공학박사 학위논문

Client Ahead-Of-Time Compiler for App-Downloading Systems

어플리케이션 다운로딩 시스템을 위한 클라이언트 선행 컴파일러

2014 년 7월

서울대학교 대학원

전기.컴퓨터 공학부

홍 성 현

Client Ahead-Of-Time Compiler for App-Downloading Systems

어플리케이션 다운로딩 시스템을 위한 클라이언트 선행 컴파일러

지도 교수 문 수 묵

이 논문을 공학박사 학위논문으로 제출함
2014 년 7월

서울대학교 대학원
전기.컴퓨터 공학부
홍 성 현

홍성현의 공학박사 학위논문을 인준함
2014 년 7월

위 원 장 백 윤 흥 (인)

부위원장 문 수 묵 (인)

위 원 이 혁 재 (인)

위 원 이 재 진 (인)

위 원 김 수 현 (인)

Abstract

Client Ahead-Of-Time Compiler for App-Downloading Systems

SungHyun Hong
School of Electrical Engineering and Computer Science
The Graduate School
Seoul National University

App-downloading systems like DTV and smart phone are popularly used. Virtual machine is mainstream for those systems. One critical problem of app-downloading systems is performance because app is executed by interpreter. A popular solution for improving performance is Just-In-Time Compiler (JITC). JITC compiles to machine code at runtime. So, JITC suffers from runtime compilation overhead. We suggested client Ahead-Of-Time Compiler(c-AOTC) which improves the performance by removing runtime compilation overhead. c-AOTC saves machine code of method generated by JITC in persistent storage and reuses it in next runs.
The machine code of a method translated by JITC is cached on a persistent memory of the device, and when the method is invoked again in a later run of the program, the machine code is loaded and executed directly without any translation overhead. One major issue in c-AOTC is *relocation* because some of the address constants embedded in the cached machine code are not correct when the machine code is loaded and used in a different run; those addresses should be corrected before they are used. Constant pool resolution complicates the relocation problem, and we propose our solutions. The persistent memory overhead for saving the relocation information is also an issue, and we propose a technique to encode the relocation information and compress the machine code efficiently. We developed a c-AOTC on Oracle's CDC VM, and

evaluation results indicate that c-AOTC can improve the performance as much as an average of 12% for benchmarks.
And we adopted c-AOTC approach to commercial DTV platform and test the real xlet applications of commercial broadcasting stations. c-AOTC got average 33% performance improvement on the real xlet application test.
V8 JavaScript VM does not use interpreter. Apps are executed only by JITC. We adopted c-AOTC to V8 VM. But we cannot get any good performance result because of V8 VM's characteristics. V8 VM components are generated as internal objects. Internal objects are used for compiling and running of JavaScript program. The machine code of V8 VM addresses internal objects which are different for each run. Because internal objects be accessed in each run, c-AOTC must recreate those objects. Because most of compilation overhead of V8 VM is internal object creation overhead, c-AOTC does not get enough improvements.

keywords **:** Virtual Machine, Java, JavaScript, Just-In-Time Compiler, Ahead-Of-Time Compiler, client Ahead-Of-Time Compiler
Student Number : 2003-30354

Contents

List of Tables

List of Figures

Chapter 1 Introduction

App-downloading system is a platform running applications which is downloaded. These app-downloading systems are popularly used like DTV, android platform and web apps. Virtual Machine (VM) is mainstream platform for app-downloading systems due to advantage in portability and security. DTV and Blu-ray disks [1] use Java xlets, Google's android platform uses Java-based applications, and web apps use JavaScript-based applications.

One critical problem of these systems is performance. Java application is compiled to bytecode executed by interpreter and JavaScript application is distributed by source codes which are parsed and executed by software. Generally Java application is 3 times slower than C++ application doing same action and JavaScript application is 10 times slower. A popular solution accelerating the performance in VM is Just-In-Time Compiler (JITC) [2, 3, 4] which translates to machine code at runtime. But, JITC suffers from runtime compilation overhead when many methods should be compiled. In applications of app-downloading systems, this is true.

In order to reduce the runtime translation overhead of JITC, we propose a client Ahead-Of-Time Compiler (c-AOTC) [5, 6, 7] inspired by the IBM's Quicksilver, a quasi-static compiler for server systems [8]. Ahead-Of-Time Compiler (AOTC) [9, 10, 11] performs the translation before runtime in the server and the translated machine code is installed and used on the client device. Unlike this traditional AOTC, c-AOTC performs AOTC on the client

device using the JITC module of the Virtual Machine that is installed on the device. That is, the machine code of a method translated by JITC is cached on a persistent memory in the device such as flash memory, and when the method is invoked again in a later run of the program, the machine code is loaded and executed directly without any translation. In this way, we can omit the JITC overhead and achieve a better performance than the original JITC-based execution.

This paper addresses some of the issues in designing and implementing the c-AOTC. The primary issue is relocation [12], which is required since some of the addresses used by the machine code saved in a file in a previous run are not correct any more when the machine code is loaded and used in a different run. We need to modify them before the machine code is executed [8, 13], and this requires saving the relocation information as well as the machine code on the persistent memory. We also propose encoding the relocation information and compressing the machine code, which reduces the space overhead without affecting the performance seriously.

We design and implement our c-AOTC approach on Java circumstance at first and we also adapt c-AOTC on Java platform of commercial DTV and on JavaScript virtual machine.

The rest of this paper is organized as follows. Section 2 introduces the approach of c-AOTC. Section 3 briefly overviews Java virtual machine and our JITC implemented on the CVM. Section 4 describes our c-AOTC and relocation solutions as well as the

embedding and compression scheme. Section 5 describes c-AOTC design and implementation adapted to different JVM which is a platform of commercial DTV. Section 6 describes c-AOTC design and implementation adapted to JavaScript VM. Section 7 shows our experimental results. Section 8 includes related work. A summary follows in Section 9.

Chapter 2 client-AOTC Approach

Our proposed client-AOTC (c-AOTC) targets app-downloading system which is an embedded software platform that can download applications at runtime. Because most popularly used platform for app-downloading system is Java, we will explain our c-AOTC approach based on Java DTV platform in this section.

For example, a software platform for digital TVs (DTV) is typically composed of two components: a Java middleware called OCAP or MHP which is statically installed on the DTV set-top box, and Java classes called xlets which can be dynamically downloaded thru the cable line [14]. Also, a software platform for mobile phones is composed of the MIDP [15] middleware on the phone and midlets downloaded wirelessly. Blu-ray disks [16] consist of the BD-J [16, 17] middleware on the BD player and xlets on the BD titles. We believe this Java software architecture composed of static Java middleware on the client devices and dynamic Java classes downloaded will be a mainstream for embedded systems.

Another trend of these dual-component systems is that both the Java middleware and the downloaded classes become more complex and substantial. The initial downloaded Java classes were mainly for displaying idle screen images or for delivering simple contents, but now more substantial Java classes such as games or interactive information are being downloaded with a longer execution time. In order to reduce the network bandwidth (wired or wireless) for downloading, the Java middleware also gets more substantial to

absorb the size and the complexity of downloaded classes. In mobile phones, for example, the first MIDP middleware provided libraries for user interfaces only, yet its successor middleware called JTWI [18] provided an integrated library with music players and SMS. Now a more substantial middleware called MSA [19] is being introduced with more features.

For achieving higher performance on these substantial, dual-component systems, it would be desirable to employ a hybrid solution for Java acceleration such that the Java middleware is handled by AOTC while the downloaded classes are handled by JITC [20]. The c-AOTC can complement the JITC performance for downloaded classes by obviating JITC overhead.

There can be many possible scenarios to perform c-AOTC in app-downloading embedded systems. For example, we can perform c-AOTC for downloaded applications before execution when the system is idle. Or, we can perform c-AOTC at the end of a regular, JITC-based execution by saving JITC methods in files. Or, we can even perform c-AOTC in middle of execution when we need to evict some JITC methods due to the shortage of runtime memory space [21]; instead of throwing away the evicted JITC methods, we can save them for use later in the same run in case they are called again.

Let us see the scenarios in the case of DTVs, for example. When we turn on a DTV and select a TV channel, a set of xlets and data files prepared for the channel and for the current program will be downloaded from the service provider through the data carousel

protocol, and some indicator (e.g., a red dot) will appear on a corner of the TV screen when the download is over and xlets are ready for execution. If we click on the indicator, the initial xlet will be executed, displaying a menu on the screen. And clicking on a menu item will execute the corresponding xlet. If we change the TV channel, a new set of xlets and data files will be downloaded and the same procedure will repeat.

In this case, our first scenario means performing c-AOTC for the downloaded xlets, while delaying the appearance of the indicator. This delay would not be a big problem unless it is too long, since the TV viewer would regard it as part of the downloading time (which often takes a couple of tens of seconds already). Once the indicator appears on the screen, however, xlets will be executed much faster, improving the TV viewer's response time, which is often more important than the delay.

Our second scenario means saving the JITC code in the persistent memory when the DTV is turned off. And when the DTV is turned on again in the future, we can load and execute the saved JITC code after confirming that the newly downloaded xlets do not differ from the old xlets from which the saved code is derived. In fact, both the xlet application of each channel and the xlet application of each program in the channel are not changed frequently, so caching their JITC code will make the red dot appear much faster in addition to improving the execution time of the xlets.

The third scenario of saving JITC code of xlets in the persistent memory in the middle of a DTV session would be useful when the

memory is tight. In fact, when we change to a different channel, the JITC code of the xlets of the old channel would better remain in the memory so that when we return back to the old channel, we can reuse the saved JITC code of its xlets, improving both the red-dot delay and the xlet execution time. Consequently, the JITC code is accumulated in the memory (and old xlets are supposed to be saved in the memory as well), and if there is a memory pressure, it would be useful to save some of the JITC code in the persistent memory, instead of removing it completely.

Among these scenarios this paper elaborates on the second one. That is, in regular JITC-based execution, machine code of a JITC method exists in the runtime memory only during the execution of the program and is thrown away when the program ends, so the method should be JITCed from scratch if the program runs again later. On the other hand, we can load and execute the machine code directly, if it is saved via c-AOTC in the previous run. The concept of c-AOTC for the DTV is illustrated in Figure 2-1.

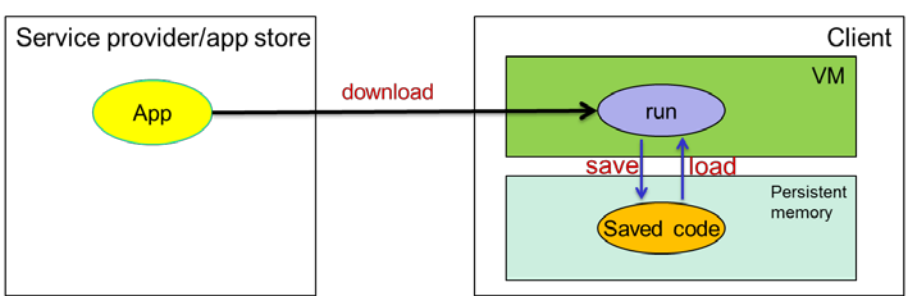


Figure 2-1 Concept of c-AOTC

We are primarily interested in the performance benefit of the "loading mode" of execution, but we are also interested in the performance degradation of the "saving mode" since it cannot be used as a valid, regular run mode if its degradation is too serious, especially considering that we compress the machine code in the saving mode and uncompress it in the loading mode.

Chapter 3 Java Virtual Machine and Our JITC

In this section, we briefly review Java virtual machine, especially the Sun's CVM [22]. Then, we describe our JITC on the CVM, focusing on those features relevant to the client-AOTC.

3.1 Overview of JVM and the Bytecode

Java has been employed as a most popular software platform for many embedded devices. The advantage of Java as an embedded software platform is three folds. First, the virtual machine provides a consistent runtime environment for a wide range of diverse client devices that have different CPUs, OS, and hardware components (e.g., different-sized displays). Secondly, Java has an advantage in security such that it is extremely hard for a malicious Java code to break down a whole Java-enabled device. Finally, it is much easier to develop software contents with Java due to its sufficient, mature APIs and its language features that increase software reliability such as garbage collection (GC) and exception handling. Java also provides platform independence, so the contents providers do not need to develop multiple binaries for multiple platforms for the same Java content.

The advantage of platform independence is achieved by using the Java virtual machine (JVM), a program that executes Java's compiled executable called bytecode [23]. The bytecode is a virtual instruction set which can be executed by the interpreter on any platform without porting. Since this software-based execution is

much slower than hardware execution, compilation techniques for translating bytecode into machine code have been used, such as just-in-time compiler (JITC) [2, 3, 4] and ahead-of-time compiler (AOTC) [9, 10, 11, 24]. In embedded systems, JITC translates bytecode into machine code at runtime on the JVM installed on the client device, while AOTC performs the translation before runtime in the server and the translated machine code is installed and used on the client device. A thorough comparison of JITC and AOTC can be found in [25].

AOTC is more advantageous in embedded systems since it can obviate the runtime translation and memory overhead of JITC, which consume the limited computing power and memory space of embedded systems. Also, AOTC can produce better quality code by employing powerful, offline code optimizations for all methods (even with offline, profile-based optimizations), which will be useful even when there are little dominant hot methods in a program (in contrast, JITC often translates only hot methods due to compilation overhead).

On the other hand, many embedded systems such as digital TVs and mobile phones may download classes dynamically at run time, which cannot be handled by AOTC but should be executed by the interpreter. This can affect the overall performance since slower interpretation is likely to dominate the whole execution time. Therefore, it would be desirable to employ JITC as well to handle downloaded classes for complementing AOTC. One issue is how to mitigate the translation overhead of JITC.

JVM is an abstract computing machine designed to run Java programs. It takes a form of stack-based machine model, where local variables including parameters and its operand stack are pushed into a newly created stack frame when a method is invoked and are popped when it returns. All computations are performed on the operand stack and temporary results are saved in local variables, so there are many pushes and pops between them.

As Java being an object-oriented language, all the data and means of manipulating the specific data are captured into classes and objects, and are saved as class files. A class file consists of stream of JVM instructions called bytecodes, and a constant pool (CP) which is similar to conventional symbol table and is used for dynamic linking. So a JVM's typical duty is to take the class file as its input and run the program as described in the bytecodes using an interpreter. For higher performance, JITC can be employed to translate the bytecode into equivalent machine code.

Dynamic linking using the CP has some important implications to c-AOTC, so it would be worthwhile to briefly mention those bytecodes who access the CP here. They include:

- Field access bytecodes such as getstatic/putstatic for class variables and getfield/putfield for instance variables.
- Method invocation instructions such as invokestatic for static methods, invokevirtual for instance methods, and others (invokeinterface and invokespecial).
- Constant loading instructions such as ldc, ldc_2, and ldc2_w
- Object creation instructions such as new and anewarray

- Miscellaneous object handling instructions such as checkcast and instanceof.

All these bytecodes do not have specific addresses (references) as operands but they include indexes to a runtime CP, a runtime data structure loaded into memory representing the CP in a class file. In the runtime CP entries, classes and fields information accessed by these bytecodes is expressed as name strings instead of addresses. When these bytecodes are executed by the interpreter so that these entries are accessed first time, these CP entries are resolved, meaning that their real addresses or field offsets are retrieved after performing appropriate actions (e.g., loading classes and checking validity of accesses or permissions). The resolved addresses or offsets are saved in the runtime CP in order to quicken future accesses to the same entries, and the executed bytecode may be replaced by a quickened version so as to quicken its future execution (this process is called as quickening) [12]. CP resolution and quickening are dependent on a VM implementation and its object model.

Figure 3-1 (a) shows the object model of the CVM. The first field of an object is a pointer to the class object which includes pointers to a method array, a field array, and a method table, where static methods, static variables, and instance methods can be found, respectively, by chasing a couple of pointers. Instance variables are saved in the object itself.

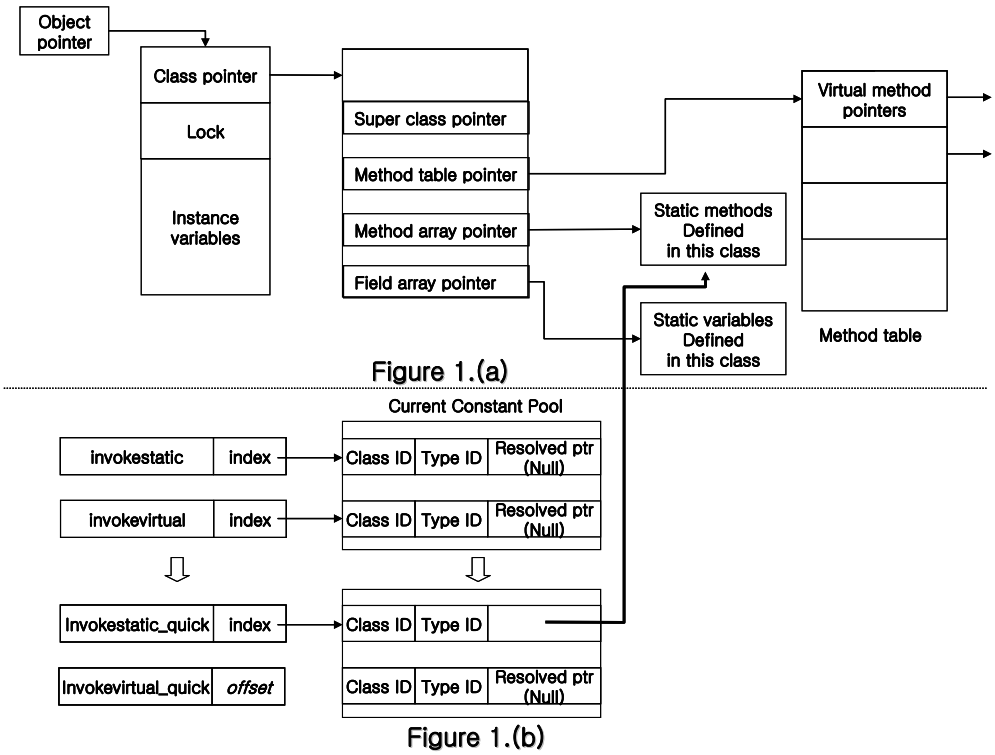


Figure 3-1 CVM object model and the CP resolution examples

Figure 3-1 (b) shows an example of CP resolution. When invokestatic is executed, the address of the target method is obtained and saved in the CP entry, which makes the entry be resolved. The bytecode is replaced by invokestatic_quick which makes its next execution obviate CP resolution. Similarly, the execution of invokevirtual obtains the offset of the target method in the method table, which is then saved in the operand of the bytecode after replacing it by invokevirtual_quick. Other bytecodes and their CP entries are quickened and resolved in a similar way by either saving offsets or addresses in the bytecode operands or in the CP entries.

3.2 Our JITC on the CVM

Since the c-AOTC is heavily dependent on the JITC and the JVM implementations, we briefly review our JITC on the CVM, which targets MIPS/Linux platform. Our JITC uses adaptive compilation method [26], where Java methods are initially executed by the CVM interpreter until they are determined as hot spot methods by some heuristics. When a method is selected as a hot spot method, the JITC module is invoked from the CVM interpreter to compile the method into native instructions. After the compilation, native code for the method resides in the memory called a code cache and is re-used whenever the method is called again thereafter.

In our implementation, JITC process undergoes three different phases. In the first phase, bytecode stream is parsed into appropriate data-structures (normally called as basic blocks) and control flow graph is created. Besides that, method inlining is also performed in this stage for optimization purposes. In the second phase, we generate intermediate representation (IR) by traversing previously formed data-structures. After the translation from bytecode to IR, following conventional code optimization techniques are performed on the IR: copy propagation, common sub-expression elimination, and check eliminations. Finally, efficient register allocation is performed and native code corresponding to each IR is emitted and saved in the memory.

