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.*

This paper presents the simple programming interface, the code generation method of MetaVM, and our experimental performance results. The results demonstrate efficiency of remote operations.

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

1 Introduction

A distributed object system is an instrument to develop a network distributed system in object-oriented programming languages. One of important benefits of such systems is to release programmers from the burden of exchanging information via a network. Programmers can write a distributed system in an object-oriented manner without concern for communication protocols.

Java [1] has recently been rapidly adopted as a programming language to develop distributed systems. Java language and virtual machine (JVM) [2] have many dynamic features desirable for use in distributed systems. For example, a JVM can load code from another computer via a network, and run the code safely without affecting the data and users on the computer where it is run.

MetaVM is a distributed object system we have developed. The system strongly depends on a runtime compiler to achieve its location transparency. For performance reasons, most JVMs have a runtime compiler called Just-In-Time (JIT) compiler. A runtime compiler converts Java bytecode to processor native code during program execution, 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.

In this paper, we describe the structure, the simple programming interface, and the code generation method of MetaVM. We also present experimental results that include the latency of remote operations and the performance of local execution.

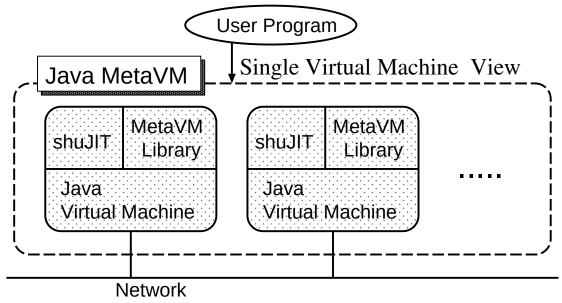

Figure 1: Structure of MetaVM

2 Overview of MetaVM

Figure 1 shows the structure of MetaVM. The system consists of computers connected with TCP/IP. Each computer has a JVM, a runtime compiler called shuJIT [3][4], and a MetaVM library.

shuJIT 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. Its code generation method is discussed in Section 4.

MetaVM library A class library written in Java. It performs most of the roles for MetaVM that include communications, forwarding operations to remote objects, and supplying class definitions over a network.

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. Note that the invocation does not affect other threads at all. 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 [5][6][7][8][9][10] for C++ and Java. We also mention a few of the remaining differences between a true single JVM and a single machine image provided by MetaVM.

3.1 Transparency provided by MetaVM

3.1.1 Absence of Special Preparation

A programmer has to only compile a source code with a normal Java compiler. A few of the existing systems are capable of doing this.

With most of these existing systems, a programmer has to specify methods or functions that are to be invoked remotely. The declaration should be written in IDL (interface definition language) for CORBA [5] and in the `interface` of the Java language for Java RMI [8]. Additionally, RMI needs preprocessing by a `rmic` stub compiler. HORB [10] and CORBA implementations also require similar preparation.

3.1.2 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.

With RMI, any remote object is dealt in the predeclared `interface` type. Voyager uses the same way as RMI, except it provides an interface generation tool.

3.1.3 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.

3.1.4 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.

3.1.5 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.

3.2 Incompatibility with a True Single JVM

3.2.1 Distributed Class Objects

Each JVM individually loads a classfile which has the definition of a class. Therefore, class objects (which are instances of `java.lang.Class` class) represent one class, but they are distributed to each JVM. Consequently, class variables belonging to class objects may have different values, although they should have a sole value in a true single JVM.

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 for a bytecode instruction that operates an object:

```
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. In short, the operation is delegated to the remote computer if the target of the operation is a remote object and the flag is set.

A different native code is generated for a bytecode instruction which creates a new instance:

