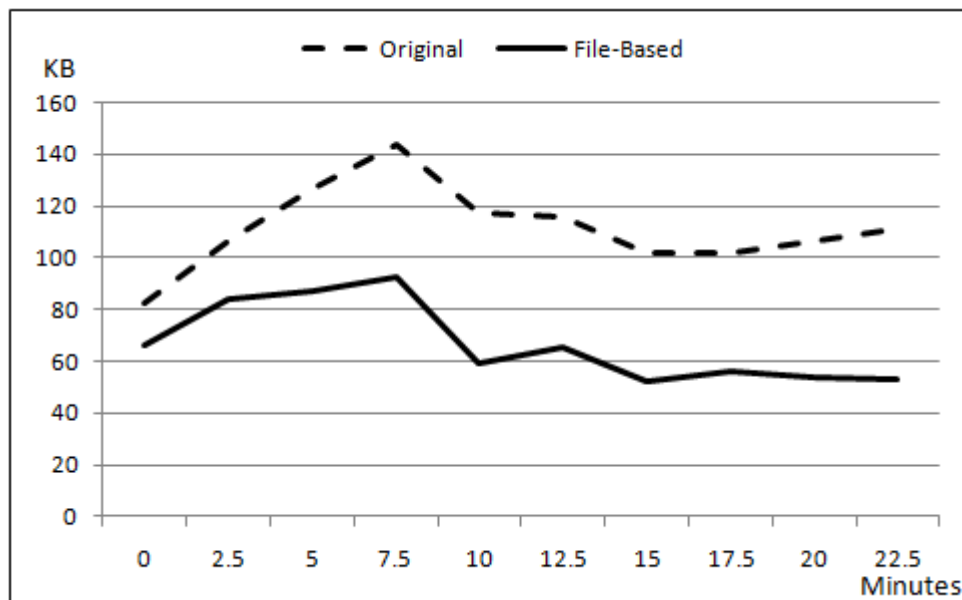


Figure 4.2: Code Cache Size Result.

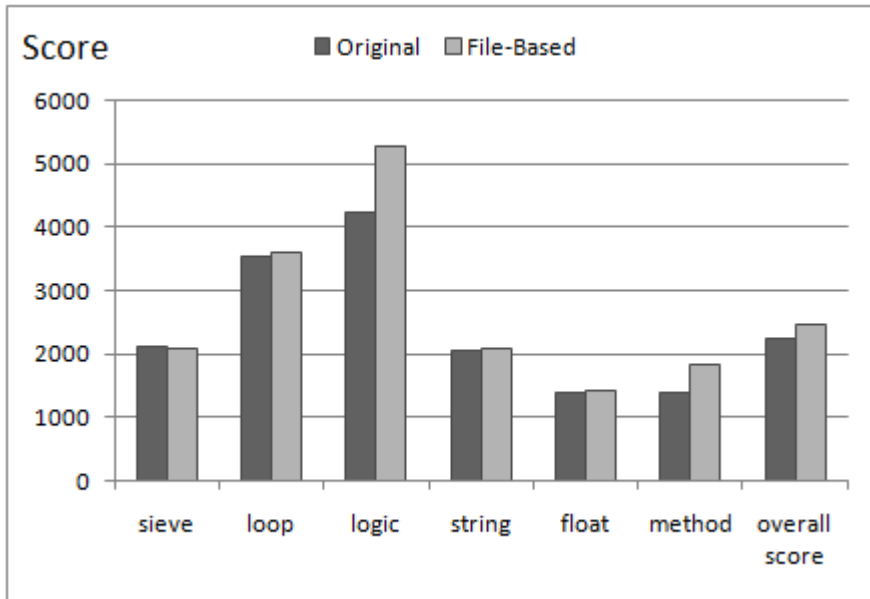


Figure 4.3: CaffeineMark Result.

of total code cache size during our experiment. As the figure shows, the total code cache size under our mechanism is smaller than under the original model. There are two reasons: (1) all applications can utilize the native code generated by previous applications; and (2) we reuse the template code for every Dalvik VM. In summary, code cache size is reduced by 45% in our experiment.

