A

Client-Ahead-Of-Time Compilation for Digital TV Software Platform

SUNGHYUN HONG, Seoul National University
SOO-MOOK MOON, Seoul National University

The software platform of the digital TV (DTV) is based on Java. A TV station broadcasts a Java xlet application to the DTV, which then executes interacting with the Java system and middleware installed on the DTV. Just-in-time compiler (JITC) is often employed for accelerating the DTV Java execution, yet its performance impact is relatively lower for the xlet application because there are fewer hot spots, suffering more from the JITC overhead. One solution to reduce the JITC overhead is saving the JITC-generated machine code when finishing the xlet execution (e.g., when turning off the TV or switching to a different channel). When we turn on the TV or switch back to the channel to execute the same xlet again, we can reuse the saved machine code directly without JITC overhead. This is an attempt to employ our previous work on the client ahead-of-time compilation (c-AOTC) in the context of the DTV. We implemented the c-AOTC on a commercial DTV and experimented with real xlets, which leads to an average of 33% performance improvement.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors optimization, runtime environments, code generation

General Terms: Performance, Experimentation, Languages.

Additional Key Words and Phrases: Java, Digital TV, client ahead-of-time compiler, just-in-time compiler, relocation, Java virtual machine

ACM Reference Format:
DCE 2014 V, N, Article A (January 2014), 12 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Java has been employed as a popular software platform for app downloading embedded devices including mobile phones, digital TVs, and Blu-ray disks [SUN 2003] . The advantage of Java as an embedded software platform is platform independence, achieved by using the Java virtual machine (JVM), a program that executes Java's compiled executable bytecode [Gosling et al. 1996] . 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-based execution, just-in-time compiler (JITC) has been popularly used, which translates the bytecode to machine code at runtime for execution and saves it to the code cache in the JVM for reuse [Aycock 2003; Krall 1998; Adl-Tabatabai et al. 1998]. Since the translation overhead is a part of the running time, only hot methods are compiled and there should be enough hot spots that can justify the JITC overhead. Unfortunately, some Java applications suffer from the JITC overhead due to fewer hot spots, and one such example is the digital TV (DTV) xlet applications.

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.

One solution that we proposed to reduce the JITC overhead is client ahead-of-time compilation (c-AOTC) [Hong et al. 2007]. The idea of c-AOTC is saving the machine code generated by the JITC at the end of the first run of a Java application. When the same application is run again, the machine code can be loaded and used directly

This work was supported by the Ministry of Trade, Industry & Energy (MOTIE) through the Electronics and Telecommunications Research Institute (ETRI) (Project No. 10045344)

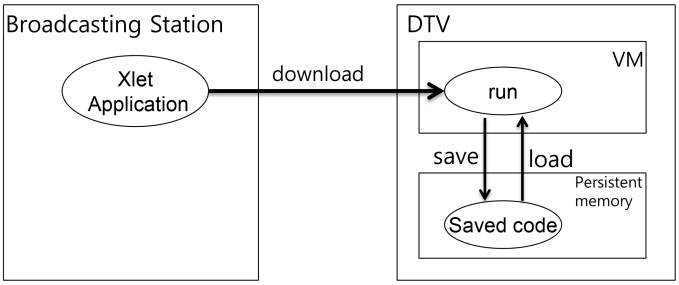

Fig. 1. Concept of c-AOTC

without the JITC overhead. In this paper, we propose employing c-AOTC for the DTV 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. With our c-AOTC, however, we 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. Since the xlet applications rarely change, it is safe to use the machine code. The concept of c-AOTC for the DTV is illustrated in Figure 1.

One important issue of c-AOTC is relocation. When we load the saved machine code to the code cache in the JVM in Figure 1, 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.

The rest of this paper consists of as follows. Section 2 describes the DTV software platform. Section 3 describes the design and implementation of c-AOTC on the DTV. Section 4 shows the experimental results. Section 5 describes related work. A summary is in Section 6.

2. DTV SOFTWARE PLATFORM

The DTV sends digital signals which consume less bandwidth than analog signals [Interactive TV]. 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) [ATSC], 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.

