SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Experiences Implementing Efficient Java Thread Serialization, Mobility and Persistence

S. Bouchenak, D. Hagimont, S. Krakowiak,
N. De Palma, F. Boyer

INRIA (French National Institute for Research in Computer Science and Control)
655, avenue de l'Europe, Montbonnot, 38334 St-Ismier Cedex, France

{*Sara.Bouchenak,Daniel.Hagimont,Sacha.Krakowiak,Noel.Depalma,Fabienne.Boyer*}*@inria.fr*

SUMMARY

Today, mobility and persistence are important aspects of distributed computing. They have many fields of use such as load balancing, fault tolerance and dynamic reconfiguration of applications. In this context, Java provides many useful mechanisms for the mobility of code via dynamic class loading, and the mobility or persistence of data via object serialization. However, Java does not provide any mechanism for the mobility/persistence of computation (i.e., threads).
We designed and implemented a new mechanism, called *Java thread serialization*, 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.
Several projects have recently addressed the issue of Java thread serialization, e.g., Sumatra, Wasp, JavaGo, Brakes, JavaGoX, Merpati. Some of them have attempted to minimize the overhead incurred by the thread serialization mechanism on thread performance, but none of them has been able to completely avoid this overhead.
We propose a generic Java thread serialization mechanism that does not impose any performance overhead on serialized threads. This is achieved thanks to the use of type inference and dynamic de-optimization techniques. In this paper, we describe the design and implementation details of our thread serialization prototype in Sun Microsystems' JDK. We report on experiments conducted with our prototype, present a comparative performance evaluation of the main thread serialization techniques, and confirm the elimination of the performance overhead with our thread serialization mechanism.

KEY WORDS: mobility; persistence; threads; type inference; dynamic deoptimization; JVM

Received 04 December 2002
Revised 17 March 2003

1. INTRODUCTION

Today, mobility and persistence are important aspects of distributed applications and have several fields of use [35, 13]. Application mobility can be used to dynamically balance the load between several machines in a distributed system [38, 20], to reduce network traffic by moving clients closer to servers [16], to dynamically reconfigure distributed applications [24], to implement mobile agent platforms [41, 11], to tackle user nomadism in mobile computing environments [3], or as a machine administration tool [39]. Application persistence can be used for fault tolerance [26, 48] or for application debugging.

In the context of distributed applications, the object paradigm has proven to be well suited and the Java Virtual Machine (JVM) is now considered as a reference platform [21, 31]. Today, the JVM is ported on almost every platform and can therefore be viewed as a universal machine. Among the services provided by the JVM to facilitate the development of distributed applications are:

- Object serialization. The serialization service allows the transfer of Java objects between several nodes or the storage of objects on disk.
- Dynamic class loading. The dynamic class loading service enables the transfer of Java code between several nodes.

Therefore, Java provides useful services for the mobility and the persistence of code and data. However, Java does not provide any service enabling mobility or persistence of running applications (control flows, i.e., processes/threads). Thus, if a running Java application migrates to a new location, only using object serialization and class loading, the execution state of the application is lost. In other words, when arriving on its new location, the migratory application can access its code and its re-actualized data but it has to restart the execution from the beginning. Consequently, the provided Java services are not sufficient for enabling the mobility or persistence of Java control flows.

Several projects have recently addressed the issue of Java thread mobility or Java thread persistence. Some of them have attempted to minimize the overhead incurred by their mechanisms on thread performance, but none of them has been able to completely avoid this overhead. Such an overhead has several reasons:

- Additional instructions inserted in the application code (code executed by the thread); this is the case for the Wasp [19], JavaGo [43], Brakes [47] and JavaGoX [42] thread mobility systems.
- Extension of the Java interpreter, as in the Sumatra thread mobility system [1], and the Merpati [44] and ITS [6] thread mobility and persistence systems.
- Non-compliance with Java JIT (Just-In-Time) compilation (execution optimization), e.g., CIA [28], Sumatra, ITS and Merpati.

All the above-mentioned systems impose a performance overhead because of code injection, Java interpreter extension or non-compliance with JIT compilation.

1.1. Contributions

We designed and implemented new services that make Java threads, i.e. executions, mobile or persistent. With these services, 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 a possible subsequent recovery. With these services, migrating a Java 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. At a lower level, we propose a new mechanism, called *Java thread serialization*. Similarly to object serialization, thread serialization allows Java programmers to access the state of threads and transfer it between several nodes (for mobility), or to store it on disk (for persistence).

The proposed Java thread serialization/mobility/persistence services do not affect the "normal" performance of threads. In this paper, we describe how we built such services. The scientific contributions of the 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.
3. Our performance evaluation comparing several thread serialization approaches and confirming that our mechanism is the unique system that does not affect the "normal" performance of threads.

Our prototype is freely available from:
http://sardes.inrialpes.fr/research/JavaThread/
It has been successfully integrated into the Suma metacomputing platform for fault tolerance purpose, where it was used as a basic service for the implementation of global uncoordinated checkpointing/recovery for parallel computations [10]. In addition to Suma's designers, there were about 200 downloads from users, testers, students and researchers working with our thread serialization service.

1.2. Roadmap

The rest of the paper is structured as follows. Section 2 discusses 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 our overall design to support Java thread serialization. Following this, sections 5 and 6 respectively focus on the implementation details of thread serialization and thread mobility/persistence. Sections 7 and 8 are respectively devoted to the experiments and performance evaluation. Finally, section 9 presents our conclusions.

2. RELATED WORK

Many systems have been developed providing mobility or persistence of control flows, i.e., processes/threads, considering either homogeneous or heterogeneous processor architectures, e.g., Charlotte [2], Sprite [15], Emerald [30], Ara [40]. There are a number of surveys discussing these features [35, 13]. 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, that is 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 de-serialization 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 here 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 instead of object serialization in order to transform agents' weak mobility (i.e., data mobility) into strong mobility (i.e. computation/thread mobility). In the Aglets mobile agent system [27], 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 [12].

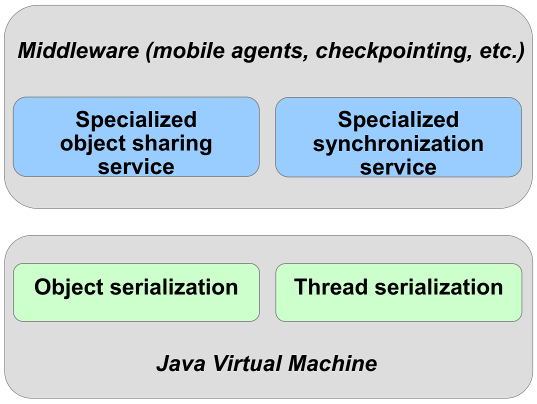

Middleware (mobile agents, checkpointing, etc.)
Specialized object sharing service
Specialized synchronization service
Object serialization
Thread serialization
Java Virtual Machine

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

- A shared object system, e.g., Ajents [29] and Javanaise [22]. The latter project is a Java distributed replicated object system, where synchronization of replicas is based on the entry consistency protocol [5]. 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 [49] and Condor [32] 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.

2.2. Related work

In this section, we focus on the research conducted in the area of Java thread serialization, in a centralized JVM[§], where some projects propose Java thread mobility systems, e.g., Sumatra [1], Wasp [19], JavaGo [43], Brakes [47], JavaGoX [42], CIA [28], and others propose both Java thread mobility and thread persistence, e.g., ITS [6] and Merpati [44].

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 and is not directly accessible to Java programmers. In order to address this issue, two main approaches are followed:

- JVM-level approach.

[§]Further details on distributed Java virtual machines can be found in [50, 10].

- 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 [44], ITS [6] and CIA [28] 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.

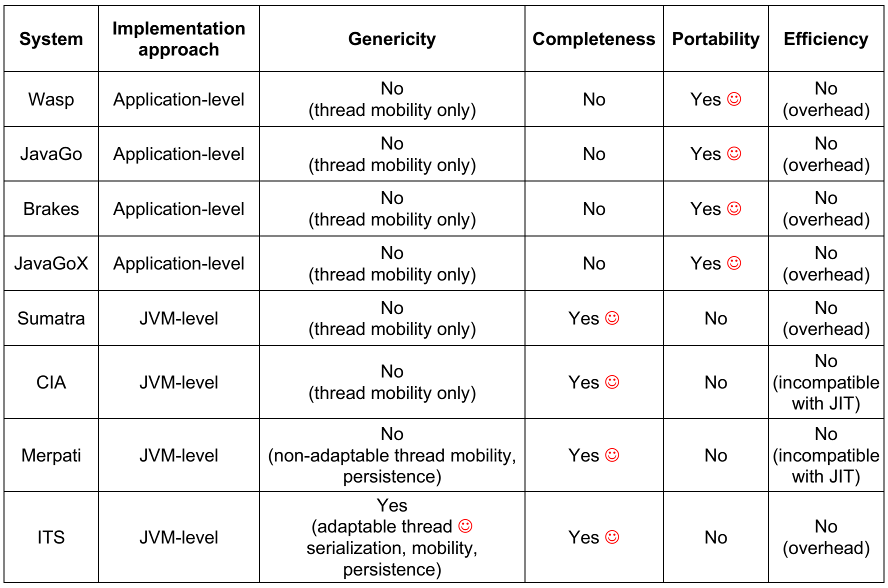
In order to address the issue of non-portability of the thread serialization mechanism on multiple Java environments, some projects propose a solution at the application level, without relying on an extension of the JVM. In this approach, the application code is transformed by a pre-processor, prior to execution, in order to attach a backup object to the Java program executed by the thread, and to add new statements in this program. 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 [19] and JavaGo [43] provide a Java source code pre-processor while Brakes [47] and JavaGoX [42] 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 [8]. The resulting systems are therefore incomplete.