```
if (remote_flag &&
     !(clazz is a class whose instances
        should be passed by copy))
   creates an instance on the remote computer
```

`else`
creates an instance on the local computer

Here the `clazz` is a class whose instance should be created. The generated native code judges whether the target object of the operation stays on a remote computer or a local computer. Because the judgement is performed in runtime, it introduces overheads to execution performance, even if there are no remote operations. We measured the overheads and the results are discussed in Section 5. The size of a generated code may be larger than the code which does not support remote operation.

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

- Creation of objects
  - Class (not array)
    `new`
  - Array class
    `newarray`, `anewarray`, `multianewarray`
- Access
  - Field
    `getfield`, `putfield`
  - Array element
    `[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

We measured the performance of two kinds of remote operations and overheads introduced to local operations. All experiments were done using 233-MHz Pentium equipped computer, Linux 2.2.1, and JDK 1.1.7. A runtime compiler TYA [11] 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.

5.1 Remote Operation

Remote operation performance was measured and compared. Each experiment was performed for both one computer and two computers connected via a network. The one computer experiment should show only overheads introduced by remote operation support. It should not show any latency of network. Conversely, if the operation can exploit the parallelism between caller and callee and they overlap, the parallelism is spoiled.

In our experiment using two computers, the second computer had a 350 MHz Pentium I-I, Linux 2.2.9, and JDK 1.1.7. Remote operations were initiated by the former computer and referred to the latter one. The network used was a 10 Mbps Ethernet.

5.1.1 Method Invocation

Figure 2 and 3 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)`).

RyORB had the smallest latency. The latency of MetaVM was almost the same as that of the other systems.

5.1.2 Field Access

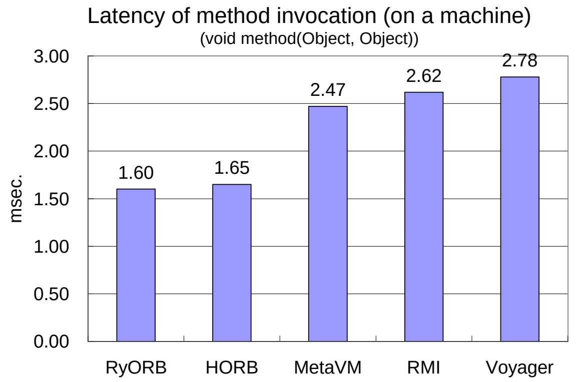

Figure 2: Latency of remote method invocation (one computer)

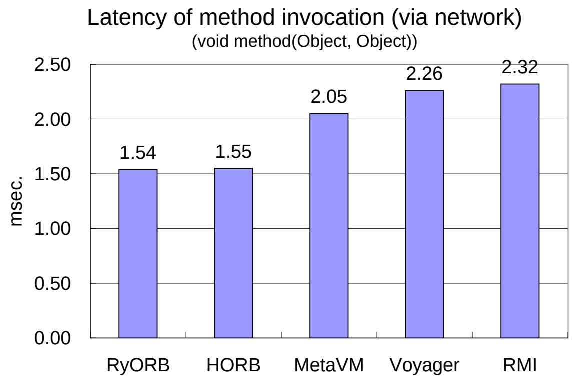

Figure 3: Latency of remote method invocation (two computers)

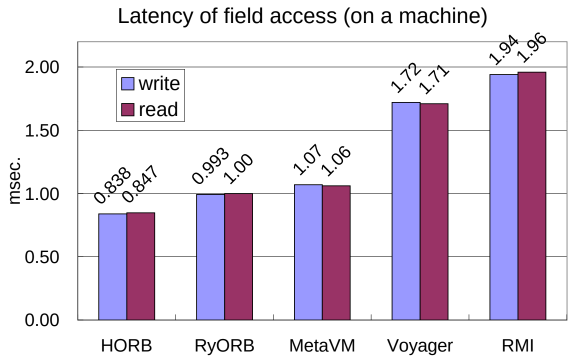

Figure 4: Latency of remote field access (one computer)

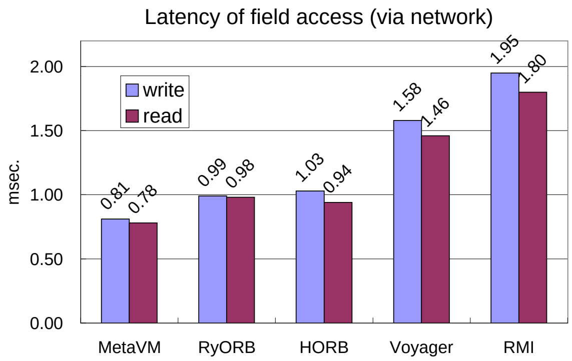

Figure 5: Latency of remote field access (two computers)

Figure 4 and 5 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.

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.

5.2 Local Operation

As mentioned in Section 4, 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. We measured how much overhead was introduced.

Figure 6 shows the results for CaffeineMark 3.0 [12] and Figure 7 shows the results for the Linpack benchmark [13]. CaffeineMark shows the number of times the JVM can perform a job in a specific time. 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 'no array'), with MetaVM operating with its full features ('full spec.').

The 'Float' of CaffeineMark was the benchmark that experienced a significant decrease. But, the decrease introduced by MetaVM was limited to 53%, which is the ratio of MetaVM