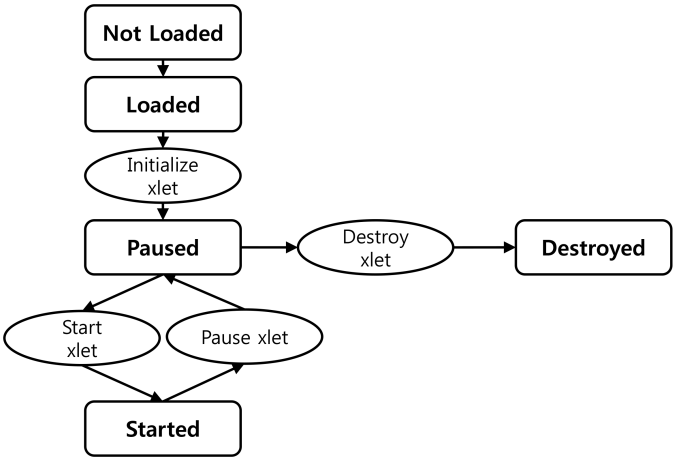

Fig. 2. Lifecycle of xlet application

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 2. 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 2, 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.

3. CLIENT-AOTC ON DTV PLATFORM

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

3.1. Design of c-AOTC on DTV

We implement our c-AOTC on a commercial DTV which has the PhoneME Advanced MR2 JVM [PhoneME]. The PhoneME JVM has a JITC based on HotSpot technology as other JVMs [Arnold et al. 2000], 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 3 shows the scenario of c-AOTC on the DTV platform.

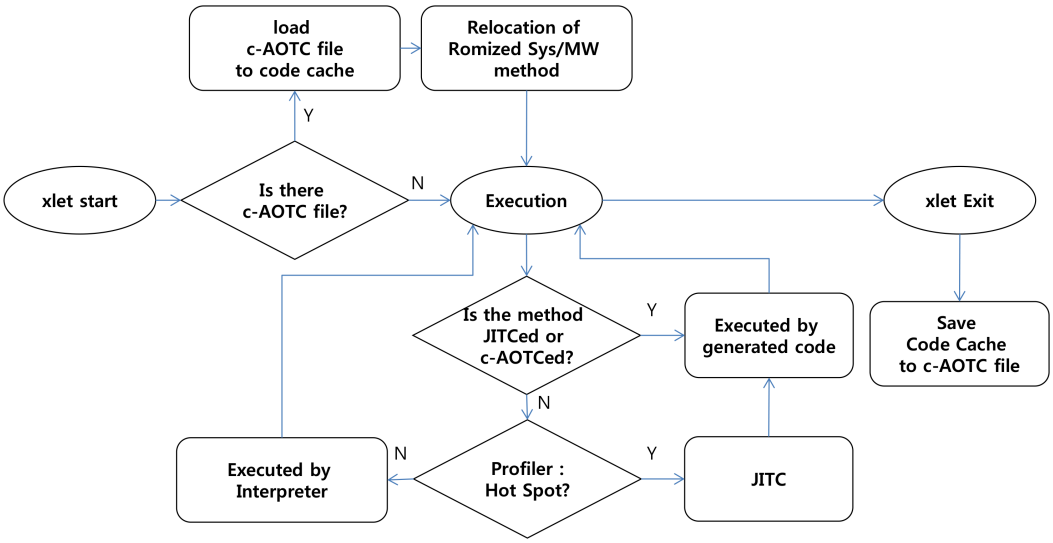

Fig. 3. c-AOTC implementation on DTV

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 previous c-AOTC takes the first approach but our DTV c-AOTC takes the latter.

Another issue is the target methods of c-AOTC. Our 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 c-AOTC, though, because the xlet methods are rarely hot, as will be described in the experimental results.

This 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 [Proebsting et al. 1997; Muller et al. 1997; Weiss et al. 1998].

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

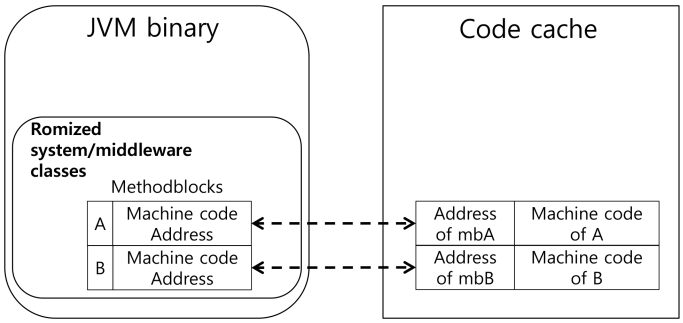

Fig. 4. JVM architecture of DTV

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.

