Bytecode Transformation for Portable Thread Migration in Java

Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa

Department of Information Science, Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113-0033
{takas, cocoa, yonezawa}@is.s.u-tokyo.ac.jp

Abstract. This paper proposes a Java bytecode transformation algorithm for realizing transparent thread migration in a portable and efficient manner. In contrast to previous studies, our approach does not need extended virtual machines nor source code of target programs. The whole state of stack frames is saved, and then restored at a remote site. To accomplish this goal, a type system for Java bytecode is used to correctly determine valid frame variables and valid entries in the operand stack. A target program is transformed based on the type information into a form so that it can perform transparent thread migration. We have also measured execution efficiency of transformed programs and growth in bytecode size, and obtained better results compared to previous studies.

1 Introduction

Mobile computation is a promising programming paradigm for network-oriented applications where running computations roam over the network. Various kinds of applications are proposed such as electric commerce, auction, automatic information retrieval, workflow management and automatic installation.

The ability to preserve execution states on migration is an important criterion of classifying programming languages for mobile computation [3, 4]. Migration is called *transparent* if a mobile application is resumed at the destination site with exactly the same execution state as before [7]. From the viewpoint of *programming* mobile applications, the notion of transparent migration is especially important since it allows the programmer to write mobile applications in the same way as writing ordinary non-mobile applications. It substantially supports the mobile application programmer to understand program behavior. Early mobile language systems such as Telescript [8] and Agent Tcl [7] had a mechanism of transparent migration.

Transparent migration, however, is not adopted in major Java-based mobile language systems (e.g., IBM Aglets [9] and Voyager [12]) though Java has been very popular among people who are interested in mobile computation. There is a difficulty in implementing transparent migration in Java. In order to move computation transparently, the call stack needs to be preserved across migration. But the Java security policy forbids Java bytecode itself to manipulate the stack.

Two different approaches have been proposed for realizing transparent migration in Java: extending a Java virtual machine and transforming source code.

Both approaches, however, have their own difficulties. The former approach requires mobile applications to run only on modified virtual machines. This nullifies one of the advantage of Java, which is *ubiquity of common virtual machines*. On the other hand, the latter approach is not applicable when source code is not available. In fact, Java source codes are often unavailable.

In contrast to the two approaches above, our approach can avoid these drawbacks by *bytecode transformation*. In our approach, bytecode instead of source code is transformed into the form that makes transparent migration possible.

Bytecode transformation has several advantages compared to source code transformation. The Java language forces clean programming, where only structured control transfer is allowed. When a method is resumed, we want to transfer the control to a suspended point in the method. A suspended point can be in a compound statement, but the Java language forbids a control transfer into a compound statement. In contrast, there is `goto` instruction in the Java bytecode set. The control can be transferred to any program point in a method if it is allowed by the bytecode verifier. The use of `goto` instruction can reduce the size of inserted code fragments for control transfer, and hopefully it can improve execution efficiency of transformed codes. Actually, as shown later, codes produced by our bytecode transformer show better execution efficiency on JDK 1.2.2 than those produced by a source code transformer [13].

There are two difficulties in bytecode transformation. (1) Transformed codes must pass a bytecode verifier. (2) It is difficult to know the set of values to be saved and restored. In the bytecode level, values are passed by frame variables and the operand stack. There are neither variable declarations nor scoping rules. To obtain necessary information for bytecode transformation, we had to adopt a type system of Java bytecode [15], and to devise a static program analyzer on it.

The rest of this paper is organized as follows: Sect. 2 gives an overview of our mobile agent system. Sect. 3 describes our implementation scheme for thread migration. Sect. 4 explains a static program analysis for the bytecode transformation. Sect. 5 describes our scheme of bytecode transformation for transparent thread migration and gives an example of a transformed bytecode. Sect. 6 shows some experiments with our current implementation of the bytecode transformation. Sect. 7 discusses related work. Sect. 8 concludes this paper.

2 Overview of Our Mobile Agent System

This section gives an overview of our model of mobile computation and our mobile agent system.

The model of mobile computation adopted in our mobile agent system is simple, plain and direct so that it accommodates wide applicable domains. The subject and the unit of mobility in our system are a *thread*. A thread migrates to a remote site, preserving its execution states such as the call stack and a part of the heap image. The thread at the departure site will vanish and an equivalent thread will appear at the destination. In this sense, our model of migration is similar to those of Arachne threads system [5] and Emerald [16] rather than major Java-based mobile agent systems.

