

A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation

Tatsurou Sekiguchi¹, Hidehiko Masuhara², and Akinori Yonezawa¹

¹ Department of Information Science, Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113-0033

² Department of Graphics and Computer Science, Graduate School of Arts and Sciences, University of Tokyo
`{cocoa,masuhara,yonezawa}@is.s.u-tokyo.ac.jp`

Abstract. A scheme has been developed that enables a Java program to be migrated across computers while preserving its execution state, such as the values of local variables and the dynamic extents of try-and-catch blocks. This scheme provides the programmer with flexible control of migration, including transparent migration. It is based on source-code-level transformation. The translator takes as input code a Java program written in a Java language extended with language constructs for migration, and outputs pure Java source code that uses JavaRMI. The translated code can run on any Java interpreter and can be compiled by any just-in-time compiler. We have measured some execution performance for several application programs, and found that the translated programs are only about 20% slower than the original programs. Because migration is completely controlled by using only three language constructs added to the Java language (*go*, *undock* and *migratory*), the programmer can write programs to be migrated easily and succinctly. Our system is available in the public domain.

1 Introduction

Mobile agent systems are a promising infrastructure for distributed systems, in which communication is based on *migration*. Migration is called *transparent* [7] if a migrated program resumes its execution at a destination site with exactly the same execution state as that before migration began. Compared to non-transparent migration, transparent migration is more desirable [3, 5, 7] for writing programs to be migrated and understanding the semantics of migration. The migration transparency of existing mobile systems is, however, not satisfactory.

- Almost all existing mobile systems [9, 10] based on Java do not preserve the calling stack.
- Telescript [17, 18], an early mobile agent system, provides transparent migration, but the unit of migration is restricted to only one object (agent) [6] at a time.

- In the few systems [3, 5, 7, 13] that support transparent migration, the continuation (the complete execution state of the remaining computation) always migrates to the destination, which incurs considerable performance loss, as pointed out by Cejtin et al. [2]

We have extended the Java language by adding three constructs for controlling migration. They are based on our mobile calculus [11] and on mobile languages, such as that developed by Watanabe [16], and allow more flexible control of migration, including transparent migration. We call this flexible migration mechanism *controllable transparent migration*.

With Java, it is difficult to implement a transparent migration mechanism without degrading portability or efficiency for several reasons.

- Consider the case where the runtime system is extended in order to support migration. In this case, we can expect that the execution performance of a migrating program is almost the same as before, and that existing class files can be used for the extended system without recompilation. In return, however, migrating programs are not portable because they can only be executed on a specific interpreter or on the runtime system of a specific just-in-time compiler. This scheme thus does not fit the philosophy of Java, that is, “write once, run anywhere”.
- When a migration mechanism is provided as a class library, it offers nearly the same benefits as when a runtime system is extended. Unfortunately, there are several obstacles that are difficult to overcome in this approach. Transparent migration requires transmission of the stack. Dynamic inspection of the stack by Java bytecode itself is forbidden by the *Java security policy*.¹ Therefore, it is difficult to implement transparent migration as a class library.
- In an approach based on source-code-level translation (e.g., Arachne [3]), a Java program is transformed in such a way that the translated Java program can explicitly manage its execution states as Java objects, which enables transparent migration. The major drawback is a slowdown in execution speed due to the code fragments inserted to maintain the information for migration. In a straightforward implementation [3], execution of a transformed program was about twice as slow as that of the original program.

Our migration scheme is based on a source-code-level translation that offers the following features:

- portability** Programs translated using our scheme require only standard Java and JavaRMI, so a translated program can be run on any Java interpreter and can be compiled by any just-in-time compiler.
- efficiency** In our scheme, most of the execution state is saved at the time of migration by using an exception-handling mechanism. Therefore, the overhead for ordinary execution is low (about 20% in most cases).

