行政院國家科學委員會專題研究計畫　成果報告

Java 虛擬機器支援之數值運算

計畫類別：個別型計畫
計畫編號：NSC93-2213-E-002-114-
執行期間：93 年 08 月 01 日至 94 年 07 月 31 日
執行單位：國立臺灣大學資訊工程學系暨研究所

計畫主持人：陳俊良

報告類型：精簡報告

處理方式：本計畫可公開查詢

中　華　民　國 95 年 1 月 23 日

行政院國家科學委員會專題研究計畫成果報告
Java 虛擬機器支援之數值運算

計畫編號：NSC 93-2213-E-002-114
執行期限：93 年 8 月 1 日至 94 年 7 月 31 日
主持人：陳俊良　國立台灣大學資訊工程學系

中文摘要

Java 技術在數值運算上的改良已經被廣泛的討論。大致可區分為兩個類別：第一、著重在利用新的 Java 編譯器，或者更改 Java 語言來達到增進效能。第二、著重在修改 Java 虛擬機器 (JVM) 並配合自定的函式庫來增進效能。

但無論何者，都有其不可避免的優缺點。修改 Java 語言或 class 檔案格式將失去 Java 跨平台的特性，而只修改 JVM 而不配合修改編譯器，將會增加許多額外的負擔來檢查執行時物件的狀態。到目前為止，仍然沒有一個完整的架構能整合 Java 編譯器及 JVM。因此我們提出一套讓 Java 編譯器和 JVM 能更緊密配合的架構，來支援像數值運算這樣的特殊運算環境。

我們的概念是，不更改 Java 語言，只修改 Java 編譯器在 class 檔案中加入有用的 attribute 資訊，並修改 JVM 去讀取這些資訊並做對應的動作來提升整體的效率。如此一來我們不但能保有 class 跨平台的特性，也能省去 JVM 在執行時的許多負擔，以增進效能。

關鍵詞： Java、數值運算、編譯器、虛擬機器。

Abstract

Enhancements of Java techniques on numerical computation have already undergone wide discussion. These attempts may be divided into two categories: through utilizing new Java compilers or modifying the Java language, and through modifying Java Virtual Machine (JVM) as well as providing with self-defined class libraries.

By altering the Java language or its class format, one risks losing the advantage of cross-platform portability, which is an important trait of Java. By altering only the JVM but not the compiler, one would have to put more work into examining the status of objects at execution time. Until now there has not been a complete architecture capable of integrating Java compiler and JVM, and it is the reason why we shall present a set of infrastructure that will bind the compiler and JVM more closely so that they could support special computing environments such as numerical computing.

Our concept is to keep the Java language unaltered, and to change only the Java compiler so that it would add useful attribute information to the class file during compilation. Then, we modify the JVM to read in the information and to react correspondingly. In this way, we not only preserve class portability, but also free JVM from excess status judgment at execution time, which greatly improves its performance.

Keywords: Java, numerical computation, compiler, virtual machine.

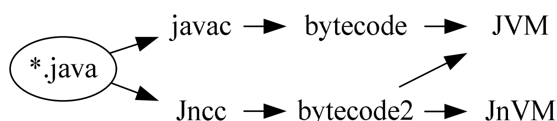
Introduction

Java has many irreplaceable advantages, and these advantages give Java an edge over other programming languages. However, the Java technology has never been widely used in the field of numerical and scientific computing. The main reason is that Java is not designed specifically for numerical computing: it supports neither complex primitive data type nor regular arrays.

Since 1998 there have been numerous publications dedicated to the application of Java on numerical and scientific computing. The proposed methods greatly improved the performance of Java on the tasks, but each of them still has its own Achilles' heel. For example, [1] solved the problems that Java

lacks complex data type. But, because they made changes to the language, the source code became incompatible with the original Java compiler, affecting its portability. On the other hand, [2] used "Semantic Expansion" for improvement, but it took a long time to examine all possible situations and therefore resulted in an even longer compilation time in JIT (Just-In-Time Compiler).

We learn from these methods that we need an integrated architecture to bind the Java compiler and Java Virtual Machine closely together. This architecture must increase system performance without destroying Java's virtues, for example, object-orientated nature and cross-platform ability. Here we propose the architecture as follows.