The first phase of JITC includes one more job to do, which is early binding. According to Java specification [23], we are supposed to

perform quickening (hence CP resolution) for only executed bytecodes on an as-needed basis, thus practicing late binding. When a method is JITCed, however, quickening should be done for all CP-accessing bytecodes in the method, even for not-yet executed bytecodes by the interpreter, because translation of those bytecodes requires CP resolution first to get the offsets or the addresses which will be included in the translated code. This leads to early binding, meaning that quickening for some bytecodes is done early, even before it is executed.

Early binding might lead to a failure for some CP resolutions, though. For example, some classes might not be available yet when we perform early binding, if they are supposed to be downloaded later. Even when all classes are available, there are some complications in implementing early binding, especially when JITC and interpreter are used concurrently, as in our CVM.

In order to reduce this complication of early binding, when we attempt quickening for a CP-accessing bytecode during its translation, we simply check if its CP entry is resolved. If it is not resolved yet, we generate quicken-and-patch code in the translated code which will perform late binding when it gets executed during the execution of the translated code.

Chapter 4 Design and Implementation of c-AOTC on JVM

Although JITC can increase the performance, it is involved with the translation overhead, which can be mitigated by using the c-AOTC. This section describes issues in implementing the c-AOTC and our proposed solutions.

4.1 Architecture of the c-AOTC

Figure 4-1 depicts the architecture of the proposed c-AOTC scenario. A method is first executed by the interpreter and when it is determined to be a hot spot, we check if there is a cached version of the translated method in the c-AOTC file of the class (whose name is class-name.aotc) where it belongs. If there is one, we load and execute it directly, and if not, we simply translate it using the JITC module and execute it. At the end of the program execution, each JITC method is saved in an .aotc file unless it is already saved there. In this way, our c-AOTC can be an incremental system such that a new method which was not JITCed in the previous run but is JITCed in the current run (i.e., it became a new hot spot due to a different input data set from the previous run) can also be added to c-AOTC files.

We can even skip the interpretation for those cached methods to execute their machine code earlier (i.e., search for the c-AOTC file occurs earlier than the hot spot check in Figure 4-1), but this

raises an early binding issue as will be discussed in Section 4.2.1. In fact, loading and executing methods earlier even before they are found to be hot spots might not be desirable because hot spots may vary depending on the input set, so some of the previous hot spots cached in the c-AOTC files might not be hot spots in the current run, thus wasting the limited JITC code cache without any real performance benefit. Therefore, we follow the scenario in Figure 4-1.

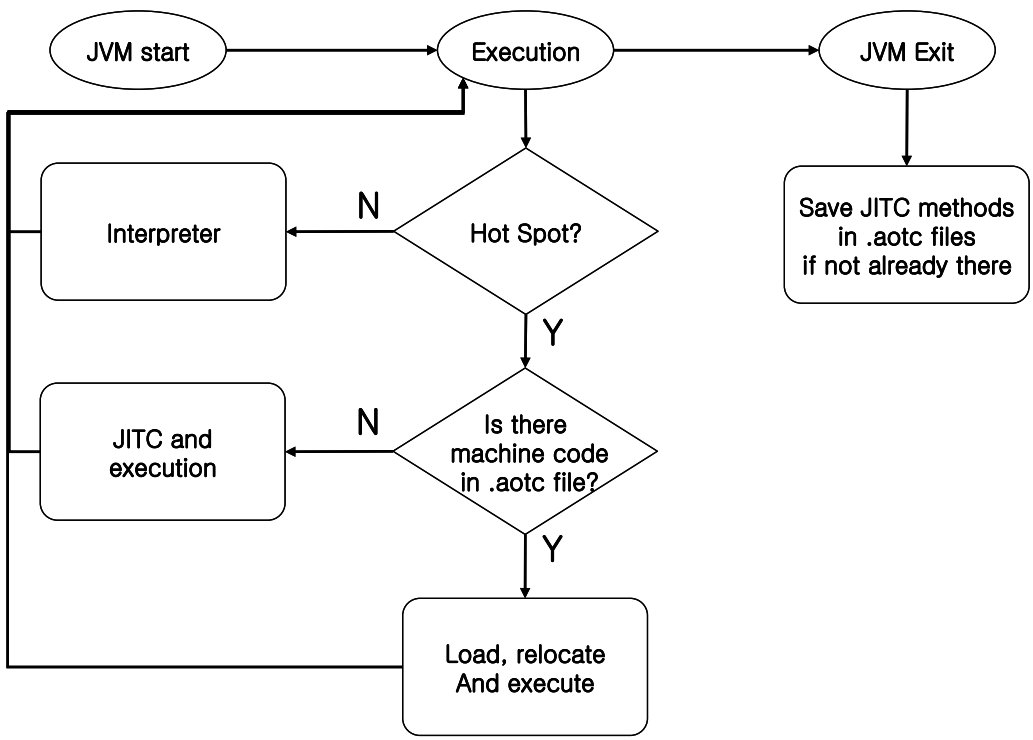


Figure 4-1 Structure of the proposed c-AOTC

Before we proceed with details of c-AOTC, we should mention an important issue briefly, which is validation of the saved machine code. We need to perform compatibility check with the current JVM,

security check if the machine code meets the Java specification, and validity check since classes might have been updated in between when the machine code is saved and when it is used later. These issues have been elaborated in [8] in the server environment, yet the first two issues would not be a big problem in the embedded environment since the xlet machine code is generated only within the embedded system. For example in the DTVs, the machine code for xlets saved in the persistent memory is generated for the same JVM in the DTV set-top box, so compatibility check would not be needed. Moreover, no machine code of xlets can be downloaded into the DTV from outside, so the security check would not be needed, either (on the other hand, in the server environment the JVM update or the downloading of the machine code from different machines can cause such compatibility or security problems).

The last issue may occur in the DTVs since the service provider can send updated xlets while the DTV is turned off, which would make the saved JITC code of old xlets invalid. This check would better be done by the data carousel processing module in the set-top box, which is already responsible for checking the validity of xlets; when the old channel is re-visited and if its xlets and data files are still in the memory, the module checks if they can be re-used, using the version information downloaded from the service provider. In our c-AOTC, we can save the version information in .aotc files so that the module can check the validity of saved .aotc files first and then decide if they can be used or new xlets should be downloaded.

There is one more compatibility issue which is related to optimizations. If some input-specific dynamic optimization was performed for the saved machine code, it cannot be used in a different run with a different input set. For example, if some optimization was performed based on a runtime array size [27], the machine code cannot run correct with different array sizes. Consequently, c-AOTC should not allow such dynamic optimizations based on information that is available only at runtime. In fact, our JITC does not perform any such optimizations, so this compatibility issue is not considered in our c-AOTC.

4.2 Relocation

The most fundamental issue of c-AOTC is relocation because the cached machine code may include addresses that differ from run to run. We first show the instructions which need relocation in the machine code. We then describe what is saved as the relocation information and how the relocation is performed, with some of issues related to relocation. Finally, we show how the relocation information and the machine code are saved in .aotc files efficiently.

4.2.1 Translated Code Which Needs Relocation

There are two types of translated code which requires relocation in our c-AOTC. One is the translated code corresponding to those bytecodes who access the CP, as shown in Section 3.1. The translated code of these bytecodes may include static, “real”

address constants in their instructions. For example, the translated code for invokestatic and getstatic/putstatic includes an address constant corresponding to the address of a static method and a static variable, respectively, whose value is dependent on where they are loaded in memory, thus requiring relocation.

Not all CP-accessing bytecodes require relocation for their translated code, though. The translated code for invokevirtual (invokeinterface) and getfield/putfield include a relative address for a virtual method and an instance variable, respectively, which is described as an offset to some dynamic address decided at runtime. If there are no class updates as we assumed, we do not have to relocate them, but if there are, we must relocate the offsets.

Other CP-accessing bytecodes who need relocation are as follows. For constant-loading instructions such as lwc, the static address constant of the memory location where the constant is saved should be relocated. Object creation bytecodes such as new and anewarray or object check bytecodes such as checkcast and instanceof require accessing the class objects, whose addresses change depending on where they are loaded, requiring relocation.

The other type of translated code requiring relocation is calls to CVM functions and code blocks or accesses to CVM global variables. The translated code generated by our JITC often includes calls to CVM functions or interpreter handlers. For example, when an exception is caught by the exception check code in the translated code (e.g., NullPointerException), it is supposed to jump to the corresponding exception handler located in the CVM's interpreter

code area. The address of the handler should be relocated, though. Similarly, we call CVM's functions to handle locks (monitorenter and monitorexit), to check class information (checkcast and instanceof), or to allocate objects or arrays (new, newarrray), which also need relocation.

Our translated code also needs to access some global variables of the CVM. One example is for synchronizing GC with other threads. That is, each thread is supposed to check if there is any pending request for GC from other threads when it makes a jump (e.g., loop back edges or method calls), which requires accessing a CVM global variable. The address of this global variable should be relocated. Similarly, when there is an exception in the machine code, the address of the corresponding bytecode is passed to the interpreter using the stack frame (the difference of this address and the start address of the bytecode array is used for consulting with the exception table), so this address should also be relocated.

For both types of translated code, there is always a pair of instructions which set a register with a static address constant. For the case of MIPS CPU, for example, an instruction which sets the upper 16 bits of a 32-bit register (load-upper-immediate, LUI) is generated, followed by an instruction which sets its lower 16-bit (logical-or-immediate, ORI) from a given 32-bit address constant. Then a load/store or a jump is performed using the register. So these two instructions should be corrected with a new address constant when they are relocated.

4.2.2 Relocation Information and Relocation Process

When JITCed methods are saved at the end of program execution, we need to save the relocation information as well as the machine code. The relocation information required for each relocation-target instruction pair in a method should include the following items:

1) The location of the relocation-target instruction pair in the method
2) The type of the relocation needed for the instruction pair
3) Additional information needed for relocation which depends on the relocation type, such as the PC of the corresponding bytecode or the index to the constant pool, etc.
4) The identifier of a method where the instruction pair belongs, if the instruction pair came from an inlined method (this will be described shortly in Section 4.2.3)

We should save these relocation data of each relocation-target instruction pair in the .aotc file, along with the translated machine code.

The relocation process is composed of simply scanning thru the translated code and correcting the address constants used in relocation-target instruction pairs by consulting with their relocation information. For the case of accessing CVM functions or global variables, we can easily replace the address constants by new ones. For the case of translated code of CP-accessing bytecodes, we need to perform CP resolution first with the

bytecode in order to obtain a new address constant (or a new offset if there are updates of classes).

There is one issue in CP resolution performed in the relocation process. Let us assume that a method is executed by the interpreter n times before it is JITCed and saved in an .aotc file. In a later run with the same input, if the method is executed by the interpreter exactly n times and then is loaded, the status of the CP would be exactly the same as when the method was JITCed. In this case, all the CP entries accessed by this method are all resolved, so the new address constants can be readily obtained from the CP.

However, if the method is interpreted less than n times before being loaded, some CP entries might not have been resolved yet when we perform relocation. In fact, when the input set is different from the one used when saving the JITC method, unresolved CP entries can also be met during the relocation process, even after the method is interpreted n times (depending on the execution paths that are taken). So, we can have the early binding problem again during relocation.

Quicksilver can also meet unresolved CP entries during its relocation process [8]. Its solution is adding a patch code, which performs CP resolution when get executed. This is similar to our quicken-and-patch code added for early binding during JITC.

We take a different approach. We simply give up relocation, interpret the method one more time, and then try relocation again, hoping that the failed entry is resolved during the interpretation. The process continues until relocation succeeds. If the input is the

same, relocation should succeed within the original n times of interpretation. If the input is different, this process may continue beyond n times. In this case, we give up using the machine code and perform JITC for the method from scratch.

4.2.3 Relocation for Inlined Methods

Inlining is an important optimization technique for JITC, which embeds callee methods into the caller's body, removing call overheads and allowing optimizations across call boundaries. In fact, our JITC performs inlining aggressively such that the original number of method calls in the interpreter mode is reduced by around 44% after JITC.

Inlining complicates the relocation process, though. Inlined machine code may include instructions that came from callee methods which access the callee's CP not the caller's CP. So, relocation for such instructions should access the callee's CP. In order to handle this problem, we add an inlining table to the relocation information of a method in the .aotc file, which lists what methods are inlined in the method and how to find their CPs (by including the invocation bytecode address). When the caller method is relocated, the CPs of all inlined methods are obtained first using the inlining table, and the relocation-target instructions from the callee method are relocated using these CPs. An index to the inlining table is included in the relocation information of the relocation-target instructions of inlined methods so as to access the CPs of inlined methods.

4.3 Reducing the Size of the .aotc Files

The .aotc files can simply consist of the translated machine code and the relocation information for each method, but its persistent memory overhead is more than 10% compared to the original persistent memory usage (which includes JVM executables, libraries, and program class files). In order to reduce this overhead, we embed and encode the relocation information in the machine code, and compress the machine code as described below.

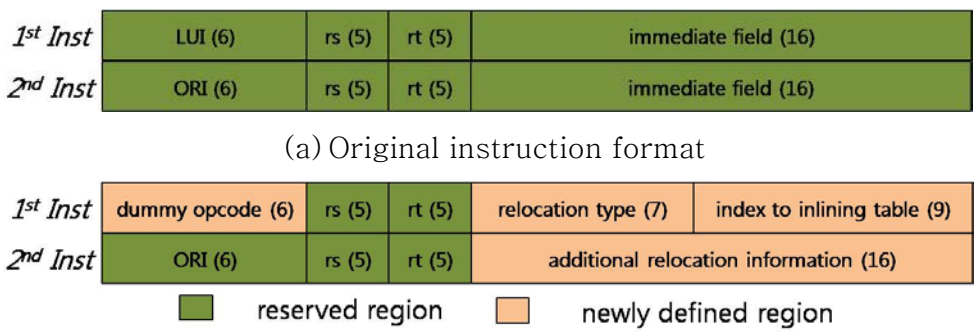
4.3.1 Encoding Relocation Information

The relocation information described in Section 4.2.2 is implemented as a relocation table with four fields such that a relocation table entry exists for each relocation-target instruction pair. And, there is a relocation table for each method and it is saved separately from the machine code of a method. Since there are many relocation-target instructions in a method, these tables are large and constitute a significant portion of the .aotc files.

In order to reduce the space overhead of relocation information, we take a different approach in this paper. Instead of constructing a separate relocation table, we embed the relocation information directly into the relocation-target instruction pairs. This is possible since those instruction pairs include address constant fields which will be corrected anyway during the relocation process; we embed the relocation information in place of those address constants.

As explained previously, our relocation-target instruction pair in

MIPS is composed of LUI and ORI, as shown in Figure 4-2 (a). Both instructions include a 16-bit immediate field. Figure 4-2 (b) shows a new format for both instructions with relocation information embedded. The relocation type and the index to the inlining table (if the instruction came from an inlined method) are encoded in the first instruction, and the additional relocation information is encoded in the second instruction. The opcode field of the first instruction was originally LUI but it is replaced by an unused opcode in MIPS. This is to identify the instruction pairs during the relocation process, when scanning thru the instruction stream.



(a) Original instruction format

(b) New instruction format

Figure 4-2 The old and new instruction format for relocation

When a JITC method is saved at the end of program execution, the relocation-target instructions are replaced following the new format before being saved. When the method is loaded in a later run, those instructions are updated and relocated after the relocation information encoded in them is extracted.

4.3.2 Machine Code Compression

The translated code size is generally much larger than the original bytecode size, so it is desirable to compress the machine code. There are two issues, though. One is that uncompressing the compressed machine code when it is loaded might affect the performance due to the uncompressing overhead. Fortunately, we found that the uncompressing overhead is negligible such that it affects performance little [28]. The other issue is that the compressed version of the machine code may be bigger than the original machine code for some small methods. In this case, we save the original machine code and mark it, so that the loading process can choose whether to uncompress the saved machine code or not.

4.3.3 Structure of the .aotc File

Figure 4-3 depicts the overall structure of an .aotc file, which is composed of a header part and a body part. The header part includes information on the class and the methods whose machine codes are saved in this file, which is used by the CVM to check if the machine code for a method exists in the .aotc file. The body part includes the compressed or the original machine code as well as the inlining table.

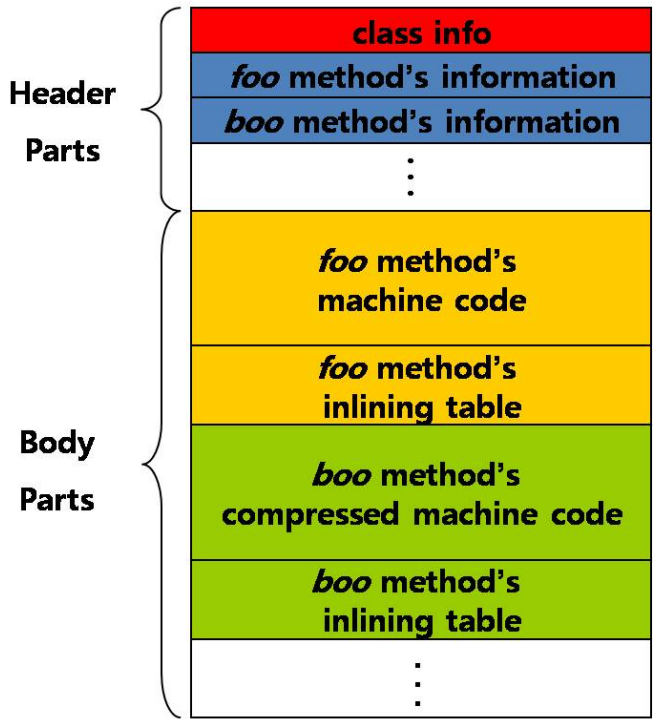


Figure 4-3 Structure of an .aotc file

Chapter 5 c-AOTC for DTV JVM platform

DTV allows data broadcasting based on Java such that the TV stations send xlet applications which execute on the DTV to display user-chosen information such as stock, weather, traffic, or news. Our measurement shows that the xlet applications include loops which iterate fewer times and methods which are invoked fewer times than benchmarks. This would reduce the performance impact of JITC.

So we employ c-AOTC for the DTV JVM platform which uses downloaded xlet applications. When the user executes the xlet application for a given channel, the machine code generated by the JITC is thrown away when the user turns off the TV or switches to a different channel. Our c-AOTC save the machine code on a persistent memory on the DTV such as flash memory so that when the user turns on the TV or switches back to the same channel, the machine code can be loaded from the persistent memory to the code cache and executed directly without any translation overhead. In this way, we can omit the JITC overhead and achieve a better performance than the original JITC-based execution.

One important issue of c-AOTC is relocation. When we load the saved machine code to the code cache in the VM of DTV, some of the addresses in the machine code might need to be updated because they might be different from the addresses when the machine code is saved. The relocation problem of c-AOTC is dependent on how the JITC produces the machine code. For

example, when a method A calls a method B, the machine code of method A must have a jump to the address of the method B. The address can be a real address constant of method B, which then needs to be corrected, or the address can be accessible from some data structure of the JVM whose address is fixed, which does not need to be updated. We will address these relocation issues for our DTV environment.

5.1 DTV software platform

The DTV sends digital signals which consume less bandwidth than analog signals [14]. The remaining bandwidth can be used for broadcasting data such as news, traffic, weather, stock, game or program-specific information. Data broadcasting in DTV is based on Java, so there are many Java open standards including Advanced Common Application Platform (ACAP) [29], which is employed in our target DTV platform.

The Java-based data broadcasting is programmed using the xlet application, which is composed of the java class files and image/text files. The xlet application is broadcasted to the DTV set-top box and executed with the system and the ACAP middleware classes installed on the set-top box. Each TV channel has a different xlet application. So if the channel is switched, a new xlet application is downloaded.

When the user turns on the DTV, the JVM starts and a Java program called an application manager initiates. Then, the xlet

application for the current channel will start its lifecycle, as depicted in Figure 5-1. When the xlet application starts being downloaded, it is in the "Not Loaded" state. When the application manager loads xlet's main class file and creates the xlet object, the xlet is in the "Loaded" state. Then, the application manager initializes the xlet to the "Paused" state. Finally, the application manager starts the xlet, entering to the "Started" state.

