Noncooperative Migration of Execution Context in Java Virtual Machines

Kazuyuki Shudo Yoichi Muraoka

School of Science and Engineering, Waseda University,
3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, JAPAN
{shudoh,muraoka}@muraoka.info.waseda.ac.jp
http://www.muraoka.info.waseda.ac.jp/

Abstract. The migration of the execution context has been applied to remote execution and mobile agents, and noncooperative migration can be applied to even more applications, such as load balancing. We have therefore designed a system for the migration of Java threads, one that allows noncooperative and heterogeneous migration of the execution context of the running code. This paper describes an overview of the system, the problems we have faced in designing its facilities, and the results of preliminary evaluations of it performance.

1 Introduction

The migration of execution context, known as process and thread migration, has long been applied to load balancing and remote execution[3][7][2], and has recently been applied to mobile agents[16][13][12]. One of the challenges remaining in this area is to design facilities that support noncooperative and heterogeneous migration, as well as the execution of native code.

We are designing a migration system for Java threads, and it will provide all three of these functions. The system, called MOBA, supports the noncooperative migration of Java threads in heterogeneous various platforms but the facility for execution of native code has not been implemented yet. The system allows the migration of an execution context to be ordered using simple programming and user interfaces. Programmers can instruct a thread to migrate with one statement: "goTo(destination)". A migrant in the system is a thread with its execution context. An execution context consists of the states of methods being executed, and it resides in stack frames that contain the program counter, operands, and local variables. The context can be saved and moved to other Java virtual machine(JVM)s without cooperation of the running code, and its execution can be resumed.

"Noncooperation" means that the migration can be carried out without the awareness of the running code. Noncoopeative migration allows entities outside of the migrating thread to give the order to migrate. Users and threads other than the migrant can issue orders. Accordingly, an appropriate program, such as a job scheduler, can attempt to balance loads of JVMs with migration.

Heterogeneity of machines, such as differences in processors and operating systems, surely complicate the migration approach. Our system make heterogeneous migration[15] possible by handling the execution context in the JVM rather than in a particular processor or in an operating system. Threads in our system can thus migrate between JVMs on different platforms.

In the rest of this paper, we describe an overview of the designed system and discuss design and implement issues of thread migration facilities. In the section 2, we try to compare MOBA with some existent systems and methods. In the section 5, we show the results of preliminary performance evaluations.

2 Related Work

system	language	context	execution	heterogeneity	noncooperation
MOBA	Java	yes	interpreter	yes	yes
Sumatra	Java	yes	interpreter	yes	no
JavaGo	Java	yes	JIT or interpreter	yes	no
Voyager, Aglets	Java	no	JIT or interpreter	yes	no
TeleScript	TeleScript	yes	interpreter	yes	no†
Emerald	Emerald	yes	native code	no	yes
Arachne	C, C++	yes	native code	yes	no

† The mechanism does not prevent noncooperative migration. But the way to issue an order from the outside of the running code is not provided.

Table 1: Properties of various migration systems.

The properties of systems supporting the migration of execution context and mobile agents are listed in Table 1. The entry in the *context* column shows whether or not the system supports migration of execution context, that in the *execution* column shows how the code is executed, that in the *heterogeneity* column shows whether or not the system allows migration between machines differing in processor and underlying OS, and that in the *noncooperation* column shows whether or not the system allows noncooperative migration.

Sumatra[11] and TeleScript[16] are the systems most similar to MOBA. They differ in programming language — TeleScript has its own language, whereas MOBA and Sumatra adopt Java — but their mechanisms are similar. The code is executed by an interpreter, and the execution context can migrate. Only MOBA, however, provides noncooperative migration. Migration can be carried out without awareness of the running code. In other words, orders to migrate can be issued by entities other than the running code. This means that the timing of migration can be determined at run-time. Although the mechanism of TeleScript doesn't prevent noncooperative migration, TeleScript does not offer programming or user interfaces for it. And Sumatra allows only explicit migration using a `go()` method.

JavaGO[13][12] and Fünfrocken's method[4] also can save and move the execution context of a Java program and restore it on another machine. Their approach is based on pre-process or source code translation. Arachne[2], which is a thread system for the C and C++ languages, is also based on the sort of method. Their method applied to Java has the advantage of being able to work with fast existing JIT compilers. Dedicated JVMs, either that are extended or built from scratch, can not benefit from existing JIT compilers. One of the problems of this translation approach is that its area of application is more limited than that of a runtime-system approach. Because the translation approach requires the timing of migration to be described in migratory codes, it cannot be applied to some of the applications that need noncooperative migration, such as load balancing.