On the other hand, whatever the level of implementation (JVM or application), all the aforementioned solutions impose a performance overhead on threads. Indeed, the JVM-level systems suffer from inducing a significant overhead on thread performance (+335%, +340%, cf. section 8) because some of them extend the Java interpretation process (e.g., Sumatra, Merpati, ITS) and none of them supports Java JIT compilation (execution optimization). And the application-level systems impose a non negligible performance overhead (+88%, +250%, cf. section 8) 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 use it in different contexts 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 are predefined and can not be



System	Implementation approach	Genericity	Completeness	Portability	Efficiency
Wasp	Application-level	No (thread mobility only)	No	Yes ☺	No (overhead)
JavaGo	Application-level	No (thread mobility only)	No	Yes ☺	No (overhead)
Brakes	Application-level	No (thread mobility only)	No	Yes ☺	No (overhead)
JavaGoX	Application-level	No (thread mobility only)	No	Yes ☺	No (overhead)
Sumatra	JVM-level	No (thread mobility only)	Yes ☺	No	No (overhead)
CIA	JVM-level	No (thread mobility only)	Yes ☺	No	No (incompatible with JIT)
Merpati	JVM-level	No (non-adaptable thread mobility, persistence)	Yes ☺	No	No (incompatible with JIT)
ITS	JVM-level	Yes (adaptable thread ☺ serialization, mobility, persistence)	Yes ☺	No	No (overhead)

Figure 2. Existing Java thread serialization systems

adapted to applications' needs; while ITS proposes a generic implementation of adaptive Java thread serialization. Finally, the existing Java thread serialization systems are summarized in Figure 2.

3. BACKGROUND: JVM CHARACTERISTICS

This section recalls the JVM characteristics that are necessary to understand the rest of the paper. The Java Virtual Machine is the runtime environment on which applications developed in the Java object-oriented language can run. A program developed in Java is generally compiled in order to generate bytecode, a binary format which can be interpreted by a JVM. As the JVM is ported on most contemporary machines, a compiled (bytecode-based) Java program is portable between different machines.

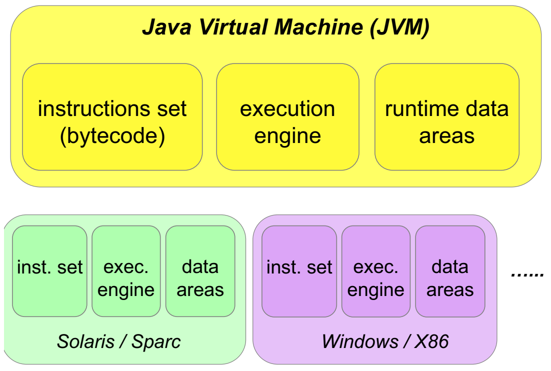


Figure 3. Architecture for the Java Environment

The architecture of the Java environment is illustrated by Figure 3 where the JVM is presented as an abstraction of a homogeneous machine with a defined set of instructions (*bytecode*), an *execution engine* (an equivalent of a hardware processor) and *runtime data areas* used, for example, for memory and process management. In the following, we detail the bytecode properties we are interested in, and the operating principles of the JVM's execution engine 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* (cf., the prefixing letter), and return a result of the same type.

The execution of bytecode in the JVM is based on a stack, called the operand stack. Figure 4 illustrates the execution of the *iadd* instruction which adds two integer operands. 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 (the binary/machine code executed by the underlying processor). In order to improve performance, the second generation of JVM has integrated Java Just-In-Time (JIT) compilers, which dynamically compile Java methods, i.e., bytecode, into native code [45]. The subsequent calls and executions of these

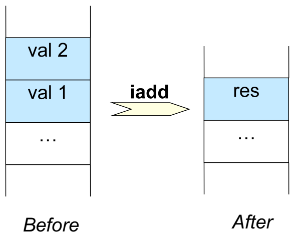


Figure 4. Addition of two integers in the JVM

methods are no more based on the Java interpreter; they directly rely on the underlying processor and therefore perform much faster.

3.3. Runtime data areas: Thread state

Timothy Lindholm and Franck Yellin define in the Java Virtual Machine Specification several runtime data areas [31]. Here, we focus on the data areas describing the execution state of a Java thread, as illustrated by Figure 5:

- *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 by the thread 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-mentioned data areas, and in order to support native methods, the JVM specification mentions a native stack associated with a thread [31]. 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.

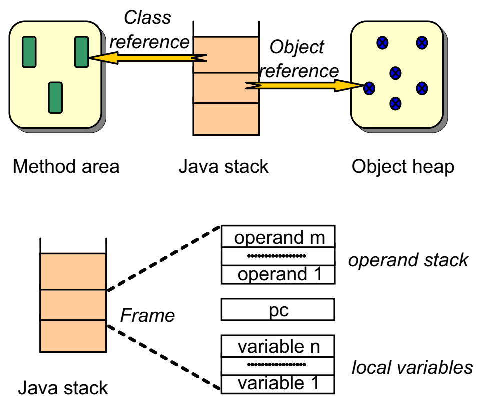


Figure 5. Java thread state

4. DESING OF THREAD SERIALIZATION

Here are the design principles and choices of our Java thread serialization mechanism.

4.1. Design principles

The thread state serialization/de-serialization service enables, on the one hand, the *capture* of the current state of a running thread, and on the other hand, the *restoration* of a previously captured state in a new thread: the new thread starts running at the point at which the execution of the previous thread was interrupted.

Thread serialization consists, more precisely, in interrupting the thread during its execution and extracting its current state. This 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 these frames, 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.

Symetrically, thread de-serialization consists first in creating a new thread and initializing its state with a previously captured state. After that, the Java stack, 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.

4.2. Main issues and design choices

The design of a Java thread serialization mechanism faces serveral issues, such as:

- the accessibility of the execution state of Java threads,
- the portability of this state,
- and the provision of an overhead-free
- and a generic thread serialization mechanism.

In the following, we discuss each of these issues and present our solutions to face them.

4.2.1. Non-accessible thread state

The state of Java threads (Java stack, heap, method area) is internal to the JVM. In other words, there is not a standard Java API that allows the programmer to access a thread's Java stack, heap (objects used by a thread) or method area (the classes used by a thread). This state can therefore not be directly captured in order to implement thread serialization. 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).

4.2.2. Non-portable thread state

Unlike the heap and the method area that consist of information portable on heterogeneous architectures (respectively Java object and bytecode), the Java stack is implemented in most JVMs as a native data structure (C structure). Therefore, the representation of the information contained in the Java stack depends on the underlying architecture. For a serialized thread to be portable on heterogeneous platforms, the thread serialization mechanism must translate the non-portable data structure representing a state (C structure) into 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 operands (c.f., section 3.3) 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 most intuitive 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., on a type stack associated with the thread). Our first prototype of Java thread serialization follows this approach, it is called ITS (Interpreter-based Thread Serialization) [6]. But the drawback of this solution is that it introduces a significant 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) [7, 9].

4.2.3. Overhead-free thread serialization

As discussed in section 2.2, the existing Java thread serialization mechanisms either focus on providing a complete solution at the JVM-level or proposing a portable system at the Java language-level; but none of them tackles the performance overhead issue. 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, our primary objective has been to provide a thread serialization mechanism that does not impose any overhead on the execution of serialized threads. In order to avoid 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 5.1.
- 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 de-optimization technique as described in section 5.2.

4.2.4. Generic thread serialization

One of our motivations was to provide a generic Java thread serialization mechanism which allows the programmer to adapt the serialization policy in order to meet applications' needs. With a generic thread serialization mechanism, various high level services can be built, such as thread mobility or thread persistence; and particular policies can be implemented such as mobility on wireless terminals or persistence using data base systems.