At this point, a message is on the TV screen, indicating that the xlet application is ready for execution. When the viewer presses a button on the remote control, a menu appears on the screen where the xlet items like news, weather, traffic, and stock are displayed. If the user chooses one item by moving the cursor, the corresponding information will appear on the screen after executing the corresponding xlet code.

If the viewer switches to a different channel, or if some xlet file of the current channel is updated (i.e., a modified xlet file is sent), the xlet is stopped and its state moves to a "Destroyed" state in Figure 5-1, where all the resources for the xlet is released. A new xlet application for the changed channel or the updated channel will start its lifecycle. When we adopt the c-AOTC to DTV, the saving phase of c-AOTC is processed during the "Destroyed" state and the loading phase is processed during the initialization of the xlet between the "Loaded" state and the "Paused" state.

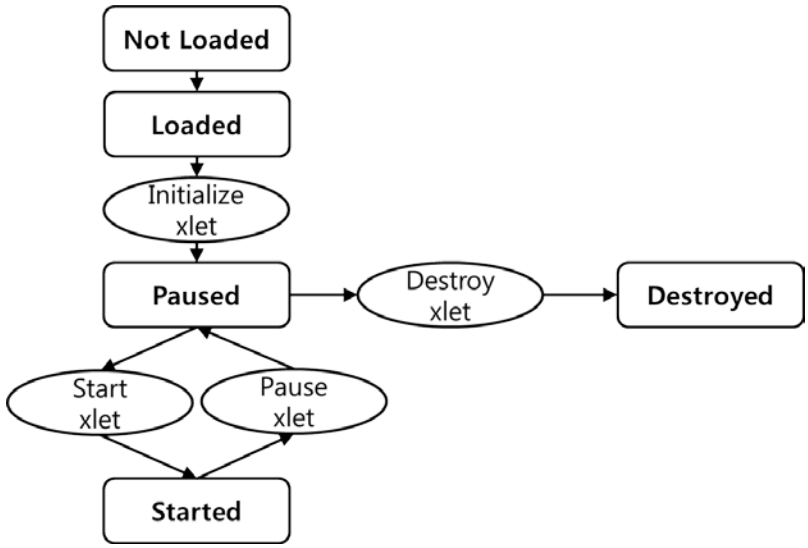


Figure 5-1 Lifecycle of xlet application

5.2 c-AOTC on the DTV

This section describes our design and implementation of c-AOTC for the DTV platform. It also addresses the relocation issue compared to our previous JVM c-AOTC of section 4

5.2.1 Design of c-AOTC on DTV

We implement our c-AOTC on a commercial DTV which has the PhoneME Advanced MR2 JVM [30]. The PhoneME JVM has a JITC based on HotSpot technology as other JVMs [26], so a method is interpreted initially and when it is found to be a hot spot, the method is compiled and saved in the code cache. Our c-AOTC system will save the code cache in a file called the c-AOTC file when leaving the current channel, and loads the c-AOTC file to the code cache when revising the previous channel. Figure 5-2 shows

the scenario of c-AOTC on the DTV platform.

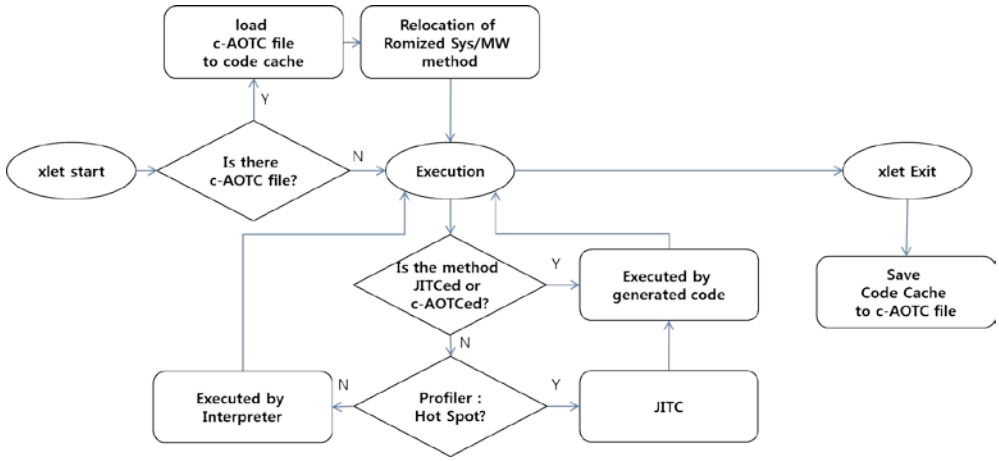


Figure 5-2 c-AOTC implementation on DTV

When an xlet application for the current channel starts, we check if there exists the c-AOTC file for the xlet. If there is none, the ordinary interpretation and JITC will be used during execution. If there is one, however, it is loaded to the code cache after relocation and the methods having loaded machine code will be executed by the machine code. When a c-AOTCed method is invoked during execution, its machine code will be executed without interpretation and translation by JITC. When the execution of xlet application terminates, the code cache is saved to the c-AOTC file if the application does not have previous c-AOTC file.

There are a couple of issues. One is the unit of the loading. When we load the c-AOTC file to code cache and do relocation, there can be two approaches. One is loading one method from the c-AOTC file when it is called. The other is loading all methods form the c-

AOTC file at once when the execution of the xlet application starts. Our c-AOTC implementation of chapter 4 takes the first approach but DTV c-AOTC takes the latter.

Another issue is the target methods of c-AOTC. Our DTV c-AOTC saves only the system and middleware methods; we do not save the methods of the downloaded xlet application. This is related to relocation since xlet methods are difficult to relocate, as will be described in Section 3.2. This does not affect the performance of DTV c-AOTC, though, because the xlet methods are rarely hot, as will be described in the experimental results.

This DTV JVM has an AOT system. The AOT performs the translation before runtime usually in the server and the translated machine code is installed and used on the client device [9, 10, 11].

But, This AOT system of PhoneME Advanced MR2 JVM translates pre-selected system/middleware methods using compilation module of JITC on client device and saves the code cache to file. When an application runs on the JVM, JVM loads the file to the code cache and uses the pre-translated machine code.

This AOT system can increase the performance by reducing runtime compilation overhead. But, it has some disadvantages. At first, some major optimizing techniques (e.g. inlining) using dynamic profiling information are not used by this AOT system. So, the code quality of AOT is lower than JITC. At second, this AOT system use big file and big code cache, because this AOT system translates pre-selected methods including non-used methods. The default number of pre-selected methods is over 950.

5.2.2 Relocation Problem

The most fundamental issue of c-AOTC is relocation [12] because the saved machine code may include addresses that differ from run to run. In our previous JVM c-AOTC work of chapter 4, we have diverse relocation targets. But in current JVM, we only need to relocate the address of machine code of callee method. JVM has a methodblock structure which has the data of a method including the address of translated machine code of a method. So the loading phase of c-AOTC relocates the methodblocks because the address of translated machine code copied to code cache by c-AOTC loading phase is different from the encoded address.

In c-AOTC file, it has machine code binary and the data according to the saved methods by c-AOTC. An important data is a descriptor. This descriptor has the address of methodblock. For relocation, we search the methodblock from descriptor and modify the address of methodblock to current address of the machine code.

Why the relocation process is simple and easy is because of following two characteristics of PhoneME JVM and DTV platform. In Figure 5-3, the JVM architecture of DTV platform romizes the system/middleware and binds the romized structure with JVM binary. The romization takes a list of class files, loads the classes and resolves them, and then takes a snapshot of the layout of all resulting data structures in memory. In data structures, there is a methodblock. So, the address of methodblock is fixed. And translated machine code uses indirection for calling a method. For

calling method B from method A, the machine code of method A does not have the address of the machine code of method B, but have the address of methodblock of method B. So, method A gets the address of the machine code of method B from methodblock and jump to it indirectly.

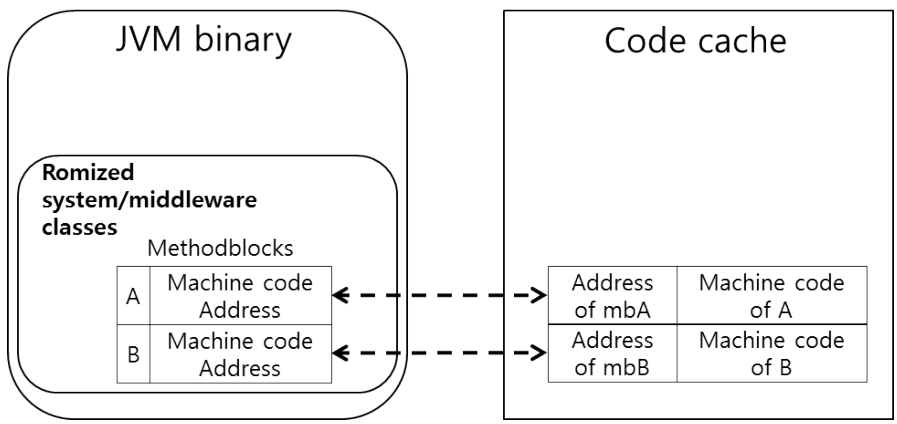


Figure 5-3 JVM architecture of DTV

Each class of Java has a constant pool (CP). The constant pool is where most of the literal constant values are stored. This includes values such as numbers of all sorts, strings, identifier names, references to classes and methods and type descriptors. All CP-accessed bytecodes do not have specific addresses (references) as operands but they include indexes to a runtime CP, a runtime data structure loaded into memory representing the CP in a class file. In the runtime CP entries, classes and fields information accessed by these bytecodes is expressed as name strings instead of addresses. When these bytecodes are executed by the interpreter so that these entries are accessed first time, these CP entries are resolved,

meaning that their real addresses or field offsets are retrieved after performing appropriate actions. The resolved addresses or offsets are saved in the runtime CP in order to quicken future accesses to the same entries, and the executed bytecode may be replaced by a quickened version so as to quicken its future execution (this process is called as quickening).

For example, Figure 5-4 (a) shows the invokestatic bytecode is changed to invokestatic_quick after its first interpreter execution with resolution which saves the address of static method to CP entry.

When we translate invokestatic_quick bytecode with JITC, the compilation module checks the methodblock of callee method and encodes the address to generated code. So, the resolved address of callee is encoded to the machine code like Figure 5-4 (b). If the address of the static method is changed at next run, we must relocate it for reusing the saved machine code by c-AOTC.

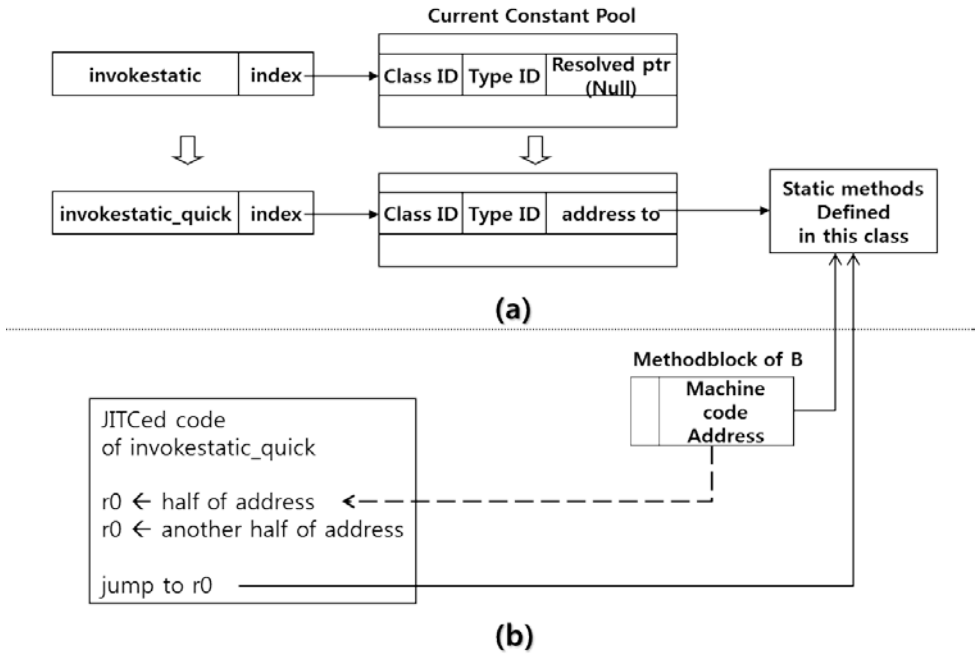


Figure 5-4 Example of access to constant pool

DTV/PhoneME JVM platform which we implemented our c-AOTC approach has the fixed address of the static method of system/middleware classes because these classes are romized and bound with JVM binary.

Figure 5-5 shows the machine code of invokestatic_quick bytecode which is same bytecode of Figure 5-4 (b). The code does not access the static method directly. The saved code does not jump to the machine code directly. Above code accesses the methodblock of callee method and gets the address of machine code of callee method for jumping to the machine code.

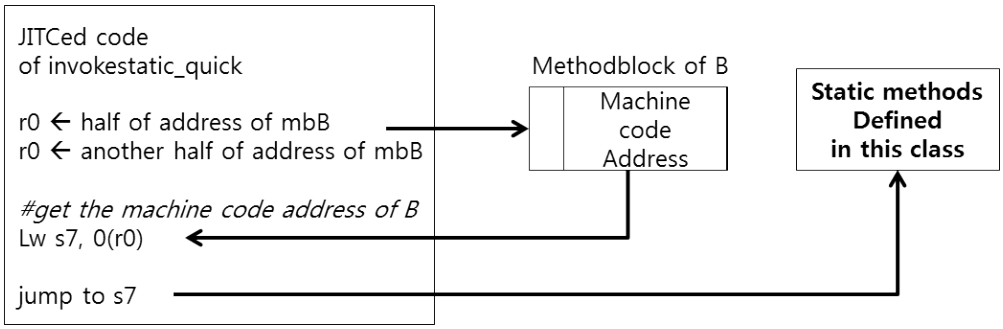


Figure 5-5 Example code on PhoneME JVM on DTV

Different from system/middleware classes, the xlet classes which are downloaded from the broadcasting station do not have the fixed location. So, if we want to reuse of the machine code of xlet classes, we must relocate the address of methodblock of xlet classes. In our implementation environment, only a few methods of xlet classes are JITCed. In fact, the JITCed methods of xlet classes are average 5% of total JITCed methods. So the overhead for reusing the machine code of xlet class methods is big, but the benefit is very small. So, we do not reuse the machine code of xlet classes.

5.2.3 Example of Relocation

We will compare generated machine code between JVM of section 4 and PhoneME JVM of DTV platform of this section.

5.2.3.1 Relocation Example of JVM c-AOTC

• checkcast_quick bytecode

checkcast_quick bytecode is the quicken version of checkcast bytecode. checkcast bytecode checks that the top item on the

operand stack (a reference to an object or array) can be cast to a given type.

The JITC of previous JVM c-AOTC work generates the machine code of checkcast_quick bytecode like following code.

```
r0 ← half address of class object
r0 ← another half address of class object
r1 ← half address of a VM function “isSubclassOf”
r1 ← another half address of a VM function
jump to r1
r1 ← half address of a VM function “isAssignable”
r1 ← another half address of a VM function
jump to r1
```

checkcast accesses the class object which has address changed by where it is loaded. So, the address must be relocated. In this code there are two encoded address for VM internal functions. The address of VM functions is fixed during VM running. So the address of VM functions is different from previous run to current run. For reusing the saved machine code, we must modify the encoded address to current address. For getting new address of callee method, we save some relocation information that the address notifies which method in saving phase.

• invokestatic_quick bytecode

invokestatic_quick bytecode is the quicken version of invokestatic bytecode. invokestatic bytecode calls a static method (also known as a class method). The JITC of previous JVM c-AOTC work generates the machine code of invokestatic_quick bytecode like following code.

```
r0 ← half address of a static method
r0 ← another half address of a static method
jump to r0
```

The machine code includes the encoded address of static method's machine code in code itself. This address is changed from previous run to current run. For correct execution, this address must be relocated.

• getstatic_quick bytecode

getstatic_quick bytecode is the quicken version of getstatic bytecode. getstatic bytecode accesses the static variable of class.

The JITC of previous JVM c-AOTC work generates the machine code of getstatic_quick bytecode like following code.

```
r0 ← half address of static field
r0 ← another half address of static field
lw r1, 0(r0)
```

The machine code has encoded address of static variable. And the address is different from previous run to current run. For correct execution, c-AOTC must relocate the address to current address.

5.2.3.2 Relocation Example of DTV c-AOTC

• checkcast_quick bytecode

In PhoneME JVM for our DTV c-AOTC work, The JITC translated checkcast_quick bytecode to following machine code.

```
r0 ← half address of classblock
r0 ← another half address of classblock
r1 ← half address of VM function "runtimeCheckCastGlue"
r1 ← another half address of VM function
jump to r1
```

checkcast bytecode use the name of class as its argument. But the translated machine code accesses the classblock. The classblock is made for each java class. In our environment, the classblock of system/middleware classes is romized and has a fixed address like methodblock. So address of classblocks are fixed by romization and do not need to modify. And the address of VM function “runtimeCheckCastGlue” is also fixed because it is the helper function bound with PhoneME JVM binary.

• invokestatic_quick bytecode

In PhoneME JVM for our DTV c-AOTC work, it translates invokestatic_quick bytecode to following machine code.

```
r0 ← high address of methodblock of static method A
r0 ← low address of methodblock of static method A
#get the machine code address of method A
LW s7, 0(r0)
jump to s7
```

The machine code loads the methodblock of method and gets the address of machine code from the methodblock and jumps to the machine code of callee method. Different from previous c-AOTC work, the encoded address in machine code is the address of methodblock, not the address of callee method’s machine code. Because the system/middleware classes are romized and the produced data structures (including methodblock) are bound with JVM binary, the address of methodblock is fixed.

So when we reuse this machine code, we don’t need any modification to the machine code. But the methodblock do not have the real address of the machine code because the location of the

machine code is different from previous saving run of application. So we need to do relocation by filling the methodblock with the real address of the machine code. When we copy c-AOTC file to the code cache in loading mode, we can know the real address of the machine code. And we can know the correct methodblock of the machine code because we also save the descriptor which has the fixed methodblock address.

• getstatic_quick bytecode

The generated code of getstatic_quick by PhoneME JVM JITC is not different from the code from previous JVM c-AOTC work.

```
r0 ← half address of static field
r0 ← another half address of static field
lw r1, 0(r0)
```

But, the encoded address of static field is fixed in current PhoneME JVM because the system/middleware class is romized and binds with JVM binary. So, the address does not need to be modified.

Chapter 6 c-AOTC for JavaScript VM

In web circumstances, JavaScript language is popularly used as client-side script language because it is independent to platform and easily developed. By language's platform independency, JavaScript is used not only for web page, but also for application ecosystem of smartphone and smart TV which are app-downloading systems.

Because market size of JavaScript platform is growing, the performance issue of JavaScript is coming to the fore. For improving the performance, JITC is popularly used. Mostly used JavaScript platforms like JavaScriptCore (JSC) of Webkit engine [31] and V8 engine [32] of Google adapt JITC as performance improvement technique.

But sometimes JITC cannot improve the performance because of the runtime compilation overhead, especially for many warm spot and few hot spot situation. In case of running JavaScript web applications, there are few hot spot methods. [33, 34] So, it is hard to introduce good performance to JavaScript platform by JITC.

So, we try to adapt client-AOTC for improving the performance by reducing the runtime compilation overhead.