In our system, a place or a location to which a mobile agent migrates is a *Java virtual machine*. A mobile agent (thread) hops around a group of Java virtual machines.

Basically, a heap image that a migrating thread can refer to is duplicated to the destination. This may cause a serious security flaw because a secret data may be duplicated to a remote site implicitly. Our mobile language system does not provide a protection mechanism for that kind of flaws, but our system can be *combined* with various proposed techniques [2, 11] that prevent security flaws.

Though objects on a heap can be transmitted to a remote site, resources such as files, windows and sockets cannot be. These stationary resources, thus, cause a problem on migration if a mobile agent has references to them. We have two options for dealing with these resources. The first one is to force the programmer to use *transient* references for these stationary resources. A transient reference is automatically nullified on migration. The second one is to use the class library [20] adapted for mobile environment instead of the default libraries in JDK. This class library shares a common interface with JDK and it enables transparent access of stationary resources. A stationary resource is either accessed remotely by Java RMI or duplicated automatically to a destination on migration.

The bytecode transformation scheme described in this paper can be used in the class loader for mobile code. A class loader fetches class files from both local and remote sites if they are not loaded yet into a Java virtual machine. When a class loader detects a class file not modified yet, our bytecode transformer automatically transforms it at load time into a form in which its execution states can be saved and restored.

3 How to Move a Thread Over the Network

Our basic mechanism of transparent thread migration on Java virtual machines is, in principle, similar to other schemes based on source-code-level transformation [1, 6, 13, 17, 18]. A thread migration is accomplished by three steps:

- The execution states of a target thread are saved at the departure site into a machine-independent data structure. The thread terminates itself when the migration succeeds.
- The data structure representing the execution states of a target thread and a part of the heap image are transmitted through the network to the destination site.
- A new thread is created at the destination. Equivalent execution states of the target thread are reconstructed for the new thread.

The above whole process is implemented by using only standard mechanisms of Java and Java RMI.

3.1 Saving Execution States

The execution states of a thread consist of those of the methods in execution. The execution states of a method consist of (1) a program counter, (2) valid local (frame) variables, and (3) valid entries in the operand stack. The execution states of the methods are encoded into a data structure. Note that we assume that each method is transformed so that it can save its execution state *when a special exception is thrown.* When a migration is in operation, an exception is thrown. If a method captures the exception, the method stores its execution state to a newly created state object defined for each method, and then it propagates the exception to the caller of the method. This process is repeated until the exception reaches the bottom of the stack.

3.2 Transmitting Execution States

When the exception is captured at the bottom of the call stack, all the state objects are transmitted to the destination site by using Java RMI. All values on the heap that can be reached from the target thread are also transmitted to the destination by a mechanism of Java RMI.

3.3 Restoring Execution States

The execution states of a target thread is reconstructed from the state objects. The call stack is reconstructed by calling the methods in the order in which they were invoked at the departure site. Each method is transformed in advance so that it can restore its execution state from the state object. When a method is called with the state object, it restores the values of the stack frame, and it continues execution from the suspended point.

4 Bytecode Analysis

To transform bytecode for transparent migration, we need information on a set of all *valid* frame variables and entries in the operand stack for each program point. A variable or a slot is valid if a value on it is available for every possible control flow. Types of frame variables and entries in the operand stack are also necessary. In addition, a transformed code must pass a Java bytecode verifier if the original code passes it. To obtain such necessary information on bytecode, we adopt a type system for Java bytecode.

4.1 A Type System for Java Bytecode

Our bytecode transformer exploits exactly the same information for bytecode verification [10]. We adopt the formulation of bytecode verifier by Stata and Abadi [15]. It is a type system for a small subset of Java bytecode called JVML0.

If a bytecode is well-typed, it tells that the bytecode is verifiable. A type judgment is written in their type system as follows:

$$F, S, i \vdash P.$$

where P denotes a sequence of instructions that constitutes a method. F is a mapping from a program point to a mapping from a frame variable to a type. S is a mapping from a program point to an ordered sequence of types. Finally i denotes a program point or an address of code. Intuitively, the map $F(i)$ gives a type of a local variable at program point i. The string $S(i)$ gives the types of entries in the operand stack at program point i.

These F and S are useful to our bytecode transformation since they contain typing information about valid local variables and entries in the operand stack, respectively.

