Efficient Java Thread Serialization

Sara Bouchenak
Swiss Federal Institute of Technology
IC / IIF / LABOS / INN 315
CH-1015 Lausanne, Switzerland
+41 (0)21 693 47 07
Sara.Bouchenak@epfl.ch

Daniel Hagimont
INRIA
655, av. de l'Europe, Montbonnot
38334 St-Ismier Cedex, France
+33 (0)4 76 61 52 62
Daniel.Hagimont@inria.fr

Noël De Palma
INPG
655, av. de l'Europe, Montbonnot
38334 St-Ismier Cedex, France
+33 (0)4 76 61 55 16
Noel.Depalma@imag.fr

ABSTRACT

The Java system supports the transmission of code via dynamic class loading, and the transmission or storage of data via object serialization. However, Java does not provide any mechanism for the transmission/storage of computation (i.e., *thread serialization*). Several projects have recently addressed the issue of Java thread serialization, e.g., Sumatra, Wasp, JavaGo, Brakes, Merpati. But none of them has been able to avoid the overhead incurred by thread serialization on thread performance. We propose a Java thread serialization mechanism that does not impose any performance overhead on serialized threads. In this paper, we describe our implementation of thread serialization in Sun Microsystems' JVM, and present the techniques that allowed us to cancel the performance overhead, namely *type inference* and *dynamic de-optimization*.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, polymorphism, control structures.*

D.3.4 [**Programming Languages**]: Processors – *Compilers, Interpreters, Run-time environments.*

D.4.1 [**Operating Systems**]: Process Management – *Threads.*

D.4.5 [**Operating Systems**]: Reliability – *Checkpoint/restart.*

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Software libraries.*

General Terms

Performance, Design, Experimentation, Languages.

Keywords

Mobility, persistence, checkpoint/restart, type inference, dynamic de-optimization, performance, threads, JVM.

1. INTRODUCTION

In JDK 1.0, the Java system supports the transmission of code via dynamic class loading. In JDK 1.1, Java allows data to be transmitted or stored thanks to object serialization. These mechanisms enable a computation to be started at new hosts, with an initial state, but always starting at the same point in the computation (i.e., the beginning of the computation). However, if we need to resume a computation at the point at which it was prior to transmission/storage, we need to transmit/store the state of the execution (i.e., thread) as well. Java thread serialization consists of capturing the current execution state of a Java thread for the purposes of transmission or storage, and thread de-serialization is the complementary process of restoring the execution state of a thread. Java thread serialization/de-serialization (hereafter referred to as "thread serialization") has many applications in the areas of persistence and mobility, such as checkpointing and recovery for fault tolerance purpose, mobile agent platforms, dynamic reconfiguration of distributed applications, administration of distributed systems, dynamic load balancing and user nomadism in mobile computing environments.

We designed and implemented a Java thread serialization mechanism that is used to build thread mobility or thread persistence. Therefore, a running Java thread can, at an arbitrary state of its execution, migrate to a remote machine where it resumes its execution, or be checkpointed on disk for possible subsequent recovery. With our services, migrating a thread is simply performed by the call of our *go* primitive, and checkpointing/recovering a thread is performed by the call of our *store* and *load* primitives[1].

In this paper, we propose a thread serialization mechanism that does not affect the "normal" performance of applications. Indeed, in some applications such as administration of distributed systems or dynamic reconfiguration of distributed systems, thread serialization is necessary but occurs rarely; it must therefore be overhead-free. The implementation of our zero-overhead Java thread serialization mechanism is mainly based on two techniques: (i) Type inference, and (ii) Dynamic de-optimization.

2. RELATED WORK

The main issue when building Java thread serialization is to access the thread's execution state, a state that is internal to the Java virtual machine (JVM) and not directly accessible to Java programmers. A possible solution is to extend the JVM with new

[1] Additional details about the API of our Java thread serialization mechanism an its use for mobility and persistence purposes are available from http://sardes.inrialpes.fr/research/JavaThread/

mechanisms that capture a thread state in a serialized and portable form, and later restore a thread from its serialized state, e.g., Sumatra **[1]**, Merpati **[10]**, ITS **[2]** and CIA **[6]**. Another solution is based on a pre-processor that transforms thread's application code prior to execution in order to add statements that follow thread's execution and manage its state capture, e.g., Wasp **[5]**, JavaGo **[9]**, Brakes **[11]** and JavaGoX **[8]**. These Java thread serialization mechanisms are characterized by four properties:

- The *completeness* of the accessed thread state.
- The *genericity* of thread serialization: its ability to adapt to different uses, e.g., mobility, persistence.
- The *portability* of the serialization mechanism across different Java platforms.
- The *efficiency* of the mechanism, i.e., its impact on the performance of thread execution.