3.2. Relocation Problem

The most fundamental issue of c-AOTC is relocation [SUN 2002] because the saved machine code may include addresses that differ from run to run. In our previous c-AOTC work, 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 4, 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.

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,

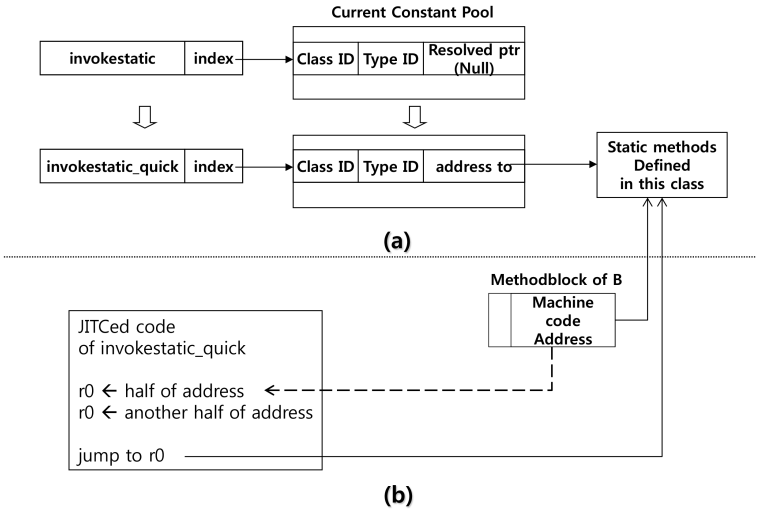

Fig. 5. Example of access to constant pool

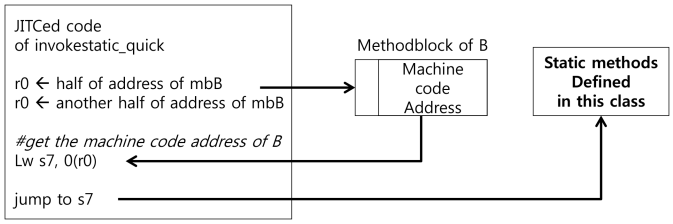

Fig. 6. Example code on PhoneME JVM on DTV

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 (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 (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.

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 6 shows the machine code of invokestatic_quick bytecode which is same bytecode of Figure 5 (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.

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.

3.3. Example of Relocation

3.3.1. Relocation example of previous c-AOTC work

(1) 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 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.

(2) 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 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.

(3) 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 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.

3.3.2. Relocation example of current c-AOTC work

(1) checkcast_quick bytecode
In PhoneME JVM for our current 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.

(2) invokestatic_quick bytecode
In PhoneME JVM for our current 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.

(3) getstatic_quick bytecode
The generated code of getstatic_quick by PhoneME JVM JITC is not different from the code from previous 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.

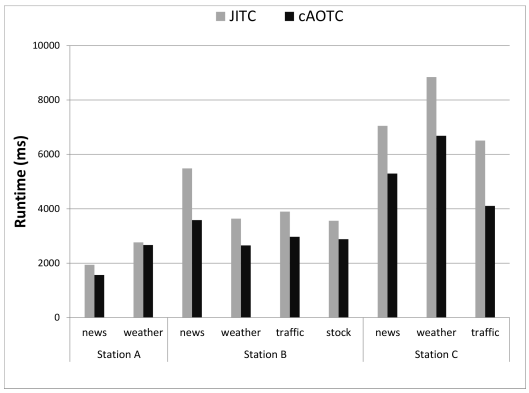

Fig. 7. Runtime of xlet application (ms)

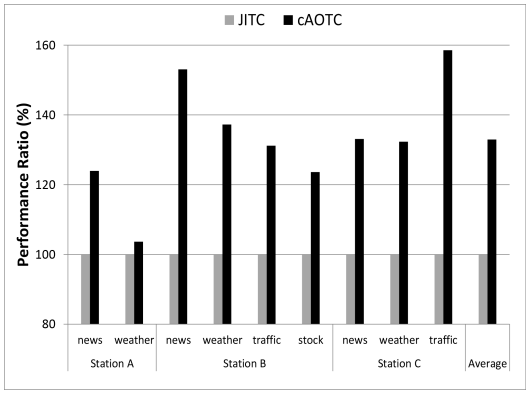

Fig. 8. Runtime performance of xlet application (%) (Running by JITC is 100 %.)

