Research on Embedded Java Virtual Machine and its Porting

Jun QIN, Qiaomin LIN, Xiujin WANG

College of Media Communications Technology,
Nanjing University of Posts and Telecommunications, P.R.China

Summary
Embedded Java Virtual Machine has seen wide applications with the fast development of embedded field. More and more embedded platforms choose the KVM porting to support Java Virtual Machine. This paper proposes a set of general methods for the KVM porting on the basis of analyzing the platform demands, source code organizing structure and main technical issues. According to the above methods, we have ported KVM onto the platforms like Windows and Linux successfully.

Key words:
Embedded, JAVA Virtual Machine, KVM, Porting

1. Introduction

The computing environment based on network of smart devices and computers not only provides a new platform for software but also brings on challenges on software development. Java technology which is just designed for the network computing environment has such virtues as platform-independence, network mobility and security[1]. It is suitable for software development in the embedded field. J2ME from SUN Company is an embedded system and helpful for developing embedded applications. Therefore the research on the core of java technology—virtual machine including its structure, operation mechanism and porting technology takes on the academic and application values significantly.

KVM (K Virtual Machine) is the J2ME's core and the execution engine. It is mainly used for small and resources-restricted devices like cellular phone, beep pager, PDA and etc. KVM is the smallest Java Virtual Machine which size is only dozens of KB during operation, while it has the full java features and can be easily ported to the other platforms. KVM porting has few requirements of the platform. Besides KVM implementation is characterized by modularization.

This paper proposes a set of general KVM porting methods after analyzing platform requirement, source code organizing structure and major technical issues. By using these methods we have successfully ported KVM onto the platforms like Windows and Linux.

Manuscript received September 5, 2007
Manuscript revised September 20, 2007

2. Analysis of SUN KVM Implementation

A feature of java technology is that its external behavior is definitely regulated by the specification, but how to implement the defined external behavior is not illuminated in the specification[2][3]. This feature offers possibility for people to implement it, but there is also difficulty in the implement process.

There are two approaches to implement the Java Virtual Machine. One is to start from scratch, design and implement according to the specification. Another is to port the implementation onto new target platform. For any approach, SUN reference implementation is always a good reference. In fact the most of J2ME porting are based on SUN reference implementation because of its being the open-source[4].

2.1 Program Execution Flow within KVM

Within J2ME program flow starts from java source code (.java files). Platform-independent java bytecode files are generated with compiling of java source files. The validity of the bytecode files is verified through an outer tester. After testing the bytecode will be executed within KVM. The program execution Flow within KVM is as figure 2-1.

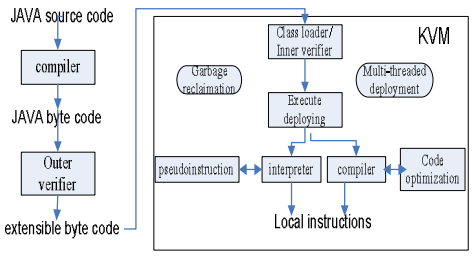

Fig. 2-1 Code Execution Flow

2.2 KVM Modularization

KVM is designed in five major modules such as class loader module, bytecode execution module, thread deploying module, memory management and garbage collecting module and class verifying module. The connections between these modules are as figure 2-2.

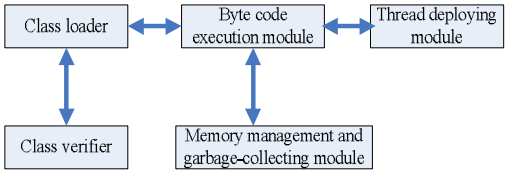

Fig. 2-2 Modules of KVM Implementation

The main usage of Class Loader Module is to load and link class files. In others words binary class files are interpreted and loaded into run-time data structure. Meanwhile all super classes and its interfaces are loaded and linked. There are four phases for class loading. In different phases the class is in the different states. These states can be CLASS_RAW, CLASS_LOADING, CLASS_LOADED and CLASS_LINKED。

In CLDC1.04 two structures are defined in KVM source code VmCommon/h/class.h. They are the inner run-time representation of the class in KVM implementation. Figure 2-3 shows the relation between instanceClassStruct and classStruct.

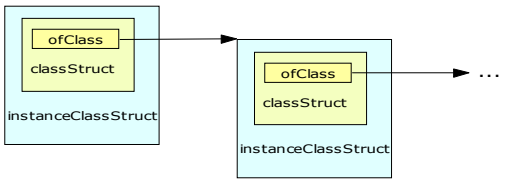

Fig. 2-3 Relation between instanceClassStruct and classStruct

When a piece of memory is assigned to contain instanceClassStruct and initiated as 0 by KVM, the class is in CLASS_RAW state. When the contents of the class are read by KVM, the class is in CLASS_LOADING state. After reading the class state is refreshed as CLASS-_LOADED. At this time the information needed by execution is not been fully understood such as those interfaces and their contents. So some necessary computations should be carried out to fetch other state informations. After getting these informations the class state will be refreshed as CLASS_LINGKED by KVM. The states conversion is showed in figure 2-4.

