Techniques for Implementing Efficient Java Thread Serialization

Sara Bouchenak
Swiss Federal Inst. of Technology
IC / IIF / LABOS / INN 315
CH-1015 Lausanne, Switzerland
Sara.Bouchenak@epfl.ch

Daniel Hagimont
INRIA
655, av. de l'Europe, Montbonnot
38334 St-Ismier Cedex, France
Daniel.Hagimont@inria.fr

Noël De Palma
INPG
655, av. de l'Europe, Montbonnot
38334 St-Ismier Cedex, France
Noel.Depalma@inria.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.

Keywords

software, mobility, persistence, type inference, dynamic de-optimization, threads, Java.

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 purposes [16], mobile agent platforms [27], dynamic reconfiguration of distributed applications [14], administration of distributed systems, dynamic load balancing [25] and user nomadism in mobile computing environments [3].

Several projects have recently addressed the issue of Java thread serialization, in the form of thread mobility or thread persistence. But none of them has been able to completely avoid the overhead incurred by the serialization mechanisms on thread performance. Such an overhead has several reasons:

- Additional instructions inserted in the application code; this is the case for the Wasp [12], Brakes [32] and JavaGo [28] thread mobility systems.
- Extension of the Java interpreter, as in the Sumatra thread mobility system [1], and the Merpati [30] and the ITS [6] thread mobility and persistence systems.
- Non-compliance with Java JIT compilation (execution optimization), e.g., CIA [18], Sumatra, Merpati and ITS.

All the above-mentioned systems impose a performance overhead. 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.

1.1 Contributions

The scientific contributions of this paper are:

1. The design of an extended Java virtual machine that supports Java thread serialization with the following properties:
   (a) The Java language syntax is not modified.
   (b) The Java compiler is not modified.
   (c) The existing Java API is not affected.
   (d) A new Java API is proposed for a generic thread serialization mechanism.
   (e) A high-level Java API for thread mobility and thread persistence is provided on top of thread serialization.
2. The implementation details of a zero-overhead Java thread serialization mechanism. This implementation is mainly based on two techniques:
   (a) Type inference.
   (b) Dynamic de-optimization.

Our prototype is freely available from : http://sardes.inrialpes.fr/research/JavaThread/

It has been successfully used in the Suma metacomputing platform where it was used as a basic service for the implementation of global uncoordinated checkpointing/ recovery for parallel computations [7]. In addition to Suma's

designers, there were about 200 downloads from users, testers, students and researchers working with our thread serialization service. Furthermore, we propose a performance evaluation comparing several serialization techniques [5]; but this evaluation was unfortunately omitted here in order to fit the paper within the space limitations.

1.2 Roadmap

The rest of the paper is structured as follows. Section 2discusses the related work and section 3 presents the Java Virtual Machine's characteristics that are necessary to understand the rest of the paper. Section 4 describes the overall design to support Java thread serialization. Following this, sections 5 and 6 respectively focus on the implementation and use of the type inference and dynamic deoptimization techniques. Finally, section 7 gives the implementation status of our Java thread serialization system and section 8 is devoted to our conclusions.

2 RELATED WORK

Many systems have been developed providing mobility or persistence of processes/threads, considering either homogeneous or heterogeneous processor architectures, e.g., Charlotte [2], Sprite [10], Emerald [20], Ara [26]. There are a number of surveys discussing these features [24][9]. In this paper, we focus our attention on providing such mechanisms in the Java environment. Our objective was to answer the following questions:

- Is it possible to provide thread serialization/mobility/ persistence in Java?
- At which conditions regarding performance?

In the following, we first place our research in the context of complementary works in the area of middleware systems, and then focus on related work in the area of Java thread serialization.

2.1 Context of our research

Our work focuses on the design and implementation of a Java thread serialization mechanism on top of which thread mobility and persistence are built.

What the mechanism does. As Java object serialization, thread serialization allows a thread execution state to be saved in a data structure, copied on a disk to implement persistence or transmitted to a remote machine for mobility purpose.

What the mechanism does not do. As object serialization, thread serialization does not deal with distribution, object sharing between threads, synchronization, nor the management of IO objects (sockets or files). Thread serialization is intended to be a basic mechanism used for the implementation of a middleware environment which addresses the above problems. The middleware may implement a distributed object space or a higher level distributed synchronization service, and thus ensure that the deserialization of a thread is consistent with the implemented distributed object management or synchronization service.