¹ A native method can inspect the execution stack at runtime, but the memory layout of objects in the stack heavily depends on the implementation of the runtime system.

accuracy In migration schemes based on source-to-source transformation, it is difficult to simulate exception handling (try-and-catch and throw statements) appropriately. In our scheme, the dynamic extents of the try-and-catch blocks are completely preserved at the time of migration.

Testing of our scheme showed that it can migrate Java programs with a graphical user interface (like migratory applications [1]) between IBM-PC (Pentium / Windows-NT) and Sun workstation (SPARC / Solaris).

The rest of this paper is organized as follows. In Sect. 2, we describe the language constructs we use for migration and explain how the programmer's intention for migration can be described by using the constructs. In Sect. 3 we briefly overview our migration scheme. In the subsequent three sections, we describe our program transformation for transparent migration in detail. In Sect. 7 we discuss the difficulties induced by the source-code-level transformation, and in Sect. 8 we present experimental results showing the execution performance and growth in code size. In Sect. 9, we present photographs in which Java applications are migrated. In Sect. 10, we summarize our scheme and compare it with a related one. We also briefly discuss the need for extending JavaRMI.

2 Java Language Extension

In this section we give a simple example of migration and describe how the programmer specifies migration in source code. The programmer can describe flexible migration by using three language primitives (*go*, *undock* and *migratory*) added to the Java language.

2.1 Starting Migration by Using Go Primitive

Migration takes place by executing a “go” statement. Suppose a method of some object has the following lines of code:

```
System.out.println ("bye!");  
go ("//ritsuko:2001/JavaGoExecutor");  
System.out.println ("nice to meet you!");
```

The argument of a go statement is the name of a migration server registered in the JavaRMI registry. The migration server is an object that manages migration. When the go statement is executed, the execution state and object are transmitted to the destination host, where the object continues executing just after the go statement. In the above example, “bye!” is displayed on the departure host, and “nice to meet you!” is displayed on the destination host (ritsuko).

The migration mechanism in our system is transparent. (1) Even in cases where a go statement appears in a compound statement, such as a for statement, try statement, switch statement, or if statement, the migrated program resumes executing at the destination host. (2) If callee method *foo* including a go statement is invoked by caller method *bar*, control returns to method *bar* on the *destination* host after the method *foo* returns. This behavior is quite

different from that of remote evaluation [14], remote procedure call (RPC) and remote method invocation (RMI). For instance, a migratory Fibonacci method can be written in our language as follows:

```
boolean Moved = false;
public migratory int fib( int n ) {
    if ( n == 0 ) {
        if ( !Moved ) {
            go ( "//aki:2001/JavaGoExecutor" );
            Moved = true;
        }
        return 1;
    }
    else if ( n == 1 )
        return 1;
    else
        return fib (n-1) + fib (n-2);
}
```

Method `fib` computes a Fibonacci number by recursively invoking itself. During the computation, `fib` (and the object executing it) migrates only once (when n is equal to 0), that is, at the point where the execution stack is the deepest. Many systems (e.g., those described in Refs. [9] and [10]) that support mobile computation do not allow the transmission of recursive function invocations. The programmer must therefore write a method and auxiliary class definitions that represent the rest of the computation explicitly. (When we wrote a Fibonacci method in Voyager [10], we had to add extra 35 lines and a class definition representing the execution state.) The keyword `migratory` at the head of the method above is explained in Sect. 2.3.

2.2 Controlling Transparency by Using Undock Primitive

An undock statement serves as a *marker* that specifies the range of the area to be migrated in the execution stack. It makes it possible to control migration transparency. The brackets of the undock statement restrict the effect of a `go` statement to the enclosed statements. When the `go` statement in the undock statement is executed, the rest after the `go` statement in the undock statement resumes executing at the destination host, while the statements after the undock statement are concurrently executed on the departure host.

```
undock {
    go ( "//ritsuko:2001/JavaGoExecutor" );
    System.out.println ( "nice to meet you!" );
}
System.out.println ( "bye!" );
```