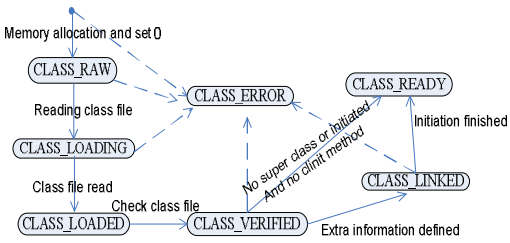

Fig. 2-4 Class State Switching

The main task of the class verifying module is to verify that loaded classes are provided with the right inner structure and harmonize with each other. The exceptions will occur if some errors are found in the checking phase by class verifier module. The main target of the implementation of class verifier module is to guarantee the programs robustness.

There are two steps for class verifying in KVM. The first step takes place by class outer verifier before class files are loaded into memory. Generally the extra attributes will be added to the classes to represent these classes that have passed the outer checking. The second step is carried out by inner verifier during execution. After the classes being loaded into memory the inner verifier in KVM implementation will start continued checking according to run-time informations and messages added by outer verifier. It's to make sure that class files are not tampered after passing through outer verifier checking.

The bytecode execution module is used mostly for java bytecode execution. The core of this module is an interpreter, which ceaselessly executes the bytecodes pointed out by program counter in the memory.

The advantages of this implementation are provided with the simplicity, reliability and readability as well. After satisfing the criterion this implementation uses the simple approach to realize the bytecodes execution.

The memory management and garbage-collecting module charge the allotting and releasing of memory. Due to memory limitation of the Java Virtual Machine the implementation approach of this module has the great influence on the performance of the whole virtual machine. One of the important tasks in this module is to reclaim garbage memory. Especially the whole virtual machine's performance can be better by the improvement of the garbage reclamation algorism. The location of the memory management system in the architecture of the virtual machine is showed in figure 2-5.

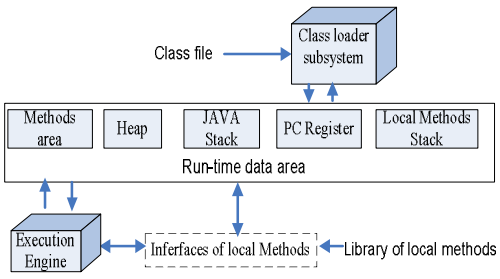

Fig. 2-5 Inner Architecture of Java Virtual Machine

The thread deployer module is used for the implementation of a platform-independent, portable and snatching thread deploying model. This model is designed as multi-threaded using java and supports the asynchronous I/O. The thread deploying module of KVM is aimed to be platform- independent, portable and snatching. This thread mechanism is implemented through a run-time data structure threadQueue, which is used in KVM, and utilizes the circular chain link to organize all the different threads in the virtual machine.

The basic thread operations include constructing thread, startup thread, suspending thread, restoring thread, destroying thread and switching threads. KVM Implementations of these basic operations are located in VmComm/src/Thread.c. Figure2-6 describes the states switching of the threads.

In figure 2-6 the function BuildThread() is used for creating a VM-layered thread. The function DismantleThread() is used to release a thread. The function startThread() is used for starting a thread. The function stopThread() is used for stopping a thread and release the occupied resources. The function suspendThread() is used for suspending thread. The function resumeThread() is used for resuming thread. The function addThreadToQueue() and removeFromQueue() are used for adding or deleting specified thread to or from the queue respectively. The SwitchThread() is to switch the current thread to the next one.

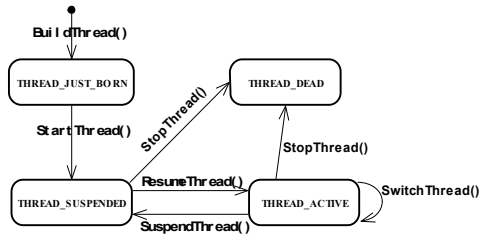

Fig. 2-6 The State Switching of Java Threads

According to the analysis of the aforementioned modules KVM implementation has the followed important features.

(1) The platform-independence and portability is the most important goal of this implementation. All the modules adopt general resolutions to meet the succinctness. But the bad performance is the serious drawback.

(2) The implementation is modularized and all the functions are independent and complete. Then the framework and module function are easy to understand. Meanwhile it offers great convenience for the module optimization. For example, the performance of the whole virtual machine can be improved by modifying the models of interpreter or garbage-collector and this modification will not cause few influence on other modules.