4. EXPERIMENTAL RESULTS

4.1. Experimental Environments

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.

4.2. Performance Result

Figure 7 and Figure 8 show the performance improvement of c-AOTC.

Figure 7 shows the runtime of the xlet applications. And figure 8 shows the runtime performance ratio of xlet application running by c-AOTC over running by JITC runtime. In Figure 8, c-AOTC gets average 33% performance improvement.

Two major causes of performance improvement of c-AOTC are to remove the JITC overhead of hot-spot methods and the interpretation of c-AOTCed methods. According to Figure 7 and Figure 8, 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.

4.3. Analysis of JITCed method

Figure 9 shows the number of JITCed methods in each benchmark.

Figure 10 shows the ratio of JITCed methods. In Figure 9 and Figure 10, gray bar is the JITCed methods included in system/middleware classes and black bar is the JITCed methods included in xlet application classes.

In Figure 10 , 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

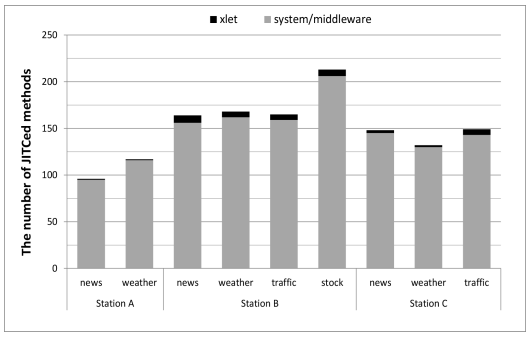

Fig. 9. Number of JITCed methods

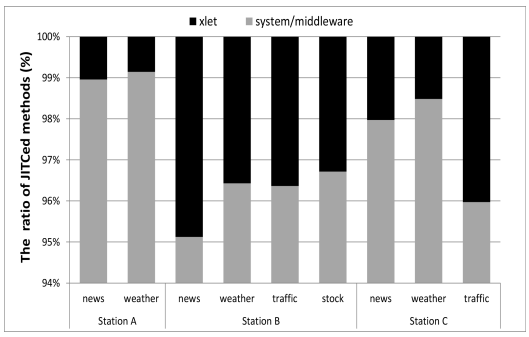

Fig. 10. Ratio of JITCed methods

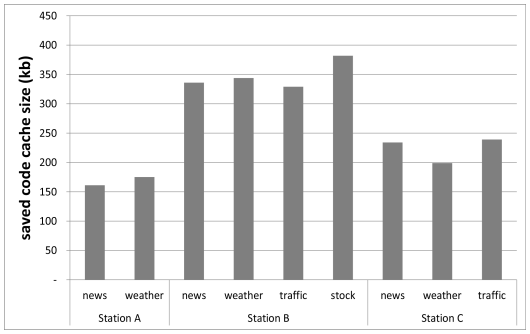

Fig. 11. Size of saved c-AOTC file

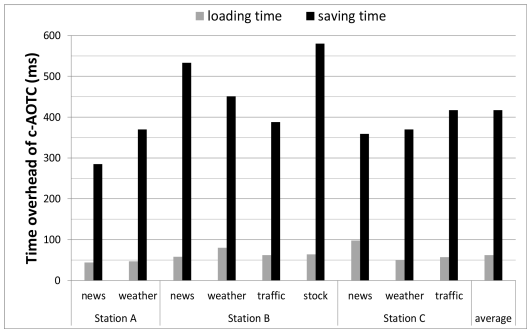

Fig. 12. The overhead of c-AOTC (loading time & saving time) (ms)

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.

4.4. Space Overhead

Figure 11 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.

4.5. c-AOTC Overhead

Figure 12 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 c-AOTC file and copy it to code cache in memory.