In the above lines of code, only the statements enclosed in the `undock` brackets are migrated. Therefore, "bye!" is displayed on the departure host, not on the destination host. An undock statement has a dynamic extent similar to a try-and-catch statement. When a `go` statement is executed in the nested extents

of several undock statements, the migrating part is the inside area of the most recently executed undock statement. An area to migrate can be specified by an undock statement in any method defined in any class.

```

o4.d() { ... go(...) ... }
o3.c() { ... o4.d() ... }
o2.b() { ... undock { ... o3.c() ... } ... }
o1.a() { ... o2.b() ... }

```

Fig. 1. Conceptual diagram of a stack.

Figure 1 depicts the migrating areas in a stack where methods *a*, *b*, *c*, and *d* are called in this order. The stack grows upwards according to a method invocation. The *o.m()* denotes the invocation of method *m* of object *o*. The portions underlined show the migrating areas. The stack frame of the *o1.a()* is not migrated because the execution of the *go* statement in method *d* is in the extent of the undock statement in method *b*. The stack frames corresponding to the migrating methods are transmitted to the destination host by using the JavaRMI mechanism. In addition to the stack frames, the receiver objects executing the migrating methods are also transmitted. Object *o2* executes on both the departure and destination hosts.

2.3 Declaring Method by Using Migratory Primitive

When migration takes place in the extent of a method, the method itself must be declared as such. This is done using the *migratory* primitive. That is, the *go* statement and the invocation of a migratory method must be written in a migratory method. If they are put in a non-migratory method, they must be enclosed within an undock statement.

3 Overview of Our Transparent Migration

To achieve transparent migration, a method being executed must be suspended, transmitted, and then resumed. In the next three sections we describe our migration scheme based on source-level transformation. In this section we give a brief overview.

The unit of our source-to-source translation is a *method*. A method is transformed in such a way that the translated method can explicitly manage its execution state as Java objects. The migratory modifier of a method is used to determine if the method is to be transformed for migration. Migratory declarations make source-level translation easier. The system proposed by Fünfroeken [5] is also based on source-to-source translation, but it requires the fixed-point iteration of a call-graph to determine the set of methods that should be translated because it lacks this kind of declaration.

3.1 Saving the Execution State

The execution state of each method consists of the execution point from which the method resumes and the values of all local variables. A method is transformed in such a way that (1) all the variable declarations are elevated and moved to the head of the method and (2) it captures `NotifyMigration` exceptions. The current execution point is continuously saved to a newly introduced special variable during ordinary execution of the method. When migration takes place, an exception is raised and captured by the method. This causes the method to store the values of all local variables into a state object. The method then propagates the exception to the caller. A state object is defined for each method, and all the state objects created during the state-saving process are connected as a chain. The details of this state-saving will be described in Sect. 5.

3.2 Transmitting the Execution State

Eventually, the exception is either captured by an undock statement or it reaches the bottom of the stack. Then, the chained state objects are serialized and transmitted to the destination host by using JavaRMI. The receiver objects executing the migrating methods are also transmitted.

3.3 Restoring the Execution State

The execution state is recovered from the chain of state objects. A method is transformed so that (1) it takes the corresponding state object as an extra parameter and (2) it can resume its execution from any statement in its body, as will be described in Sect. 4. When a method m was being suspended by calling another method m' at the time of migration that was caused by calling m' , a code fragment is inserted so that m' is called before resuming the execution of m . The value of each local variable is properly restored before executing the body of the method, as described in Sect. 5. The state of the execution stack (and the dynamic extents of the try-and-catch statements) is reconstructed at the destination host by calling the method at the bottom of the stack with the corresponding state object as the extra argument. This reconstruction is described in Sect. 6.

4 Resuming a Method

To enable a migrated method to resume execution, a mechanism that enables execution to jump to any program point is needed. In other words, a way to resume *control* is needed.