(3) In this implementation all codes are partitioned in the platform-dependent and platform-independent. But it causes the other problem that the optimization of the whole design can not be easy resolved after the porting.

3. Key techniques of KVM Porting

SUN reference implementation has few extra demands on target platform since it is designed and realizes as platform-independent. In fact KVM porting should focuses on C compiler on target platform according to reference [5]. For KVM porting we must distinguish the platform-dependence code and the platform-independence code. The key research should aim to the platform-dependence code.

3.1 Directory of KVM Implementation

Via the Decompress of the distribution package in SUN reference implementation, we can get a directory named j2me_cldc. This directory includes all CLDC related codes and a subdirectory in SUN reference implementation. The content of the subdirectory shows as table 3-1.

The KVM directory includes the source codes of KVM implementation. The VmCommon subdirectory is composed of the head files and the implementation files which are designed as platform-independent code. Therefore these files within the subdirectory need not modification in most cases.

Table 3-1 J2ME_CLDC Subdirectory and its Contents

Subdirectory Name	Description
api	including source code of java library of the version
bin	including all binary executable files and compiled java class library
build	including makefile to compile the reference implementation
doc	including some explanative documents
jam	including optional source code of java application manager（JAM）supported by JVM
kvm	including source code of KVM implementation
tools	including source code of needed tools（JCC、Previrifier etc.）

3.2 Porting Files and Porting Functions

All porting of KVM should offer the file VmPort/h/machine_md.h. This file function is to override the compiling definitions and declarations including those particular declarations needed by target platform. These particular declarations, function prototypes, typedef statements, #include statements and #define statements must be defined directly in this file or other files that can be included in the target platform. There are the fourteen functions like void AlertUser (const char *message) that must be defined on any porting platform. Generally C codes are placed in the file VmPort/src/runtime_md.c.

In the porting process KVM needs the special C library functions such as character string manipulation functions, memory operation functions, print functions, random digits creatation functions and exception process functions. If the porting platform does not offer these functions, you should define them by yourself or map those identical functions in porting platform to the above functions by using macro.

3.3 KVM Compiling Switch Option

Many features of KVM implementation are set as switch by using macro definition during compilation. It is very convenient to choose to or not to support any feature in the process of KVM porting by switch on or off these switches according to characteristics of target platform. Therefore the meanings of these compiling switches or options should be understood completely before the porting. Then we can decide whether to support these features or not. The reference [6] gives the detailed explanations on these switches or options.

3.4 General KVM Porting Scheme

The main steps of KVM porting are as follows.

(1) Identify the necessity and capability of target platform for porting. The necessity means whether the target platform needs J2ME technology or not. The capability means whether the platform meet the requirements for KVM porting or not.

(2) Understand the code structure. It should be made clearly that which part can be used without any modification and which part need be implemented for the target platform.

(3) Begin to port. The porting sequence is generally from bottom layer to higher layer. So the first part to be ported is CLDC. The switches and options should be carefully checked according to target platform during the CLDC porting, so as to control many platform-related features.

(4) After CLDC is ported successfully, then MIDP and Optional Packages can be ported continued.

(5) In the end, the strict professional testing (like TCK, JDTS and etc.) should be carried out to guarantee the robustness and correctness of the porting.

4. KVM Porting on Windows and Linux

Generally speaking embedded Java Virtual Machine is not an absolute operating system. It is just an application program based on target platform. On its bottom-layer there are the target platform hardware and operating system. On its top-layer there are all kinds of java applications. Owing to KVM isolation between the java applications and the lower part (hardware and operating system), it is possible to realize platform-independence. The structure of embedded Java Virtual Machine is showed as figure 4-1.

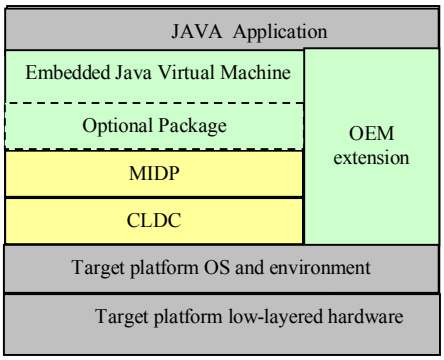

Fig. 4-1 Structure of Embedded Java Virtual Machine

Since Windows and Linux are targeted in desktop and server domain, the research on KVM porting for these platforms is valuable. In fact these platforms have wide applications and extensions such as WinCE, embedded Linux, which makes it a good reference for other system porting.