3 Overview of the System and the Scheme

MOBA is implemented as a plug-in to the JVM, that is implemented by Sun Microsystems and dealt out distributed as Java Development Kit(JDK) and Java Runtime Environment(JRE). The most part of MOBA is written in Java. Although some of its code is in C language, the system supports any UNIX platform where Sun JVM can run. MOBA is a plug-in, not a JVM built from scratch, so a program utilizing MOBA functions can also utilize plenty of functions provided by the JDK.

3.1 Programming and User Interface

The programming interface provided by MOBA is so simple that only a few changes to the original code are needed to make the code migratory. To make the thread movable, we use the `MobaThread` class instead of the normal `Thread` class to instantiate the thread. To migrate to another machine, call the following method.

`MobaThread.goTo(`*destination*`)`

Unlike programmers working with existent mobile agent systems for Java[8][10][5], programmers working with MOBA have to pay little attention to the particular programming interface.

Migration can be ordered not only by the migrant but also entities outside of the migrant, such as other threads and users. In this case, no statement to migrate is required in the migrant's code. Other threads in the same JVM, where the migrant stays, call the following method to move the thread.

<*target thread*>`.moveTo(`*destination*`)`

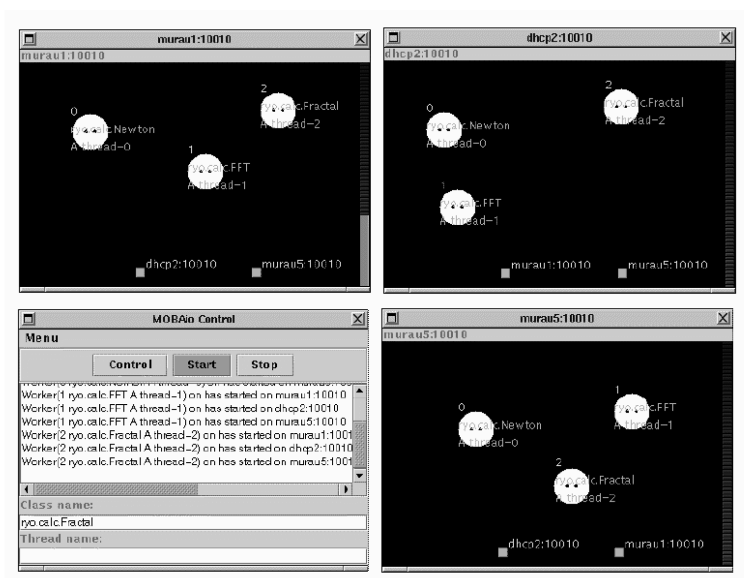

Figure 1: Graphical user interface to visualize mobile threads and order to migrate.

Furthermore, users can issue the order to migrate by using some user interfaces, either character-based or graphical(Fig. 1).

3.2 Organization of the Facilities

The migration facilities MOBA consist of some libraries, introspection, object marshaling, thread externalization, and thread migration. Their relation and dependency are shown in Fig. 2. The

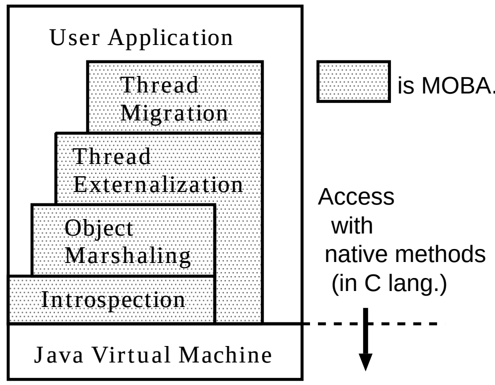

Figure 2: Organization of thread migration facilities.

introspection library provides the same function as the reflection library which is part of the standard library of Java. Similarly, object marshaling provides the function of serialization. Thread externalization translates a state of the running thread to a byte stream, and it is used by the thread migration library for moving threads between JVMs. Thread externalization can be used not only for migration, but also for persistence and fault tolerance. We, or an appropriate daemon program, can save states of running threads to a disk or a database in order to provide for an unforeseen fault of an underlying OS or machine.