Migration in the C language by using source-code translation [3] exploits a simple jump mechanism: `goto` statements. Java, however, does not have `goto` statements. Its `break` and `continue` statements permit only escaping from a block statement, there is no way to jump into a compound statement. Our scheme

implements the jump facility with low overhead by using switch-case statements and the unfolding technique. This facility is based on transforming a method into a form in which the method can be resumed from any program point. Doing this requires two preprocessings.

4.1 Preprocessings

Splitting an Expression with Side-Effects. Resuming execution of a statement needs special care that contains side effects. Consider the following assignment statement (where `foo` is a migratory method):

```
y = a[x++] + foo(x) + b[x++];
```

If the method containing this statement is to be resumed immediately after method `foo` is invoked, we must know the values of `a[x++] + foo(x)` before migration and assigns the sum and `b[x++]` to variable `y` after migration. Although these intermediate values do not appear as local variables in the method, they must be captured and restored after migration. To handle these intermediate values, a statement including side effects is decomposed into a sequence of atomic operations. The above statement is decomposed as follows.

```
tmp3 = a; tmp4 = x++;
tmp5 = tmp3[tmp4];
tmp6 = foo(x);
tmp7 = b; tmp8 = x++;
tmp9 = tmp7[tmp8];
y = tmp5 + tmp6 + tmp9;
```

New variables (`tmp3`,...,`tmp9`) are generated to keep track of the intermediate values. This transformation guarantees that we can avoid resumption from within an expression. This does not seem to be the case in Fünfroeken's scheme [5].

Elevating Variable Declarations. Next, we identify all the local variables (including the intermediate variables introduced in the first preprocessing) in the method, elevate the variable declarations, and move them to the head of the method. Unlike the C language, Java allows variable declarations to be placed almost anywhere in a method. Because the statements in a method may be reordered or duplicated during the subsequent transformations, elevating the variable declarations avoids the difficulties of code reordering and duplication. If we duplicated a block of statements including a variable declaration, the resulting code fragment would contain two declarations for the same variable. Elevating avoids this. When a variable is declared with its initial value, we elevate only its declaration and leave the assignment of the initial value at the original program position.

4.2 Jump Facility

Our scheme enables execution to jump to any top-level statement in the method. A top-level statement is one that is not enclosed in an other statement, such as

a if statement or a for statement. Note that the goto facility of the C language can be simulated in Java, but only at the top-level. Consider the following Java code fragment.

```

TopLevel: for( ;; ) switch (EntryPoint) {
    case 0:
        ...
    case 1:
        ...
    case 2:
        ...
}

```

Each case statement can be considered a label used as a destination of a goto statement. An occurrence of `goto n`; is encoded as `{EntryPoint = n; continue TopLevel;}`. By following this approach, a method in which every statement appears at the top level can be transformed into a resumable form. Consider the following method.

```

void bar() {
    foo(); // migratory method invocation
    System.out.println("after foo");
}

```

To resume this method after invoking `foo`, we make the following transformation by adding a switch statement, assuming variable `EntryPoint` is set to 1.

```

void bar() {
    TopLevel: for( ;; ) switch (EntryPoint) {
        case 0:
            foo(); // migratory method invocation
        case 1:
            System.out.println("after foo");
            return;
    }
}

```

In our migration scheme, a method does not have to be resumable from every statement because a method is resumed only immediately after a migratory method is invoked. The reason for this will be explained in Sect. 6.