According to the general porting methods, it is easy to port SUN reference implementation to Windows and Linux. Now that the implementation is for reference, those platform-dependent and platform-independent codes have been divided separately in the organization of source codes. The CLDC and upper MIDP have been divided and realized in the local codes. The task required in the porting process is to set environment variables and

compile corresponding project step by step. The figure 4-2 and figure 4-3 are the sketch maps of KVM after porting to Windows and Linux respectively.

We utilized the techniques of the direct threaded interpreter and the lazy binding to improve the KVM performance. With the KVM testing tool JBenchmark we have inspected the general porting methods described in this paper. The test result shows that the application of the general porting methods and the optimization strategy can enhance the KVM performance up to 40% averagely. The figure 4-4 and figure 4-5 give the comparison in JBenchmark1 and Jbenchmark2.

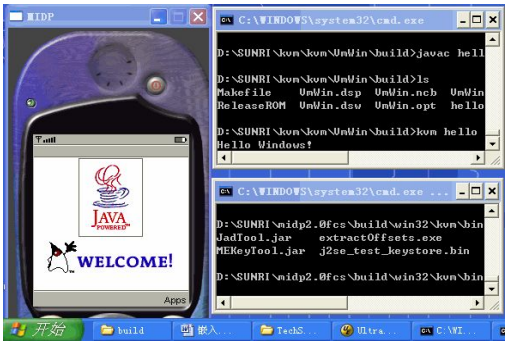

Fig. 4-2 KVM Port to Windows

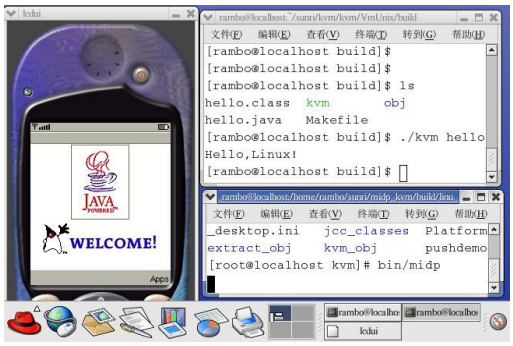

Fig. 4-3 KVM Port to Linux

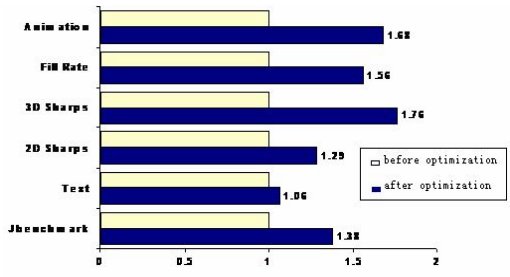

Fig. 4-4 the scale of JBenchmark1 test

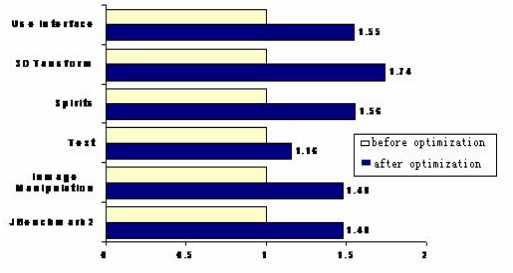

Fig. 4-5 the scale of JBenchmark2 test

5. Conclusion

Through the porting research in theory and engineer, we analyze the target platform requirement and code structure. Then this paper proposes a set of general porting methods. The KVM porting instances testify that these methods take on the good feasibility and practicability.

Acknowledgments

This research work is supported by Jiangsu University Natural Science Foundation under Grant No. 06KJB520079.

References

[1] B. Veners, Inside the Java Virtual Machine, 2nd edition, NewYork: McGraw-Hill

[2] James Gosling, Bill Joy, Guy Steele, Gilad Brancha, The Java Language Specification Third Edition, Addison-Wesley

[3] Tim Lindholm, Frank Yellin, The Java™ Virtual Machine Specification, Second Edition

[4] Sun Microsystems, Inc, J2ME CLDC Reference Implementation, Release Notes, CLDC 1.0.4. http://java.sun.com/products/cldc/

[5] BillVenners, McGraw-Hill , Inside Java Virtual Machine, Second Edition,

[6] Sun Microsystems, Inc, KVM porting Guide, CLDC, Version1.1. http://java.sun.com/products/cldc/

Jun QIN is an associate professor of computer science and Technology at Nanjing University of Posts and Telecommunications. Her research interests include software engineering, multimedia technology and so on.

Qiaomin LIN is a lecture of computer science and Technology at Nanjing University of Posts and Telecommunications. His research interests include software engineering, multimedia technology and so on.

Xiujin WANG received the M.S. Degree from Nanjing University of Posts and Telecommunications in 2007. His research interests include software engineering, multimedia technology and so on.