4.2.5. Synthesis

To summarize, the main issues that we faced when extending the JVM with Java thread serialization are the following:

(a) To have access to threads' execution state. This is necessary to build a Java thread serialization mechanism.
(b) To provide a portable Java thread state. This is necessary to follow the Java philisophy regarding portability (code portability, data portability, and in this case, execution portability).
(c) To provide a thread serialization mechanism that is compatible with Java JIT-compilation. This is necessary to provide an effective solution for today's Java environments.
(d) To propose a generic thread serialization service. This approach was followed to respect the genericity and reusability necessary to Java services.

The respectively proposed solutions to these issues are:

(a) An extension of the JVM in order to externalize the thread state, and the provision of a new API that allows the programmer to access this state.
(b) The provision of a type inference mechanism that transforms a non-portable data structure of a thread state to a portable data structure. In a JVM that does not provide any type information (e.g., standard JVM), using a type inference mechanism is the unique solution that tackles the portability issue without incurring a performance overhead.
(c) The use of dynamic de-optimization techniques for JIT-compilation compatibility. Using dynamic de-optimization is the only one technique to revert from JIT-compiled methods, and therefore to provide a Java thread serialization mechanism that is compatible with JIT compilation.
(d) A generic design that follows the object-oriented approach and class hierarchy to propose a generic, adaptable and reusable thread serialization mechanism.

4.2.6. 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 (self-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 operation 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 for Java thread self-serialization without native methods. Further details on how the above issues are tackled can be found in [8].

5. IMPLEMENTATION OF THREAD SERIALIZATION

As described earlier, the main issues that we faced and the main design choices that we made when designing Java thread serialization are the following:

- To have access to a Java thread state, we extend the JVM.
- To provide portable thread state, we propose a type inference technique.
- To build a zero-overhead thread serialization, we combine type inference with dynamic de-optimization techniques.
- To propose several uses of thread serialization, we provide a generic design of the serialization mechanism.

In the following, we describe how we extended the JVM with the type inference and dynamic de-optimization techniques, before giving an overview of the API of our generic Java thread serialization mechanism, and finally describing its current implementation status.

5.1. Type inference

The proposed 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 that 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 thread serialization time, an empty type stack is initially associated with the thread's Java stack. And for each frame on the Java stack, an empty type frame is initially pushed onto the associated type stack. The types of the local variables and operands of the Java frame are then inferred as follow. The bytecode of the associated method is parsed from the beginning of the method to the exit point of the method (the exit point is given by the Java frame's *pc* and represents 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, as local variable or 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? It is important to notice here that different code paths may assume different types for a same item (local variable or operand) on the Java stack. Let us illustrate this problem through an example of a Java method m represented by a Java source code, its equivalent bytecode and the associated execution flow graph (see Figure 7). In this program, local variables i and j are declared in *block 1* and represent values of type *int*, and 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).

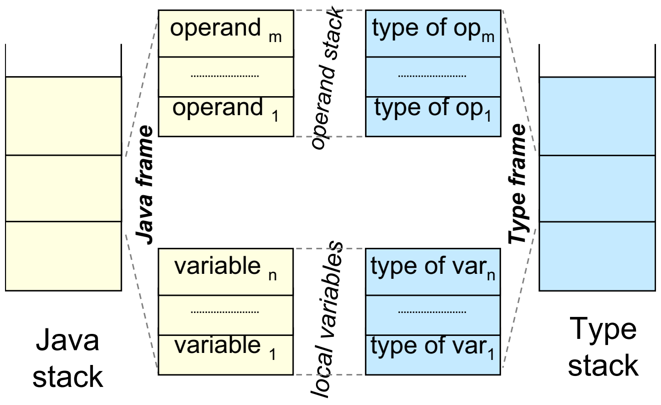
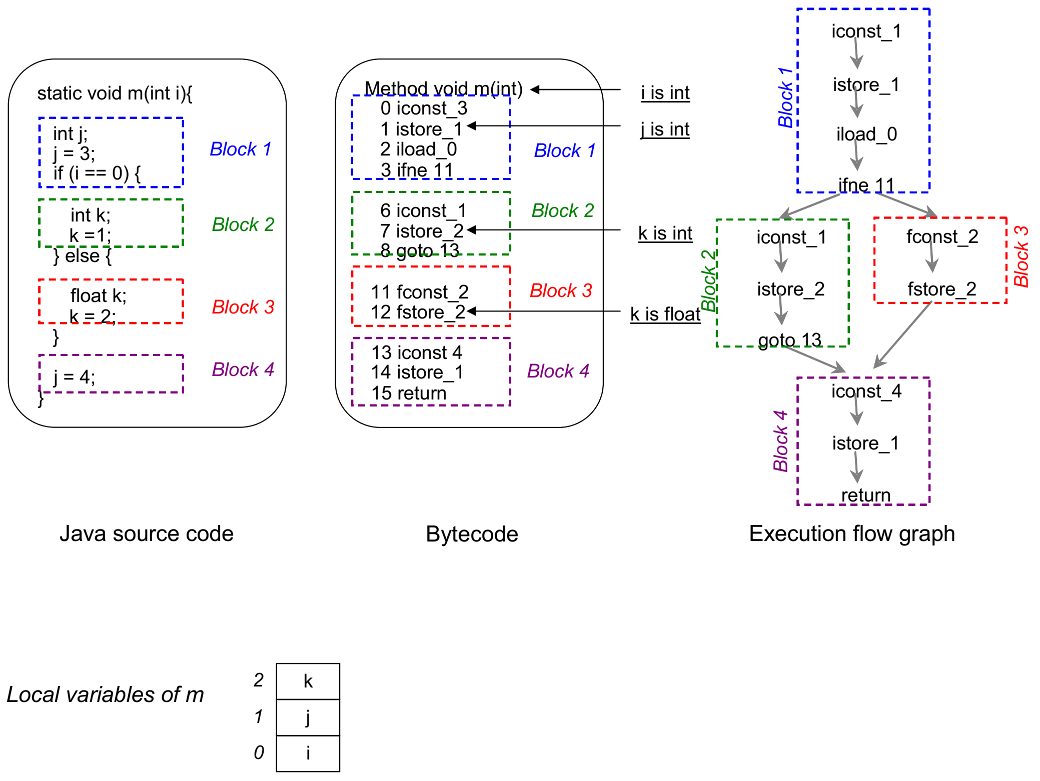


Figure 6. Type stack vs. Java stack

When serializing the thread executing method *m*, and given the non-typed Java frame and bytecode of *m*, how are the types of the local variables/operands of method *m* determined? Four cases are possible here:

1. The exit point (*pc* value) is in *block 1*. In this case, there is only one possible path from the beginning of the code to the exit point. The analysis of this path determines 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 of the bytecode[†] (see Figure 7).
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[†].
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? Is variable *k* of type *int* or of type *float*?

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



```
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 7. Example of bytecode execution

Our solution to this problem is based on two correctness properties of the Java bytecode [17]:

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 either path *block 1-block 2-block 4* or path *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. Therefore, any of the existing code paths can be used for type inference.

To summarize, we implemented an algorithm that infers the types of the values (local variables and operands) on a Java frame, and more generally on a thread's Java stack, in one pass of the bytecode. This algorithm is applied at thread serialization time; it 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. Furthermore, type inference is a technique used in bytecode verification, which is generally applied at Java class loading time [31]. Thus, a possible optimization of the thread serialization system can be made if the class loading system keeps the type information that it builds during bytecode verification: this would prevent thread serialization from re-building the necessary type information.

5.2. Dynamic de-optimization

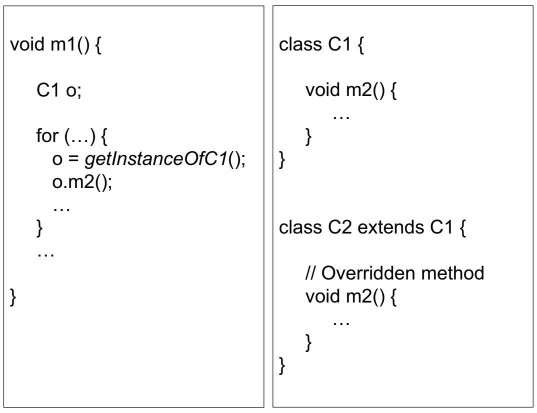
The type inference technique described in the previous section requires access to the thread's Java stack. But the Java stack may sometimes not reflect the current execution state of the thread, because of Java JIT compilation. In this case, the execution of JIT compiled methods is no longer based on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

Here, the thread serialization mechanism would need functions that allow it to restore Java frames from native frames produced by the JIT compiler, and then to 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 [34].

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 [25]. 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. Figure 8 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 become invalid if *C2*, a subclass of *C1* that overrides *m2*, is dynamically loaded and if the *getInstanceOfC1* called in method *m1* return an instance of *C2*. Here, dynamic de-optimization is used to revert from inconsistent optimized (i.e., compiled/inlined) code to a valid interpreted code.