Regarding the existing solutions, the thread serialization systems based on a JVM-level implementation verify the completeness requirement but lack in efficiency and portability. And the thread serialization systems proposed at the application level are portable but they are neither efficient nor complete. Furthermore, except Merpati and ITS, all the existing implementations propose Java thread serialization mechanisms that are restricted to thread mobility. Merpati allows Java threads to benefit from both mobility and persistence but it lacks in genericity because the proposed mobility/persistence services can not be adapted to applications' needs; while ITS proposes a generic implementation of Java thread serialization.

3. JVM CHARACTERISTICS

Three Java Virtual Machine's characteristics are mainly necessary to understand the rest of the paper: the bytecode, execution engines (equivalent of hardware processors) and runtime data areas.

Bytecode. The Java bytecode provides an instruction set that is very similar to the one of a hardware processor. Each instruction specifies the operation to be performed, the number of operands and the types of the operands manipulated by the instruction. For example, the *iadd*, *ladd*, *fadd* and *dadd* instructions respectively apply on two operands of type *int*, *long*, *float* and *double*, and return a result of the same type. The execution of bytecode in the JVM is based on a stack, called the *operand stack*. For example, before the invocation of the *iadd* instruction, two integer operands are pushed on the stack, and after the operation is completed, the integer result is left on top of the stack.

Execution engine. The first generation of JVM was based on an interpreter which translates each bytecode instruction into the execution of native code. In order to improve performance, the second generation of JVM has integrated Java Just-In-Time (JIT) compilers, which compile Java methods into native code. The subsequent JVM's execution engines perform much faster.

Runtime data areas. The JVM' data areas that describe the execution state of a Java thread are illustrated by Figure 1:

- *Java stack*. A Java stack is associated with each thread in the JVM. A new frame is pushed onto the stack each time a Java method is invoked and popped from the stack when the method returns. A frame includes a table with the local variables of the associated method, and an operand stack that contains the partial results (operands) of the method. A frame also contains registers such as the program counter (pc) and the top of the stack.

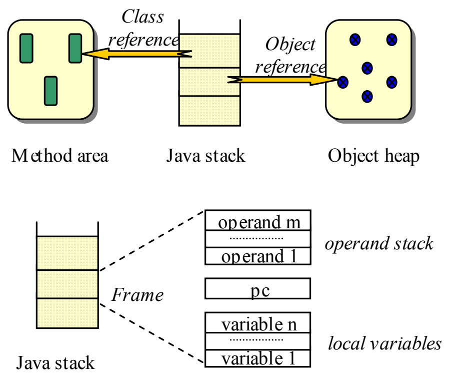


Figure 1. Java thread state

- *Object heap*. The heap of the JVM includes all the Java objects created during the lifetime of the JVM. The heap associated with a thread consists of all the objects used by the thread (objects accessible from the thread's Java stack).
- *Method area*. The method area of the JVM includes all the classes that have been loaded by the JVM. The method area associated with a thread contains the classes used by the thread (classes where some methods are referenced by the thread's stack).

In addition to the above data areas, a native stack is associated with a thread, in order to support native methods. A Java stack is used when the underlying execution engine is a Java interpreter; but when a Java method is JIT compiled, the invocation frame of this method is based on the native stack.

4. OVERALL DESIGN

Java thread serialization consists in interrupting the thread during its execution and extracting its current state. The extraction amounts to build a data structure (a Java object) containing all information necessary for restoring the Java stack, the heap and the method area associated with the thread. To build such a data structure, the Java stack associated with the thread is scanned in order to identify its current Java frames, the objects and classes that are referenced from the frames' local variables and operand stack, and the bytecode index for each frame (i.e., a portable value of the *pc*). After thread serialization, the resulting data structure can be transmitted to another virtual machine in order to implement thread mobility or it can be stored on disk for persistence purposes. Symmetrically, Java thread de-serialization consists first in creating a new thread and initializing its state with a previously captured state. After that, the Java stack (Java frames, local variables, operand stacks, pc), the heap and the method area associated with the new thread are identical to those associated with the thread whose state was previously captured. Finally, the new thread is started, it resumes the execution of the previous thread.

