MetaVM: A Transparent Distributed Object System Supported by Runtime Compiler

Kazuyuki Shudo Yoichi Muraoka
School of Science and Engineering
Waseda University
Okubo 3-4-1, Shinjuku-ku, Tokyo 169-8555, Japan

Abstract *MetaVM is a distributed object system for Java virtual machine. It allows programmers to deal with remote objects in the same way they do local objects. Therefore, it can provide a single machine image to programmers. We implemented a runtime compiler of Java bytecode to provide the facilities. The runtime compiler generates a native code which can handle remote objects beyond the network besides the local objects. The compiler uses semantic expansion, which is a technique that changes the original semantics of a Java bytecode.*

Keywords: distributed object system, network transparency, Java Just-In-Time compiler

1 Introduction

MetaVM is a distributed object system, which strongly depends on a runtime compiler to achieve its location transparency. For performance reasons, most Java virtual machines (JVM) [1] have a runtime compiler called Just-In-Time (JIT) compiler. A runtime compiler converts Java bytecode to processor native code, and an underlying processor actually executes the native code. Therefore, we can say that the runtime compiler defines how Java bytecode is recognized, and controls the semantics of the bytecode. The runtime compiler we have developed generates a native code which operates remote objects as well as local objects. Such a modification of program semantics is called "*semantic expansion*".

MetaVM achieves object location transparency by applying semantic expansion. Programmers, and even bytecode instructions, do not have to be aware of the distinction between local and remote objects (Section 3). Therefore, programmers would be able to see a single machine image of computers.

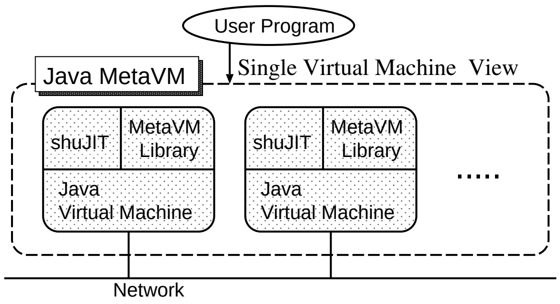

Figure 1: Structure of MetaVM

2 Overview of MetaVM

MetaVM consists of computers connected via a network. Each computer has a JVM, a runtime compiler called shuJIT [2][3], and a MetaVM library (Figure 1).

The shuJIT is a runtime compiler of Java bytecode. While shuJIT can work as an ordinary JIT compiler for FreeBSD and Linux on Intel IA–32, it can also support MetaVM. The shuJIT generates native code which is capable of handling remote as well as local objects. We can use an ordinary Java compiler and a JVM with the runtime compiler because any change or addition to the JVM and the language has not been introduced.

2.1 Programming Interface

Programmers have to know only a method to specify the place where sequent object creations are done. The following code segment is an example of how to create an object on a remote computer.

```
// A designation of the place
// on which objects should be created.
VMAddress addr = new VMAddress("hostname");
MetaVM.instantiationVM(addr);

// An instantiation.
Object obj = new Object();

// A designation which specifies that
// instantiations should be done locally.
MetaVM.instantiationVM();
```

An invocation to the method `instantiationVM()` associates the place specified as the argument with the thread currently being executed. A programmer can exploit all functions of the Java language to deal with the `obj` object even though it is a remote object.

3 Transparency

In this section, we discuss what we consider as transparency by comparing MetaVM with the existing distributed object systems for C++ and Java including CORBA [4], Voyager [5], RMI [6], RyORB [7].

3.1 Type of Remote Reference

By using MetaVM, a programmer and an already-compiled bytecode can deal with a remote reference in the same type as it would be the actual remote object. Semantic expansion of type-checking instructions (`instanceof` and `checkcast`) enables this.

3.2 Remote Reference to Array

In a JVM, arrays are also objects. Existing systems have not allowed the referring of an array remotely, but MetaVM does.