```
void m1() {

    C1 o;

    for (…) {
      o = getInstanceOfC1();
      o.m2();
      …
    }
    …

}
```

```
class C1 {

    void m2() {
        …
    }
}


class C2 extends C1 {

    // Overridden method
    void m2() {
        …
    }
}
```

Figure 8. Method inlining and dynamic class loading

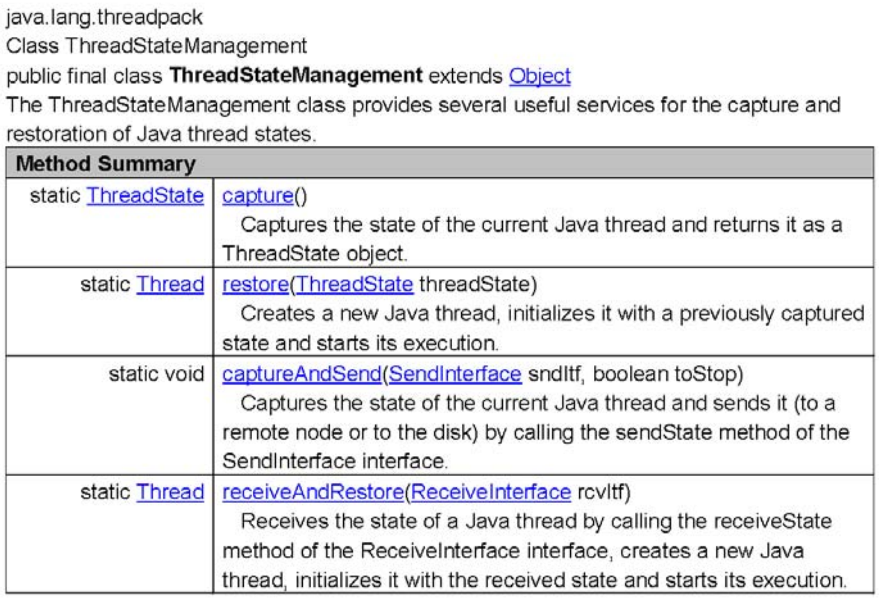
Dynamic de-optimization was used in the context of debugging systems and dynamic class loading systems. Here, we use it in a thread serialization system as follows. At serialization time, in case some Java methods were JIT compiled, dynamic de-optimization is invoked on the thread's JIT compiled frames. This leads to retrieve the Java frames that would have been produced by the Java interpreter. Therefore, the type inference algorithm described in section 5.1 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 performance of serialized threads. Finally, Java applications that use our thread serialization mechanism continue to benefit from JIT compilation, before and after thread serialization: they execute exactly in the same conditions as on an unmodified JVM[¶].

5.3. API of thread serialization

Our Java thread serialization mechanism is proposed in a new Java package, called *java.lang.threadpack*. This package provides many classes such as the *ThreadState* class whose instances represent the execution state of Java threads and the *ThreadStateManagement* class that provides the necessary features for Java thread serialization.

Figure 9 illustrates a part of the application programming interface (API) of the *ThreadStateManagement* class. The *capture* method performs the serialization of a Java thread and returns the captured thread state as a result of the method, as a *ThreadState* object. Symmetrically, the *restore* method performs thread de-serialization. It creates a new Java

[¶]In [50], a thread migration system compatible with JIT compilation is also proposed. It is not based on dynamic de-optimization but on a particular extension/implementation of the JIT compiler.


java.lang.threadpack
Class ThreadStateManagement
public final class **ThreadStateManagement** extends Object
The ThreadStateManagement class provides several useful services for the capture and restoration of Java thread states.

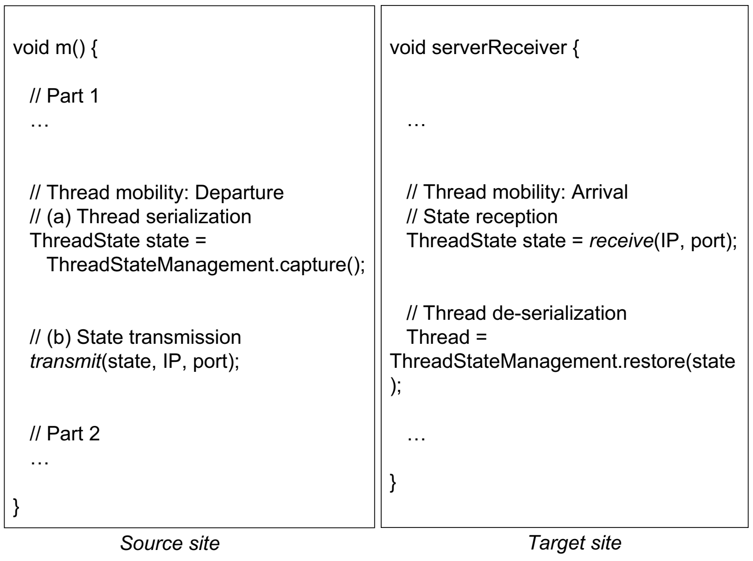
Method Summary	
static ThreadState	capture() Captures the state of the current Java thread and returns it as a ThreadState object.
static Thread	restore(ThreadState threadState) Creates a new Java thread, initializes it with a previously captured state and starts its execution.
static void	captureAndSend(SendInterface sndItf, boolean toStop) Captures the state of the current Java thread and sends it (to a remote node or to the disk) by calling the sendState method of the SendInterface interface.
static Thread	receiveAndRestore(ReceiveInterface rcvItf) Receives the state of a Java thread by calling the receiveState method of the ReceiveInterface interface, creates a new Java thread, initializes it with the received state and starts its execution.

Figure 9. Thread serialization mechanism

thread, initializes its state with the *ThreadState* argument, starts the new thread and returns it as a result of the method. The de-serialized thread resumes the execution of the thread whose state was previously captured and passed as an argument of the *restore* method.

With the proposed thread serialization mechanism, it is possible to build higher-level services such as specialized thread mobility or thread persistence, thanks to our *captuteAndSend* and *receiveAndRestore* methods. To motivate the usefulness of these methods, let us consider an example. The implementation of thread mobility upon thread serialization could naively be performed as described in Figure 10 where:

- On the source site, a thread starts executing *Part 1* of method *m* and then migrates to a target site by first performing thread serialization using our *capture* primitive, before transmitting the thread state to the target site using, for example, Java object serialization.
- On the target site, the migrating thread is received by first receiving its execution state (e.g., using object de-serialization and dynamic class loading) and then performing thread de-serialization using our *restore* primitive. Here, the de-serialized thread would resume its execution starting at *Part 2* of method *m*.



```
void m() {

  // Part 1
  …


  // Thread mobility: Departure
  // (a) Thread serialization
  ThreadState state =
    ThreadStateManagement.capture();


  // (b) State transmission
  transmit(state, IP, port);


  // Part 2
  …

}
```

Source site

```
void serverReceiver {


  …


  // Thread mobility: Arrival
  // State reception
  ThreadState state = receive(IP, port);


  // Thread de-serialization
  Thread =
ThreadStateManagement.restore(state
);

  …

}
```

Target site

Figure 10. Naive implementation of thread mobility

But with this solution, the de-serialized thread will resume its execution at the point following the thread serialization operation that is state transmission (part *(b)* of method *m* in Figure 10 and not *Part 2*). This behavior is similar to the UNIX *fork*.

In order to tackle this problem, we propose the *captureAndSend* method that allows the programmer to specify the way a thread state is handled after a *capture* operation: the captured state can for example be sent to a remote machine for mobility purpose, it can be stored on disk to implement persistence, etc. The specialization of the handling of the captured state is specified by the first argument of the *captureAndSend* method. Indeed, this argument implements our *SendInterface* interface and so provides a *sendState* method that is called by our *captureAndSend* method just after the capture of the thread state (see Figure 11). The second argument of the *captureAndSend* method is a boolean that specifies if the thread whose state is captured is stopped or resumed. This argument is, for example, set to true in the case of thread migration and is set to *false* for remote thread cloning.

Symmetrically, the *receiveAndRestore* method specifies the way a thread state is received before it is restored: the state can for example be received from a remote machine, or it can be read from disk, etc. The specialization of the way the thread state is received is possible thanks to the argument of the *receiveAndRestore* method: this argument implements our *ReceiveInterface* interface and so provides a *receiveState* method that is called by our *receiveAndRestore* method just before the restoration operation (see Figure 11).

Finally, *captureAndSend* and *receiveAndRestore* are proposed as generic methods that can specialize thread serialization to application needs.