Figure 1 illustrates how thread serialization takes place in such a middleware. Let us consider three examples:

- A mobile agent system. In such a middleware, agents are generally well encapsulated Java object containers that migrate using object serialization. Thread serialization could therefore be used to transform agents' weak mobility (i.e., data mobility) into strong mobility (i.e. computation/thread mobility). In the Aglets system [17], interactions between agents are based on message exchanges, and Java object sharing management is thus avoided. A detailed work on the isolation of Java applications is presented in [8].
- A shared object system, e.g., Ajents [19] and Javanaise [13]. The latter is a Java distributed replicated object system, where synchronization of replicas is based on the entry consistency protocol [4]. Such a system could be combined with thread serialization in order to build a complete distributed thread migration service that benefits from replication and synchronization.
- A distributed system responsible for managing IO objects. Accent/Mach [33] and Condor [22] are examples of systems that respectively provide transparent access to communication channels and files (i.e., location/distribution are hidden). The same functionalities could be implemented by a Java based middleware, where thread mobility and persistence would benefit from transparent access to IO objects.

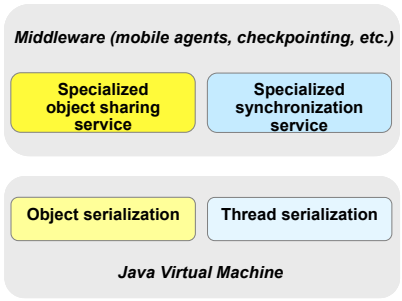

Figure 1. Thread serialization: a basic component in a middleware environment

2.2 Related work

In this section, we focus on the research conducted in the area of Java thread serialization, where some projects propose Java thread mobility systems, e.g., Sumatra [1], Wasp [12], JavaGo [29], Brakes [32], JavaGoX [28], CIA [18], and others propose both Java thread mobility and Java thread persistence systems, e.g., ITS [6] and Merpati [30].

The main issue when building Java thread serialization is to be able 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. In order to address this issue, two main approaches are followed:

- JVM-level approach.

- Application-level approach.

The most intuitive approach to access the state of a Java thread is to add new functions to the Java environment in order to export the thread state from the JVM. In the Sumatra [1], Merpati [30], ITS [6] and CIA [18] projects, the JVM is extended with new mechanisms that capture a thread state in a serialized and portable form, and later restore a thread from its serialized state. This solution grants full access to the entire state of a Java thread. But its main drawback is that it depends on a particular extension of the JVM; the provided thread serialization mechanism can therefore not be used on existing virtual machines.

In order to address the issue of non-portability of the thread serialization mechanism on multiple Java environments, some projects propose a solution at the application level, without relying on an extension of the JVM. In this approach, the application code is transformed by a pre-processor, prior to execution, in order to attach a backup object to the Java program executed by the thread, and to add new statements in this code. The added statements manage the thread state capture and restoration operations and store the state information in the backup object. The backup object can therefore be serialized using object serialization. Several Java thread migration systems follow this approach: Wasp [12] and JavaGo [29] provide a Java source code pre-processor while Brakes [32] and JavaGoX [28] rely on a bytecode pre-processor. The key advantage of application-level implementations is the portability of the provided mechanisms to all Java environments. However, they are not able to access the entire execution state of a Java thread, because some part of the state is internal to the JVM [6]. The resulting systems are therefore incomplete.

On the other hand, whatever the level of implementation (JVM or application), all the existing solutions impose an important performance overhead on the threads. Indeed, the JVM-level systems suffer from inducing a significant overhead on thread performance (+335%, +340%, c.f., [5]) because some of them extend the Java interpretation process (e.g., Sumatra, Merpati, ITS) and none of them supports Java execution optimization (JIT compilation). And the application-level systems impose a non negligible performance overhead (+88%, +250%, c.f., [5]) due to the statements added to the original code of the thread.

To summarize, Java thread serialization mechanisms are characterized by four properties:

- The *genericity* of thread serialization, i.e., the ability to adapt it to different uses such as mobility, persistence,
- the *completeness* of the accessed thread state,
- the *portability* of the serialization mechanism across different Java environments,
- and 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

This section describes the Java Virtual Machine's characteristics that are necessary to understand the rest of the paper: a defined set of instructions (bytecode), an execution engine (an equivalent of a hardware processor) and runtime data areas.

3.1 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.