6.1 V8 JavaScript VM

We implement our c-AOTC approach to V8 engine of google [32]. V8 engine has no interpreter and translates all methods to machine code by JITC and executes the machine codes. Figure 6-1 shows

brief explanation of the JITC internal of V8 engine. JITC of V8 engine has two parts – parsing phase and translating phase. Because JavaScript applications are distributed by source level, all JavaScript VMs need to parse the source of applications. When JavaScript application is JITCed in V8 engine, JITC parses JavaScript application method to Abstract Syntax tree (AST) as intermediate representation and compiles the AST to machine code.

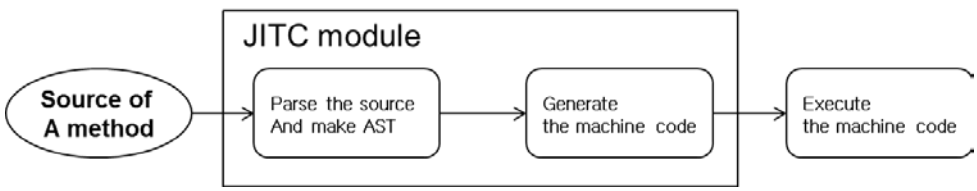


Figure 6-1 Simple structure of JITC in V8 engine

In c-AOTC implementation, it skips the generation phase not the parsing phase because of the characteristics of V8 engine. Many VM components of V8 JavaScript are generated as internal objects which are used for compiling and running of JavaScript program. Most of internal objects are created during parsing phase, but some of them are created during generation phase. The translated machine code accesses internal objects directly. Because the addresses of internal objects changes between VM runs, relocation issue is coming to the fore. And when we load c-AOTCed machine code, the accessed internal objects must be existed. Because parsing phase not only generates AST but also creates the internal objects, c-AOTC cannot skip parsing phase. If c-AOTC skip

parsing phase, c-AOTC must load AST and re-create internal objects. Because c-AOTC cannot get any benefit from skipping parsing phase, we adopt c-AOTC just to generation phase.

6.2 Issue and Solution of c-AOTC on V8 JavaScript VM

Major issue of c-AOTC in V8 engine is relocation of internal objects which are accessed by loaded machine code. Figure 6-2 simply depicts the relocation problem.

The first problem is that the address accessing internal object from loaded machine code can be incorrect. The internal objects created during parsing phase have different addresses between VM runs. If a machine code accessing internal object is loaded, the encoded address accessing internal object will be not valid.

The second problem is that the internal object accessed by loaded machine code can be not existed. Because c-AOTC skips the generation phase, the internal objects which must be created during the generation phase are not created. So, loaded machine code cannot access the internal objects.

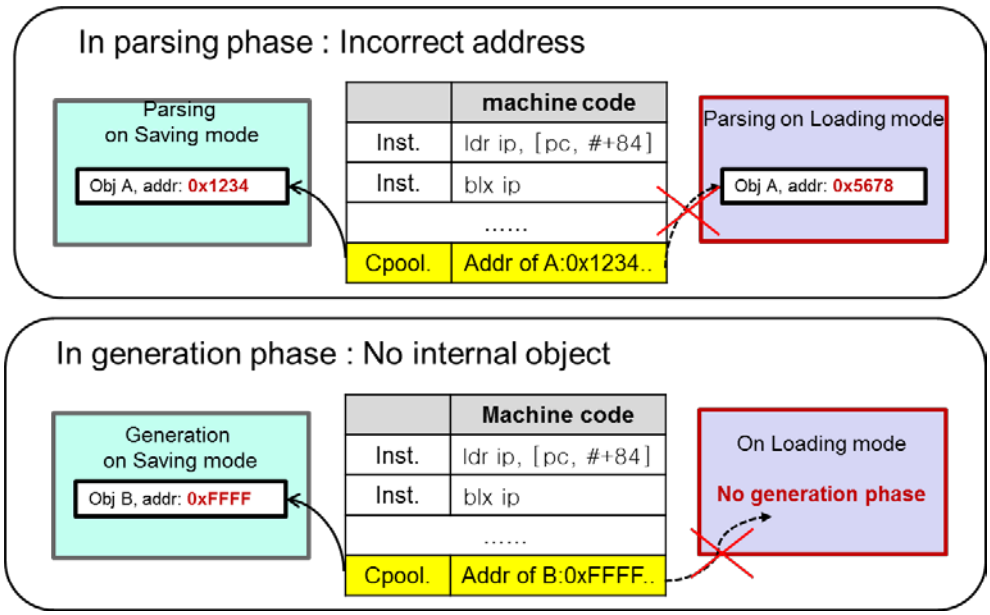


Figure 6-2 The problem accessing invalid internal objects

Because there are two problems about relocation of internal object, they have different solutions.

In the case of internal objects created during parsing phase, the address of loaded machine code is incorrect but the internal object is existed. By modifying the encoded address to newly created one, we can solve the relocation problem. For getting correct address, c-AOTC must do preprocessing using mapping table. Figure 6-3 shows the solution for relocation of internal object created during parsing phase. Figure 6-3 (a) is the original run of V8 engine. Internal object A is created during parsing phase and accessed by machine code directly. Figure 6-3 (b) shows the mapping table. The mapping table has elements which are a specific key index and address of the internal object. When an internal object is created, VM gives a specific index key and writes the address of internal

object to mapping table. This mapping table is made every VM run. In Figure 6-3 (c), c-AOTC saves the machine code after modifying the address of internal object to its specific index key value. Figure 6-3 (d) shows the loading mode. When the machine code is loaded, an internal object accessed by loaded machine has different address from previous run but its index key is same. VM can change the index key value to its valid address from mapping table. So the loaded machine code accesses valid internal object.

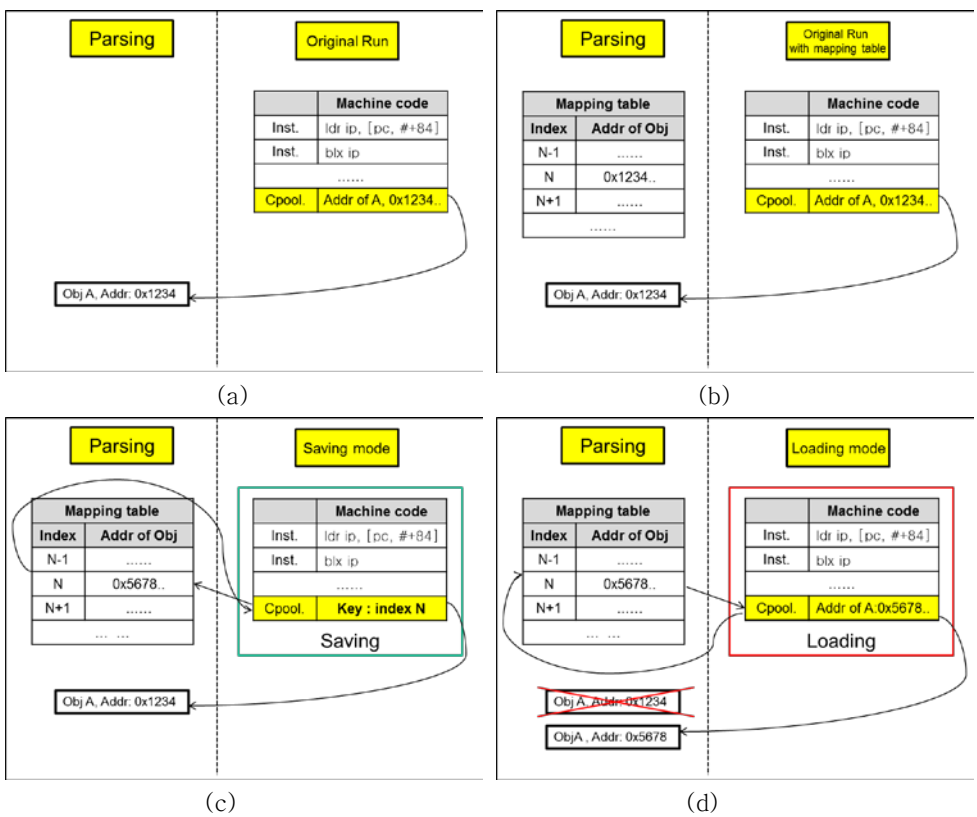


Figure 6-3 Solution for accessing incorrect address

In the case of internal objects created during generation phase, the internal object is not existed because c-AOTC skips the generation

phase. Because there is not internal object, VM must recreate it. For recreating the internal object, c-AOTC does preprocessing using unique id of internal object. Figure 6-4 shows the solution for relocation of internal object created during generation phase. Figure 6-4 (a) is the original run of V8 engine. Internal object B is created during generation phase and accessed by machine code directly. In Figure 6-4 (b), VM makes unique id from internal object B based on its information. And VM saves the machine code after modifying the address to its unique id. In Figure 6-4 (c), VM loads the machine code and re-creates the internal object accessed by loaded machine code based on the encoded unique id.

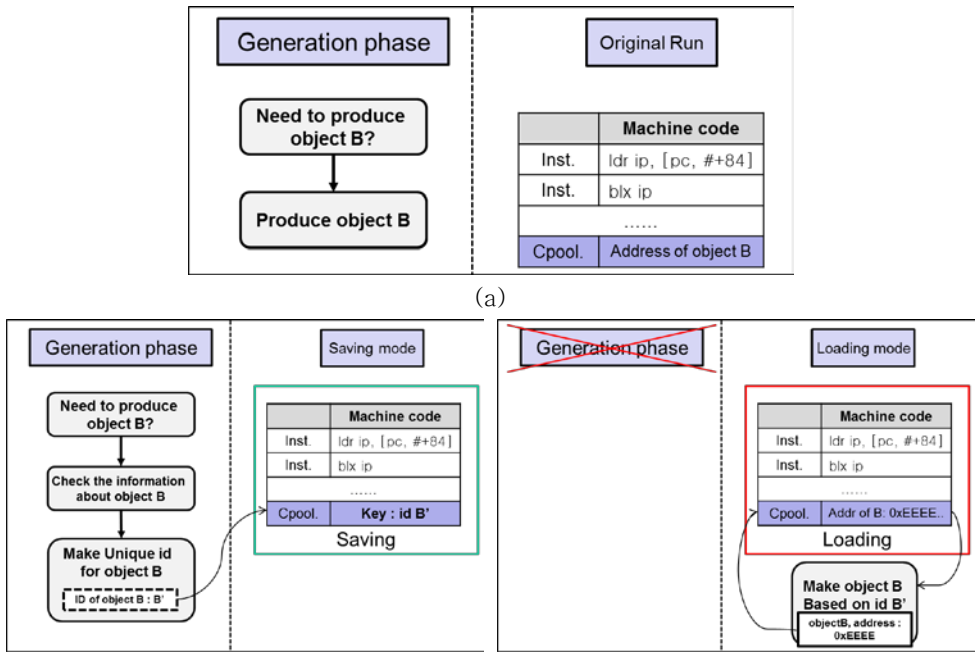


Figure 6-4 Solution for accessing non-existed internal object

Chapter 7 Experimental Results

Previous sections described our proposed c-AOTC which saves JITC methods of a previous run for faster execution of later runs of a program. In this section, we evaluate the proposed c-AOTC.

7.1 Experimental Environment of JVM

The experiments were performed with our JITC implemented on CVM RI version build 1.0.1_fcs-std-b12. Our JITC passed most of the compatibility tests, which was a really demanding work due to many corner tests (e.g., intentional early binding failures which should be handled correctly with JITC), yet we did not perform a full-compatibility test with c-AOTC.

Our CPU is a MIPS-based SoC called ATI Xilleon which is popularly employed in Digital TVs. The MIPS CPU model is 4Kc V0.7 with a clock speed of 300MHz. It has an I-cache of 16KB, a D-cache of 16KB, and a 128MB main memory. We installed a local disk of 40GB (7200RPM) for storing the .aotc files. The OS is an embedded linux (kernel v2.4.18).

The benchmarks we used are SPECjvm98 (except for javac.) [35] and EEMBC [36]. Since the xlets or the OCAP middleware, for example, are proprietary information of the service providers or the set-top box manufacturers, we could not access their source code. On the other hand, the purpose of our experiments is evaluating the benefit and the penalty of c-AOTC, which could possibly be estimated using the conventional benchmarks to some degree, since

the same kind of JITC overhead, saving overhead, loading overhead and relocation overhead would be involved when experimenting with them.

For performance evaluation, we measured 10 times for each benchmark, choose five numbers in the middle, and took their average. We did this because of severe fluctuations in some benchmarks (e.g., regex in EEMBC or mtrt in SPECjvm), which would be due to the embedded environment whose performance is more sensitive and fluctuating than in the desktop environment. When we measure 10 times for each benchmark, we actually run the CVM 10 times, not iteratively running the benchmark 10 times on a single CVM run which might distort the performance impact of the c-AOTC. Before each benchmark runs, we removed all of previous .aotc files (including those from standard library methods shared by other benchmarks) to measure the impact of the c-AOTC more precisely.

In this c-AOTC implementation, we do not implement the techniques of section 4.3.1 and section 4.3.2. Those techniques for reducing file size are adapted to c-AOTC implementation of section 7.5

7.2 Performance Impact of c-AOTC

For each benchmark, we ran four modes of execution. The first one is a normal execution mode based on interpretation and JITC. The second one is a saving mode where normal execution is performed but at the end of program execution, machine code and relocation information are saved in .aotc files. The third one is a loading mode with normal interpretation such that interpretation is performed first as usual and when a method is to be JITCed, the machine code is loaded and executed if it is available, obviating the JITC overhead. The last one is also a loading mode but with no interpretation attempt such that the first time a method is executed, its machine code is loaded and executed if it is available even without any interpretation, obviating both the JITC and the interpretation overhead. However, relocation might fail due to the lack of CP resolution as explained in Section 4.2.2, which leads to interpreting the methods incrementally until relocation succeeds.

Figure 7-1 and Figure 7-2 depicts the performance ratio of the saving mode and the two loading modes compared to the normal mode, for EEMBC and SPECjvm, respectively. The saving mode suffers from performance penalty due to collecting relocation information and storing them with machine code in the files, yet it is an average of 2~3%, which is small enough to make the saving mode still a practically competitive run mode.

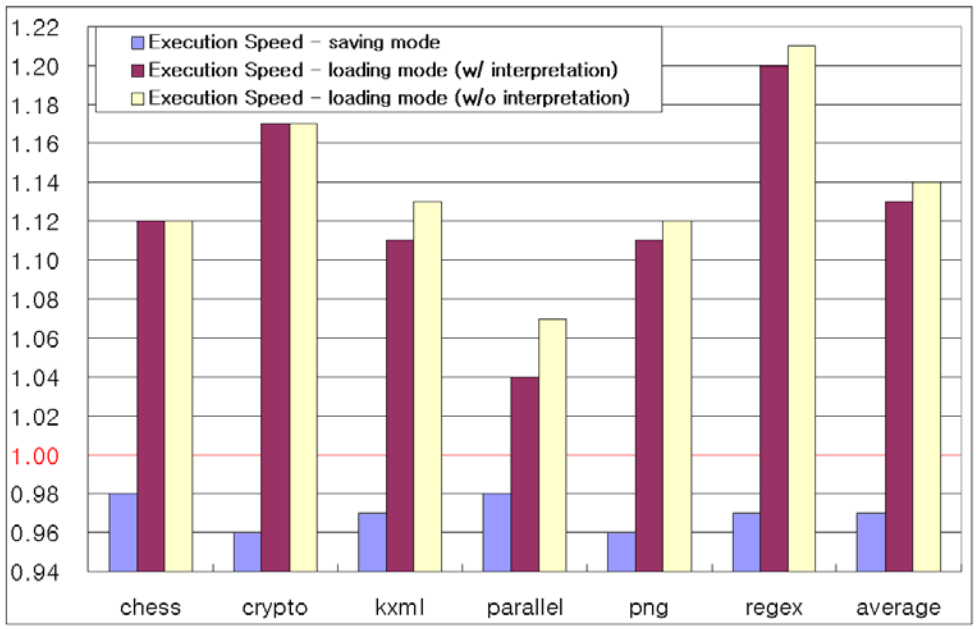


Figure 7-1 Performance ratios in EEMBC

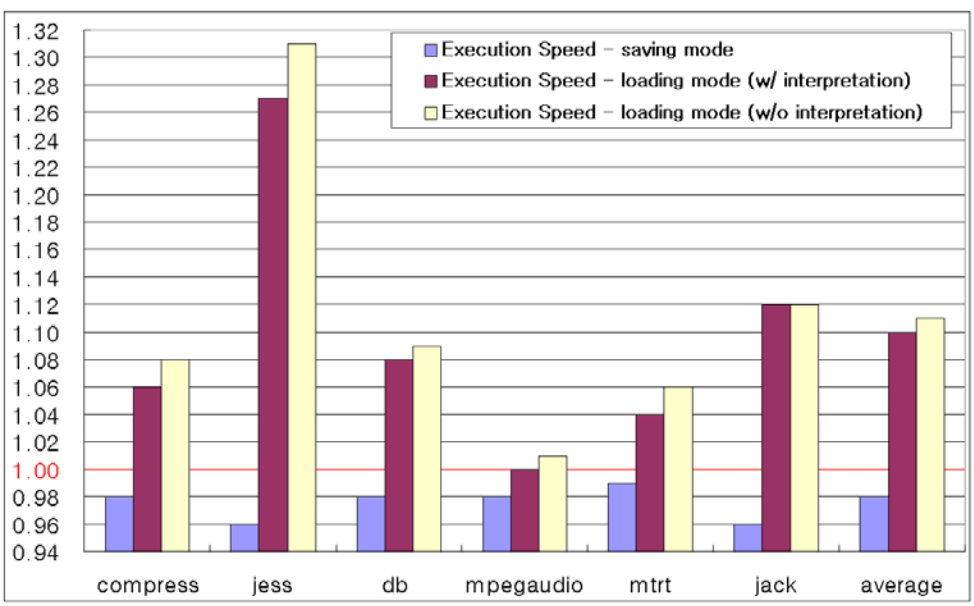


Figure 7-2 Performance ratios in SpecJVM98

The performance benefit of the loading mode with normal interpretation is an average of 10~13%, which would roughly correspond to the JITC overhead. The loading mode with no interpretation attempt achieves an additional benefit of 1% by obviating some interpretations (we could not measure the interpretation overhead separately in the normal mode since its running time for each method is too short to measure precisely).

Table 7-1 shows the total number of executed methods and JITC methods, respectively, with the number of files generated. Around 10% of executed methods are JITCed and an .aotc file contains an average of 2.2 methods in it.

Table 7-1 Total number of executed methods, JITCed methods and JITCed files.

Programs	Total # of executed methods [a]	Total # of JITC methods [b]	Total # of c-AOTC files [c]	b/a (%)	b/c
chess	968	108	53	11.2	2.0
crypto	962	106	46	11.0	2.3
kxml	953	111	55	11.6	2.0
parallel	884	67	39	7.6	1.7
png	897	67	42	7.5	1.6
regex	943	68	40	7.2	1.7
compress	759	44	26	5.8	1.7
jess	1171	165	71	14.1	2.3
db	776	78	43	10.1	1.8
mpegaudio	942	142	55	15.1	2.6
mtrt	907	157	54	17.3	2.9
jack	1012	200	57	19.8	3.5
Average	**931**	**109**	**48**	**11.7**	**2.2**

For the loading mode with no interpretation attempt, Figure 7-3 shows the cumulative percentage of methods that succeed in relocation as a function of the interpretation count for each benchmark. An average of 64% methods succeed in relocation without any interpretation (the percentage when the interpretation count is zero). This is possible because these methods do not access the CP or their accessing CP entries are already resolved when we relocate other methods in the same class which share the CP. An average of 94% of methods succeed in relocation within a single interpretation (the percentage when the interpretation count is one). This result indicates that our approach to CP resolution based on interpretation for relocation is efficient. There are several methods which require many interpretations, though, yet we confirmed that those methods still succeed in relocation before reaching their original interpretation count, as we expect.