```
public static void captureAndSend
     (SendInterface sndItf,
     boolean toStop) {

     ThreadState state;


     // Thread state capture
     state =
ThreadStateManagement.capture();


     // Thread state handling
     sndItf.sendState(state);


     // Resuming or stopping the thread
     if (toStop)
        Thread.currentThread().stop();

}
```

```
public static Thread receiveAndRestore
     (ReceiveInterface rcvItf) {


     ThreadState state;
     Thread          thread;


     // Thread state handling
     state = rcvItf.receiveState();


     // Thread state restoration
     thread =
ThreadStateManagement.restore(state);


     return thread;

}
```

Figure 11. Generic thread serialization

5.4. Implementation status

The type inference system, described in section 5.1, has been implemented in Sun Microsystems' JDK 1.2.2. Our first prototype of Java thread serialization is therefore proposed as an extension of JDK 1.2.2.

We have then 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 sections 5.1 and 5.2. The performance evaluation presented in section 8 is thus based on our extended JDK 1.2.2.

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

6. THREAD MOBILITY AND THREAD PERSISTENCE

One of our main motivations was to provide a generic thread serialization mechanism that can be used to implement various higher level services such as thread mobility and thread persistence. Therefore, besides our mechanism for capturing/restoring the state of Java threads, we provide higher-level services for the mobility and the persistence of Java threads.

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 I. Implementation results of Java thread serialization

Making a thread mobile is the action of capturing the current execution state of the thread, sending this state to a target machine and restoring the state in a new thread on the target machine: the new thread resumes the execution in the state left by the original thread.

In the same way, making a thread persistent is, first, the action of capturing the current state of the thread and saving it on disk and then, the ability to restore the saved state in a new thread: the new thread resumes the execution of the previous thread.

Java thread mobility and Java thread persistence facilities are respectively provided by our *MobileThreadManagement* and *PersistentThreadManagement* classes, in the *java.lang.threadpack* package.

6.1. API and implementation of thread mobility/persistence

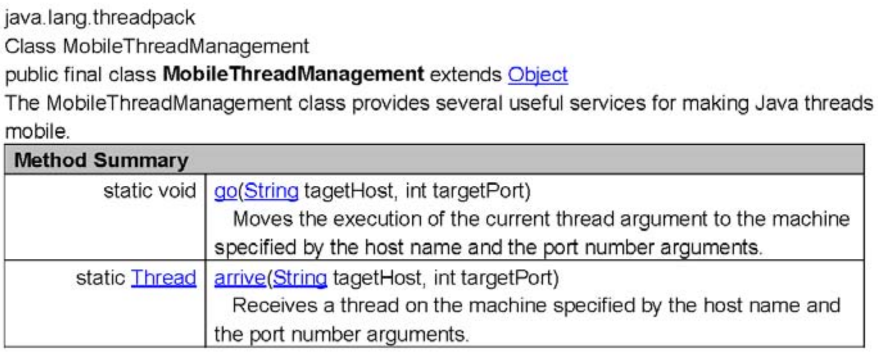
Our *MobileThreadManagement* class provides the necessary services for the mobility of Java threads. Figure 12 illustrates a part of the API of this class. The *go* method transfers the execution of a running Java thread to a Java virtual machine identified by an IP address and a port number. And the *arrive* method enables the reception of a migrating Java thread.

The *go* method is implemented as a combination of our thread serialization mechanism with Java object serialization in order to transmit the captured thread state (see Figure 13):

- The go method calls our *captureAndSend* method which first captures the current state of the thread.
- As presented in section 5.3, *captureAndSend* is a generic method; it is adapted here using an instance of the *MySender* class.
- The *MySender* class implements our *SendStateInterface* interface and thus provides a *sendState* method. Here, this method aims at establishing a connection to a machine and sending the *ThreadState* object using object serialization.

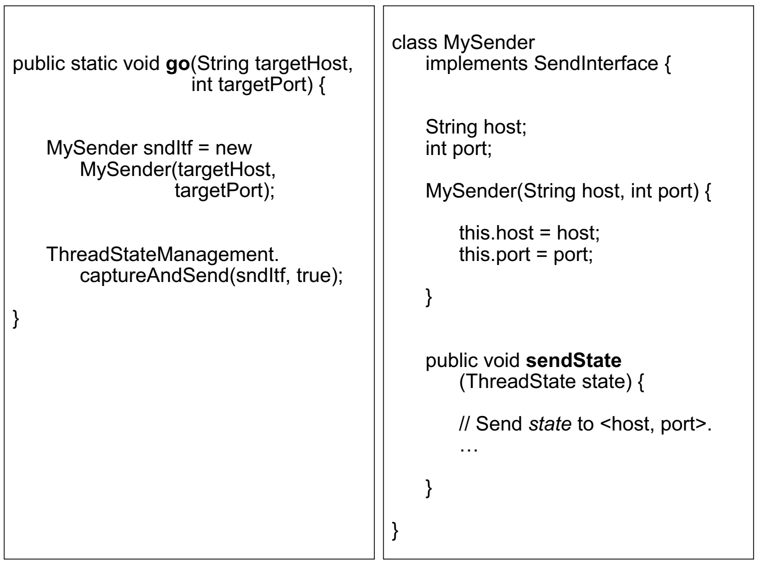
The *arrive* method is implemented as a combination of our thread de-serialization mechanism with Java object de-serialization and dynamic class loading in order to receive the thread state (see Figure 14):

- The *arrive* method calls our *receiveAndRestore* method.


java.lang.threadpack
Class MobileThreadManagement
public final class **MobileThreadManagement** extends Object
The MobileThreadManagement class provides several useful services for making Java threads mobile.

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

Figure 12. Java thread mobility service


```
public static void go(String targetHost,
                      int targetPort) {

    MySender sndItf = new
        MySender(targetHost,
                 targetPort);

    ThreadStateManagement.
        captureAndSend(sndItf, true);
}
```

```
class MySender
    implements SendInterface {

    String host;
    int port;

    MySender(String host, int port) {

        this.host = host;
        this.port = port;

    }

    public void sendState
        (ThreadState state) {

        // Send state to <host, port>.
        …

    }
}
```

Figure 13. Implementation of the *go* method

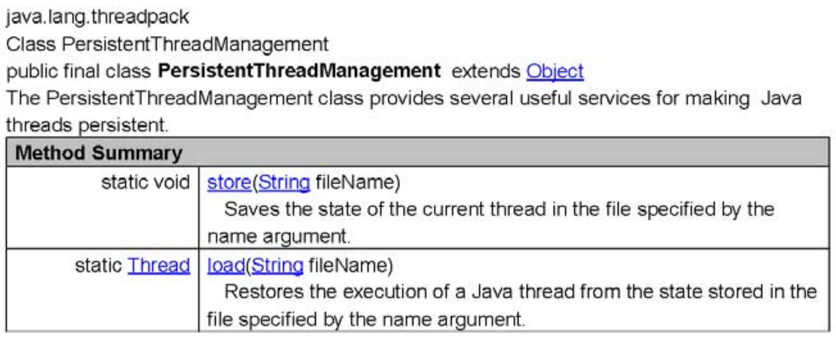
```
public static Thread arrive
     (String targetHost,
      int targetPort) {


     MyReceiver rcvItf = new
          MyReceiver(targetHost,
                         targetPort);


     return ThreadStateManagement.
          receiveAndRestore(rcvItf);


}
```

```
class MyReceiver
     implements ReceiveInterface {

     String host;
     int port;

     MyReceiver(String host, int port) {
          this.host = host;
          this.port = port;

     }


    public ThreadState receiveState()
     {
          // Receive a state on
         // <host, port> and return it.
         …