When a type judgment $F, S, i \vdash P$ is true, it tells that program P is *verifiable* at program point i. The whole program is verifiable if the program is verifiable for every program point in it. This is denoted by $F, S \vdash P$.

The type reconstruction problem for JVML0 is to find appropriate F and S such that $F, S \vdash P$ is true for given P. It is actually a verification algorithm itself. We have implemented a type reconstruction algorithm for the extended JVML0 to be explained next.

4.2 Extending JVML0 to the Full Set of Java Virtual Machine

Since JVML0 includes only a small subset of the Java virtual machine, we have extended it so that it incorporates the full set of Java virtual machine except bytecode subroutines. In doing so, the following points are important.

Instruction. The Java virtual machine has around 200 instructions. A typing rule is defined for each instruction.

Type. Stata and Abadi's type system has only three kinds of types: an integer type, a reference type and a return address type. We have to add primitive types, and the notions of inheritance and subtyping to reference types. In order to pass a bytecode verifier, our bytecode transformer has to know the most specific type of a value when the value is restored from a state object.

Exception. Since Stata and Abadi's type system lacks the mechanism of throwing and catching exceptions, we have to add a facility of handling exceptions.

5 Bytecode Transformation

This section describes our bytecode transformation algorithm. It takes a method in bytecode and produces a method that has instructions for saving and restoring its execution state. Because the produced bytecode also consists of the standard JVM instructions, the transformed method including the state handling mechanism is compatible with any standard JVM and any JIT compiler.

5.1 Overview of Bytecode Transformation

The transformation algorithm changes the signature of a given method to take a state object as an extra parameter and inserts the following code fragments in the method:

- An exception handler for each method invocation. The occurrence of migration is notified by a special exception. The exception handler is responsible for saving an execution state. The program counter to be saved is known since an exception handler is unique for each suspended point. The set of valid local variables and their types (whether it is a primitive type or a reference type) are found by the bytecode analysis described in Sect. 4 (from F and S). Even if a migration takes place in a `try` statement with the `finally` clause, the `finally` clause is not executed on migration, because we do not insert a `jsr` instruction to the `finally` clause in the exception handler.
- Instructions for saving valid entries on the operand stack. The contents on the operand stack are defined to be discarded when an exception is thrown, which means that their values cannot be fetched from an exception handler. The basic idea for saving values on the operand stack is to make the copies of them in the extended local variables before the method invocations. The valid entries on the operand stack are found from S.
- Instructions at the head of the method that restore all valid frame variables. When the execution state of a method is restored, a state object is passed to the method in the extended parameter. The inserted code restores all valid frame variables at the suspended point. After restoring the frame variables, the control is transferred to the suspended point.
- Instructions that put a state object as an extra parameter for a method invocation instruction.

5.2 State Class

```
public class STSamplefoo extends javago.StackFrame
  implements java.io.Serializable {
    public int      M_EntryPoint;
    public int[]    ArrayI = new int[1];
    public long[]   ArrayL = new long[2];
    public float[]  ArrayF = new float[3];
    public double[] ArrayD = new double[4];
    public Object[] ArrayA = new Object[5];
}
```

Fig. 1. A state class.

Our transformation algorithm defines a state class for each method. An execution state of a method is stored into an instance of the state class. Fig. 1

shows all field variables of a state class, where the array sizes are the maximum numbers of the values of the corresponding type in the execution states. These numbers are found by the bytecode analysis. The type of a value to be saved is either one of the primitive types (`int`, `long`, `float`, and `double`) or a reference type (`Object`).

5.3 Extending Method Frame for Local Variables

The set of the local variables are extended for saving valid entries in the operand stack and some other purposes. The amount of local variables is determined by the information S, which is obtained by the bytecode analyzer, and another eight local variables are used for special purpose in our current implementation of transformation. These reserved frame variables are used to keep the state object for the current method, the state object for the caller of the current method, a special exception that notifies migration, and array references in the current state object (that is `ArrayI`, ..., `ArrayA` in Fig. 1).

5.4 Example of a Transformed Bytecode

```
public class test {
    public static int foo(int x, int y) {
        int z = x + y;
        may_migrate();
        return z;
    }
}
```

Fig. 2. A toy method that may cause a migration.

We illustrate the bytecode transformation by using a toy method in Fig. 2. This method is compiled into the (pseudo) bytecode listed in the left part in Fig. 3, and it is transformed into the one listed in the right part. An address with a subscript or a prime denotes an inserted code.