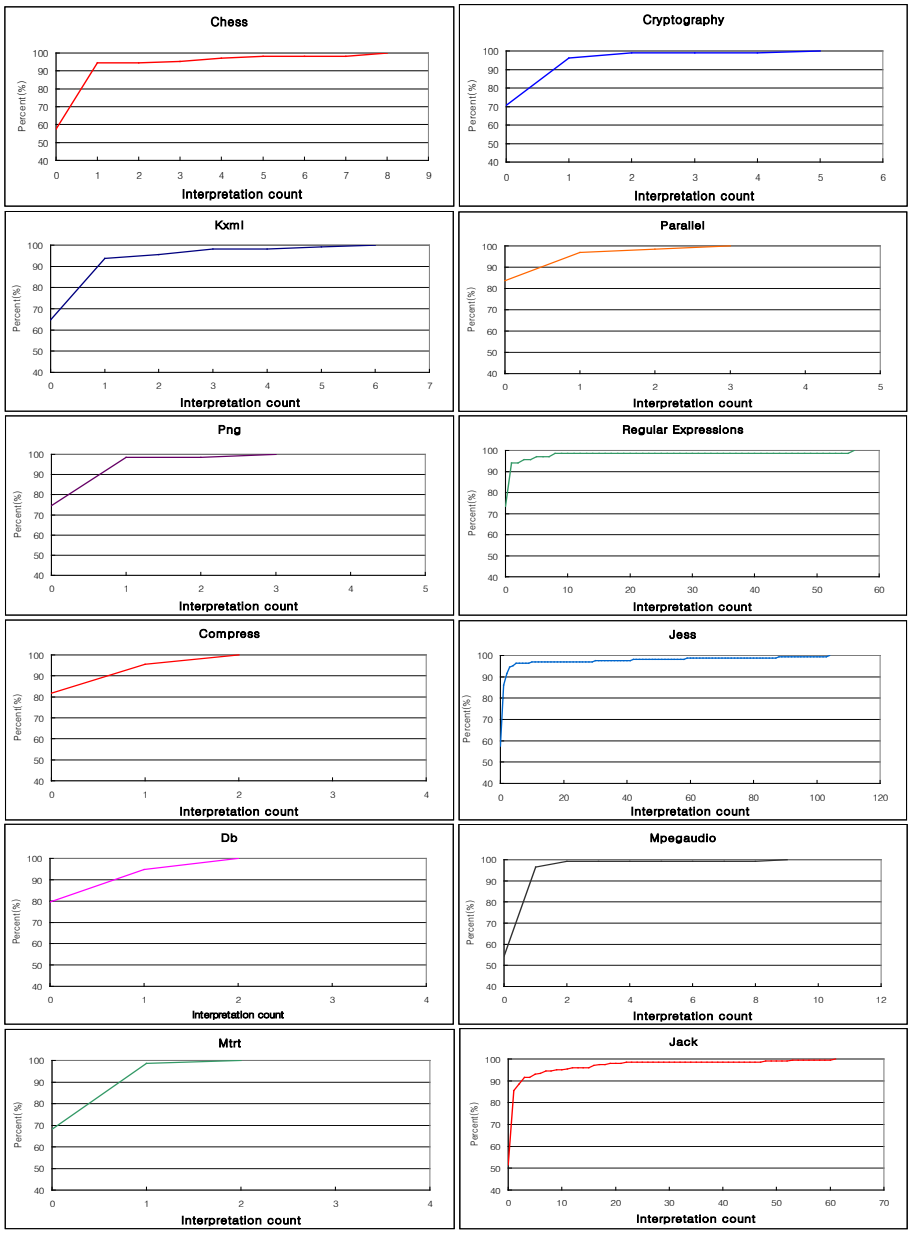


Figure 7-3 Cumulative percentages of methods which succeed in relocation as a function of the interpretation count

7.3 Space Overhead of c-AOTC

Table 7-2 depicts the cumulative size of each part of .aotc files for each benchmark. On average, 71% and 28% of files are used for storing machine code and relocation information, respectively, and the header part takes a tiny portion.

Table 7-2 Sizes of .aotc files

programs	Header Parts (KB)	Body Parts		
		Machine Code (KB)	Relocation Information	
			Inlining Tables (KB)	Relocation Entries (KB)
chess	4.4	403.5	5.2	139.9
crypto	4.3	339.2	3.2	123.4
kxml	4.4	418.7	4.7	160.3
parallel	2.7	173.7	2.4	65.3
png	2.7	233.5	2.6	89.3
regex	2.8	208.0	2.6	76.0
compress	1.7	111.4	1.1	39.6
jess	6.4	745.2	6.4	309.4
db	3.0	163.8	2.5	62.6
mpegaudio	5.6	504.7	5.0	177.8
mtrt	6.3	414.7	5.6	147.1
jack	7.8	479.9	7.0	194.4
Average	4.4 (0.9%)	349.7 (71.3%)	4.0 (0.8%)	132.1 (26.9%)

We measured how many instructions in the machine code require relocation, which is shown in Table 7-3. It shows that an average of 12% of instructions requires relocation. We need to optimize the data structures for relocation entries in order to reduce the space overhead further.

Table 7-3 Number of Relocated Instructions

programs	Total Number of instructions in .aotc files (thousands)	Number of relocated instructions (thousands)	[b/a] (%)
chess	103.3	11.9	11.6
crypto	86.8	10.5	12.1
kxml	107.2	13.7	12.8
parallel	44.5	5.6	12.5
png	59.8	7.6	12.8
regex	53.2	6.5	12.2
compress	28.5	3.4	11.9
jess	190.8	26.4	13.8
db	41.9	5.3	12.7
mpegaudio	129.2	15.2	11.7
mtrt	106.2	12.5	11.8
jack	122.9	16.6	13.5
Average (%)			12.6

Figure 7-4 shows the ratio of the space overhead of all .aotc files compared to the original static size of the system, which includes the CVM binary, the CVM library, and the benchmark class files. The graph indicates that on average around 10% of additional persistent storage is needed for exercising the c-AOTC.

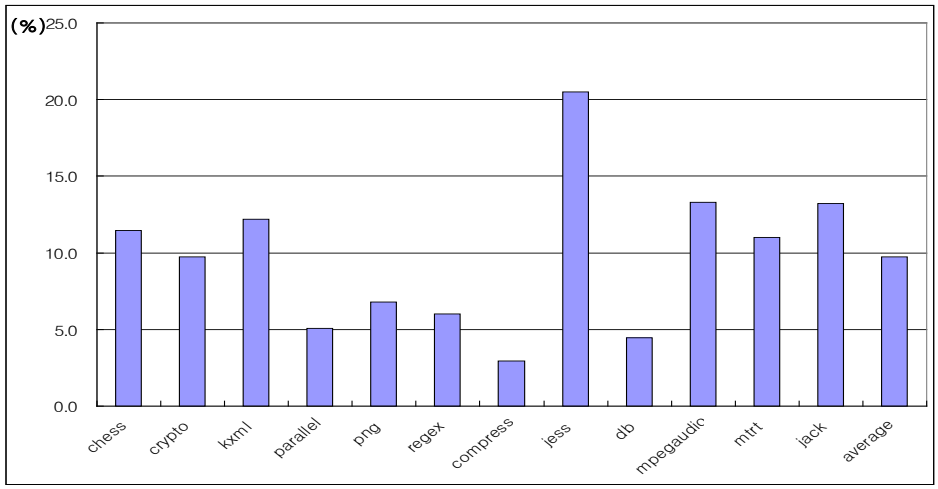


Figure 7-4 Static space overhead of .aotc files compared to the original static size

7.4 Reducing Number of c-AOTC Methods

If the persistent memory space is tight, we can save only parts of the JITC methods. The problem is which methods we should save such that the performance degradation due to this is minimized while maximizing the reduction of space, achieving graceful degradation of performance. If we assume that all of our JITC methods saved will be JITCed anyway in the loading mode, which is true when we use the same input with the same CVM, we can achieve this goal by saving those methods whose interpretation and JITC overhead is high and whose machine code size is small.

In our experiments, we estimated the interpretation and JITC overhead of a method by its bytecode size since the JITC overhead generally increases proportional to the bytecode size (i.e., we ignored the interpretation overhead in this heuristic). We sort the JITC methods with a decreasing order of their bytecode sizes

divided by their machine code sizes, and choose to save them in this order as much as the given space allows.

Figure 7-5 depicts the performance ratio of the saving mode and the loading mode (with interpretation) compared to the normal mode, when the space size is 95%, 90%, 85%, 80%, 75%, and 70% of the original space for each benchmark. The graph shows that the performance of the saving mode remains almost the same (since the saving of fewer methods would affect the running time little), while the performance ratio of the loading mode decreases, but not as sharply as the decrease of the space. The average performance ratio of the loading mode is 1.11 (100%), 1.10 (95%), 1.10 (90%), 1.08 (85%), 1.08 (80%), 1.07 (75%), and 1.07 (70%), respectively. This shows that our heuristic tends to allow relatively graceful degradation of performance as space gets tight.

The loading mode graph of some benchmarks (e.g., mpegaudio, db, regexp) shows a small glitch even when the space gets tighter, meaning that saving fewer JITC methods takes shorter running time. It is partly due to fluctuation, but we found that some method affects the loading mode performance negatively when it is saved. We do not exactly understand the reason, but it might be due to some cache anomaly.

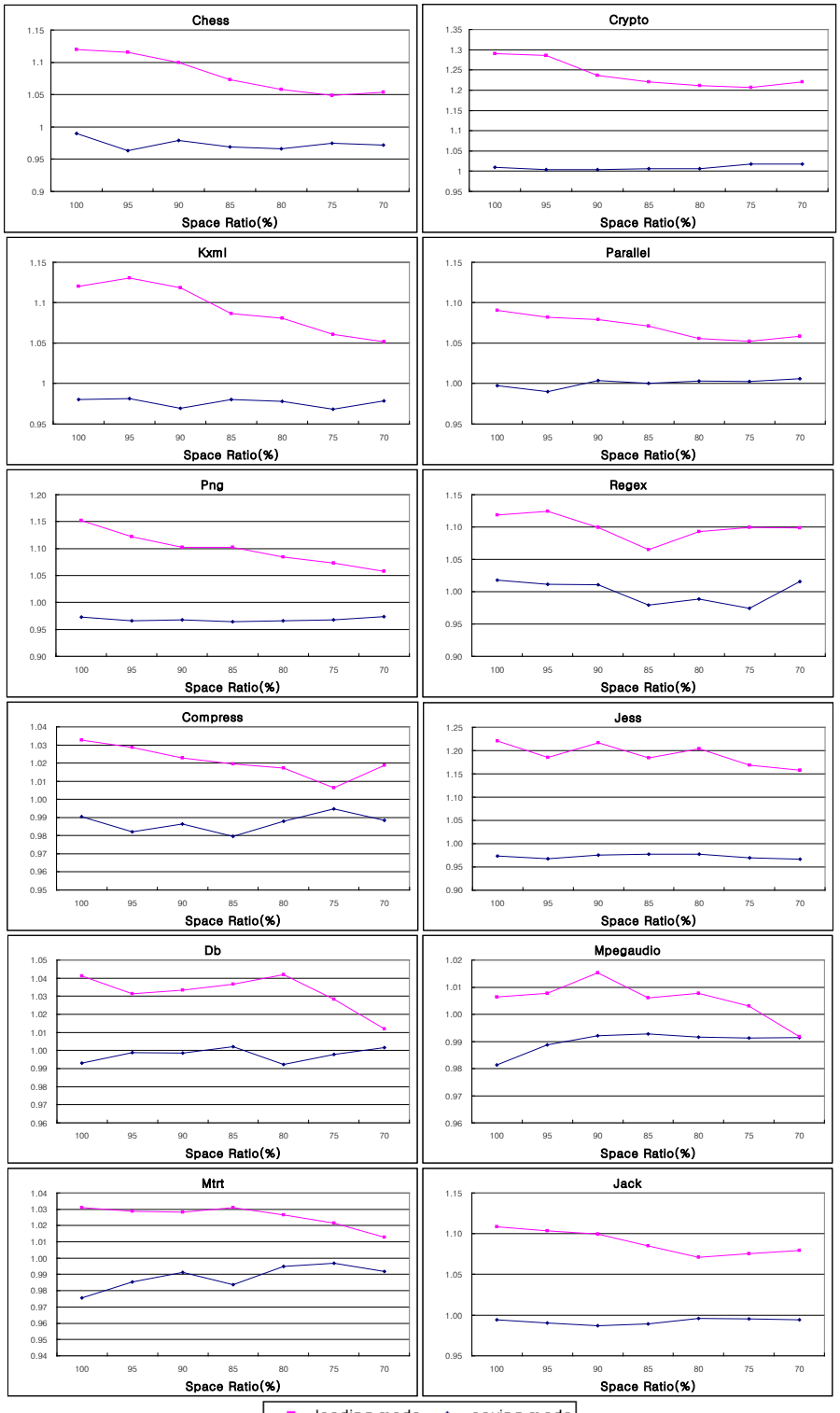


Figure 7-5 Performance impact of reducing JITC methods saved

7.5 c-AOTC with new hot-spot detection heuristics

We employ better hot spot detection heuristic than section 7.1. The previous heuristic estimates the running time of a method statically based on its code structure (e.g., loops), while the new one can estimate more precisely by dynamically counting only important bytecodes interpreted, but with a simple arithmetic calculation it can obtain the precise count of all interpreted bytecodes [37]. The number of JITC methods is reduced by half and many of those JITC methods are compiled earlier, achieving a better JITC performance. This will lead to a more rigorous evaluation of the c-AOTC since no bogus hot spot methods will be saved.

And we adopt the techniques of section 4.3.1 and section 4.3.2 for reducing file size in this c-AOTC implementation.

7.5.1 Performance Impact of c-AOTC with new hot-spot detection heuristics

For each benchmark, we ran three modes of execution. The first one is a normal execution mode based on interpretation and JITC. The second one is a saving mode where normal execution is performed but at the end of program execution, machine code and relocation information are saved in .aotc files (so its execution time includes the saving and compression overhead). The third one is a loading mode where interpretation is performed first as usual and when a method is to be JITCed, the machine code is loaded and executed if it is available (so its execution time includes the

relocation un-compression overhead).

Figure 7-6 and Figure 7-7 depicts the performance ratio of the saving mode and the loading mode compared to the normal mode, for EEMBC and SPECjvm, respectively. The saving mode suffers from performance penalty due to collecting relocation information and storing them with compressed machine code in the files, yet it is an average of 4%, which is small enough to make the saving mode still a practically competitive run mode.

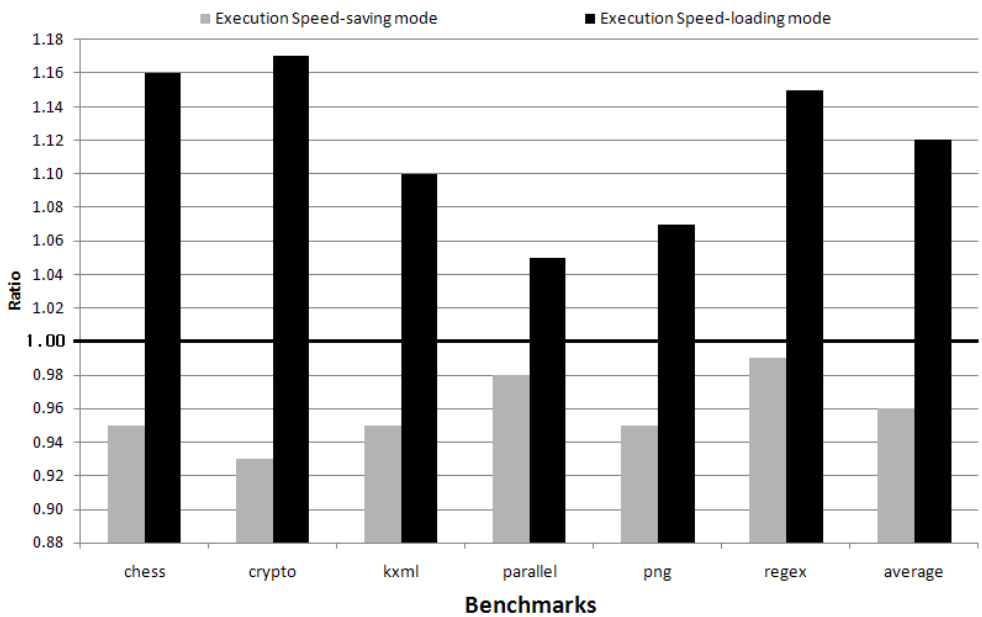


Figure 7-6 Performance ratio in EEMBC with new hot-spot heuristics

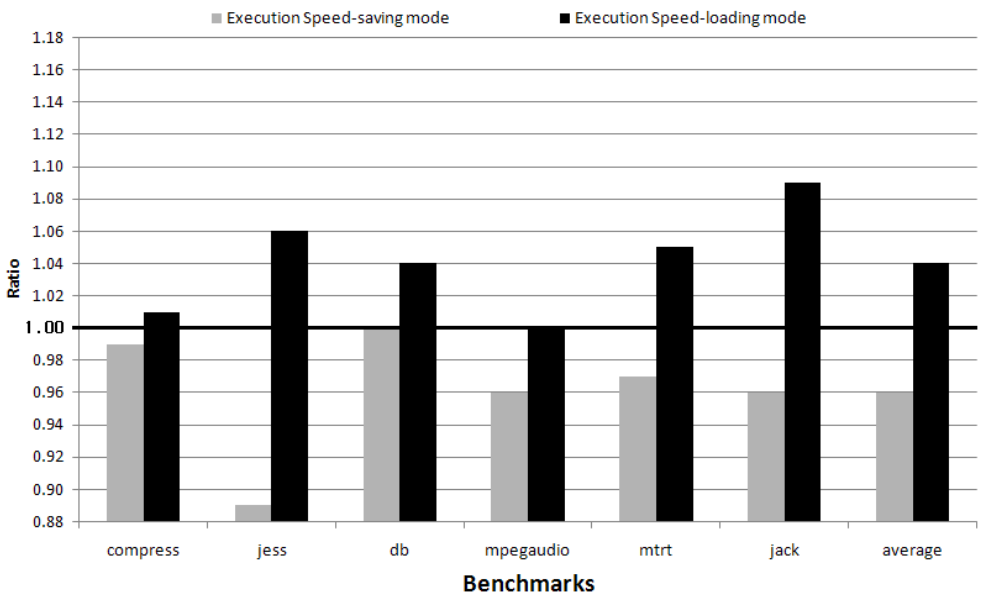


Figure 7-7 Performance ratio in SpecJVM98 with new hot-spot heuristics

The performance benefit of the loading mode is an average of 12% for EEMBC and 4% for SPECjvm98, which would roughly correspond to the JITC overhead in the normal mode. This is slightly less than the previous benefit since fewer methods are saved and un-compression overhead is added, yet the space overhead for the persistent memory is much smaller, as will be seen shortly. The higher benefit in EEMBC compared to the one in SPECjvm98 appears to be related to its shorter total running time. Table 7-4 shows the actual running time of the normal JITC mode and the loading mode, and their time differences. Here, EEMBC runs relatively shorter than SPECjvm98 while the time differences (i.e., the benefit of c-AOTC) are similar, which makes the benefit more

pronounced. In fact, embedded software tends to run relatively shorter than desktop software, so the benefit of c-AOTC in EEMBC would be more likely the case.