     }
}
```

Figure 14. Implementation of the *arrive* method

- The *receiveAndRestore* method is a generic method that is adapted here using an instance of the *MyReceiver* class.
- The *MyReceiver* class implements our *ReceiveStateInterface* interface and therefore provides a *receiveState* method; this method aims at accepting a connection on a particular machine/port, and then receiving a *ThreadState* object using de-serialization. The classes associated with this *ThreadState* object are received relying on the Java dynamic class loading mechanism.
- After that, the *receiveAndRestore* method restores the received thread state in a new Java thread (cf., section 5.3).

It is important to notice that the presented Java thread mobility service proposes a default behavior, where all Java objects and classes used by the mobile thread are transmitted with the thread. In other words, default thread serialization behaves as default object serialization and class loading, i.e., leaving decisions related to object sharing, static fields, synchronization and non-serializable objects to the application programmer. And because Java object serialization and dynamic class loading are themselves generic facilities, they can be specialized in order to build various thread transmission and storage policies. Object serialization can, for example, be specialized in order to specify a particular management of IO objects included in the thread's heap, such as *Socket* objects that may be closed at serialization time, and recreated and reconnected at de-serialization time. And dynamic class loading can be specialized in order to use a particular URL for fetching thread's classes. We can also imagine *go* and arrive methods


java.lang.threadpack
Class PersistentThreadManagement
public final class **PersistentThreadManagement** extends Object
The PersistentThreadManagement class provides several useful services for making Java threads persistent.

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 15. Java thread persistence service

that rely on wireless transport protocols instead of IP in order to perform thread migration between JVM installed on wireless hosts [14].

In the same way, the *PersistentThreadManagement* class provides several functions for the persistence of Java threads. A part of its API is illustrated by Figure 15. The *store* method saves the current state of a Java thread in a file specified by a name and the *load* method restores a Java thread from a state saved in a file identified by a name. These two methods are also implemented using our *captureAndSend* and *receiveAndRestore* generic methods.

Finally, the *MobileThreadManagement* and *PersistentThreadManagement* classes are two possible adaptations of our generic Java thread serialization service. In the same way and depending on application requirements, the generic thread serialization service can be adapted to build other tools that meet specific applications' needs.

7. EXPERIMENTS WITH THREAD MOBILITY/PERSISTENCE

This section gives an idea of how our thread mobility and persistence services can be used by applications; through three experiments. The first experiment shows the usefulness of strong mobility (mobility of computation/thread), the second experiment shows how to build a dynamic reconfiguration tool on top of our mobility service, and the third experiment describes how the Suma metacomputing platform's designers use our thread persistence mechanism for fault tolerance purpose.

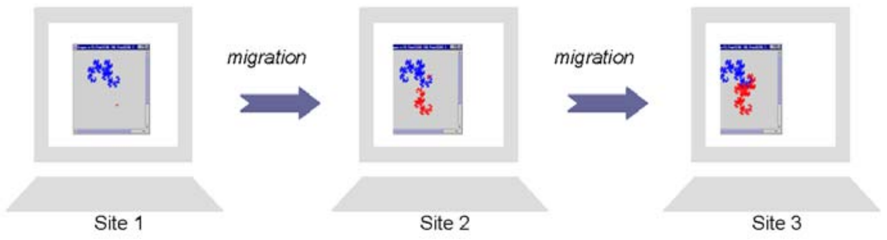


Figure 16. Mobile Fractal

7.1. Strong mobility: Mobile recursive *Fractal*

Two degrees of application mobility can be distinguished: *weak mobility* and *strong mobility* [18]. With weak mobility, only data state information and application's code are transferred. Therefore, on the new location, the mobile application has its actualized data but restarts execution from the beginning. With strong mobility, the code of the application, the state of data and execution are transferred: the application on the destination location resumes its execution at the point where it was interrupted on the source location.

The use of weak or strong mobility depends on applications' needs. Let us consider a recursive Java application; how such an application is made mobile?

- Weak mobility does not consider the state of execution (thread's state and in particular thread's Java stack), so frames corresponding to recursive calls and previously pushed onto the Java stack are lost after transmission and the execution restarts from the beginning.
- Strong mobility captures the execution state and allows the execution to be resumed after transmission.

For demonstration purpose, we experimented a recursive graphical Java application: the Dragon fractal curve where a small dragon appears at a certain depth of recursion [33]. We implemented a Java Dragon application and used our thread mobility service in order to transfer the application, when it is running, on several sites. Figure 16 illustrates this experiment. Here, the Dragon application is first started on a first site, then transmitted to a second site where it resumes its execution and finally transferred to a third site where it completes its execution. The transfer of the thread calculating the fractal is performed by calling the *go* method of our *MobileThreadManagement* class. Finally, this Dragon Fractal demonstration application illustrates the usefulness of thread mobility (i.e., strong mobility).

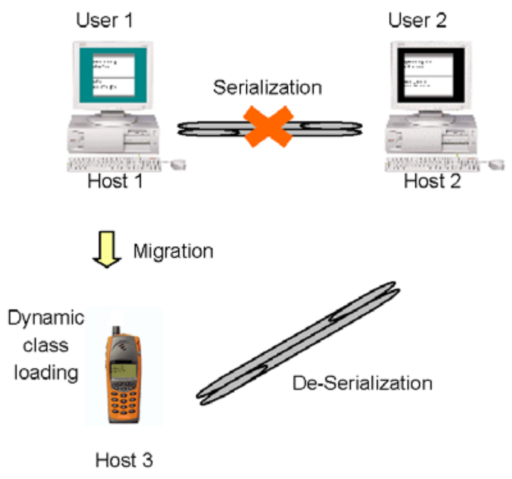


Figure 17. Mobile Talk

7.2. Dynamic reconfiguration: Mobile *Talk*

This second experiment shows how our mobility service can be combined with adapted object serialization and dynamic class loading, in order to build a dynamic reconfiguration tool.

We consider a *Talk* application where two remote users exchange messages. Initially, each user starts an instance of the *Talk* application on its personal computer with a graphical user interface. Each user has two communication channels: an input channel to receive messages from the remote user and an output channel to send messages to the remote user. During the talk, one of the users decides to transfer its application to a minimal host with limited physical characteristics (a mobile phone for example) and to resume its execution. This dynamic reconfiguration of the *Talk* application is illustrated by Figure 17; it has the following requirements:

- Moving a running application from one host to another. This is performed by our thread mobility mechanism which takes into account the current state of the application.
- Handling communication channels during transfer. This is achieved by specializing Java object serialization. Indeed, serialization of the communication channel objects can be adapted in order to send a particular message to the remote user informing him about the next migration and then to close the connections. Symmetrically, de-serialization of the communication channel objects can be adapted in order to recreate new channels and re-establish the connection with the remote user.

- Replacing the graphical user interface by a textual user interface when arriving on the destination host because of the limited physical characteristics. This can be performed by adapting Java dynamic class loading in order to use a textual user interface instead of the graphical one on the mobile phone.

Finally, this dynamic reconfiguration experiment shows how our thread serialization/mobility mechanism can be combined with other Java mechanisms (object serialization, dynamic class loading) in order to build higher level services.

7.3. Fault tolerance: Global checkpointing/recovery

SUMA (Scientific Ubiquitous Metacomputing Architecture) is a distributed platform that supports the execution of parallel Java computations [23]. These parallel computations communicate through message passing that is implemented in SUMA using *mpiJava*, a Java interface to the Message Passing Interface (MPI) [4].