4.1 Objectives

Our first objective was to provide a *generic* Java thread serialization mechanism which allows the programmer to adapt the serialization policy in order to meet applications' needs. Therefore, various high level services can be built, such as thread mobility or thread persistence. Another objective was to provide a *complete* thread serialization mechanism that takes into account the complete state of a Java thread. On the other hand, one of the first criticisms addressed to Java was its poor performance; therefore, an important effort was made by Java/JVM designers in terms of execution optimization which led to today's efficient JVM. Consequently, for a new Java facility to be widely accepted, it must not degrade the performance of the applications which use it. Therefore, one of our main objectives has been to provide a thread serialization mechanism that *does not impose any overhead* on the execution of serialized threads. Finally, regarding the portability of the thread serialization mechanism, this property is, from our point of view, not the main issue. Our approach was to give ourselves the opportunity to propose a complete and efficient Java thread serialization system that would be widely used and could become a standard Java feature in future JVM implementations (as for RMI).

4.2 Main issues and design choices

Genericity. We propose a generic design of Java thread serialization thanks to which we are able to build several higher level services such as thread mobility and thread persistence. Indeed, the implementation of our thread mobility and thread persistence mechanisms is a combination of our Java thread serialization to standard Java mechanisms such as object serialization and dynamic class loading.

Completeness. The state of Java threads is not entirely accessible by Java programs. For facing this problem, we extended the JVM in order to be able, on the one hand, to externalize the state of Java threads (for thread serialization), and on the other hand, to initialize a thread with a particular state (for thread de-serialization).

Portability of thread state. Unlike the heap and the method area that consist of information portable on heterogeneous architectures (thanks to Java object serialization and bytecode definition), the Java stack is implemented in most JVMs a as native data structure (C structure). The representation of the information contained in the Java stack depends on the underlying architecture. The thread serialization mechanism must translate this non portable data structure (C structure) to a portable data structure (Java object), and thread de-serialization must perform the symmetric process. Translating the Java stack into a portable data structure consists more precisely in translating the native values of local variables and partial results into Java values. This translation requires the knowledge of the types of the values. But the Java stack does not provide any information about the types of the values it contains: a four bytes word may represent a Java reference as well as an *int* value or a *float* value. Therefore, the main issue here is to infer the types of the data stored in the Java stack.