Table 7-4 The runtime of JITC-mode and Loading-mode and the difference of those with new hot-spot heuristics

Benchmark	JITC-Mode (s)	Loading-Mode (s)	Difference (s)
chess	18.8	16.2	2.6
crypto	8.8	7.5	1.3
kxml	19.4	17.7	1.7
parallel	10.6	10.1	0.5
png	8.3	7.7	0.6
regex	13.2	11.5	1.7
EEMBC average	**13.2**	**11.8**	**1.4**
compress	25.1	24.8	0.3
jess	14.9	14.0	0.8
db	13.5	13.0	0.5
mpegaudio	190.5	189.8	0.7
mtrt	111.5	105.8	5.6
jack	29.6	27.1	2.5
SpecJVM98 average	**64.2**	**62.4**	**1.7**

Table 7-5 shows the total number of executed methods and JITC methods, respectively, with the number of files generated. Around 6.2% of executed methods are JITCed (it was 11.7% in Table 7-1) and an .aotc file contains an average of 2.7 methods in it.

Table 7-5 Total number of executed methods, JITCed methods and JITCed files with new hot-spot heuristics

Programs	Total # of executed methods [a]	Total # of JITC methods [b]	Total # of c-AOTC files [c]	b/a (%)	b/c
chess	968	51	20	5.3	2.6
crypto	962	44	11	4.6	4.0
kxml	953	65	22	6.8	3.0
parallel	884	26	14	2.9	1.9
png	897	20	9	2.2	2.2
regex	943	25	10	2.7	2.5
compress	759	14	9	1.8	1.6
jess	1171	73	34	6.2	2.1
db	776	47	27	6.1	1.7
mpegaudio	942	95	33	10.1	2.9
mtrt	907	107	29	11.8	3.7
jack	1012	135	38	13.3	3.6
Average	**931**	**59**	**21**	**6.2**	**2.7**

7.5.2 Space Overhead of c-AOTC with new hot-spot detection heuristics

Table 7-6 depicts the cumulative size of each part of .aotc files for each benchmark. On average, 94.9% of files are used for saving the machine code of methods (most are compressed), and inlining s and header parts take a tiny portion.

We measured how many instructions in the machine code require relocation, which is shown in Table 7-7 (each pair of LUI and ORI is counted as two instructions). It shows that an average of 47% of instructions requires relocation, so separate relocation information tables would be too large.

Table 7-6 Sizes of .aotc files with new hot-spot heuristics

programs	Header Parts (KB)	Body Parts			Total Size (KB)
		Machine code		Inlining Tables (KB)	
		Non-compressed methods (KB)	Compressed methods (KB)		
chess	2.4	0.8	51.6	2.3	57.2
crypto	2.1	2.0	39.4	0.9	44.4
kxml	3.1	2.5	60.2	3.0	68.8
parallel	1.3	1.0	10.5	0.5	13.3
png	0.9	0.4	22.6	0.7	24.7
regex	1.2	0.9	16.1	0.6	18.7
compress	0.7	0.9	7.9	0.3	9.7
jess	3.4	1.8	75.9	2.8	84.0
db	2.2	2.5	20.5	1.2	26.4
mpegaudio	4.5	2.1	113.3	4.0	123.8
mtrt	4.5	4.2	72.8	3.8	85.3
jack	6.2	8.3	65.1	4.3	83.9
Averages	**2.7 (5.1%)**	**2.3 (4.3%)**	**46.3 (86.9%)**	**2 (3.7%)**	**53.4**

Table 7-7 Number of Relocated Instructions with new hot-spot heuristics

programs	Total Number of instructions in .aotc files (thousands)	Number of relocated instructions (thousands)	[b/a] (%)
chess	47.2	18.6	39.4
crypto	56.4	16.4	45.1
kxml	53.1	26.8	50.5
parallel	5.5	2.7	49.1
png	19.2	9.9	51.6
regex	11.3	4.9	43.4
compress	5.1	2.0	39.2
jess	83.8	47.7	56.9
db	13.1	6.5	49.6
mpegaudio	108.6	48.6	44.8
mtrt	73.0	33.1	45.3
jack	52.2	27.6	52.9
Average (%)			**47.3**

Figure 7-8 shows the ratio of the space overhead of all .aotc files compared to the original static size of the system, which includes the CVM binary, the CVM library, and the benchmark class files. The first bar in the graph shows the ratio with the .aotc file format used in our experiments, which indicates that on average around 1% of additional persistent storage is needed for exercising the c-AOTC. The second bar shows the ratio with no compression and the third bar shows the ratio with no encoding as well, whose averages are 3.5% and 5.2%, respectively (it was 10% in Figure 7-4). This result indicates the space overhead of c-AOTC is much lower with encoding and compression, and seems to be reasonable enough.

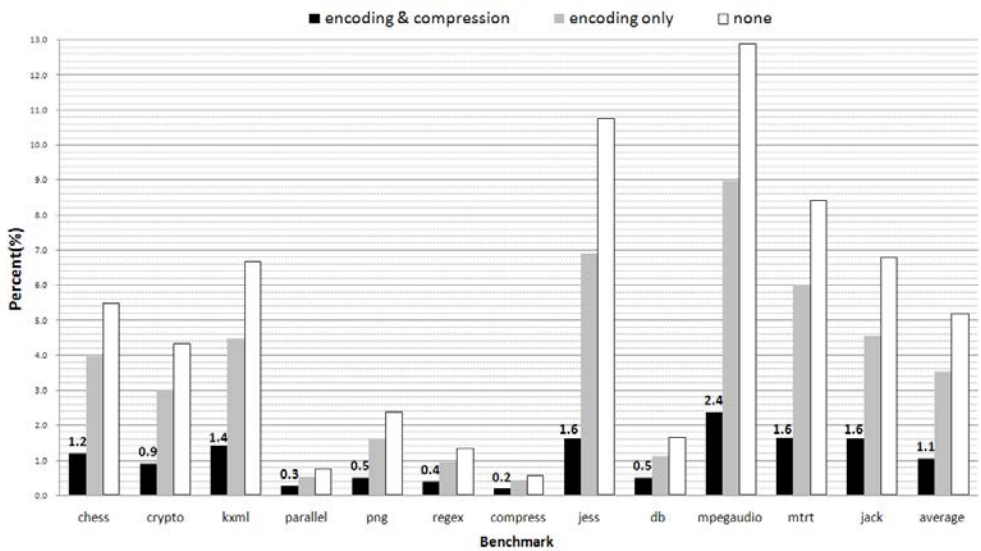


Figure 7-8 Static space overhead of .aotc files compared to the original static size with new hot-spot heuristics

7.6 c-AOTC of DTV JVM platform

Our target DTV set-top box includes a 333MHz MIPS CPU with a 128MB memory. Its software platform has the Sun's PhoneME Advanced MR2 version with advanced common application platform (ACAP) middleware, running on the Linux with kernel 2.6.12. There are three terrestrial TV stations in Korea, each of which broadcasts a different xlet application. We designate them as A, B, and C in this paper. Station A has news and weather menu item. Station B has news, weather traffic and stock menu item. Station C has news, weather and traffic menu items. We are primarily interested in the running time of displaying the chosen information on the TV screen when each menu item is selected using the remote control.

7.6.1 Performance result of DTV platform

Figure 7-9 and Figure 7-10 show the performance improvement of c-AOTC. Figure 7-9 shows the runtime of the xlet applications.

Figure 7-10 shows the runtime performance ratio of xlet application running by c-AOTC over running by JITC runtime.

In Figure 7-10, c-AOTC gets average 33% performance improvement. Two major causes of performance improvement of c-AOTC are removing the JITC overhead of hot-spot methods and removing the interpretation of c-AOTCed methods.

According to Figure 7-9 and Figure 7-10, c-AOTC does not decrease the performance because there is no extra overhead

caused by c-AOTC on runtime. The reasons of c-AOTC's overhead are saving the code cache and loading the code cache. Saving the code cache is happened after the xlet terminates. And loading the code cache is happened before the xlet starts. So the xlet user can not feel the overhead of c-AOTC when he watches the DTV and runs the xlet application.

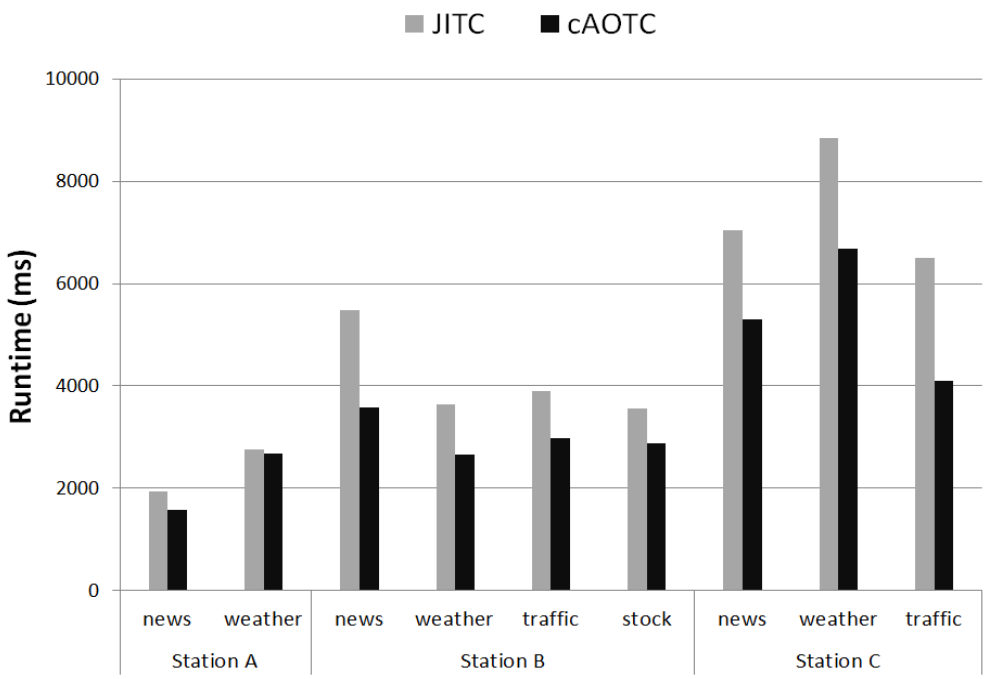


Figure 7-9 Runtime of xlet application on DTV(ms)

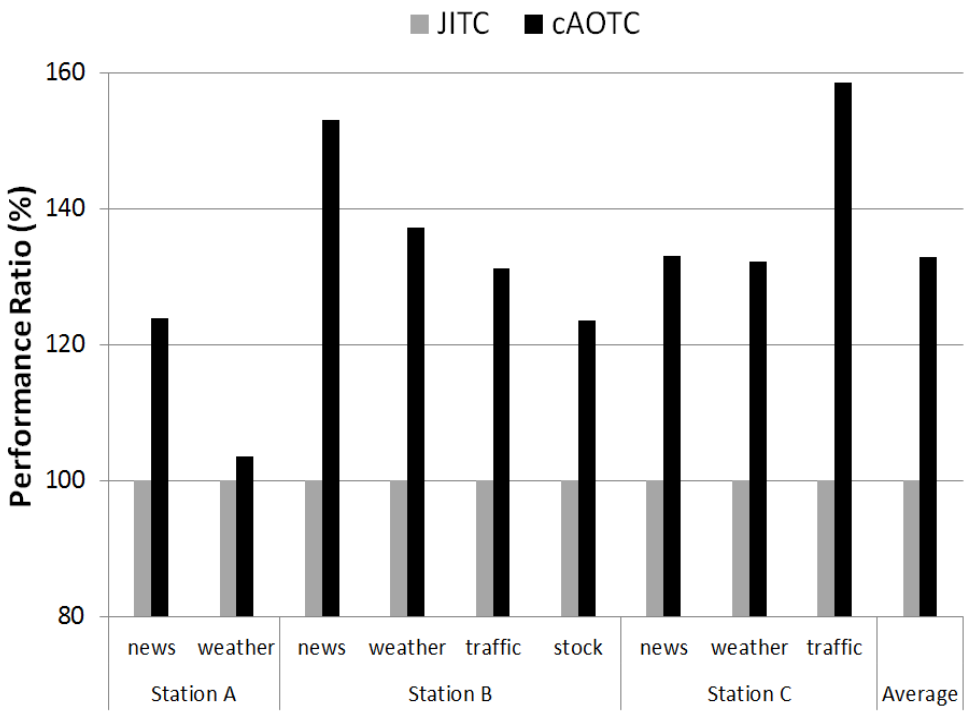


Figure 7-10 Runtime performance of xlet application on DTV(%)
(Running by JITC is 100 %.)

7.6.2 Analysis of JITCed method of DTV platform

Figure 7-12 shows the ratio of JITCed methods. In Figure 7-11 and Figure 7-12, blue bar is the JITCed methods included in system/middleware classes and red bar is the JITCed methods included in xlet application classes.

In Figure 7-12, JITCed methods of xlet application classes are average 3% of total JITCed methods. So the overhead to compile the methods of xlet classes is small. And the possible benefit to adopt c-AOTC to the methods of xlet classes is also small. But the overhead to relocate these un-romized and non-fixed methods is very big compared the benefit. So, we decided not to re-use the

methods of xlet classes.

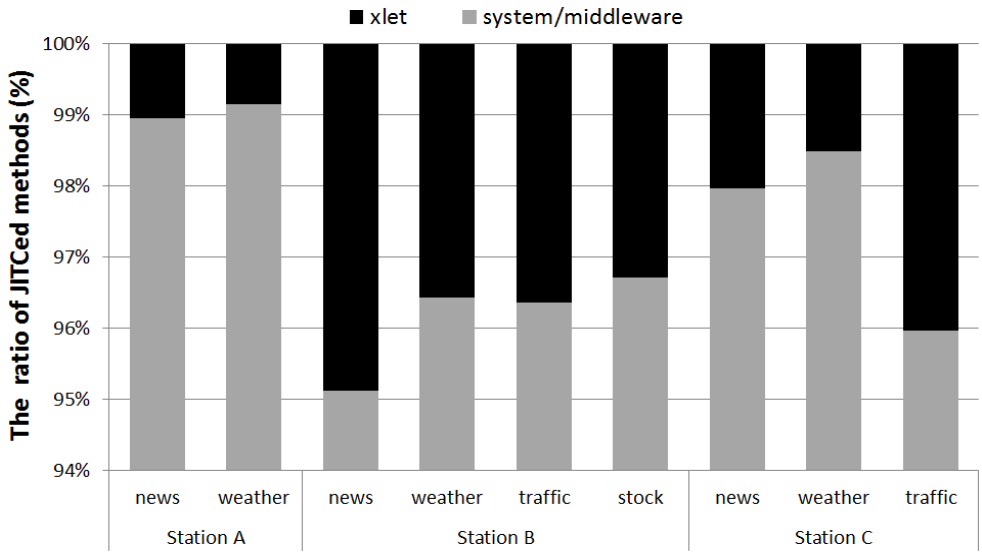


Figure 7-11 Number of JITCed methods on DTV

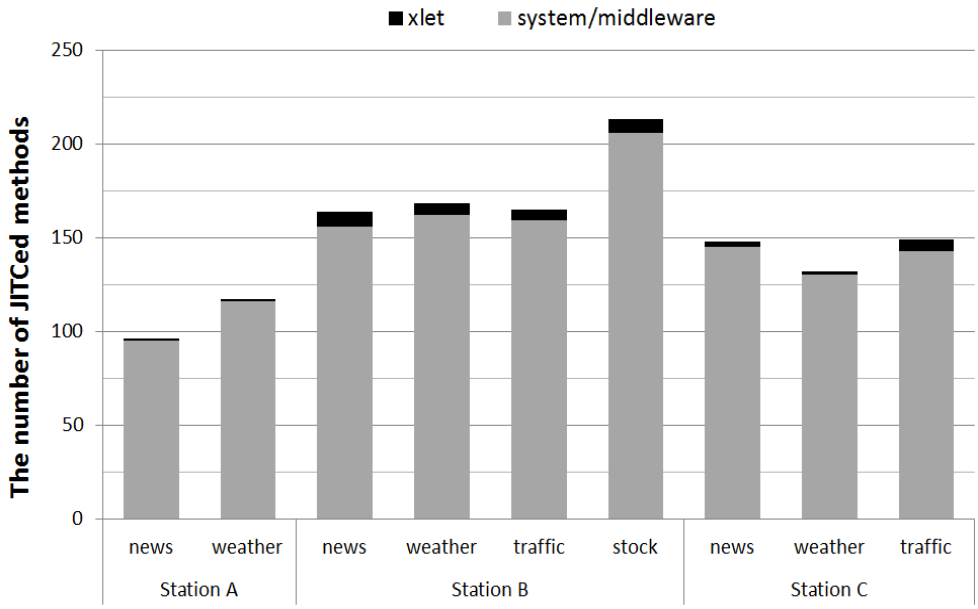


Figure 7-12 Ratio of JITCed methods on DTV

7.6.3 Space overhead of DTV platform

Figure 7-13 shows the c-AOTC file size of each xlet applications. The size is the sum of two stuffs. One is the saved machine code and the other is some information for c-AOTC (e.g. the size information of code cache, the location of xlet class and so on). The size of VM binary in DTV is 70 Mbytes and the size of saved c-AOTC file is just 0.5% of the VM binary. Because current PhoneME JVM includes the system and middleware classes and their romized data, the size of VM is big.

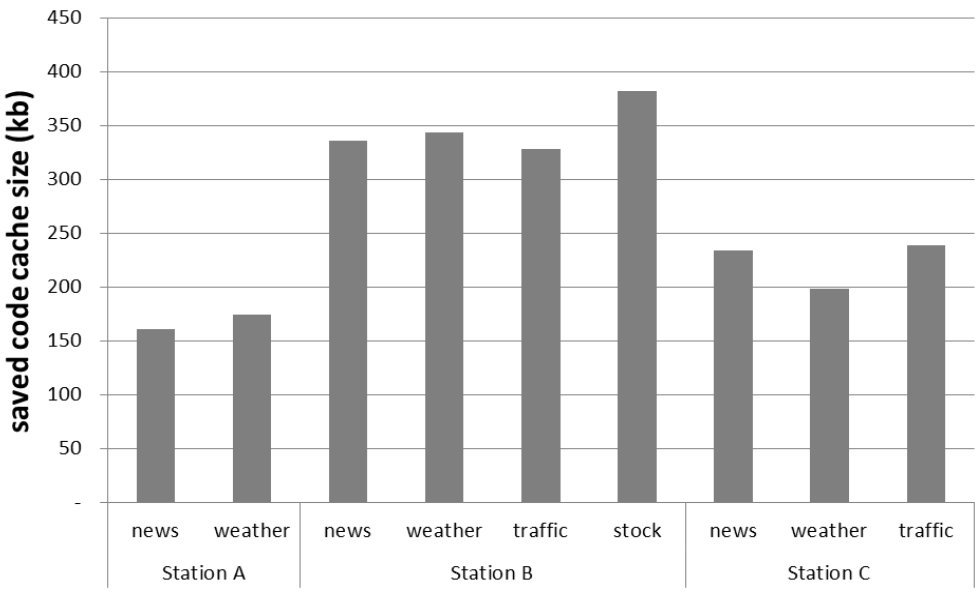


Figure 7-13 Size of saved c-AOTC file on DTV

7.6.4 c-AOTC overhead of DTV platform

Figure 7-14 shows the saving time which c-AOTC saves the code cache of xlet application to c-AOTC file and the loading time which c-AOTC reads the file and load it to code cache in memory.

The saving time and loading time of Figure 7-14 is directly proportional to the size of c-AOTC file of Figure 7-13. The saving time takes average 400ms and the loading time takes average 60ms.