3.3 Semantics of Argument Passing

When a method is invoked in Java, a reference to an argument object is passed to the callee method. MetaVM preserves the semantics in the case of passing objects via a network. Furthermore, if the passed reference points to a local object, the remote reference is automatically replaced with a local reference. The replacement of a reference is difficult to do without assistance by a JVM or a runtime compiler. This preservation of the semantics is one of the notable attributes of MetaVM.

3.4 Access to Array Elements and Fields

Besides allowing remote method invocation, MetaVM enables access to the elements of a remote array and the fields of a remote object. Existing systems for Java and C++ do not permit such operations because the systems perform remote operations using polymorphism of method invocation.

4 Code Generation Method

Native code generated by shuJIT transmits an instruction for an object operation to a remote JVM if the target object of the operation is a remote object. The native code invokes the MetaVM library and actual transmission and communication are performed by the library. The shuJIT generates the following native code:

```
if ((obj instanceof Proxy) && remote_flag)
   delegates the operation to the MetaVM library
else
   operates the object normally
```

The `obj` is a reference to the target of the operation and the `Proxy` is a class for representing a remote reference. The `remote_flag` is an internal boolean flag that indicates whether MetaVM should show a remote reference as a raw `Proxy` object or as a remote object. This flag belongs to each thread.

For the following bytecode instructions, shuJIT will generate a code different from what shuJIT working as a normal JIT compiler will generates.

- Instantiation:
  `new`, `newarray`, `anewarray`, `multianewarray`
- Field access: `getfield`, `putfield`
- Array access:
  `[ailfdbcs]aload`, `[ailfdbcs]astore`
- Get array length: `arraylength`
- Method invocation:
  `invoke`{`virtual, special, interface`}
- Type check: `checkcast`, `instanceof`
- Monitor handling:
  `monitorenter`, `monitorexit`

'[. . .]' equals one of characters in the brackets and '{. . .}' equals one of words separated by comma.

Although a JVM has over 200 bytecode instructions, Only the 30 instructions listed above have to be treated. Furthermore, many segments of native code can be shared by several instructions.

5 Experimental Results

All experiments were done using 233-MHz Pentium, Linux, and JDK 1.1.7. A runtime compiler TYA was used with all systems except for MetaVM which required shuJIT. The version of Voyager used was 2.0.2, HORB was 1.3beta4, and RMI was the one attached to the JDK. The network used was a 10 Mbps Ethernet.

5.1 Remote Method Invocation

Figure 2 show the latency of an one time remote invocation for the method which accepts two arguments and returns nothing (`void method(Object obj1, Object obj2)`). The latency of MetaVM was almost the same as that of the other systems.

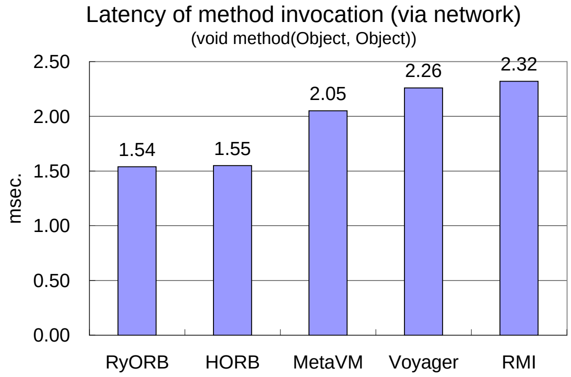

Figure 2: Latency of remote method invocation

5.2 Remote Field Access

Figure 3 show the latency of an one time remote read and write to a 32-bit integer field. Actually, except for MetaVM, an invocation to an accessor method that we prepared was done instead of a field access because they do not support remote field access.