3.2 Execution engine

The first generation of JVM was based on an interpreted scheme in which the Java interpreter 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 [31]. The subsequent JVM's execution engines perform much faster.

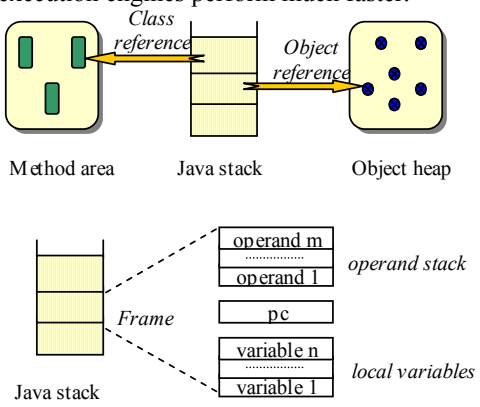

Figure 2. Java thread state

3.3 Runtime data areas

The JVM specification defines several runtime data areas [21]. Here, we focus on the data areas describing the execution state of a Java thread, as illustrated by Figure 2:

- *The Java stack*. A Java stack is associated with each thread in the JVM; it consists of a succession of frames. 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 containing the local variables of the associated method and an operand stack that contains the partial results (operands) of the method. The values of local variables and operands may be of several types: integer, float, Java reference, etc. A frame also contains registers such as the program counter (pc) and the top of the stack.
- *The 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).
- *The method area*. The method area of the JVM includes all 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, and in order to support native methods, the JVM specification mentions a native stack associated with a thread [21]. The structure of the native stack is not specified, it depends on the underlying operating system. Notice that the Java stack is managed for the execution of bytecode by a thread, i.e., when the underlying execution engine is a Java interpreter. But when a Java method is JIT compiled, the invocation frame of this method is not managed on the Java stack anymore but 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 purpose.

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.

To illustrate the use of our thread serialization and de-serialization mechanisms, an example is shown in Figure 3. Our *java.lang.threadpack* Java package contains several classes[1] such as the *ThreadState* class whose instances represent the state of Java threads and the *ThreadStateManagement* class that provides the necessary features for capturing and restoring Java threads state.

```
// Serialization of the current thread
ThreadState ts = ThreadStateManagement.capture();

// De-serialization of a thread
Thread new_thread = ThreadStateManagement.restore(ts);
```

Figure 3. Thread serialization and de-serialization example

4.1 Objectives

As discussed in section 2.2, the existing Java thread serialization mechanisms either propose a complete solution or provide a portable system but none of them focuses on the performance and genericity issues.

One of our objectives was to provide a generic Java thread serialization mechanism which allows the programmer to adapt the serialization policy in order to meet applications' needs. Indeed, with a generic thread serialization mechanism, 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

4.2.1 Generic thread serialization

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.

Figure 4 and Figure 5 respectively present a part of the API[1] of our thread mobility and thread persistence mechanisms. The implementation of these mechanisms relies on:

- The combination of our thread serialization mechanism with Java object serialization in order to transmit or store the resulting *ThreadState* object.
- The use of object de-serialization and dynamic class loading in order to receive or recover the *ThreadState* object (objects and classes) and then use it for thread de-serialization.

Method Summary	
static void	go(String targetHost, int targetPort) Transfers the execution of the current thread to the machine specified by the host name and the port number arguments.
static Thread	arrive(String targetHost, int targetPort) Receives a thread on the machine specified by the host name and the port number arguments.

Figure 4. Java thread mobility service

Method Summary	
static void	store(String fileName) Saves the state of the current thread in the file specified by the name argument.
static Thread	load(String fileName) Restores the execution of a Java thread from the state stored in the file specified by the name argument.

Figure 5. Java thread persistence service

Because object serialization and dynamic class loading are generic facilities, they can be specialized in order to build various thread transmission and storage policies for distributed system. Object serialization can, for example, be specialized in order to specify a particular management of IO objects that are part of the thread's heap, such as closing *Socket* objects at serialization time and recreating new *Socket* objects and connecting them at de-serialization time. And dynamic class loading can be specialized in order to use a particular URL for fetching thread's classes.
Further details on the design of our thread mobility and persistence services can be found in [5]. In this paper, we discuss our experience in implementing Java thread serialization.

4.2.2 Complete thread state

The complete state of Java threads is internal to the JVM. This state is not 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).