When the `may_migrate` method wants to migrate, the method throws a special exception. The exception handler at line $4'$ catches the exception and saves the execution states (x, y and z). Then it propagates the exception to the caller method. On resumption, this method is invoked with a state object. It restores the execution state from the state object at line 0_3. The control is transferred to line $2'$ and then the execution state of the `may_migrate` method will be restored.

5.5 Which Method Should be Transformed?

Since our bytecode transformer changes the signature of a method, the transformer must modify method invocation instructions accordingly. Some methods,

Original bytecode.	Transformed bytecode.
1: `z = x + y;`	0_1: `if not resumption, jump to 1;`
2: `call may_migrate;`	0_2: `tablejump` 0_3`;`
3: `return z;`	0_3: `restore x,y,z; jump to` $2'$`;`
	1: `z = x + y;`
	$2'$: `push state_object;`
	2: `call may_migrate;`
	3: `return z;`
	$4'$: `save x,y,z; throw;`

Fig. 3. Original and transformed pseudo bytecode for the toy method.

however, preserve their signatures as mentioned in Sect. 5.6. An important question, thus, arises: which method should be transformed? There are two solutions: annotation by the programmer, and construction of a call graph. In the first solution, methods to be transformed are specified by the programmer. This scheme is simple and able to realize the programmer's intention correctly, but it cannot deal with class files written by others. In the second solution, every method that contains migration instructions is transformed. Besides, every method that *indirectly* invokes migration instructions is also transformed. The second kind of methods is found by constructing a call graph. Though this scheme can deal with class files written by others, it requires *all* class files before starting execution. It is difficult to achieve this requirement in cases that a mobile agent visits some location for the first time and that two unknown agents meet somewhere.

Our solution is to predesignate the set of methods not to be modified. Our transformer does not modify system classes and the signatures of callback methods in user code. This scheme does not need annotation nor a call graph, but an implementor of the transformer must have a good knowledge of system classes.

5.6 Limitations

Our transformation algorithm and its current implementation has some limitations.

First, our current transformer cannot handle programs that may migrate when a live return address of a subroutine is included in a local variable or an entry of an operand stack. The reason is that a return address cannot be saved arbitrarily under the restrictions of Java bytecode verifier. This limitation matters when a migration occurs in a `finally` clause of a `try` statement.

We are planning to eliminate this limitation in future. The basic idea is to keep traces of subroutine calls dynamically. The bytecode transformer inserts a code fragment in subroutines that keeps the set of subroutines in execution and their invocation order. When the execution state of a method is restored, the subroutines that were in execution at the departure site are invoked explicitly by the inserted code fragment. The code in the subroutines that were already executed is skipped by dynamic checking. Perhaps, this scheme degrades execution performance because many code fragments are inserted that are always executed, but this scheme lowers code growth in comparison to unfolding and it does not need to extend the exception table.

Second, as is mentioned in Sect. 5.5, we do not transform system classes because they have tight connections with native code. This decision makes migration across callback methods impossible such as the `actionPerformed` and the `finalize`. The transformer preserves the signatures of these callback methods. In the callback methods, a null state object is passed to transformed methods.

Third, our transformation changes the signature of a method in order to pass a state object as an extra parameter. This induces some programs using reflection not to work correctly.

Fourth, our current implementation of transformation removes the line number table attribute of the target method, since debug information is no longer correct after transformation. This implies that it becomes unable to trace the execution of transformed methods with source-code debuggers. This limitation, however, can be easily eliminated by maintaining the line number table.

6 Experiments

This section shows some experiments with our current implementation of bytecode transformer for transparent thread migration. We measured the cost of the transformation process and evaluated the quality of transformed codes produced by our bytecode transformer from the viewpoints of execution efficiency and code size. The transformed codes are compared with (1) those produced by our source code transformer JavaGo [13] and (2) the original code without transformation.

The implementation is written in Java using only standard libraries in JDK. The size of the source code is around 5000 lines. All the benchmark programs were generated as standalone applications in advance by this transformer.