The original compiler and VM for Java are standard Javac and JVM, respectively. We have modified them into Jncc and JnVM by the following: during compilation, Jncc adds "useful" information to the attribute fields of class file (bytecode2). During execution, JnVM reads these newly-added attributes and makes the necessary moves. Bytecode2 can also be executed by standard JVM, which will ignore the additional attributes and maintain cross-platform flexibility.

The key point in our architecture is the "attributes" in the bytecode structure. Attribute is the place where bytecode stores information. The attributes can be either added or neglected, and JVM can choose whether to use them or not. Because of this design, we believe that we could use self-defined attributes to make Java compiler and JVM work hand in hand, letting the compiler share the task by recording useful data on attributes, and giving JVM time to do other things.

When we have added self-defined attributes to the bytecode, we could still run it on normal JVM, and the attributes would be ignored. On the other hand, these self-defined attributes would be useful to the JnVM, improving its overall performance.

For example, DAXPY (ax+y) is a computation commonly seen in numerical computing. The original Java language does not provide such subroutine, and to determine where DAXPY would appear simply by JVM is next to impossible because the bytecode loses the tree structure once it is read into the JVM. Thus, we can use Jncc to record each location (Program Counter) at which DAXPY appeared and convert them into an attribute, and all JnVM needs to do is to read and process them at the given locations.

JNC Framework

To design a sound architecture, we must take Java language, Java compilers, and JVM into full-scale consideration. Here we propose a new architecture, which does not modify the Java language for the sake of source code compatibility, but binds the Java compiler and JVM more closely. The key to this binding is the utilization of attribute in the Java class format.

Without changing the bytecode structure, we let the process be done by the compiler at compilation time, and adding the compiled results as attribute into Code_attribute, alleviating JVM's workload at runtime.

A standard .java file produces bytecode after javac compilation, and is then executed by normal JVM. In the JNC framework, the java file is compiled by Jncc, producing also bytecode but different from the bytecode generated by javac. The difference is the additional self-defined attributes. Such bytecode can be executed by normal JVMs, though the result would be the same as not adding these new attributes (they are ignored). When the bytecode is executed by JnVM, however, self-defined attributes would improve the performance of execution.

In the following we will discuss problems about DAXPY. Similar issues like array or complex nimber can be discussed in a similar way. We start treating all these issues by defining suitable attributes. As we explained, these attributes

would be ignored in standard JVM, so they would not affect cross-platform ability.

DAXPY refers to a commonly seen computation in numerical computing, aX+Y. X or Y could represent complex data structure such as matrices. Java itself does not support an instruction like aX+Y, but if JVM can complete a lot of numerical computing work - multiplication, addition, and so on - all at one, the performance may be improved. For the processor with native DAXPY instruction, JnVM may get huge performance improvement from compiling aX+Y to single native instruction by JIT.

DAXPY serves as a good example because it embodies the spirit of the JNC framework. In JNC, we hope that Jncc can gather useful and relative information for JnVM in compilation time, reducing overhead for JnVM at runtime. It's easy to find a pattern like DAXPY at compilation time, but difficult in JVM.

In the Jncc compilation, all the computation is established as tree structure, so it's easy to judge the location of DAXPY appearances. When the compilation ends, JVM wouldn't spend much time reconstructing the whole tree, which is why it's extremely difficult to determine DAXPY in bytecode.

Because DAXPY patterns can be nested, so we must find DAXPY patterns in whole tree structure. But in JVM, the tree structure is difficult to re-construct.

We experiment with the most simplified situation in Jncc: when a, X, Y are all double variables. A self-defined DAXPY attribute is used for recording:

```
DAXPY_attribute_info {
   u2 attribute_name_index;
   u4 attribute_length;
   u4 daxpy_attribute_length;
   {
     u2 dmul_pos;
     u2 dadd_pos;
   } daxpy_info[daxpy_attr_length];
}
```

The attribute_name_index will point to the UTF8 string "DAXPY" in constant_pool, while dmul_pos and dadd_pos stores the PC (Program Counter) values where DMUL and DADD appeared in DAXPY. We write in the DAXPY_attribute when the class file is produced.

DAXPY_attribute is ignored in standard JVM, but in JnVM the class loader reads in this attribute, and changes the PC (in theory, it should always be DMUL) location MUL_POS points to in the code to NOP, and change ADD_POS's PC value (should always be DADD) into a new instruction - JNC_DAXPY.