[1] The complete API is available from http://sardes.inrialpes.fr/research/JavaThread/

4.2.3 Non-portable 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 3.1 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., type stack associated with the thread). Our first prototype of Java thread serialization followed this approach, it is called ITS (Interpreter-based Thread Serialization) [6]. 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.

4.2.4 Overhead-free thread serialization

In order to design Java thread serialization in such a way that it avoids any performance overhead, we followed two principles:

- No additional computation is performed in parallel with bytecode interpretation: everything is done at serialization time. This is achieved by using a type inference technique applied at thread serialization time as detailed in section 5.
- Compatibility of thread serialization with today's Java JIT compilation techniques. The problem here 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 deoptimization technique as described in section 6.

4.2.5 Miscellaneous

Complementary questions regarding the issues and design choices of thread serialization may be asked at this point:

- Is thread serialization initiated by the serialized thread itself (auto-serialization) or can it be initiated by another thread (preemptive serialization)?
- How is the execution context associated with native methods (frames on the native stack) managed when a thread serialization occurs?
- Does the introduction of a thread serialization mechanism violate Java security?

In this paper, we focus on the design and implementation details of a complete and efficient mechanism of auto-serialization without native methods. Further details on how the above issues are tackled can be found in [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* (see Figure 6). 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.

The type stack of a thread 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. In this case, which path should be followed for type inference? Different code paths may assume different types for a same item on the Java stack (local variable or operand). Let us illustrate this problem through an example of a Java program represented by a Java source code, its equivalent bytecode and the associated execution flow graph (see Figure 7). 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).

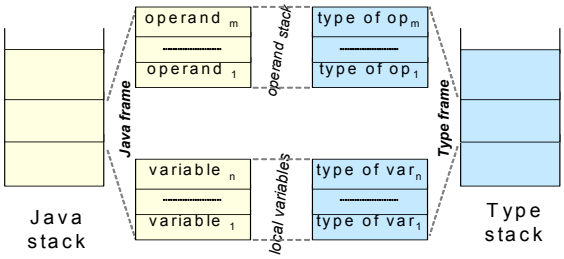

Figure 6. Type stack vs. Java stack

Let us first focus our attention on the determination of the types of the local variables of the method m. When serializing the thread executing the method m, four cases are possible:

1. The exit point (*pc* value) is in *block 1*. In this case, 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].
2. If the exit point is in *block 2*, then the only one path reaching that point is *block 1-block 2*. When analyzing this path, the local variables *i* and *j* are recognized as being *int* values (as in the first case) and the *int* type of the local variable *k* is determined thanks to the instruction *istore_2* at line 7 of the bytecode[2].
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[2].
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?

[2] In the case the exit point is after the *store* instruction, otherwise the variable is not yet used and determining its type is not necessary.