4.3 Unfolding Technique

To enable execution to be resumed in the middle of a compound statement, such as if statements and for statements, (1) the sub-statements after the resumption point and (2) the subsequent statements that should be executed until control reaches the top level are duplicated at the top level. By representing a compound statement as a while construct, we illustrate the necessary transformation below. Consider the following code fragment.


```

while(C1) {
    while(C2) {
        A;
        foo(); // migratory method invocation;
        B;
    }
}
return;

```

A and B are arbitrary non-migratory statements, and C1 and C2 are arbitrary non-migratory expressions. By unfolding the loops, we translate this code fragment into the following code fragment.

```

label1:
    while(C1) {
        while(C2) {
            A;
            foo(); // migratory method invocation;
            B;
        }
    }
    return;
label2:
    while(C2) {
        A;
        foo(); // migratory method invocation;
        B;
    }
    goto label1;
label3:
    B;
    goto label2;

```

(This unfolding duplicates the contents of the inner loop at the top level.) The resumption point is at `label3` when method `foo` is called at the time of migration. All the labels are at the top level so they can be implemented by top-level jumps. By jumping to `label3`, execution is apparently resumed immediately after `foo()`. As illustrated above, if we have top-level jumps, a method is resumable from any point by using unfolding.

The size of a transformed method increases in proportion to $O(n^2)$, where n is the maximum depth of the loops, because the body of a loop with a depth of n is unfolded n times.

Because labels must be at the top level, a try block cannot bridge several labels. If a loop appears in a try block, the block is duplicated so that it does not bridge labels. In our current implementation, to reduce the size of a transformed method, an optimization is performed such that the bodies of the catch blocks are shared among the duplicated try-and-catch statements.

5 Saving and Restoring Local Variable Values

Saving and restoring the values of local variables in a method play an important role in implementing migration. In this section, we first touch upon a related study in which migration is implemented by source-code translation and then describe our scheme.

5.1 The Arachne Scheme

In the Arachne scheme [3], migration in the C language is based on source-code translation. A native stack frame is not used to save the values of local variables; instead a stack is managed at the user-program level. A special object representing a stack frame is allocated for every function invocation. Each access to a local variable is replaced with the corresponding field access to the special object. A stack is transmitted to a remote site by transmitting these special objects. Unfortunately, applying the Arachne migration scheme to Java causes a considerable performance loss, as discussed in Sect. 8.

5.2 Our Scheme

Our scheme for saving the values of local variables is as follows. The body of each migratory method is enclosed by a try statement to capture a special exception, `NotifyMigration`, that is signaled by the occurrence of migration. When the method actually captures the exception, the values of all local variables are stored into a new state object, and the exception is raised again. This procedure repeats until the exception reaches the bottom of the stack or is captured by an undock statement.

For each method, a class for the state object is defined. The instance variables of the class record (1) the values of all the local variables, including `EntryPoint`, (2) a reference to the receiver object of the method (i.e., the value of `this`), and (3) the state object of the sub-method invoked by this method on resumption. Every state class inherits a common base class, so that state objects can constitute a chain.

When restoring the values of local variables, a state object containing the values is applied to the method as an extra argument. The values of the local variables are set to those stored in the object at the head of the method. For ordinary method invocation, the null value is passed to the extra argument.

Putting saving and restoring together, the transformation of a method looks like the following.

```

void foo() {
    int x = 0;
    ... the body of a method ...
}

⇒

void foo(State_X_foo State) {
    int x;
    if ( State == null )
        x = 0;
    else
        x = State.x;
    try {
        ... the body of a method ...
    } catch (NotifyMigration e) {
        State = new State_X_foo(this);
        State.x = x;
        e.Append(State);
        throw e;
    }
}

```

In the code on the righthand side, `State_X_foo` is the name of a state object. A state object is defined for each method and contains all the local variables of that method. Exception `e` maintains the chain of state objects. A state object is appended to the chain by executing `e.Append(State)`. The chain of state objects is transmitted to a destination site by using the JavaRMI mechanism. At the destination, the execution state is reconstructed based on the chain of state objects, and execution resumes there.