The saving time and loading time of Figure 12 is directly proportional to the size of c-AOTC file of Figure 11. The saving time takes average 400ms and the loading time takes average 60ms.

5. 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 [Serrano et al. 2000] . 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 [Joisha et al. 2001] . 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 [Weiss et al. 1998]. 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 [Bak et al. 2001]. A similar code unloading is also proposed in [Zhang and Krintz 2004].

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 [Sandya 2004; Arnold et al. 2005] . 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.

A hybrid environment composed of JITC and AOTC is built in [Oh et al. 2008], 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.

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 [Ogata et al. 2010]. Because our c-AOTC removes the process of JITC, the peak memory overhead of VM is removed.

6. SUMMARY

JITC can improve the performance of xlet application on DTV. But, the translated machine code with runtime compilation overhead is removed when the station is changed. Our c-AOTC make that the translated machine code can be reused. When the user select previous station and restart the xlet application which is used before and is saved by c-AOTC, the c-AOTC has average 33% performance improvement.

REFERENCES

ADL-TABATABAI, A.-R., CIERNIAK, M., LUEH, G.-Y., PARIKH, V. M., AND STICHNOTH, J. M. 1998. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. ACM, New York, NY, USA, 280–290.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive optimization in the jalapeo JVM. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00)*. ACM, New York, NY, USA, 47–65.

ARNOLD, M., WELC, A., AND RAJAN, V. T. 2005. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*. ACM, New York, NY, USA, 297–311.

ATSC. Advanced tv systems committee. http://www.atsc.org.

AYCOCK, J. 2003. A brief history of just-in-time. *ACM Computing Surveys (CSUR) 35,* 2, 97–113.

BAK, L., ANDERSEN, J. R., AND LUND, K. V. 2001. Non-intrusive gathering of code usage information to facilitate removing unused compiled code.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley.

HONG, S., KIM, J.-C., SHIN, J. W., MOON, S.-M., OH, H.-S., LEE, J., AND HYUNG KYU CHOI. 2007. Java client ahead-of-time compiler for embedded systems. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES '07)*. ACM, New York, NY, USA, 63–72.

Interactive TV. Interactive tv web. http://www.interactivetvweb.org.

JOISHA, P. G., MIDKIFF, S., SERRANO, M., AND GUPTA, M. 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.

KRALL, A. 1998. Efficient JavaVM just-in-time compilation. In *Proceedings of 1998 International Conference on Parallel Architectures and Compilation Techniques*. 205.

MULLER, G., MOURA, B., BELLARD, F., AND CONSEL, C. 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'97)*. Vol. 3. USENIX Association, Berkeley, CA, USA, 1.

OGATA, K., MIKURUBE, D., KAWACHIYA, K., TRENT, S., AND ONODERA, T. 2010. A study of Java's non-Java memory. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*. ACM, New York, NY, USA, 191–204.

OH, H.-S., MOON, S.-M., AND JUNG, D.-H. 2008. On hybrid compilation environment for embedded systems. In *Proceedings of the 12th Workshop on Interaction between Compilers and Computer Architectures (Interact-12)*. 82–90.

PhoneME. Phoneme project. https://java.net/projects/phoneme.

PROEBSTING, T. A., TOWNSEND, G., BRIDGES, P., HARTMAN, J. H., NEWSHAM, T., AND WATTERSON, S. A. 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'97)*. Vol. 3. USENIX Association, Berkeley, CA, USA, 3.

SANDYA, S. M. 2004. Jazzing up JVMs with off-line profile data: does it pay? *Newsletter ACM SIGPLAN Notices 39,* 8, 72–80.

SERRANO, M., BORDAWEKAR, R., MIDKIFF, S., AND GUPTA, M. 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.

SUN. 2002. *Porting Guide Connected Device Configuration and Foundation Profile version 1.0.1 Java 2 Platform Micro Edition*. Sun Microsystems.

SUN. 2003. *CDC: An Application Framework for Personal mobile Devices*. Sun Microsystems.

WEISS, M., DE FERRIRE, F., DELSART, B., FABRE, C., HIRSCH, F., JOHNSON, E. A., JOLOBOFF, V., ROY, F., SIEBERT, F., AND SPENGLER, X. 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)*. 119–130.

ZHANG, L. AND KRINTZ, C. 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.