SUMA's designers extended their platform with a new facility: parallel checkpointing and recovery for fault tolerance purpose. This is illustrated by Figure 18, where the implementation of parallel checkpointing/recovery is based:

- On the one hand, on our Java thread persistence mechanism in order to perform local checkpoints and recoveries of individual computations.
- And on the other hand, on a protocol proposed by Mostefaoui et al. in [36, 37] to implement global uncoordinated checkpointing and recovery of parallel computations.

Therefore, our Java thread persistence mechanism was successfully integrated into the SUMA platform. The implementation details and performance evaluation regarding this integration are deeply discussed by Cardinale et al. in [10].

8. EVALUATION

This section first describes our evaluation environment, and then presents the performance figures of our Java thread mobility/persistance mechanisms before giving the results of a comparative performance evaluation that we made between several Java thread serialization systems.

8.1. Evaluation environment

The performance results presented here were obtained in the following environment:

- Pentium III, 1 GHz mono-processor, 256 MB RAM,
- Windows NT, SP 4,
- Sun Microsystems' Java Development KIT/Version 1.2.2, also known as Java 2 SDK/Version 1.2.2.

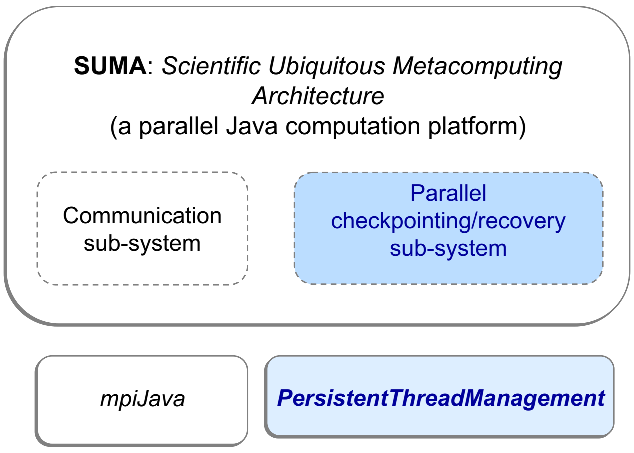

SUMA: Scientific Ubiquitous Metacomputing Architecture
(a parallel Java computation platform)
Communication sub-system
Parallel checkpointing/recovery sub-system
mpiJava
PersistentThreadManagement

Figure 18. Global checkpointing/recovery in a metacomputing system

8.2. Evaluation of thread mobility/persistence

This section presents the performance figures of the Java thread mobility and persistence mechanisms proposed by our CTS system:

- the cost of migrating a Java thread between two machines,
- and the cost of checkpointing/recovering a Java thread.

The time spent in a Java thread migration operation depends on the size of the thread state at migration time. Because a thread state consists of a heap, a method area and a Java stack (c.f., 3.3), its size varies according to the number and size of objects used by the thread, the number and size of classes used by the thread, and the number and size of frames on the thread's Java stack. In this paper, and because of space limitation, we focus our attention on the influence of the number of frames on the cost of our mechanisms. In order to vary the number of frames pushed onto the thread's Java stack, we use a recursive program (the factorial function).

Figure 19 describes, on the left-hand side, the variation of the cost of a thread migration operation according to the number of frames on the thread's Java stack at migration time. The cost of thread migration linearly varies from 30 ms to 190 ms when the number of frames on the thread's stack is between 1 and 100. Figure 19 gives, on the right-hand side, the ratio of the basic operations (i.e., state capture, transfer and restoration) to a migration operation. It shows that the cost of thread migration is mainly due to the cost of thread state transfer. Indeed, in a thread migration operation, between 2% and 9% of the time is devoted to state capture and less than 3% is required for state restoration, while 89% to 95% of the time

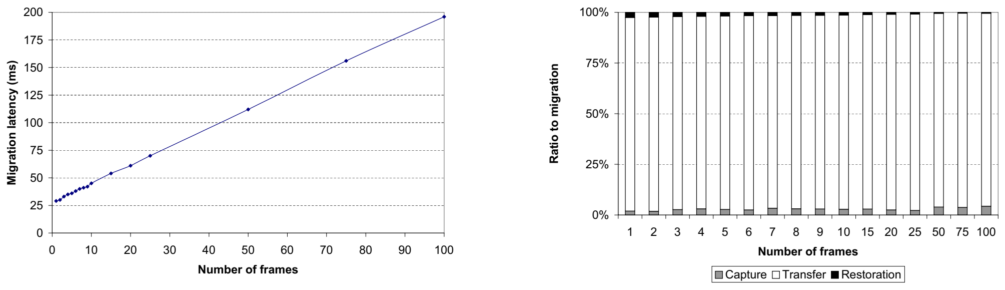


Figure 19. Java thread migration

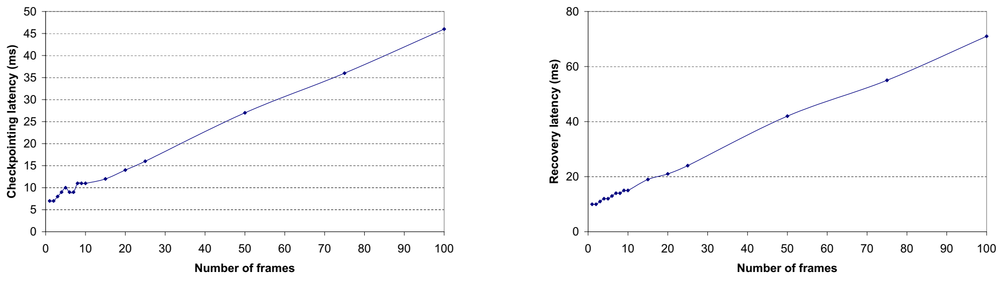
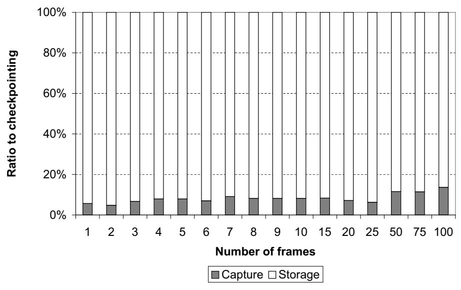


Figure 20. Java thread cehckpointing and recovery

is necessary to state transmission. Therefore, reducing the cost of state transmission would significantly reduce the overall migration latency. In our Java thread migration mechanism, the implementation of state transmission partly relies on Java object serialization. The resulting performance can be improved by using Java externalization rather than object serialization. Indeed, externalization allows applications programmer to write its own object transmission policy by only saving the information necessary to rebuild object graphs. Externalization may be until 40% faster than object serialization [46].

Besides thread migration, we measured the cost of checkpointing a running Java thread and saving its state on disk, and the cost of reading a thread state from disk and recovering the execution. Figure 20 gives the cost of a Java thread checkpointing operation and the



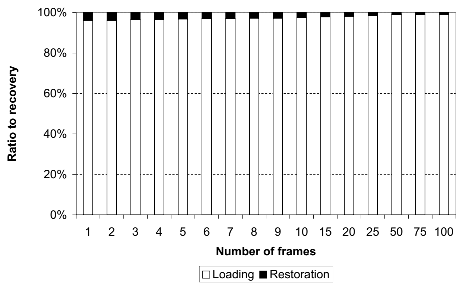


Figure 21. Ratio of state capture/write to thread checkpointing and state read/restoration to thread recovery

cost of a thread recovery operation, according to the number of frames on the thread's Java stack at checkpointing time. We notice that these two costs vary linearly when the number of frames on the thread's Java stack varies. And according to Figure 21, 86% to 95% of the time of thread checkpointing is spent in writing the thread's state on disk, and 96% to 99% of the time of thread recovery is spent in reading the thread's state from disk. Similarly to thread migration, the checkpointing and recovery latencies can be improved by improving disk read/write operations.

8.3. Comparative evaluation

In this section, we present the results of our comparative performance evaluation of several Java thread serialization systems. In this evaluation, two measurements have been reported (see Figure 22):

- *The performance overhead on code execution.* It is defined as the difference between the necessary time to execute the application code on a system that provides thread serialization (E2a + E2b) and the necessary time to execute the same code on a system that does not provide thread serialization (E1), that is (E2a + E2b) - E1.
- *The latency of serialization.* It is defined as the sum of the time necessary to thread serialization/de-serialization, that is S.

We have taken into account both performance overhead and serialization latency in order to show when a cost is paid with thread serialization. And in order to explain how these costs vary according to several thread serialization techniques, we installed and configured several Java thread serialization prototypes, which cover all the approaches to thread serialization as described in section 2.2:

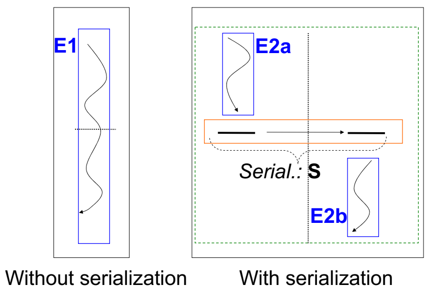


Figure 22. Performance overhead vs. thread serialization latency

- JavaGo [43], an application-level system based on a pre-processor of the Java application source code executed by the serialized Java thread,
- Brakes[‡] [47] and JavaGoX [42], two application-level systems based on a pre-processor of the bytecode executed by the serialized thread,
- ITS [6], the first JVM-level solution that we proposed, based on an extension of the Java interpreter,
- and finally CTS, our final JVM-level solution, based on type inference and dynamic de-optimization techniques integrated into the JVM.

The Sumatra [1] and Merpati [44] projects are no longer maintained, and the current implementation of thread serialization in CIA [28] is in progress. We were thus unable to include these systems in our comparative evaluation.

The following subsections 8.3.1 and 8.3.2 respectively discuss our evaluation of performance overhead and serialization latency.

8.3.1. Performance overhead

In order to evaluate the variation of the performance overhead on a thread according to the amount of computation performed by the thread, we wrote a benchmark based on the Fibonacci recursive algorithm. The performance overhead incurred by the different thread serialization systems was evaluated as follows:

- For the JavaGo, Brakes and JavaGoX thread mobility systems, the pre-processor of the systems is first applied to the Fibonacci benchmark. And because the pre-processor only applies on programs that call the "move" method of the underlying thread

[‡]In this comparative evaluation, we used the fastest implementation of Brakes' thread serialization, namely "brakes-serial".