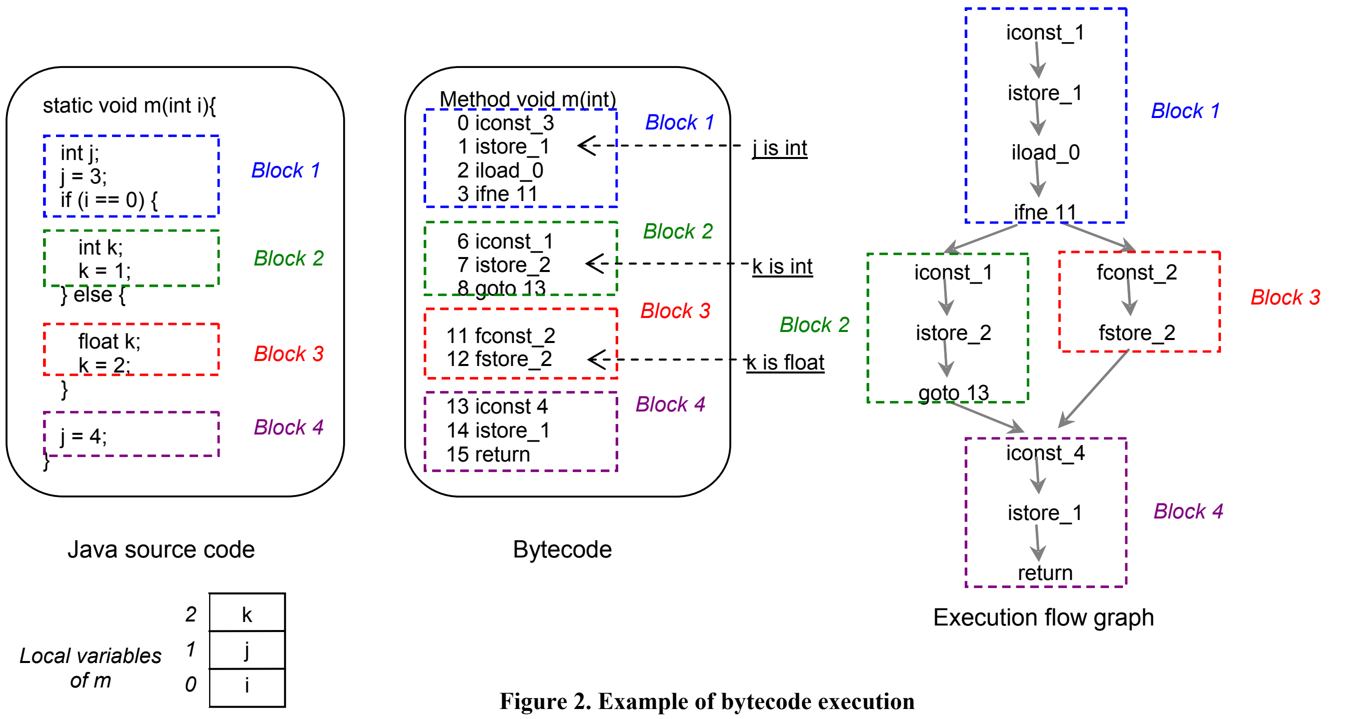
The only place where these types are known is the bytecode of the methods that push the data on the stack. As explained in section 3, a bytecode instruction which pushes a value on a Java stack is typed and determines the type of this value. The simplest solution is thus to modify the Java interpreter in such a way that each time a bytecode instruction pushes a value on the stack, the type of this value is determined and stored "somewhere" (i.e., a type stack associated with the thread). But the drawback of this solution is that it introduces an important performance overhead on thread execution, since additional computation has to be performed in parallel with bytecode interpretation. In order to avoid any overhead, type inference must not be performed during thread execution but only at thread serialization time. We propose a solution in which the bytecode executed by the thread is analyzed with one pass, at thread serialization time. With this analysis, the type of the stacked data is retrieved and used to build the portable data structure that represents the thread's Java stack. Thus, the Java interpreter is kept unchanged and no performance overhead is incurred on the serialized thread. This approach is called CTS (Capture time-based Thread Serialization); it is detailed in section 5.

Efficiency. In order to design Java thread serialization in such a way that it avoids any performance overhead, we followed two principles: (i) No additional computation is performed in parallel with bytecode interpretation, and (ii) Thread serialization is compatible with today's Java JIT compilation techniques. First, that means that everything is done at serialization time: by using a type inference technique applied at thread serialization time, as described in section 5. Moreover, regarding JIT-compatibility, the problem is to be able to perform thread serialization even if the thread's Java stack does not really reflect the current execution state of the thread. This is the case when some Java methods currently executed by the thread are JIT compiled (i.e., their execution is based on the threads' native stack and not on the Java stack). In order to face this problem, we propose to use a dynamic de-optimization technique as described in section 6.

5. TYPE INFERENCE

The type inference mechanism aims at building a *type stack* that reflects the types of the values (local variables and operands) contained in the thread's Java stack. Like the Java stack, the type stack consists of a succession of frames which we call *type frames*. A *type frame* on a type stack is associated with each Java frame on the Java stack. A type frame contains two main data structures: a table that describes the types of the local variables of the associated method and an operand type stack that gives the types of the partial results of the method. A thread's type stack is built as follows. At serialization time, for each frame on the thread's Java stack, the bytecode of the associated method is parsed from the beginning to the exit point of the method (pointed to by the Java frame's *pc* and representing the last instruction executed in the method). Following this code path, the parsed bytecode instructions are analyzed and the types of the values they manipulate are inferred and stored in the type frame, either as local variable types or as operand types.