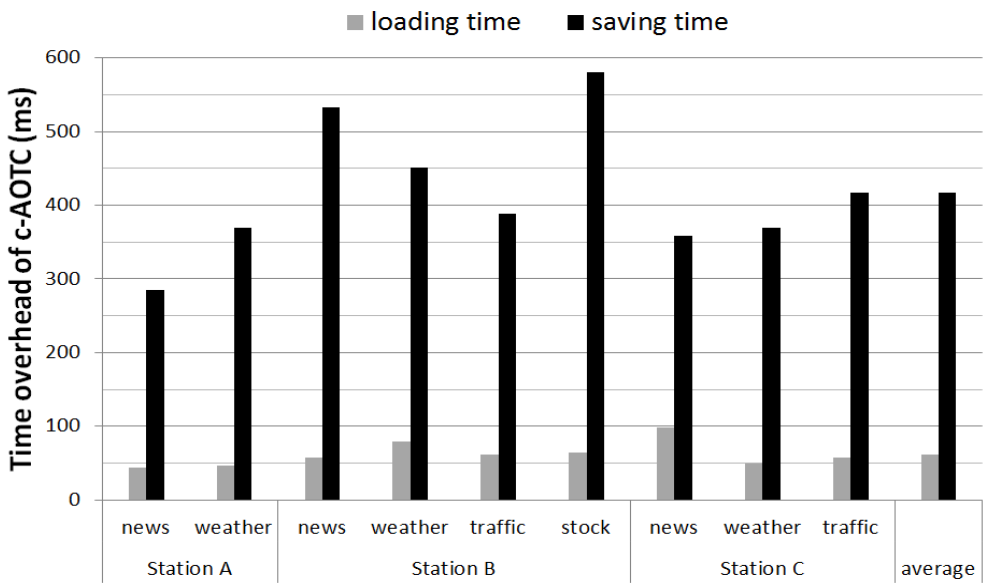


Figure 7-14 The overhead of c-AOTC on DTV (loading time & saving time) (ms)

7.6.5 c-AOTC performance using different xlet's c-AOTC file in DTV platform

The c-AOTC file includes the machine code of system/middleware classes. The machine code itself can be used for every xlet because the xlets share the system/middleware of DTV.

So we experiment our xlets with 6 c-AOTC files made by 3 station's news and weather xlet.

Table 7-8 shows the experimental result. In table 7-8, 100% means the JITC-only execution performance. And the italic bold number is the experiment using its own c-AOTC file. The average performance improvement is about 20%~27%. It is good performance but worse than the c-AOTC using its own c-AOTC file. But some experiment (e.g. Station A news using Station B weather c-AOTC file) shows better performance than using own c-AOTC file. This is lucky one because the Station B weather c-AOTC file includes not only the hot spot methods of Station A news but also used, not hot methods. So in this case, Station A news can remove the JITC overhead and some extra interpretation. But usually the experiments show worse performance like Station C weather using Station B weather c-AOTC file.

Table 7-8 Performance ratio of xlet when it uses another c-AOTC files on DTV (%)

			used c-AOTC file					
			Station A		Station B		Station C	
			news	weather	news	weather	news	weather
xlet	Station A	news	***123.9***	122.9	126.7	128.1	117.5	119.6
		weather	105.7	***103.6***	103.1	110.3	102.3	99.6
	Station B	news	145.7	151.2	***153.1***	160.8	142.1	154.2
		weather	118.3	117.8	118.4	***137.2***	123.4	116.0
		traffic	112.9	117.1	116.5	123.2	124.8	125.8
		stock	127.1	123.3	123.0	125.2	121.0	118.5
	Station C	news	123.7	125.2	128.6	129.6	***133.1***	129.0
		weather	123.4	101.3	108.9	102.3	131.1	***132.3***
		traffic	114.6	111.4	124.1	126.3	142.4	118.8
Average			121.7	119.3	122.5	127.0	126.4	123.7

Figure 7-15 show the graph of each xlet item of Table 7-8. In station B traffic, stock and station c traffic, the graphs include the result of using its own c-AOTC file. And in the average result, it includes the average result of Figure 7-10.

In Figure 7-15, the red bar means the result which uses its own c-AOTC file.

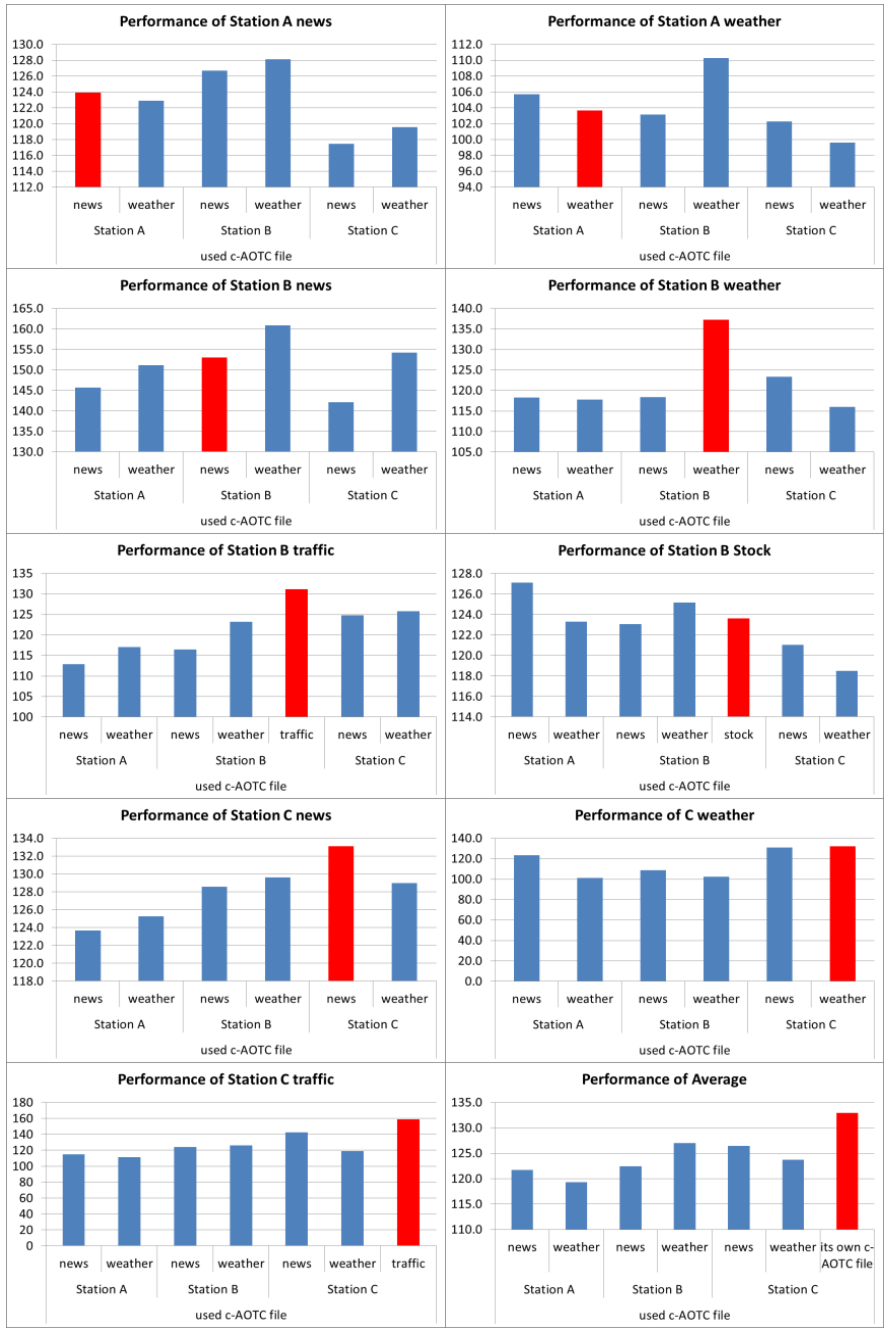


Figure 7-15 The performance ratio of xlet when it uses another c-AOTC files on DTV (%)

7.7 c-AOTC of V8 JavaScript engine

Our target board includes a cortex-A8 ARMv7 1GHz CPU with a 256MB RAM and 4GB class2 Flash memory as persistent memory. Its software platform has the V8 engine version 3.4.9 running on the Ubuntu Linux with kernel version 2.6.31. The benchmarks we used are SunSpider benchmarks [38] and v8 benchmarks [39].

We take running time by three test sets – Base run, saving mode and loading mode. Base run is original run using V8 default JITC. Saving mode is storing machine code after running original JITC-based run. In loading mode, VM loads the saved machine code and executes it.

.

7.7.1 Compilation overhead on V8 JavaScript VM

The compilation overhead of V8 JavaScript VM is two parts. One is parsing phase overhead to parse the source code to AST. And the second is machine code generation overhead to translate AST to machine code. The goal of c-AOTC implementation in this paper is to remove the machine code generation overhead.

Figure 7-16 shows the generation phase overhead of c-AOTC. In this figure, 100% is the generation overhead of base run. In saving mode, there is an overhead to make unique id of internal object. In loading mode, there is smaller generation overhead than base run because c-AOTC skip the part of machine code generation.

The remaining overhead of loading mode is the time recreating internal object assessed by loaded machine code. The additional

overhead of saving mode is average 58% and the reduced overhead of loading mode is average 30%.

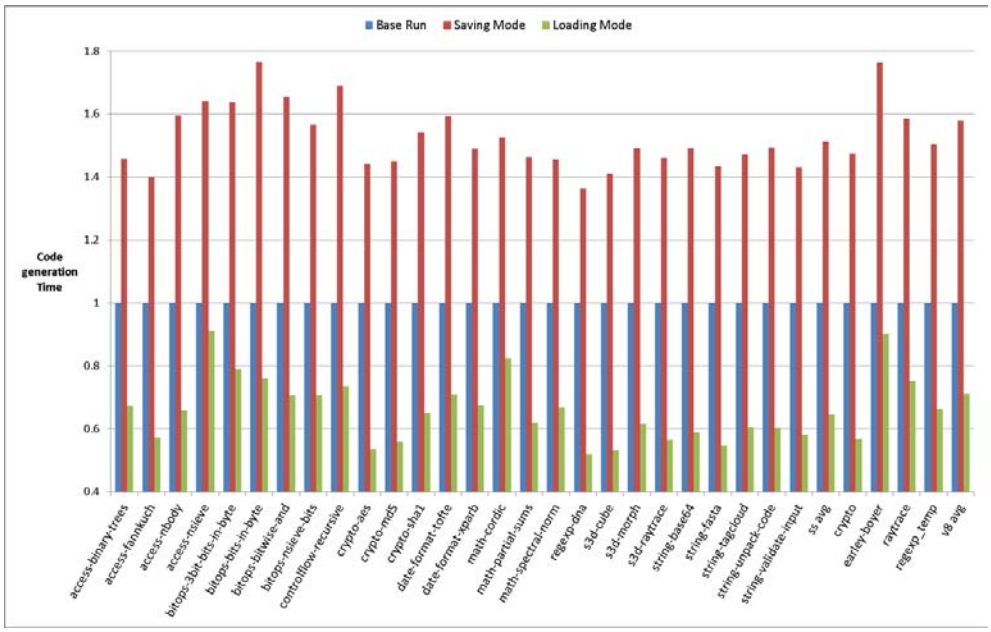


Figure 7-16 Generation overhead on JavaScript VM

Figure 7-17 shows total compilation overhead containing generation overhead of Figure 7-16 and parsing overhead. In this figure, 100% is the generation overhead of base run. In our experiments, parsing overhead and generation overhead is similar. And there is a little change in parsing overhead. So, the additional compilation overhead of saving mode is average 30% and the reduced compilation overhead of loading mode is average 20%.

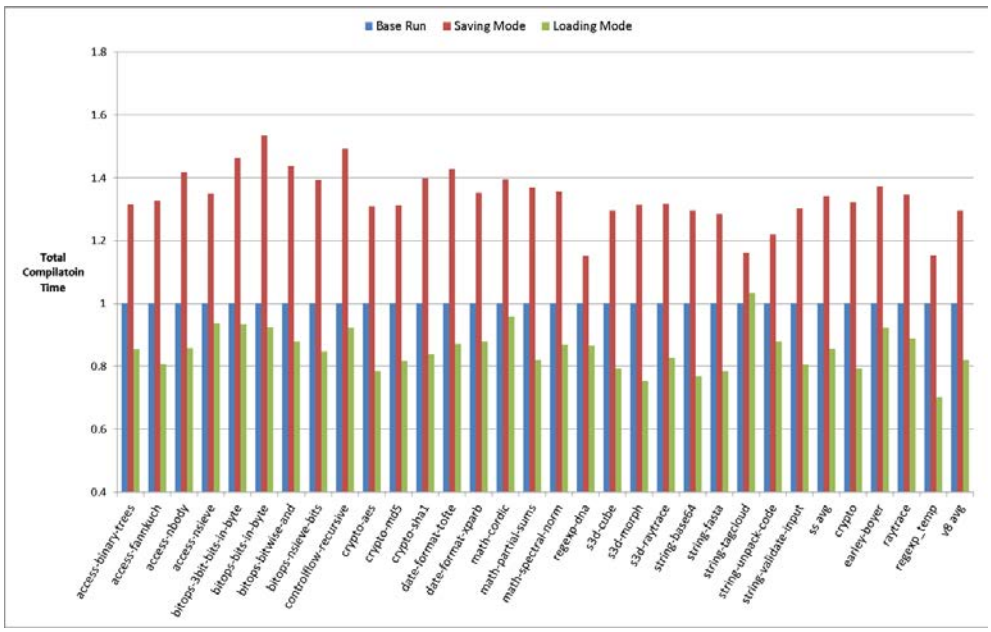


Figure 7-17 Compilation overhead on JavaScript VM

7.7.2 Performance result on V8 JavaScript VM

Figure 7-18 shows the running time of benchmarks without file I/O time. And Figure 7-19 shows the running time of benchmarks.

For ascertaining the effect of compilation overhead changes in running time, we show not only Figure 7-19 but also Figure 7-18.

In Figure 7-18, the running time of saving mode is increased by average 3% in sunspider benchmarks and average 6% in v8 benchmarks. And the running time of loading mode is reduced by average 1% in sunspider benchmarks and average 2% in v8 benchmarks. Because the compilation time is about 10% of total running time, the results of Figure 7-18 is proportional to Figure 7-17. But in Figure 7-19, the running time of saving mode is increased by average 15% and the running time of loading mode is increased by average 3%. This is because the benefit of c-AOTC is

smaller than file I/O overhead of slow flash memory. The degradation in Figure 7-19 is not big problem. We have similar experience when we developed c-AOTC on JVM platform at first. The degradation by big file I/O can be easily changeable by updating hardware. If we change the flash memory to faster one, the slowdown by file I/O will be dramatically reduced.

The small effect of c-AOTC in Figure 7-18 is real problem. The compilation overhead of benchmarks is average 10%. But, our c-AOTC just has small effect. We will discuss about this in 7.7.3 compared with c-AOTC in another JavaScript VM.

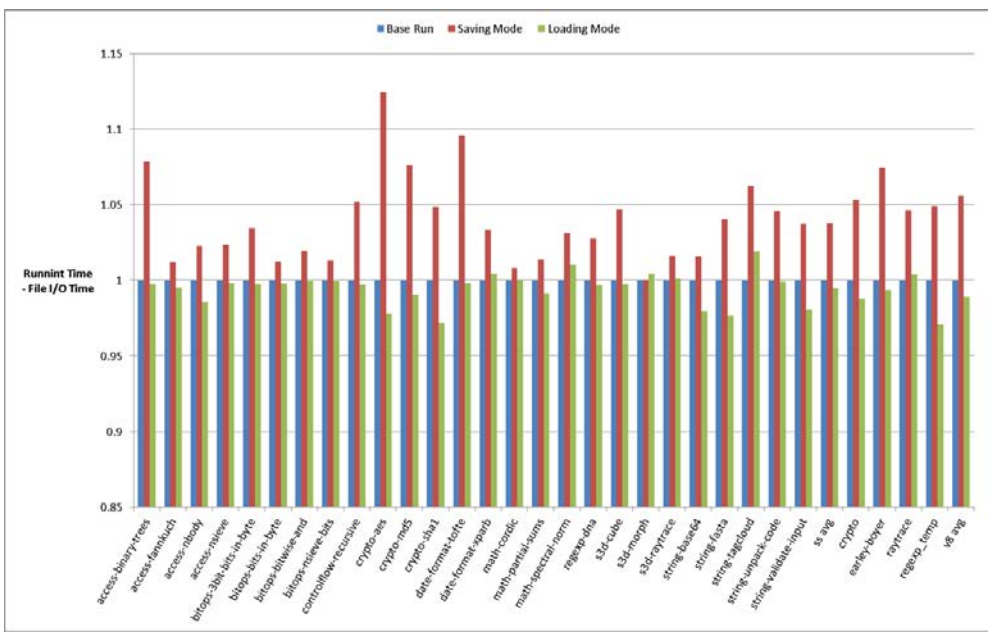


Figure 7-18 Running time without file I/O overhead on JavaScript VM

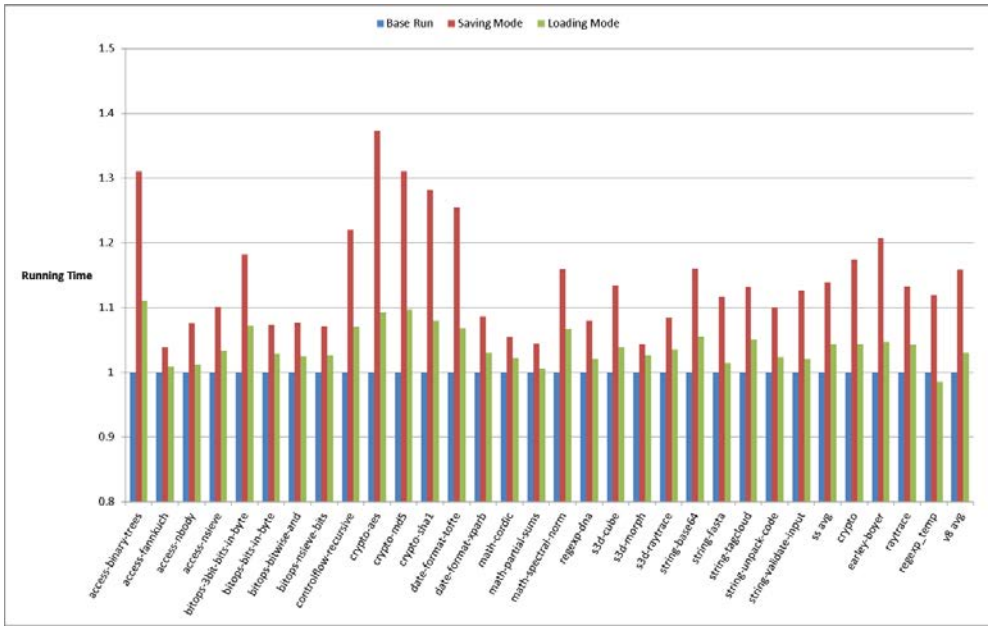


Figure 7-19 Running time with file I/O overhead on JavaScript VM

In Figure 7-19, regexp_temp benchmark has 1.5% improvement than base run. This is because regexp_temp has some big method. This is similar with the characteristics of web apps and RIA. The other benchmarks do not have big method, just have many small methods.

7.7.3 Comparison with c-AOTC of JavaScriptCore VM

Our lab developed c-AOTC on another JavaScript VM which is JavaScriptCore (JSC) engine of WebKit. Figure 7-20 shows that c-AOTC of JSC has average 9% performance improvement.