JnVM adds a new JNC_DAXPY instruction. When the instruction is executed, there are at least three double values on the stack, namely a, x, and y. JNC_DAXPY will pop these three double values and push the result of ax+y back onto the stack, so it would require one less instruction, one less stack push, and one less stack pop, thus achieving performance enhancement. Furthermore, if the hardware processor supports a native instruction of ax+y, we can translate a DAXPY pattern into one native instruction.

By adding JNC_DAXPY instruction into JVM, the two sequential instructions, multiply and add, are combined into one instruction, reducing one stack push and stack pop. Although this may not seem much improvement, we can get the flavor of how it works. The JNC framework binds Java compiler and JVM to be more close, expanding the function of the compiler, and lifts the workload on VM execution.

Implementation

The prototype of our implementation is based on a platform of Red Hat Linux 9, and is divided into two parts: Jncc compiler and JnVM.

The Jncc compiler uses the open source code of javac published by Sun as the underlying basis. The Sun's javac is written in Java and can compile .java source code to bytecode format. We modify javac's code to analyze tree structure in compilation time, and add attributes like DAXPY, when class files are produced. The resulting class file can work normally under standard JVM such as Sun's JVM, Kaffe's VM, and IBM's VM.

Jncc complier will analyze tree structure, and try to find the pattern of

DAXPY. Jncc checks the tree structure and looks for an AddExresssion, which is a binary expression. A DAXPY will appear in a situation that a MulExpression at the left-hand side of AddExpression.

JnVM is based on Kaffe 1.1.3. Kaffe is a clean room implementation of the Java virtual machine, plus the associated class libraries needed to provide a Java runtime environment. It contains no Sun source code at all, and was developed without even looking at the Sun source code. But Kaffe is a great choice as a base for virtual machine education and research. The Kaffe virtual machine is free software, licensed under the terms of the GNU General Public License.

We modify Kaffe 1.1.3, at class loader it reads in attributes like DAXPY which were added by jncc, and fetches the PC value of these appearances. Kaffe 1.1.3 is an open-source project and support two engines- interpreter and JIT. We also build JnVM in different engine modes.

After reading attributes, we try to improve performance by these attributes' information. In DAXPY, JnVM converts the original two-instruction set - DMUL and DADD into a new instruction, DAXPY. This simplifies the process by one instruction.

Conclusions

The goal of our proposed framework is to share VM's workload into compilation time. We let the process done by the compiler at compilation time, and add the compiled results as attribute into Code_attribute, alleviating JVM's workload at runtime. Through the architectural design of Jncc and JnVM, we have demonstrated in this study that this framework successfully improves Java's performance on numerical computing while keeping its cross-platform strength. Because jncc didn't change the Java language itself, programmers do not need to comply with specialized syntax, and the source code is compatible with compilers like Javac, Jike, etc. The class file compiled by Jncc can also be transmitted through the Internet as other standard classes do with general dynamic linking and loading. The only price to pay is a larger class file caused by additional attributes. This doesn't affect compatibility.

JnVM is specifically designed for the classes compiled by Jncc, so it is capable of expressing self-defined attributes such as DAXPY enhancing the system performance by introducing "Semantic Expansion" at the same time. By applying this framework on kaffe 1.1.3, we have demonstrated that in some extreme test programs, we have achieved at most 11.236 times better than original kaffe under JIT, while achieving at most 8.231 times better than Sun's J2SDK 1.3.1 . In Java Linpack benchmark, we have also achieved 3.45 times better than original kaffe under JIT, while having achieved 82.4% of Sun's J2SDK.

References

[1] B. Blout and S. Chatterjee, An Evaluation of Java for Numerical Computing, Unversity of North Carolina, 1999.

[2] P. Wu, S. Midkiff, J. Moreira and M. Gupta, Efficient Support for Complex Number in Java, IBM T.J. Watson Research Center, 1999.

[3] M. Philippsen and E. Ghnthner, Complex numbers for Java, University of Karlsruhe, 1999.

[4] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira, High Performance Numerical Computing in Java : Language and Compiler Issues. IBM Thomas J. Watson Research Center, 1999.

[5] R. F. Boisvert, J. Moreira, M. Philippsen and R. Pozo, Java and Numerical Computing, IEEE 2001.

[6] J. E. Moreira, S. P. Midkiff and M. Gupta, Supporting Multidimensional Array in Java, IBM T.J. Watson Research Center, 2001.

[7] J. E. Moreira, S. P. Midkiff and M. Gupta, The NINJA Project: Making Java Work for High Performance Numerical Computing, IBM T.J. Watson Research Center, 2001.