The main problem when inferring the types occurs when several paths exist between the beginning of the method's code and the method's exit point; especially when different code paths may assume different types for a same item on the Java stack (local variable or operand). In this case, which path should be followed for type inference? Let us illustrate this problem through an example of a Java program represented by a Java source code, its



```
static void m(int i){
  int j;
  j = 3;
  if (i == 0) {
    int k;
    k = 1;
  } else {
    float k;
    k = 2;
  }
  j = 4;
}
```

```
Method void m(int)
 0 iconst_3
 1 istore_1
 2 iload_0
 3 ifne 11
 6 iconst_1
 7 istore_2
 8 goto 13
11 fconst_2
12 fstore_2
13 iconst 4
14 istore_1
15 return
```

Figure 2. Example of bytecode execution

equivalent bytecode and the associated execution flow graph (see Figure 2). In this program, the local variables *i* and *j* are declared in *block 1* and represent values of type *int*, and the local variable *k* represents a value of type *int* in *block 2* and of type *float* in *block 3*. This variable is implemented by the same entry in the local variable table of the Java frame (a variable at index 2, manipulated at lines 7 and 12 in the bytecode). How are the types of the local variables of the method *m* determined? When serializing the thread executing the method *m*, four cases are possible:

1. The exit point (*pc* value) is in *block 1*. Thus, there is only one path from the beginning of the code to the exit point. The analysis of this path permits to determine that the local variable *i* is an *int* value thanks to the method signature, and the local variable *j* is an *int* value thanks to the instruction *istore_1* at line 1 in the bytecode.
2. If the exit point is in *block 2*, the only one path reaching that point is *block 1-block 2*. When analyzing this path, the variables *i* and *j* are recognized as being *int* values (as in the 1^{st} case) and the *int* type of the variable *k* is determined thanks to the instruction *istore_2* at line 7 of the bytecode.
3. In case the exit point is in *block 3*, there is only one path reaching that point: *block 1-block 3*. This case is similar to the second one; the only one difference is that path analysis recognizes the variable *k* as being a *float* value thanks to the instruction *fstore_2* at line 12 of the bytecode.
4. Finally, if the exit point is in *block 4*, then two paths exist: either *block 1-block 2-block 4* or *block 1-block 3-block 4*. In this case, which code path should be followed for type inference?

Our solution to this problem is based on two correctness properties of the Java bytecode [4]:

Correctness properties:
At any given point in the program, no matter what code path is taken to reach that point:
P1: The operand stacks built by following each code path contain the same types.
P2: The local variables built by following each code path are of the same types or are unused if the types differ.

As a consequence of the P2 correctness property, following both paths *block 1-block 2-block 4* or *block 1-block 3-block 4*, variable *k* is no more used and its type is undefined. And according to the P1 correctness property, an operand built following two different code paths has the same type. Thus, any of the possible code paths can be used for type inference. Thus, our algorithm determines the types of the values on a thread's Java stack in one pass of the bytecode. This algorithm amounts to: (i) determining, for the code of each method currently executed by the thread, any code path starting from the beginning of the method's code and reaching the method's exit point (*pc* value), and (ii) inferring the types of the manipulated values from the bytecode instructions contained in this path. Finally, the type inference algorithm builds a type stack that reflects the types of the values on the thread's Java stack. The resulting type information is then used to capture the thread's Java stack in a portable form.

6. DYNAMIC DE-OPTIMIZATION

The type inference technique described in the previous section requires access to the thread's Java stack. But the Java stack may sometimes not reflect the current execution state of the thread. Indeed, with Java JIT compilation, the execution of JIT compiled methods is no longer based on the thread's Java stack but on the native stack. Thus, the issue is to permit thread serialization even in the presence of JIT compilation. That was one of our main objectives: not to trade Java applications performance for the implementation of thread serialization.

Sun Microsystems' HotSpot virtual machine includes a mechanism which performs dynamic de-optimization. This mechanism transforms the native frames associated with JIT compiled methods into Java frames [7]. Dynamic de-optimization was first used in the Self's source-level debugging system; it shields the debugger from optimizations performed by the compiler by dynamically de-optimizing code on demand. This allows the programmer to debug his program at the source code-level even in presence of compilation optimizations. In the HotSpot VM, dynamic de-optimization was introduced in order to deal with the inconsistency problem rising from the combination of method inlining performed by JIT compilation and dynamic class loading. Here, we use dynamic de-optimization in a thread serialization system. At serialization time, we invoke dynamic de-optimization on the thread's JIT compiled frames in order to retrieve the Java frames which would have been produced by the Java interpreter. Therefore, the type inference algorithm described in section 5 can be applied to these Java frames, and the thread can be serialized. It is important to notice here that if dynamic de-optimization is used at thread serialization time, re-optimization must be used at thread de-serialization time in order not to trade thread performance. Finally, Java applications that use our thread serialization mechanism continue to benefit from JIT compilation, before and after serialization, i.e., they execute exactly in the same conditions as on an unmodified JVM.