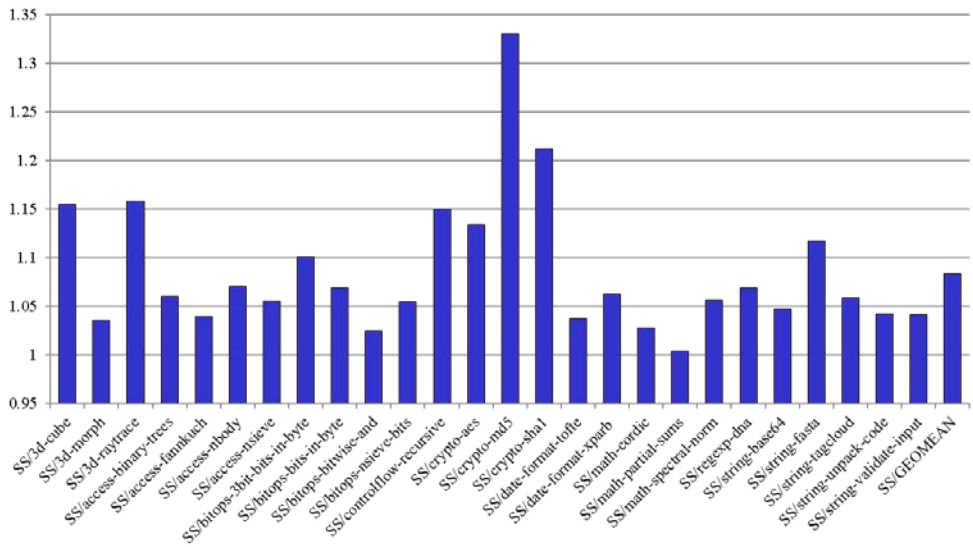


Figure 7-20 Performance result of JavaScriptCore VM

There is big difference of performance improvement between V8 c-AOTC and JSC c-AOTC. This is because V8 c-AOTC can skip only some part of generation phase but JSC c-AOTC can skip all compilation phases. The fundamental reason why c-AOTC cannot adapt to V8 is the number of internal objects. Figure 7-21 shows the number of internal objects in V8 and JSC. V8 creates average 15 times more internal objects than JSC VM. In V8 VM, the most part of compilation overhead is time to create internal objects which must be recreated during loading mode of c-AOTC.

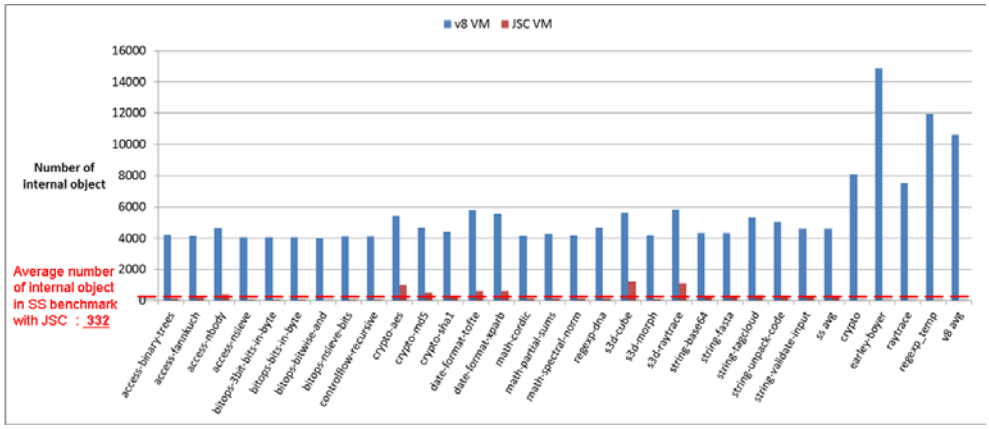


Figure 7-21 The number of internal objects of V8 and JSC

Chapter 8 Related Work

The idea of caching JITC binaries between runs is not new. Many previous works were done in the context of server JVMs.

Quicksilver is a quasi-static compiler developed for the IBM's Jalapeno system having baseline JITC and optimization JITC [8]. It saves all JITC methods in the QSI files at the end of execution and loads them after stitching (i.e., relocation) when they are used later, exactly as our scenario in this paper. However, since it is based on server systems, space overhead is not an issue at all. For example, when optimized, hotspot JITC methods of a class are saved at the end of execution, baseline, cold-spot JITC methods are also saved, after being recompiled by the optimization JITC. In our case, only hotspot methods are compiled and saved by c-AOTC. Also, Quicksilver saves additional data in files such as exception tables and GC maps, creating larger files than ours.

A follow-up work of Quicksilver attempts to obviate the stitching process using an indirection table [40]. The QSI files are generated in such a way that the code which needs stitching is modified to load values from the indirection table. So, the code in QSI files can be used without any stitching and we simply need to provide an indirection table with new address or offset values for a new run. This can contribute to reducing the runtime memory overhead by sharing the same code among multiple JVMs in a server environment or directly executing the QSI code in the ROM without copying to RAM in an embedded environment.

The .NET platform of Common Language Runtime VM employs a JITC traslating MSIL (MS intermediate language) into machine code [41]. It is possible to invoke the JITC offline so as to compile ahead-of-time. This JITC-based AOTC can save the JITC overhead in a server environment, as Quicksilver can.

US patent 6738969 proposes evicting the least-used JITC methods if the runtime memory space is not enough [21]. A similar code unloading is also proposed in [42].

There is a research proposal to save the profile of previous runs of Java programs, not the machine code, in order to improve the performance of future runs [43, 44]. The previous profile together with the current profile data can be used to decide which method to compiler at which time to allow hot spot methods to be compiled earlier and to keep bogus hot spot methods from being compiled. This idea can be complementary with the c-AOTC idea.

Some Issues of AOTC related to inlining and field accesses for meeting the real-time specification for Java are discussed in [45]. They argue that inlining and field accesses bounded early in AOTC may be invalidated at runtime if there are changes of method implementations or field orders in the meantime. Although our c-AOTC may also have similar issues, embedded systems are often supposed to perform the validation of existing downloaded Java classes before using them, so validation of the .aotc files can also be performed in the same context and JITC can be done from scratch if the validation fails.

A hybrid environment composed of JITC and AOTC is built in [20],

but it could not achieve a performance that would be normally expected, primarily due to the call overhead between JITC methods and AOTC methods; it employs a bytecode-to-C AOTC which is based on the C-stack for parameter passing, but JITC is based on the Java stack, so there is an additional overhead for cross-compiled calls. There is no such an overhead between c-AOTC methods and JITC methods, since both are generated by the same JITC based on the Java stack.

There is code reuse model designed for WebKit engine in desktop environment. [46] It saves native code generated and re-uses it.

JITC uses heavy memory usage for generating machine code. The process translating bytecode to IR and optimizing the IR introduce bigger peak memory overhead of VM [47]. Because our c-AOTC removes the process of JITC, the peak memory overhead of VM can be removed.

Chapter 9 Conclusion

Recently Virtual Machine (VM) is used as platform of app-downloading system because VM has advantage in portability and independency. But slow performance is one critical disadvantage of VM. For improving performance, VM uses JITC which is the dynamic compilation technique. But in real world application, there are few hot-spot method, many warm-spot method. This cause bigger dynamic translation overhead and less benefit executing the machine code. So, c-AOTC can improve the performance by executing the machine code without translation overhead.

In Java platform, c-AOTC improves JITC performance more than 10% for benchmarks. And we adopt our c-AOTC approach to commercial DTV platform and test the real xlet applications of commercial broadcasting stations. Our c-AOTC gets average 33% performance improvement on the real xlet application test.

As mentioned, the real-world application suffers dynamic compilation overhead more than test benchmarks. So we get better performance improvement in real-world.

In v8 JavaScript platform, it is hard to skip all compilation phases because of creating many internal objects. But clearly, we can reduce the compilation overhead. As described, c-AOTC cannot get good performance improvement in V8 engine, but it can get good performance improvement in WebKit JavaScriptCore engine. This difference is due to the design and implementation of VM internal. V8 VM makes more internal objects than JSC VM.

Our c-AOTC approach is a good approach removing compilation overhead for App-Downloading systems. But we can see that the degree of effect of c-AOTC is dependent to the design and implementation of VM and its JITC.

Bibliography

[1] Sun Microsystems, CDC: An Application Framework for Personal Mobile Devices, White Paper, Sun Microsystems.

[2] John Aycock. 2003. A brief history of just-in-time. ACM Comput. Surv. 35, 2 (June 2003), 97-113. DOI=10.1145/857076.857077 http://doi.acm.org/10.1145/857076.857077

[3] A. Krall. 1998. Efficient JavaVM Just-in-Time Compilation. In Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98). IEEE Computer Society, Washington, DC, USA, 205-.

[4] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. 1998. Fast, effective code generation in a just-in-time Java compiler. SIGPLAN Not. 33, 5 (May 1998), 280-290. DOI=10.1145/277652.277740 http://doi.acm.org/10.1145/277652.277740

[5] SungHyun Hong, Jin-Chul Kim, Jin Woo Shin, Soo-Mook Moon, Hyeong-Seok Oh, Jaemok Lee, and Hyung-Kyu Choi. 2007. Java client ahead-of-time compiler for embedded systems. SIGPLAN Not. 42, 7 (June 2007), 63-72. DOI=10.1145/1273444.1254776 http://doi.acm.org/10.1145/1273444.1254776

[6] Sunghyun Hong, Jin-Chul Kim, Soo-Mook Moon, Jin Woo Shin, Jaemok Lee, Hyeong-Seok Oh, and Hyung-Kyu Choi. 2009. Client ahead-of-time compiler for embedded Java platforms. Softw. Pract. Exper. 39, 3 (March 2009), 259-278. DOI=10.1002/spe.v39:3 http://dx.doi.org/10.1002/spe.v39:3

[7] Sunghyun Hong and Soo-Mook Moon. 2014. Client-Ahead-Of-Time Compilation for Digital TV Software Platform. 3rd workshop on: Dynamic Compilation Everywhere (Jan 2014).

[8] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. 2000. Quicksilver: a quasi-static compiler for Java. In Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00). ACM, New York, NY, USA, 66-82. DOI=10.1145/353171.353176 http://doi.acm.org/10.1145/353171.353176

[9] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. 1997. Toba: java for applications a way ahead of time (WAT) compiler. In Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3 (COOTS'97), Vol. 3. USENIX Association, Berkeley, CA, USA, 3-3.

[10] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. 1997. Harissa: a flexible and efficient java environment mixing bytecode and compiled code. In Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3 (COOTS'97), Vol. 3. USENIX Association, Berkeley, CA, USA, 1-1.

[11] Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E. Andrew Johnson, Vania Joloboff, Fred Roy, Fridtjof Siebert, and Xavier Spengler. 1998. TurboJ, a Java Bytecode-to-Native Compiler. In Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98), Frank Mueller and Azer Bestavros (Eds.). Springer-Verlag, London, UK, UK, 119-130.

[12] Sun Microsystems, Porting Guide Connected Device Configuration and Foundation Profile version 1.0.1 Java 2 Platform Micro Edition, May 2002.

[13] Chandra Krintz. 2003. Coupling on-line and off-line profile information to improve program performance. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '03). IEEE Computer Society, Washington, DC, USA, 69-78.

[14] Interactive TV Web, http://www.interactivetvweb.org

[15] Sun Microsystems, Java ME Mobile Information Device Profile(MIDP), http://java.sun.com/products/midp

[16] Blu-ray Disc Association, http://www.blu-raydisc.com

[17] Blu-ray Disc Association, BD-J Baseline Application and Logical Model Definition for BD-ROM, White Paper, March 2005.

[18] Sun Microsystems, Java ME Java Technology for the Wireless Industry (JTWI), JSR 185, http://java.sun.com/products/jtwi

[19] Sun Microsystems, Java ME Technology - Mobile Service Architecture, http://java.sun.com/javame/technology/msa

[20] Hyeong-Seok Oh, Soo-Mook Moon and Dong-Heon Jung. 2008. On hybrid compilation environment for embedded systems. In Proceedings of the 12th Workshop on Interaction between Compilers and Computer Architectures (Interact-12).

[21] Lars Bak, Jacob R. Andersen, Kasper V. Lund. 2001. Non-intrusive Gathering of Code Usage Information to Facilitate Removing Unused Compiled Code, US Patent, 6738969.

[22] Sun Microsystems, Java ME Connected Device Configuration (CDC), http://java.sun.com/products/cdc

[23] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Addison-Wesley, 1996.

[24] GNU, GNU Compiler for the Java programming language, http://gcc.gnu.org/java

[25] M. Stoodley, K. Ma, and M. Lut, Real-time Java, Part 2: Comparing compilation techniques, http://www.ibm.com/developerworks/java/library/j-rtj2/index.html

[26] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive optimization in the Jalapeño JVM. SIGPLAN Not. 35, 10 (October 2000), 47-65. DOI=10.1145/354222.353175 http://doi.acm.org/10.1145/354222.353175

[27] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. 1999. An evaluation of staged run-time optimizations in DyC. SIGPLAN Not. 34, 5 (May 1999), 293-304. DOI=10.1145/301631.301683 http://doi.acm.org/10.1145/301631.301683

[28] JinChul Kim. Reducing Client-AOTC Image Size using Space Optimization Techniques. MS Thesis, Seoul National University. (Feb 2008).

[29] Advanced TV Systems Committee, http://www.atsc.org

[30] phoneME project, http://java.net/projects/phoneme

[31] The WebKit Open Source Project, http://www.webkit.org

[32] V8 JavaScript Engine, http://code.google.com/p/v8

[33] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. SIGPLAN Not. 45, 6 (June 2010), 1-12. DOI=10.1145/1809028.1806598 http://doi.acm.org/10.1145/1809028.1806598

[34] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: comparing the behavior of JavaScript benchmarks with real web applications. In Proceedings of the 2010 USENIX conference on Web application development (WebApps'10). USENIX Association, Berkeley, CA, USA, 3-3.

[35] Standard Performance Evaluation Corporation, SPECjvm98 at http://www.spec.org

[36] The Embedded Microprocessor Benchmark Consortium, http://eembc.org

[37] SungMoo Kim. Precise Hot Spot Detection Using Trace-Sensitive Runtime Estimation. MS Thesis, Seoul National University. (Feb 2008).

[38] SunSpider JavaScript Benchmark, http://www.webkit.org/perf/sunspider/sunspider.html

[39] V8 Benchmark Suite, http://v8.google.com/svn/data/benchmarks/v3/run.html

[40] Pramod G. Joisha, Samuel P. Midkiff, Mauricio J. Serrano, and Manish Gupta. 2001. A framework for efficient reuse of binary code in Java. In Proceedings of the 15th international conference on Supercomputing (ICS '01). ACM, New York, NY, USA, 440-453. DOI=10.1145/377792.377902 http://doi.acm.org/10.1145/377792.377902

[41] R. Wilkes, NGen Revs Up Your Performance with Powerful New Features, http://msdn.microsoft.com/msdnmag/issues/05/04/NGen

[42] Lingli Zhang and Chandra Krintz. 2004. Adaptive code unloading for resource-constrained JVMs. In Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES '04). ACM, New York, NY, USA, 155-164. DOI=10.1145/997163.997186 http://doi.acm.org/10.1145/997163.997186

[43] S. M. Sandya. 2004. Jazzing up JVMs with off-line profile data: does it pay?. SIGPLAN Not. 39, 8 (August 2004), 72-80. DOI=10.1145/1026474.1026485 http://doi.acm.org/10.1145/1026474.1026485

[44] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving virtual machine performance using a cross-run profile repository. SIGPLAN Not. 40, 10 (October 2005), 297-311. DOI=10.1145/1103845.1094835 http://doi.acm.org/10.1145/1103845.1094835

[45] Mike Fulton and Mark Stoodley. 2007. Compilation Techniques for Real-Time Java Programs. In Proceedings of the International Symposium on Code Generation and Optimization (CGO '07). IEEE Computer Society, Washington, DC, USA, 221-231. DOI=10.1109/CGO.2007.5 http://dx.doi.org/10.1109/CGO.2007.5

[46] Sanghoon Jeon and Jaeyoung Choi. 2012. Reuse of JIT compiled code in JavaScript engine. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12). ACM, New York, NY, USA, 1840-1842. DOI=10.1145/2245276.2232075 http://doi.acm.org/10.1145/2245276.2232075

[47] Kazunori Ogata, Dai Mikurube, Kiyokuni Kawachiya, Scott Trent, and Tamiya Onodera. 2010. A study of Java's non-Java memory. SIGPLAN Not. 45, 10 (October 2010), 191-204. DOI=10.1145/1932682.1869477 http://doi.acm.org/10.1145/1932682.1869477

초 록

어플리케이션(앱)을 다운로드 받아서 수행하는 시스템은 DTV나 스마트폰처럼 대중적으로 사용되고 있다. 앱을 다운받아서 사용하는 시스템들은 가상 머신을 주류로 사용하고 있다. 가상 머신의 가장 큰 문제점은 인터프리터를 통한 수행에 의한 느린 성능이며, 이 성능의 향상을 위해 주로 사용되는 기술이 적시 컴파일러이다. 적시 컴파일러는 다운받은 앱의 수행 중에 동적으로 머신 코드로 번역하여 사용하는 기법으로, 동적 컴파일레이션 오버헤드를 가지게 된다. 우리는 이 동적 컴파일레이션 오버헤드를 제거하여 성능을 향상시키는 클라이언트 선행 컴파일러를 제안하였다. 클라이언트 선행 컴파일러는 적시 컴파일러가 생성하는 머신 코드를 앱의 종료될 때 지우지 않고 파일형태로 스토리지에 저장하여 이후에 앱이 다시 수행될 때 저장한 머신 코드를 재활용하여 사용함으로써 런타임 컴파일레이션 오버헤드를 제거하게 된다.

저장한 머신 코드를 재활용할 때 머신 코드에 인코딩된 주소값들은 유효하지 않기 때문에 가상 머신의 현재 값에 맞추어 변경해주는 작업이 필요하다. 이 작업은 주소 재배치이다. 주소 재배치는 저장된 머신 코드만으로 수행할 수가 없기 때문에 추가적인 정보를 머신 코드를 저장하는 과정에서 생성하여 파일에 함께 저장해 주어야 한다. 자바의 상수 풀 해석은 주소 재배치 작업을 어렵게 한다. 우리는 이에 대한 해결책을 만들었다. 주소 재배치를 위한 정보들을 저장하기위해 영구 메모를 많이 사용하는 것도 문제가 된다. 따라서 우리는 주소 재배치 정보를 머신 코드 상에 인코딩하고 압축하여 저장하는 방법을 제안했다. 우리의 클라이언트 선행 컴파일러 기법은 오라클사의 CDC 가상머신 참조구현인 CVM에 구현하였다. 우리의 클라이언트 선행 컴파일러는 벤치마크의 성능을 약 12% 향상시켰다.

또한 우리는 클라이언트 선행 컴파일러 기법을 실제로 판매하는 DTV환경에 구축하여 실제 방송국이 사용하는 어플리케이션을 수행해 보았다. 우리의 클라이언트 선행 컴파일러 방식은 사용자의 실제 환경에서 33%의 좋은 성능 향상을 얻었다.

자바스크립트 가상머신인 구글사의 V8 가상머신은 인터프리터 수행없이 적시 컴파일러만을 사용하고 있다. 우리는 V8 가상 머신에 클라이언트 선행 컴파일러는 적용하였지만, 실제 성능 향상을 얻어내지는 못했다. 이것은 V8 가상 머신의 특징인 내부 객체의 적극적인 사용에 의한 결과이다. 내부 객체는 컴파일러가 생성하여 컴파일러 과정에서 사용되며, 자바스크립트 프로그램에서도 접근하여 사용하게 된다. V8 가상 머신의 컴포넌트들은 대부분 내부 객체로 생성되어, 다른 종류의 가상 머신에 비해서 상당히 많은 내부 객체를 생성하고 있다.

V8의 적시 컴파일러가 생성하는 머신 코드에서는 이 내부 객체를 직접 접근하여 사용하게 되어, 클라이언트 선행 컴파일러에 의해 어플리케이션이 수행될 때마다 이 내부 객체는 항상 필요하기 때문에 클라이언트 선행 컴파일러는 내부 객체를 재생성해야만 한다. V8 적시 컴파일러의 런타임 컴파일레이션 오버헤드의 대부분이 내부 객체를 생성하는 오버헤드이기 때문에, 우리의 클라이언트 선행 컴파일러는 이 환경에서 충분한 성능 향상을 얻을 수 없었다.

주요어 : 가상 머신, 자바, 자바스크립트, 적시 컴파일러, 선행 컴파일러, 클라이언트 선행 컴파일러
학 번 : 2003-30354