6.1 Elapsed Time for Bytecode Transformation

Table 1. Elapsed time for bytecode transformation.

program	# of methods	analysis(ms)	code insertion(ms)	total (ms)
fib	1	235	79	314
qsort	1	285	81	366
nqueen	1	267	80	347
_201_compress	23	3454	1349	4803

(JDK 1.2.2, Sun UltraSPARC 168MHz)

Table 1 shows the times consumed by our transformer to analyze and transform all methods of the sample programs, where _201_compress is a benchmark program included in SpecJVM98. These elapsed times are rather small, but our bytecode analyzer needs at least the same costs of bytecode verification.

6.2 Execution Efficiency of Transformed Programs

Table 2. Comparison of execution efficiency.

	elapsed time (ms)					
	with JIT			without JIT		
program	original	JavaGo	ours	original	JavaGo	ours
fib(30)	111	263 (+137%)	173 (+56%)	870	2553 (+193%)	1516 (+74%)
qsort(400000)	214	279 (+30%)	248 (+16%)	2072	2856 (+38%)	2597 (+25%)
nqueen(12)	1523	2348 (+54%)	1731 (+14%)	30473	36470 (+20%)	30843 (+1.2%)
_201_compress	33685	61629 (+83%)	40610 (+21%)	365661	713936 (+95%)	433439 (+19%)

(JDK 1.2.2, Intel Celeron(TM) Processor 500MHz)

The elapsed times of transformed programs were measured and compared with those of the original programs. The purpose of this measurement is to identify the overheads induced by inserted code fragments in the original programs. Thus, migration does *not* take place during the execution of the benchmark programs. The results are shown in Table 2. As a comparison, the elapsed times of the transformed programs by JavaGo, which is a source code level transformer, is also listed in the table.

Most part of the overheads is due to the code fragments for saving the operand stack at suspended points. The overheads of the Fibonacci method is rather high because the method does almost nothing but invokes the method itself recursively. When the contents of a method is so small, the relative overheads of inserted code fragments tend to be high. This tendency is common to migration schemes based on program transformation. Our results, nevertheless, are better than those of JavaGo. In this experiment, the overheads induced by our bytecode transformation are always less than those induced by JavaGo. For quick sort and N-queen programs, the overheads were approximately 15% to the original programs when the applications were executed with Just-In-Time compilation.

6.3 Growth in Bytecode Size of Transformed Programs

The growth in bytecode size due to our bytecode transformation is shown in Table 3. To show the pure growth in the method body size, these sizes do not

Table 3. Comparison of bytecode size.

	bytecode size (in bytes)		
program	original	JavaGo	Ours
fib	276	884 (3.2 times)	891 (3.2 times)
qsort	383	1177 (3.1 times)	1253 (3.3 times)
nqueen	393	1146 (2.9 times)	976 (2.5 times)
_201_compress	13895	22029 (1.6 times)	18171 (1.3 times)

include that of the state classes. The growth rates for these programs are approximately three times. We think that these results would be the worst case because the relative amount of inserted code fragments tend to be high when an original method is small.

The size of bytecode produced by our transformation scheme is very similar to the size of bytecode produced by JavaGo scheme. But their characteristics are quite different each other. In case of JavaGo, the size of transformed bytecode is proportional to square of the deepest depth of loops. In contrast, the size of bytecode transformed by our transformation scheme is proportional to the number of suspended points.

7 Related Work

Telescript [8] was an early interpreted programming language for mobile agents developed by General Magic Inc. The Telescript interpreter had a mechanism for transparent migration. Unfortunately, General Magic does not seem to develop Telescript anymore. Agent Tcl [7] was also a mobile language system developed in the early stage of mobile computation. Agent Tcl also has a mechanism for transparent migration. It is useful to run existing Tcl scripts on a mobile environment.

However, this emphasis on transparent migration was not inherited by major Java-based mobile language systems because of the restrictions imposed by the Java virtual machine release policy. Shudo avoided the restrictions by extending a Java virtual machine and implemented a transparent thread migration system [14]. Fünfrocken [6] pointed out that an exception handling mechanism could be used for notifying occurrence of migration with low costs. He developed a scheme of transparent migration for standard Java, but his scheme had difficulties in resumption of control in a compound statement. Sekiguchi et al. eliminated these difficulties based on the idea of unfolding and developed a transparent migration system [13] in Java. Their scheme enables transparent migration on any standard Java virtual machine with Java RMI. But it requires all Java source code that constitutes a mobile program. In Java, programs are always distributed in the form of bytecode, and source code may be unavailable.