6 Reconstructing Stack of Method Invocations

A method is resumed by using a combination of the techniques described in the previous two sections. Resuming a *call stack* of method invocations, however, requires a different technique because the dynamic extents of try-and-catch blocks spanning method invocations must be reconstructed. If these dynamic extents are not preserved, the semantics of throwing an exception is violated. The call stack of method invocations is reconstructed by calling each method in the stack with its own state object as the extra argument in the order that the methods were invoked before migration. Suppose method m' was called by method m , and migration takes place. We need to insert a code fragment that executes the rest of m' at the resumption point in method m . When method m is called with its state object, m resumes m' and after returning from m' , the rest of m is executed. To implement such a behavior, method `bar` from Sect. 4.2 is translated as follows.

```

void bar(State_X_bar State) {
    TopLevel: for( ;; ) switch (EntryPoint) {
        case 0:
            EntryPoint = 1;
            foo (null); // migratory method invocation
        case 1:
            if ( State != null ) {
                foo ((State_X_foo)State.Child);
                State = null;
            }
    }
}

```

```

    }
    System.out.println("after foo");
    return;
  }
}

```

`State_X_bar` and `State_X_foo` are the names of the state classes defined for methods `bar` and `foo`, respectively. Suppose that a migration takes place when method `foo` is being called. On restart, a state object is passed to the corresponding method `bar` (the state object is referred to by variable `State` in the above code fragment). The state object shows that the value of `EntryPoint` has been set to 1. Therefore, control is transferred to case 1. Then, method `foo` is invoked with `State.Child`, where `State.Child` refers to the state object of the callee of this method, namely the state object of method `foo`.

7 Limitations due to Source-Level Translation

There are limitations in the current implementation of our migration scheme. Most of the limitations are essentially due to source-level translation. Therefore, similar limitations apply to other migration schemes based on source-level translation.

There are three areas in a program from which a `go` statement and a migratory method cannot be called, meaning that migration cannot take place.

- in a class initializer (a static initializer)
- in an instance initializer, and
- in a constructor

It is difficult to resume in a class or instance initializer because it is difficult for a user program to reconstruct the effects of executing these initializers. They are invoked by the runtime system when a class is loaded or an object is created. A constructor is also invoked by the runtime system, so it cannot be resumed. To allow invoking a migratory method from these areas, we must simulate the effects of these initializers and a constructor by combining other language constructs.

Another limitation is that locking cannot be preserved on migration. If a lock has been acquired by a synchronized statement or a synchronized method, it is released at the time of migration by the exception notifying migration. Although the locked state of an object is correctly recovered after migration, the state of the threads that were waiting on the migrating object cannot be preserved because the exception used in our migration scheme releases the object lock. The locked state is thus temporarily lost on migration.

8 Performance

To evaluate the overheads imposed by our translation scheme, we carried out benchmark testing by executing several Java application programs. The execution performance of programs transformed by our scheme was compared with

program	elapsed time(ms)			byte code size(bytes)		
	original	transformed	growth	original	transformed	growth
fib(25)	290	821	+183%	1274	3563	2.80
qsort(200000)	7691	9794	+27.3%	2765	5035	1.82
nqueen(11)	17816	20580	+15.5%	1647	2387	1.45
Richards	4777	4783	+0.125%	10868	25482	2.34
DeltaBlue	25881	28047	+8.37%	28257	50983	1.80

(JDK 1.1.7a, AMD-k6 200MHz)

Table 1. Execution performance and growth in byte code size of programs transformed by our scheme.

that of the original programs. Migration does *not* take place during the execution of the benchmark programs. The results are shown in Table 1. The overheads with the Fibonacci method is rather high because the body of Fibonacci is very small. Most of the overhead is due to extra control transfers induced by the unfolding. When the body of a method is very small, the overhead of code insertion is tend to be high. For the quick sort and N-queen applications, the elapsed times were approximately 20% higher than those of the original applications. Richards is a medium-scale benchmark program simulating the task dispatcher in an operating system. DeltaBlue [4] is a medium-scale constraint solver benchmark program (about 1000 lines in the case of Java). Java versions of Richards and DeltaBlue are available with the source code from Sun Microsystems Laboratories [15]. (Richards has seven variants. We used the one called “richards_gibbons”.)