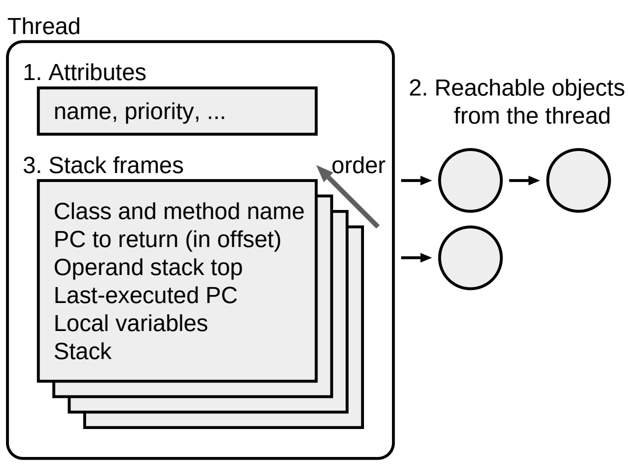

Figure 3: Procedure to externalize a thread.

The procedure to translate a thread to a byte stream is represented in Fig. 3. First of all, some attributes of the thread (name, priority and so on) are translated. Then, after objects that are reachable from the thread object are marshaled, execution context is treated. A context consists of contents of stack frames generated by a chain of method invocations. The externalizer follows the chain from older frame to newer one and serializes the contents of the frame. A frame is located on the stack in a JVM and contains the state of a called method, The state consists of a program counter, operands to the method, local variables, elements on the stack. They are serialized in machine-independent form.

Marshaling all the objects reachable from the thread object may be a problem. The migration system which handles remote references can leave some objects in the source JVM of migration. To access the objects left in the source JVM from the thread which has migrated is expensive because the access requests and replies have to pass through the network. Moving many objects takes a lot of time, but accessing the moved objects takes less time than does accessing the objects that were left at the source JVM. Deciding whether or not an object should be moved is not a trivial dicision.

4 Design Issues of Thread Migration in JVMs

4.1 Noncooperative Migration

"Noncooperative migration" is migration without awareness by the migrant. The migration in MOBA does not need any cooperation of the migrant program. Cooperative migration is invoked by the migrant program itself. This kind of migration is suitable for some applications (e.g., describing mobile agents) but not for other applications. Dynamic load balancing and saving running states for fault tolerance are two applications that need noncooperative migration.

MOBA allows noncooperative migration, but it requires nonpreemptive scheduling of Java threads. Sun JVM can utilize two kinds of libraries as an underlying thread library: OS native threads (e.g., Solaris native threads) and green threads. Scheduling in green threads is nonpreemptive, so it always allows noncooperative migration. Although scheduling of threads is preemptive, cooperative migration (e.g., `goTo(`*destination*`)`) can be carried out.

If scheduling is preemptive, threads can be suspended in the middle of the execution of a bytecode instruction. At that time, elements of the execution context (such as a program counter, a stack pointer and so on) can be inconsistent. Nonpreemptive scheduling, on the other hand ensures that the context of a suspended thread is consistent. In nonpreemptive scheduling, the thread suspends its own execution by calling methods to wake-up the scheduler. For example, the `Thread.yield()` method, and some other methods that call system calls of the OS, kick the scheduler.

4.2 Runtime Compilation

Most JVMs have a runtime compiler called a Just-in-time compiler (JIT). It translates bytecode to processor native code at runtime. A runtime system that supports the capturing of execution context, however, is incompatible with existing JIT compilers. Because the approach of MOBA and Sumatra[11] provides a runtime system supporting execution-context capture, they cannot work with existing JIT compilers.

Heterogeneous migration needs a machine-independent representation of execution context, but most existing JIT compilers don't preserve a program counter on bytecode. Only the counter on native code can be obtained during execution of the native code generated by an existing JIT compiler. But, Sun's HotSpot VM[14] may allow the execution context on bytecode to be captured during the execution of the generated native code. Its details is not documented, but capturing the program counter on bytecode seems to be needed for its dynamic deoptimization. Common JIT compilers do not allow it.

The approach based on source code translation[13][4] may work with any existing JIT compiler and thereby benefit from the JIT. But as noted earlier, this approach requires the timing of migration to be determined when writing codes, it cannot be applied to areas such as load balancing and fault tolerance.

Although none of the JIT compilers presently available can work with the runtime system, but it should in principle be possible to design a JIT compiler that supports the capture of execution context.

we are now developing the sort of JIT compiler: the native code sometimes checks a flag during its execution, and the flag indicates a request for capturing the context. This polling may have some cost in term of performance, but we expect any decrease in performance to be small.