7. CONCLUSION

Java provides most of the functions required to transmit the code (i.e., dynamic class loading), and to transmit or store data (i.e., object serialization). However, Java does not provide any mechanism for the transmission/storage of the computation (i.e., threads). We propose a generic thread serialization mechanism that we used as a basis for the implementation of thread mobility and thread persistence services. With these services, a running Java thread can, at an arbitrary state of its execution, migrate to a remote machine where resume its execution, or be checkpointed on disk and then recovered.

We implemented the CTS (Capture-time Thread Serialization) thread serialization system within Sun Microsystems' Java Virtual Machine. The lessons learned from this experiment are:

- It is possible to extend the Java Virtual Machine with thread serialization, mobility and persistence facilities without redesigning the whole JVM.
- The proposed thread serialization/mobility/persistence mechanisms do not incur any performance overhead on threads. This was possible thanks to the use of two techniques:
    - A type inference technique which permits to build a thread serialization mechanism that is totally separated from the JVM interpreter and does therefore not impact bytecode interpretation.
    - A dynamic de-optimization technique which allows thread serialization to be compliant with Java JIT compilation.

In this paper, we described our work towards the provision of basic mechanisms for an overhead-free Java thread serialization/mobility/persistence system. We restricted our discussion to the design and implementation issues in a local environment (i.e., a local JVM), and we did not discuss the problems rising from using our serialization facility to build large distributed systems, e.g., object sharing, synchronization, etc. Some elements of response are presented in [3], where the authors describe how they use our Java thread serialization mechanism for fault tolerance purpose, and how they built a checkpoint/restart facility for parallel computations in the Suma metacomputing system. Further experiments have to be conducted in order to evaluate the use of our thread serialization system to build large mobile distributed applications.

8. REFERENCES

[1] Acharya, A., Ranganathan, M., and Salz, J. Sumatra: A Language for Resource-aware Mobile Programs. 2nd International Workshop on Mobile Object Systems (MOS'96), Linz, Austria, Jul. 1996.

[2] Bouchenak, S. Mobility and Persistence of Applications in the Java Environment. Ph. D. Thesis, French National Polytechnic Institute of Grenoble (INPG), France, Oct. 2001.

[3] Cardinale, Y., Hernández, E. Checkpointing Facility on a Metasystem. European Conference on Parallel Computing (Euro-Par'2001), Manchester, UK, Jan. 2001.

[4] Engel, J. Programming for the Java Virtual Machine. Addison Wesley, 1999.

[5] Fünfrocken, S. Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs). 2nd International Workshop Mobile Agents 98 (MA'98), Stuttgart, Germany, Sep. 1998.

[6] Illmann, T., Krueger, T., Kargl F., Weber, M. Transparent Migration of Mobile Agents Using the Java Debugger Architecture. 5th IEEE Int. Conference on Mobile Agents (MA'2001), Atlanta, GA, USA, Dec. 2001.

[7] Meloan, S. The Java HotSpot Performance Engine: An In-Depth Look. Sun Microsystems, Jun. 1999.

[8] Sakamoto, T., Sekiguchi, T., and Yonezawa, A. Bytecode Transformation for Portable Thread Migration in Java. 4th Int. Symposium on Mobile Agents 2000 (MA'2000), Zürich, Switzerland, Sep. 2000.

[9] Sekiguchi, T., Masuhara, H., Yonezawa, A. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. 3rd Int. Conference on Coordination Models and Languages, Amsterdam, The Netherlands, Apr. 1999.

[10] Suezawa, T. Persistent Execution State of a Java Virtual Machine. ACM Java Grande 2000 Conference, San Francisco, CA, USA, Jun. 2000.

[11] Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., and Verbaeten, P. Portable Support for Transparent Thread Migration in Java. 4th International Symposium on Mobile Agents 2000 (MA'2000), Zürich, Switzerland, Sep. 2000.