The growth in code size due to program transformation is also shown in Table 1. The growth rate was less than twice except for the Fibonacci method and Richards, but even in these cases it was less than 3 times. In general, the code size grows in proportion to the square of the depth of the loops.

	elapsed time(ms)
original quicksort	7691
Arachne-style quicksort	16300
cost of field selection	5825
cost of state object allocation	2784

(JDK 1.1.7a, AMD-k6 200MHz)

Table 2. Performance of Arachne scheme.

For comparison, we applied the Arachne scheme to Java and measured the execution performance. As shown in Table 2, the direct adoption of the Arachne scheme to Java causes a considerable performance loss. The elapsed time for executing a method transformed by using the Arachne scheme was about twice

that of the original method. The overheads can be roughly divided into two parts. The first and major overhead is the cost of field selection. The second is the cost of allocating state objects at the head of the method.

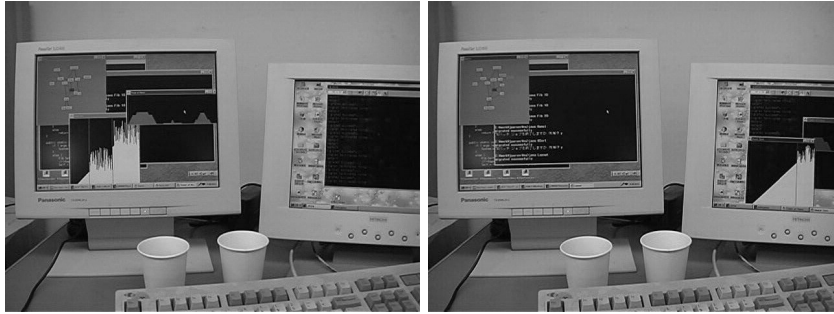


Fig. 2. Applications migrating from one computer to another.

9 Sample Mobile Applications

The two photographs in Fig. 2 show three applications migrating between two computers. The applications are the graph layout program included in JDK as a demo application, the tower of Hanoi, and quick sorting. The latter two applications are written using conventional recursive functions. We used two hosts, and the applications migrated from one to the other at different times. Each application had a window (an instance of `java.awt.Frame` class). Because a window is serializable, it can be transmitted to a remote host by using JavaRMI. When a visible window is transmitted to a remote host by using JavaRMI, the window still remains at the departure host and the moved window is not mapped on the display of the destination host automatically. Therefore, when an application migrates, the window remaining at the departure host must be explicitly deleted, and the window at the destination host must be explicitly mapped. This behavior is simply implemented by using `go` and `undock`. The code fragment that deletes the window at the departure host is as follows.

```
undock {  
    ... the body of an application ...  
}  
dispose();
```

When an application migrates, the control at the departure host exits the `undock` statement shown above. The code fragment that maps the window at the destination host is as follows.

```
go (destination);  
show();
```

If migration succeeds, `show()` is executed at the destination.

In the left picture of Fig. 2, all the applications are running on the host whose monitor is shown on the left. In the right picture, Hanoi and quick sorting have migrated to the right host.

10 Conclusion

We have developed language constructs for describing controllable transparent migration and its portable and efficient implementation based on Java source-code-level transformation. The translated code can run on any Java interpreter and be compiled by any just-in-time compiler. Nevertheless, it usually runs only about 20% slower than the original code. The transformation algorithm (1) translates the method into a form that makes it restartable from any of its statements, (2) inserts code for saving and restoring the values of local variables, and (3) inserts code for reconstructing the calling stack and the dynamic extents of the try-and-catch statements. The translator is written in Standard ML and is available in the public domain [12].

Transference of *several* threads at a time is not supported in our current migration scheme, but we believe that it can be added by simply extending the scheme.