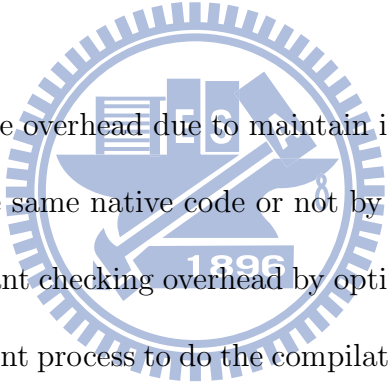
For performance evaluation, we use the CaffeineMark benchmark to evaluate performance. We execute CaffeineMark ten times to calculate the average score. 9% improvement on the overall score is obtained from the reduction of recompilations. Except the first iteration, all other iterations can benefit

from reusing the native code generated. As Figure 4.3 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.



Chapter 5

Future Work



Our work still have some overhead due to maintain information of the trace, and check whether have same native code or not by asking Query Agent. It could eliminate redundant checking overhead by optimize whole workflow, or make a single independent process to do the compilation of the whole system.

Currently, we delete the shared file when the Android system shutdown. We could reserve the shard file and related information of trace within Query Agent, and reuse it to make start up time of the Android system faster than original system.

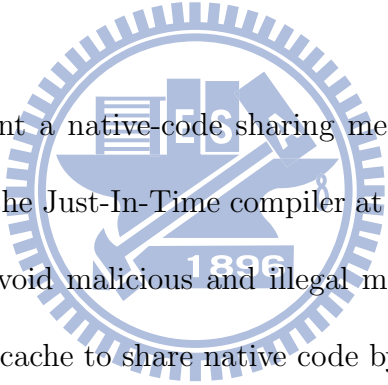
In Android, to reduce power consumption and prevention malicious attack from network, there have a restriction which every application if it need network ability, it should register toward operating system. To protect the security further and reduce duplicate native code, we apply daemon process

and communicate by socket in our mechanism. So, if we want to make every application can communicate with daemon process, we have to open the network ability for every application even this application doesn't need the network ability. However this way would make the whole system under exposure to dangerous.



Chapter 6

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 recompilaions of the same code.

Bibliography

- [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] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, John Yates, *FX!32: A Profile-Directed Binary Translator*, IEEE Micro, pp. 56-64, Mar/Apr, 1998
- [7] Beagleboard.org, "Beagleboard.org - about". [online]. Available: <http://beagleboard.org/about>.
- [8] Google Inc, "Android.com." [online]. Available: <http://www.android.com/>.
- [9] IBM Inc, "Introduction to Android development." [online]. Available: <http://www.ibm.com/developerworks/opensource/library/os-android-devel/>.

- [10] Google Inc, "Dalvik Porting Guide." [online]. Available: <http://android.git.kernel.org/>.
- [11] Wiki, "Dalvik(software)." [online]. Available: [http://en.wikipedia.org/wiki/Dalvik_\(software\)](http://en.wikipedia.org/wiki/Dalvik_(software)).
- [12] Sheng Liang, *Java Native Interface: Programmer's Guide and Specification*, Addison-Wesley, ISBN:0201325772, June 20, 1999
- [13] 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
- [14] 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
- [15] 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.
- [16] A. Gal and M. Franz. *Incremental dynamic code generation with trace trees*, Technical Report ICS-TR-06-16, University of California, Irvine, Nov. 2006.
- [17] Andreas Gal, "Tracing The Web." [online]. Available: <http://andreasgal.wordpress.com/>.
- [18] Dalvik Virtual Machine, "Dalvik Virtual Machine." [online]. Available: <http://www.dalvikvm.com/>.
- [19] Google Inc, "Google I/O 2010." [online]. Available: <http://code.google.com/intl/zh-TW/events/io/2010/>.
- [20] Oxlab, "0xdroid project." [online]. Available: <http://code.google.com/p/0xdroid/>.
- [21] Massachusetts Institute of Technology, "Project SIKULI." [online]. Available: <http://groups.csail.mit.edu/uid/sikuli/>.
- [22] Hsu Hong-Rong, Shih Wei-Kuan, "Icing: An Ahead of Time Compilation Framework Complement to the Android Dalvik JIT", Master Thesis of National Tsing Hua University, May, 2010
- [23] Wu Chih-Ying, Chung Yeh-Ching, "Dryad: An Ahead of Time Compiler Optimization Framework for Android", Master Thesis of National Tsing Hua University, 2009