4.3 Marshaling Objects Tied to the Resources

MOBA does not offer a function handling objects that reside in a remote machine, and it thus cannot by itself handle a reference to a remote object. Objects reached from the migratory thread are copied to the migration target. How to maintain objects which have some relation to resources specific to the machine is a common problem in object migration systems. File and socket descriptors are examples of the resources.

The general solution for the system to support remote reference is making the objects fixed to the machine[7][11], but MOBA cannot use that solution because so far it cannot handle remote references. To prevent accidents caused by an attempt to move the resources, MOBA disables some kinds of resources, such as a file descriptor. Consequently, the file descriptor becomes invalid (e.g., -1) at migration. Programmers have to be conscious that when writing the code.

Classes may be grouped into some categories with regard to their dependence on resources. Some classes whose instances are tied to the resources must be maintained, but others do not have to be maintained. When classes are newly written by programmers, the programmers can write specific marshaling methods for them. But, if the classes reside in the Java standard library, they are not aware of the migration of their instances. If the object migration system does not make objects stationary, it has to treat the resources tied to the classes. Current MOBA maintains some classes such as `FileDescriptor`.

4.4 Types of Values on the JVM Stack

The runtime system that supports the capture of execution context must know the types of values in the stack of a JVM. Local variables and operands of the called method stay on the stack. The values may be 32-bit or 64-bit immediate values or references to objects.

It is difficult to distinguish the types referring only the value. A Sumatra interpreter maintains a type stack parallel to the value stack[1], and distinguishes the type with it. Sumatra has its own interpreter built from scratch, so it can use this method. But MOBA is a plug-in to the existing Sun JVM, which does not have a type stack like Sumatra. If the class file has `LocalVariableTable` attributes[9], the types of local variables can be obtained in the table. But in general, in a Sun JVM there is no information about the type of values in the stack.

With a JVM like Sun's, we have either to infer the type from the value else determine the type by tracing the bytecode of the method from the head like a bytecode verifier. Tracing bytecode to determine types is computationally expensive, so MOBA infers the type from the value. It distinguishes a reference from an immediate value by utilizing the fact that all references reside in a particular area of the address space and are aligned in 64-bit boundaries. A reference cannot be mistaken for an immediate value, Although an immediate value will infrequently be mistaken for a reference. To confirm its inference, MOBA checks the validity of a value recognized as a reference. Whether the value has a right structure as a reference is checked. The validation cannot be perfect but it will be effective enough to ensure the result of recognition.

5 Performance Evaluation

We evaluated the performance of MOBA's mobility function by using two machines connected via one ethernet repeater, in a 100-Mbit/sec Ethernet. One of the machines had an UltraSPARC–II 167-MHz processor, the other had an UltraSPARC–II 296-MHz processor, and SunOS 5 ran on both machines. We used the reference implementation of JDK 1.1.8 with MOBA and used the production release of JDK 1.1.7 with other systems, and we used interpreter with MOBA and Sun JIT compiler with other systems since MOBA can't work with existing JIT compilers.

5.1 Latency of Migration

We described a simple and lightweight migrant with MOBA and with Voyager ORB 3.0[10], and then made them go and return. Migration times obtained with the systems are listed in Table 2. It is interesting whether thread migration can be comparable with the mobile agent system like Voyager, which does not support migration of execution context.

# of times go and back	1	50
MOBA	191.0	105.32
Voyager	292.5	37.08

Table 2: Latency of an one-way migration (msec).

When a go and back is performed once, MOBA is faster even though it moves the execution context in addition to the data held by the migrant.

5.2 Throughput

We also used MOBA for remote execution, measured the data transfer throughput, and compared it with the throughput obtained using two object request broker(ORB)s for Java, RMI[17] and HORB[6]. With those we used a remote method invocation with an argument and no return value. The argument was a large array of 64-bit floating point value. With MOBA, we used a migrant with a large array and return without any data.

As shown in Fig. 4, when the amount of data transferred is small MOBA takes more time than the ORBs do because it moves the execution context as well as data. Thus time taken by a migration with MOBA is larger than latency of a remote invocation of ORBs, but the data transfer throughput is better with MOBA than with the other systems.

6 Conclusion

A migration system for Java threads has been implemented as a plug-in to an existing JVM, and it supports noncooperative migration of execution context with nonpreemptive scheduling of threads. Some problems pointed and discussed here were whether objects reachable from the migrant should be moved, how the types of values in the stack can be identified, compatibility with JIT compilers, and how resources tied to moving objects should be handled.