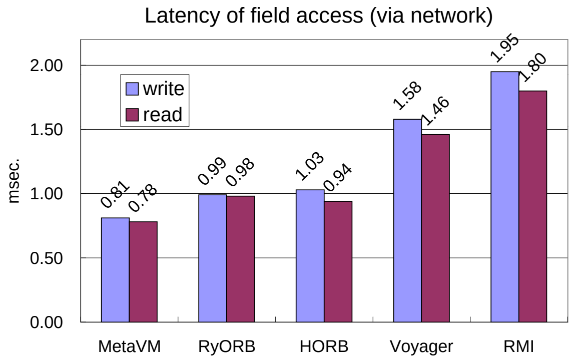

Figure 3: Latency of remote field access

MetaVM shows relatively better results compared to the remote method invocation. The reason for this was that remote field access is one of the native functions of MetaVM.

5.3 Local Operation

MetaVM introduces a certain amount of overhead into the local execution of a Java program even if the program does not perform any remote operations.

Figure 4 shows the results for CaffeineMark 3.0. The four conditions for our local operation test were as follows: without a runtime compiler, with shuJIT (which does not support MetaVM), with MetaVM whose remote array handling function was disabled (indicated as

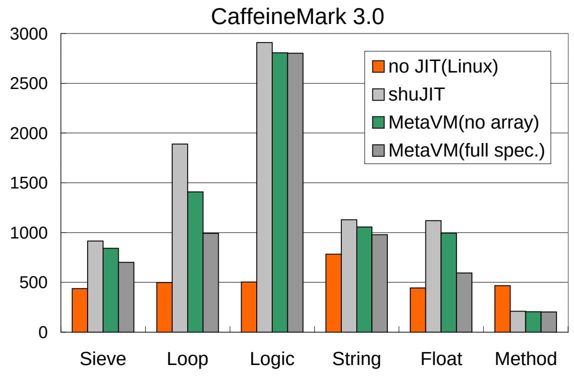

Figure 4: Performance of local execution – CaffeineMark 3.0

'no array'), with MetaVM operating with its full features ('full spec.').

If remote array handling is disabled, MetaVM copies arrays when they are passed over a network and the semantics of distributed execution can be differentiated from a local execution. The results shows that part of the decrease of performance is avoidable if we submit to such incompatibility of the program semantics.

6 Related Work

The cJVM [8] is a JVM to provide a single system image of a cluster of computers. There is currently no runtime compiler available which can work with the cJVM even though the cJVM is aimed at achieving high performance. Development of the interpreter does not affect a runtime compiler working with the interpreter. In the end we still have to develop the runtime compiler.

Obliq [9] is an object-oriented language with a distributed lexical scope. Our study suggests that the type of network transparency provided by Obliq can be achieved even in Java virtual machines. Moreover, our method basically exploits the high performance of native code unlike the interpretation, because MetaVM cooperates with a runtime compiler.

7 Conclusion

We described a method to construct a transparent distributed object system for a Java virtual machine. The method implies the *semantic expansion* of bytecode instructions by a runtime compiler. It enabled handling remote objects as well as local objects, and provided a single machine image to programmers. In results of our performance evaluations, Our MetaVM demonstrated low latency although the system achieved a higher level of transparency.

References

[1] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison Wesley, 1997.

[2] Kazuyuki Shudo and Yoichi Muraoka. Efficient implementation of strict floating-point semantics. In *Proc. of 2nd Workshop on Java for High-Performance Computing (in conj. with ICS'00)*, May 2000.

[3] Kazuyuki Shudo. shuJIT—JIT compiler for Sun JVM/x86, 1998. http://www.shudo.net/jit/.

[4] Object Management Group, Inc. CORBA/IIOP 2.3.1 Specification. http://www.omg.org/corba/corbaiiop.html.

[5] ObjectSpace, Inc. Voyager. http://www.objectspace.com/products/Voyager/.

[6] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java^{TM} system. In *The Second Conference on Object–Oriented Technology and Systems (COOTS) Proceedings*, pages 219–231, 1996.

[7] Ryo Neyama. RyORB—Ryo's object request broker for Java, 1998. http://www.shogi.ne.jp/RyORB/.

[8] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a single system image of a jvm on a cluster. In *Proc. of 1999 International Conference on Parallel Processing (ICPP-99)*, September 1999.

[9] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.