Although our scheme and Fünfroeken's [5] were developed independently, both are based on Java source-code translation and use exception handling to transmit the execution stack. An important difference between them is in the way that method execution is resumed. In Fünfroeken's scheme, a so-called *artificial* program counter is used to skip the code fragments that have been already executed. An additional code fragment that checks whether the current statement should be skipped is inserted for each statement. (Successive statements can be grouped if they do not include migratory method invocations.) Our scheme does not need such checking, but in some cases, a compound statement needs to be unfolded. In saving a state, Fünfroeken's scheme inserts additional code that records the values of local variables into the state object for each migratory *method invocation*. In contrast, additional code is inserted for each *method* in our scheme because all variable declarations are moved to the head of the method. Fünfroeken's scheme performs fixed-point iteration of a call-graph to find the set of migratory methods. Our scheme does not need such an iteration because a migratory method is declared as such. The difference in performance cannot be directly compared because Fünfroeken did not report execution performance. He did report the growth in code size due to source-code transformation. The growth factor was about 3.65 to 4.7 times compared to the original programs. In our scheme, the growth factor is about 1.45 to 2.8 times.

From our experience in writing mobile applications with graphical user interfaces, we feel that the JavaRMI mechanism needs to be extended for transmitting graphical user interfaces. When a visible window is transmitted to a remote host by using JavaRMI, the window remains at the departure host and the moved window is not mapped on the display of the destination host automatically.

Therefore, when an application migrates, the window remaining at the departure host is explicitly deleted, and the window at the destination host is explicitly mapped. The applications we wrote had only one window; if applications with several windows were written, this code insertion would be cumbersome. Therefore, we need a mechanism that specified methods are automatically invoked before and after migration, analogous to that of the `:before` and `:after` methods in the common lisp object system (CLOS) [8]. Such a mechanism would be useful for any resource that needs initialization and finalization, not for only graphical user interfaces.

References

1. Krishna A. Bharat and Luca Cardelli. Migratory Applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, 1995.
2. Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-Order Distributed Objects. In *ACM Transactions on Programming Languages and Systems*, volume 17(5), pages 704–739, 1995.
3. 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.
4. Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An Incremental Constraint Solver. In *CACM*, volume 33(1), pages 54–63, 1990.
5. Stefan Fünfrocken. Transparent Migration of Java-Based Mobile Agents. In *MA'98 Mobile Agents*, 1477, *Lecture Notes in Computer Science*, pages 26–37, 1998.
6. General Magic Inc. *Telescript Programming Guide*. Version 1.0 alpha 2, 1996.
7. Robert S. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM Workshop on Intelligent Information Agents*, 1995.
8. Guy Steele Jr. *Common LISP: The Language*. Digital Press, 1984.
9. Danny B. Lange and Daniel T. Chang. IBM Aglets Workbench: A White Paper, 1996. IBM Corporation.
10. Voyager core package technical overview, 1997. ObjectSpace Inc.
11. Tatsurou Sekiguchi and Akinori Yonezawa. A Calculus with Code Mobility. In *Proceedings of Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, pages 21–36. Chapman & Hall, 1997.
12. Tatsurou Sekiguchi. JavaGo, 1998. <http://web.y1.is.s.u-tokyo.ac.jp/amo/>.
13. Kazuyuki Shudo. Thread Migration on Java Environment. Master's Thesis, University of Waseda, 1997.
14. James W. Stamos and David K. Gifford. Remote Evaluation. In *ACM Transactions on Programming Languages and Systems*, volume 12(4), pages 537–565, 1990.
15. Sun Microsystems Laboratories. Benchmarking Java with Richards and DeltaBlue. <http://www.sunlabs.com/people/mario/java.benchmarking/index.html>.
16. Takuo Watanabe. Mobile Code Description using Partial Continuations: Definition and Operational Semantics. In *Proceedings of WOOO*, 1997.
17. James E. White. Telescript Technology: An Introduction to the Language, 1995. General Magic white paper.
18. James E. White. Mobile Agents. In Jeffrey Bradshaw, editor, *Software Agents*. The MIT Press, 1996.