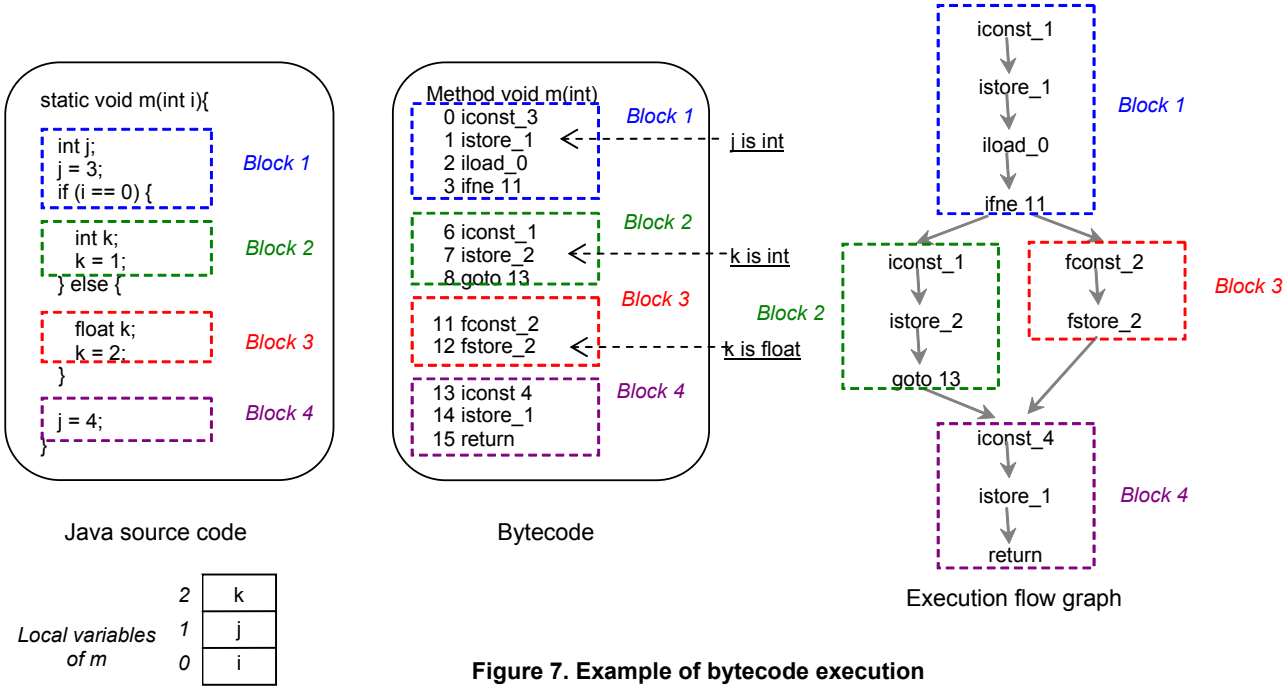

Figure 7. Example of bytecode execution

Our solution to this problem is based on two correctness properties of the Java bytecode [11]:

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.

To summarize, we implemented an algorithm that determines the types of the values on a thread's Java stack in one pass of the bytecode. This algorithm amounts to:

- 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
- inferring the types of the manipulated values from the bytecode instructions contained in this path.

Finally, our 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 in order to capture the thread's Java stack in a portable form.

6 DYNAMIC DEOPTIMIZATION

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, because of Java JIT compilation. In this case, 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.

Here, we would need functions that allow us to restore Java frames from native frames produced by the JIT compiler, and then be able to apply the type inference technique.

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 [23]. 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 [15]. 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. 0 illustrates this problem with an example where a method *m1* calls a method *m2* of a class *C1*. For optimization purpose, the JIT compiler may inline *m2* in *m1*. But this inlining may be-

come invalid if *C2*, a subclass of *C1* that overrides *m2*, is dynamically loaded. Here, dynamic de-optimization is used to revert from optimized (i.e., compiled/inlined) code to interpreted code.

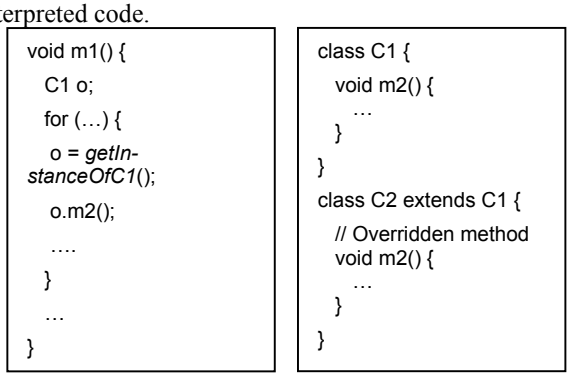

Figure 8. Method inlining and dynamic class loading

Dynamic de-optimization was used in the context of debugging systems and dynamic class loading systems. Here, we use it 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 IMPLEMENTATION STATUS

We have implemented the CTS Java thread serialization mechanism within Sun Microsystems' JVM. The associated type inference system, described in section 5, has been implemented in JDK 1.2.2. Our first prototype of Java thread serialization is therefore proposed as an extension of JDK 1.2.2. We have also experimented with the de-optimization functions provided by JDK 1.3.1 (HotSpot) and showed that we were able to retrieve the Java frames from the JIT compiled frames. We are currently completing the port of the type inference system from JDK 1.2.2 to JDK 1.3.1 in order to produce an integrated prototype of our solution as described in section 6.

Table 1 summarizes the figures of the implementation of our system for Java thread serialization, mobility and persistence.

Java code lines	2500 (+0.2% of original JDK 1.2.2's Java code)
C code lines	17500 (+3% of original JDK 1.2.2's C code)
Supported OS/processors	- Solaris 2.5.1/2.6 on Sparc - Solaris 2.5.1/2.6 on x86 - Windows NT/95/98 on x86

Table 1. Implementation results of Java thread serialization

8 CONCLUSIONS

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 thread serialization mechanism that allows Java programmers to access the execution state of a Java thread as a Java object, and thus to build Java thread transmission and storage facilities. Our thread serialization mechanism is generic: we used it 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 performance.
  - A dynamic de-optimization technique which allows thread serialization to be compliant with Java JIT compilation.