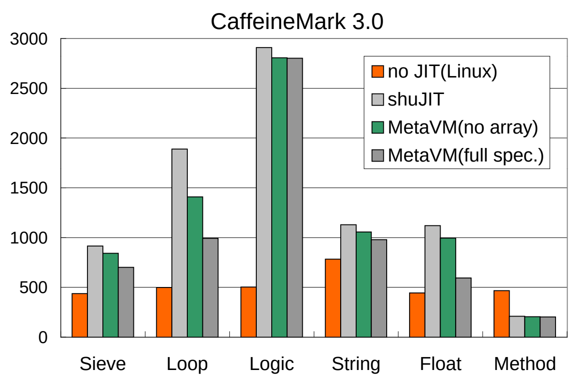

Figure 6: Performance of local execution – CaffeineMark 3.0

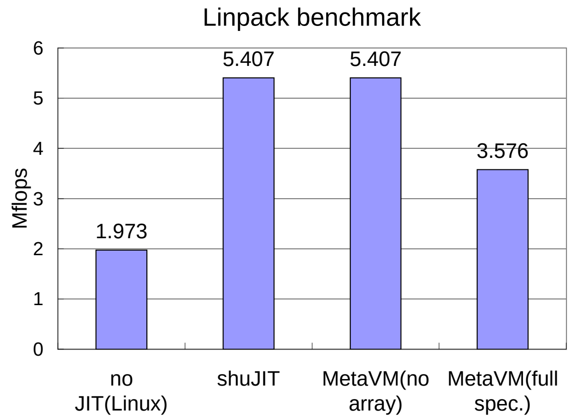

Figure 7: Performance of local execution – Linpack benchmark

to shuJIT. All results with MetaVM were better than those for the Java bytecode interpreter except for a benchmark where shuJIT was inferior to the interpreter (The 'Method' of CaffeineMark).

If remote array handling is disabled, MetaVM copies arrays when they are passed over a network. In that case, the semantics of distributed execution by MetaVM 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. For instance, performance of the Linpack benchmark did not decrease when the array handling function is disabled because the benchmark did not contain any object operations except array accesses.

6 Related Work

The cJVM [14] 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 [15] 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.

OpenJIT [16] is another runtime compiler of Java bytecode. Most of it has been written in Java so programmers can control the behavior of the compiler by using the Java language. We are not yet sure whether or not it is possible to develop a MetaVM-like system with OpenJIT. In principle, OpenJIT can provide the facilities which are necessary to implement MetaVM.

At University of California at Berkeley, research on low-latency communications via Myrinet in the Java language has been done [17]. Here shuJIT was used as the basis on which a kind of semantic expansion was built.

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. We have developed a system named MetaVM and demonstrated the semantic expansion. In results of our performance evaluations, MetaVM demonstrated low latency although the system achieved a higher level of transparency.

References

[1] James Gosling, Bill Joy, and Guy L. Steele Jr. *Java Language Specification*. Addison Wesley, 1996.

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

[3] 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.

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

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

[6] Inc. Sun Microsystems. Java^{TM} IDL Documentation. http://www.javasoft.com/products/jdk/idl/.

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

[8] 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.

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

[10] Satoshi Hirano. HORB: Distributed execution of Java programs. In *Proceedings of World Wide Computing and Its Applications*, March 1997.

[11] Albrecht Kleine. TYA Archive. http://sax.sax.de/˜ adlibit/.

[12] Pendragon Software Corporation. CaffeineMark 3.0. http://www.pendragon-software.com/pendragon/cm3/.

[13] Jack Dongarra and Reed Wade. Linpack benchmark — Java version. http://www.netlib.org/benchmark/linpackjava/.

[14] 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.

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

[16] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT — a reflective Java JIT compiler. In *Proc. of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, pages 16–20, October 1998.

[17] Matt Welsh and David Culler. Jaguar: Enabling efficient communication and I/O from Java. In *Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications*, December 1999.