Eddy Truyen et al. [19] developed independently a Java bytecode transformer for transparent migration, which shares a large part with ours. Both perform bytecode verification to determine all valid values in a frame and insert code fragments in the target bytecode based on the analysis. The major difference between their scheme and ours is execution efficiency. They use `return` and `if` instructions to roll back the stack during saving state, while we use the mechanism of exception. The former scheme significantly degrades the efficiency. Taga measured execution overheads of that scheme [18] (although it was performed in C++) and reports that the additional overheads are 27% – 137% of the original execution time compared to our scheme. In addition, their scheme uses a data area proper to a thread to pass state objects. Their scheme, therefore, can preserve the signature of transformed methods, but it also induces considerable performance loss. Truyen discusses execution efficiency and growth in bytecode size in a formal setting, while we measured with real applications.

8 Conclusion and Future Work

We have proposed a scheme for bytecode transformation that enables Java programs to save and restore their execution states including the call stack with low overheads. A bytecode transformer based on our scheme has actually been implemented. The transformer gives Java programs the ability of transparent migration. As is described in Sect. 6, the quality of the bytecode transformer is measured and we have obtained better results compared to existing schemes for transparent migration based on source code transformation. The latest implementation of the transformer described in this paper is widely available at `http://www.yl.is.s.u-tokyo.ac.jp/amo/`.

Further work is needed to eliminate the limitations mentioned in Sect. 5.6 due to bytecode subroutines. More programs can be transformed if these limitations are removed.

Acknowledgment

The authors would like to express our sincere thanks to Hidehiko Masuhara who contributed to much of the presentation of this paper.

References

1. Hirotake Abe, Yuuji Ichisugi, and Kazuhiko Kato. An Implementation Scheme of Mobile Threads with a Source Code Translation Technique in Java. In *Proceedings of Summer United Workshops on Parallel, Distributed and Cooperative Processing*, July 1999. (in Japanese).
2. Boris Bokowski and Jan Vitek. Confined Types. In *Intercontinental Workshop on Aliasing in Object-Oriented Systems in Association with ECOOP Conference*, 1999.
3. Luca Cardelli. Mobile Computation. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 3–6. Springer-Verlag, April 1997.

4. Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analyzing Mobile Code Languages. In *Mobile Object System: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 93–109, April 1996.
5. Bozhidar Dimitrov and Vernon Rego. Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. In *Proceedings of IEEE Parallel and Distributed Systems*, volume 9(5), pages 459–469, 1998.
6. Stefan Fünfrocken. Transparent Migration of Java-Based Mobile Agents. In *MA'98 Mobile Agents*, volume 1477 of *LNCS*, pages 26–37. Springer-Verlag, 1998.
7. Robert S. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, 1995.
8. White J.E. *Telescript Technology: Mobile Agents.* White Paper. General Magic, Inc, 1996.
9. Danny Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.
10. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition.* Addison-Wesley, 1999.
11. Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages*, pages 228–241, January 1999.
12. Voyager Core Package Technical Overview, 1997. ObjectSpace Inc.
13. Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. In *Coordination Languages and Models*, volume 1594 of *LNCS*, pages 211–226. Springer-Verlag, April 1999.
14. Kazuyuki Shudo. Thread Migration on Java Environment. Master's thesis, University of Waseda, 1997.
15. Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. SRC Research Report 158, Digital Systems Research Center, June 1998.
16. B. Steensgaard and E. Jul. Object and Native Code Thread Mobility among Heterogeneous Computers. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 68–78, 1995.
17. Volker Strumpen and Balkrishna Ramkumar. Portable Checkpointing for Heterogeneous Architectures. In *Fault-Tolerant Parallel and Distributed Systems*, chapter 4, pages 73–92. Kluwer Academic Press, 1998.
18. Nayuta Taga, Tatsurou Sekiguchi, and Akinori Yonezawa. An Extension of C++ that Supports Thread Migration with Little Loss of Normal Execution Efficiency. In *Proceedings of Summer United Workshops on Parallel, Distributed and Cooperative Processing*, July 1999. (in Japanese).
19. Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen and Pierre Verbaeten Portable Support for Transparent Thread Migration in Java. To appear in ASA/MA 2000.
20. Hiroshi Yamauchi, Hidehiko Masuhara, Daisuke Hoshina, Tatsurou Sekiguchi, and Akinori Yonezawa. Wrapping Class Libraries for Migration-Transparent Resource Access by Using Compile-Time Reflection. to appear in Proceedings of Workshop on Reflective Middleware, 2000. April.