Type inference and dynamic de-optimization are widely used techniques applied in the context of code verification and program debugging. We showed how to use them in the context of thread serialization.

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. Some elements of response are presented in [7], 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.

9 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] Artsy, Y., and Finkel, R. Designing a Process Migration Facility: The Charlotte Experience. IEEE Computer, Vol. 22, No. 9, 1989.

[3] Bagrodia, R., Chu, W. W., Kleinrock, L, Popek, G. Vision, Issues and Architecture for nomadic computing. IEEE Personal Communications Magazine, Vol. 2, No. 6, 1995.

[4] Bershad, B. N., Zekauskas, M. J., Sawdom, W. A. The Midway Distributed Shared Memory System. IEEE Int. Computer Conference (COMPCON'93), Feb. 1993.

[5] Bouchenak, S., Hagimont, D. Zero-Overhead Java Thread Migration. INRIA Technical Report No. 0261, May 2002.

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

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

[8] Czajkowski, G. Application Isolation in the Java(tm) Virtual Machine. 17th Annual ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA'00), Minneapolis, MN, USA, Oct. 2000.

[9] Deconinck, G., Vounckx, J., Cuyvers, R., and Lauwereins, R. Survey of Checkpointing and Rollback Techniques. Technical Report, Katholieke Universiteit Leuven, Belgium, June 1993.

[10] Douglis, F., and Ousterhout, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. Software: Practice and Experience, Vol. 21, No. 8, 1991.

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

[12] 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.

[13] Hagimont, D., Boyer, F. A Configurable RMI Mechanism for Sharing Distributed Java Objects. IEEE Internet Computing, Vol. 5, No. 1, January 2001.

[14] Hofmeister, C., Purtilo, J. M. Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement. 13th International Conference on Distributed Computing Systems, Pittsburgh, PA, USA, May 1993.

[15] Hölzle, U., Chambers, C., Ungar, D. Debugging Optimized Code with Dynamic Deoptimization. ACM SIGPLAN 92 Conference on Programming Language Design and Implementation (PLDI'92), San Francisco, California, USA, Jun. 1992.

[16] Huang, Y., Kintala, C., Wang, Y-M. Software Tools and Libraries for Fault-Tolerance. IEEE Technical Committee on Operating Systems and Application Environments (TCOS), Vol. 7, No. 4, 1995.

[17] IBM Tokyo Research Labs. Aglets Workbench: Programming Mobile Agents in Java, 1996. http://www.trl.ibm.co.jp/aglets

[18] 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.

[19] Izatt, M., Chan, P., and Brecht, T. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. Concurrency: Parctice and Experience, Vol. 12, No. 8, Jul. 2000.

[20] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-Grained Mobility in the Emerald System. ACM Transactions on Computer Science, Vol. 6, No. 1, 1988.

[21] Lindholm, T., Yellin, F. The Java Virtual Machine Specification (2nd Edition), Addison Wesley, 1999.

[22] Litzkow, M. J., Solomo, M. Supporting Checkpointing and Process Migration outside the UNIX Kernel. USENIX Winter Conference, San Francisco, CA, USA, Jan. 1992.

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

[24] Milojičić, D., Douglis, F., and Wheeler, R. Mobility: Processes, Computers and Agents. Addison Wesley, Feb. 1999.

[25] Nichols, D. A. Using Idle Workstations in a Shared Computing Environment. 11th Symposium on Operating Systems Principles (SOSP'11),Austin, TX, USA, Nov. 1987.

[26] Peine, H., and Stolpmann, T. The Architecture of the Ara Platform for Mobile Agents. 1st International Workshop on Mobile Agents (MA'97), Berlin, Germany, Apr. 1997.

[27] Picco, G. P. Mobile Agents. Proceedings of the 5th IEEE Int. Conference on Mobile Agents (MA'2001), Lecture Notes in Computer Science, Vol. 2240, Atlanta, Georgia, USA, Dec. 2001.

[28] 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.

[29] 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.

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

[31] Sun Microsystems. Java JIT Compiler Overview. Sun Microsystems, 2002. http://www.sun.com/solaris/jit/

[32] 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.

[33] Zayas, E. R. Attacking the Process Migration Bottleneck. 11th ACM Symposium on Operating System Principles (SOSP'11), Austin, TX, USA, Nov. 1987.