As a further study, we are currently designing a JIT compiler that can work well with thread migration. Hereby noncooperative and heterogeneous migration with execution of native code will be got possible.

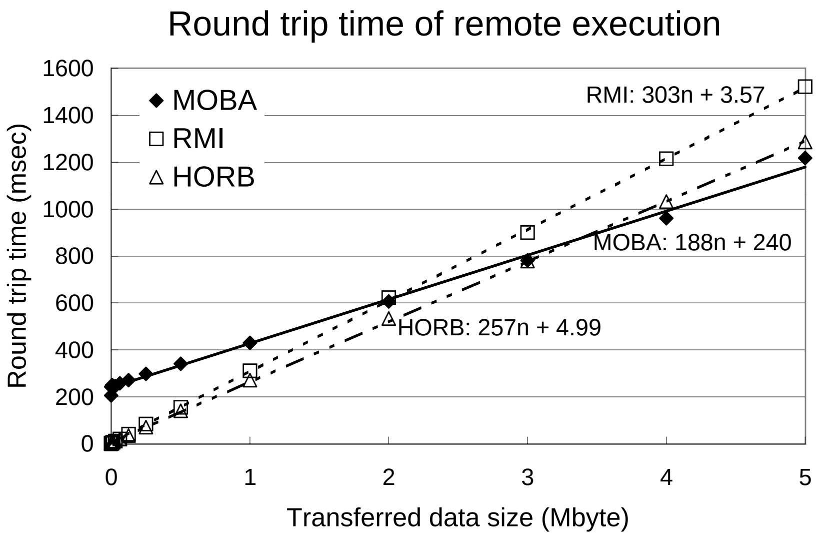

Figure 4: Round-trip time of remote execution.

References

[1] Acharya, A., Ranganathan, M., and Saltz, J. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems*, J. Vitek and C. Tschudin, Eds. Springer Verlag Lecture Notes in Computer Science, 1997.

[2] Dimitrov, B., and Rego, V. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transaction on Parallel and Distributed Systems 9*, 5 (May 1998), 459–469.

[3] Eskicioğlu, M. R. Design issues of process migration facilities in distributed system. *IEEE Technical Comittee on Operating Systems Newsletter 4*, 2 (Winter 1989), 3–13. Reprinted in *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press.

[4] Fünfrocken, S. Transparent migration of java-based mobile agents. In *Proc. of 2nd Int'l Workshop on Mobile Agents 98(MA'98)* (Sept. 1998), pp. 26–37.

[5] General Magic, Inc. Odyssey. http://www.genmagic.com/technology/odyssey.html.

[6] Hirano, S. HORB: Distributed execution of Java programs. In *Proceedings of World Wide Computing and Its Applications* (Mar. 1997).

[7] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the emerald system. *ACM Transaction on Computer Systems 6*, 1 (Feb. 1988), 109–133.

[8] Lange, D., and Oshima, M. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Longman, Inc., 1998.

[9] Lindholm, T., and Yellin, F. *The Java^{TM} Virtual Machine Specification*. Addison Wesley, 1997.

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

[11] Ranganathan, M., Acharya, A., Sharma, S., and Saltz, J. Network-aware mobile programs. In *Proceedings of USENIX'97* (Jan. 1997).

[12] Sekiguchi, T. JavaGo Manual, 1998. http://web.yl.is.s.u-tokyo.ac.jp/amo/JavaGo/doc/.

[13] Sekiguchi, T., Masuhara, H., and Yonezawa, A. A simple extension of Java language for controllable transparent migration and its portable implementation. In *To appear in a Springer Lecture Notes in Computer Science for International Conference on Coordination Models and Languages(Coordination99)* (1999).

[14] Sun Microsystems, Inc. The Java HotSpot^{TM} Performance Engine Architecture. http://www.javasoft.com/products/hotspot/whitepaper.html.

[15] Theimer, M. M., and Hayes, B. Heterogeneous process migration by recompilation. In *Proc. IEEE 11th International Conference on Distributed Computing Systems* (1991), pp. 18–25. Reprinted in *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press.

[16] White, J. E. *Telescript Technology: The Foundation of the Electronic Marketplace*. General Magic, Inc., 1994.

[17] Wollrath, A., Riggs, R., and Waldo, J. A Distributed Object Model for the Java(tm) System. In *The Second Conference on Object–Oriented Technology and Systems (COOTS) Proceedings* (1996), pp. 219–231.