```
static boolean toMove = false;

int fibo(int n) {
    if (n == 0) {
        // False migration
        if (toMove)
            move("//IP:port/");
        return 1;
    } else if (n == 1)
        return 1;
    else
        return (fibo(n – 1) + fibo(n – 2));
}
```

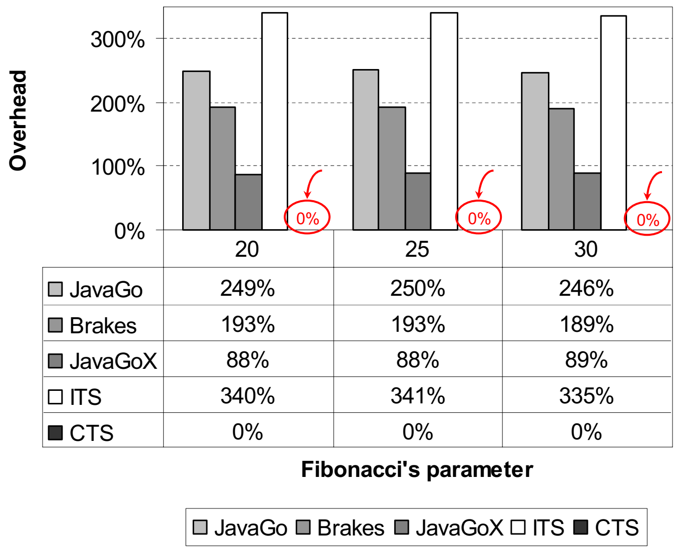
Figure 23. Extended Fibonacci program

mobility system, we extended the Fibonacci program in such a way that it calls the "move" method. And in order to evaluate the performance overhead separately from the serialization latency, the "move" method must not be executed by the benchmark, and is thus placed inside an if-block with a false condition (see Figure 23). Finally, this benchmark is run on the standard JVM.

- For the ITS system, we wrote a benchmark that is based on the original (non-extended with "move") Fibonacci program. This benchmark is run on the ITS-related extension of the JVM.
- For the CTS system, the original Fibonacci benchmark is run on the CTS-based extension of the JVM.

The above mentioned benchmarks were run using several Fibonacci's parameter values (i.e., 20, 25, 30). The obtained results were compared to the ones obtained when running the original Fibonacci benchmark on the standard JVM, in order to calculate the performance overhead incurred by each system. The resulting overheads are presented in Figure 24:

- JavaGo ░, Brakes ▒ and JavaGoX ▓ incur a non-negligible performance overhead (+88% to +250%), due to the code inserted by the pre-processor in the application code. Among these systems, JavaGo incurs the highest overhead because it adds Java code to the application's source code while Brakes and JavaGoX follow a more fine-grained approach that adds code at the bytecode level.
- ITS □ imposes a significant overhead (+335% to +340%) due to the additional processing performed by the underlying extended Java interpreter at almost each bytecode interpretation.



	20	25	30
▢ JavaGo	249%	250%	246%
▨ Brakes	193%	193%	189%
▨ JavaGoX	88%	88%	89%
□ ITS	340%	341%	335%
■ CTS	0%	0%	0%

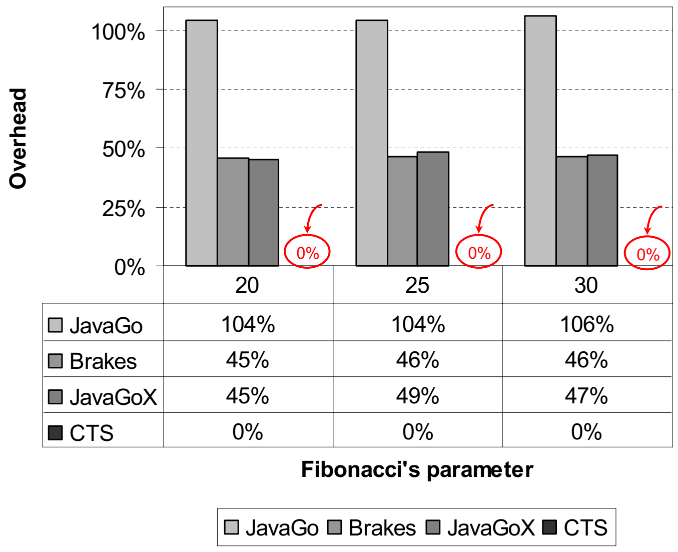
Figure 24. Performance overhead / Fibonacci benchmark (JIT disabled)

- CTS ■ does not incur any performance overhead, because it does not impose any additional computation.

In Figure 24, the illustrated performance figures result from benchmarks running without Java JIT compilation. This was necessary in order to be able to compare the ITS system with other systems. Indeed, ITS is based on an extended Java interpreter and can therefore only be used in interpreted mode (without JIT compilation). The effective performance overheads (with JIT compilation) incurred by the other systems (JavaGo, Brakes, JavaGoX, CTS) are presented in Figure 25. Finally, even if JIT compilation reduces the performance overhead incurred by JavaGo, Brakes and JavaGoX, it does not cancel it (+45% to +106%), thus heavily penalizing serializable Java threads.

8.3.2. Serialization latency

The previous section shows that CTS is the only Java thread serialization system that does not incur a performance overhead on serializable threads. This behavior is not magic: it is due to the fact that with CTS, all additional processing is transferred to thread serialization time. In this section, we aim at discussing the relationship between, on the one hand, the variation of the performance overhead on a serialized thread and, on the other hand, the variation of the thread serialization latency.



	20	25	30
▨ JavaGo	104%	104%	106%
▨ Brakes	45%	46%	46%
▨ JavaGoX	45%	49%	47%
■ CTS	0%	0%	0%

Figure 25. Performance overhead / Fibonacci benchmark (JIT enabled)

In order to be able to compare the latency of Java thread serialization using different systems, these systems must be as similar as possible. But the evaluated systems implement different thread serialization and transfer policies. To homogenize the comparative evaluation environment, and to be as close as possible to the thread serialization operation itself (and not to the network transfer), we built mid-level thread serialization operations for all the evaluated systems:

- We first implemented a common thread state "transfer" mechanism which is simply based on default Java object serialization and system class loading.
- We then built mid-level thread serialization mechanisms for JavaGo, Brakes and JavaGoX by modifying the implementation of these systems in order to replace their state transfer mechanism by the new one.
- We finally built mid-level thread serialization mechanisms for ITS and CTS by combining the new state transfer mechanism to ITS and CTS's thread serialization mechanisms.

The mid-level thread serialization mechanisms implemented for each system were then used as a basis for benchmarking these systems and comparing their thread serialization latencies.

Let us now focus on the implementation of the latency benchmark. Thread serialization latency varies when the size of the thread state varies, e.g., number and size of Java frames pushed on the Java stack, number and size of objects in the heap. The latency benchmark that we present in this paper was written in such a way that we fixed the number of objects used by the serialized thread and focused on the variation of the number of Java frames on the thread's Java stack. To vary the number of frames on the thread's stack, the serialization latency benchmark program first performs recursive calls to a Java method. When it reaches the deepest recursive call (N frames), the benchmark runs a ping-pong program using the thread serialization facility of the benchmarked system in order to measure the average latency of a thread serialization operation when the thread has a particular state size (N frames).

In addition to the comparative evaluation of the thread serialization latency using different systems, we conducted a complementary evaluation that illustrates the relationship between the variation of the serialization latency and the variation of the performance overhead on a serializable thread. In order to achieve this goal, the two evaluations were performed with similar evaluation parameters. Indeed, the variation of the serialization latency was evaluated according to the number of frames on the thread's Java stack when the thread is serialized; and the variation of the performance overhead on the execution of a program was evaluated according to the number of frames that this execution pushes on the thread's stack.

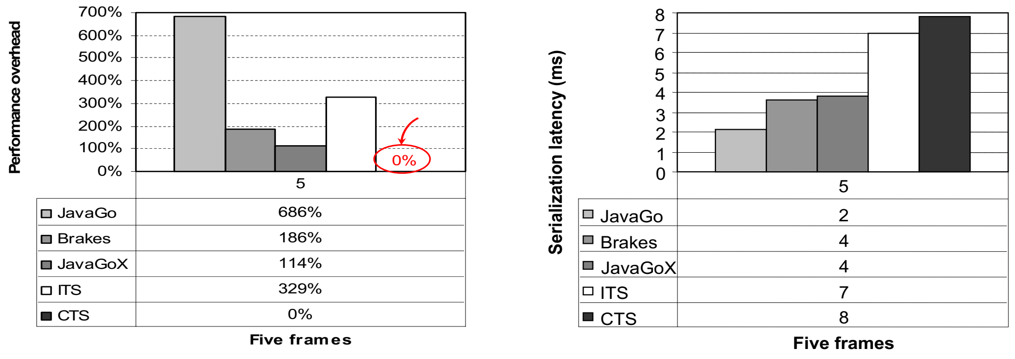
For measuring this performance overhead, we wrote a benchmark program that performs recursive calls which vary the number of frames.

Finally, the performance overhead and serialization latency were obtained as follow: -(

- With JavaGo, Brakes and JavaGoX, the benchmark programs were first written following the programming constraints of each system and then passed through the pre-processor before they were run on the standard JVM.
- For ITS and CTS, the benchmark programs were written using the thread serialization functions provided by these systems and run on the underlying extended JVM (i.e., respectively ITS and CTS).

Figure 26 and Figure 27 illustrate, for JavaGo ▢, Brakes ▨, JavaGoX ▩, ITS □ and CTS ■, the variation of, on the left-hand side, the performance overhead on a serializable thread, and on the right-hand side, the thread serialization latency. In Figure 26, the thread is serialized when it has five Java frames on its Java stack and in Figure 27, the thread is serialized when there are ten frames on its Java stack. The two figures show that:

- For the application-level systems (JavaGo, Brakes, JavaGoX), the overhead on the serialized thread and the serialization latency are inversely proportional. This is explained by the fact that the more processing is performed during the execution of the thread (overhead), the less processing is required at serialization time (latency). We also notice that with these benchmarks, Brakes and JavaGoX present similar behaviors due to their similar bytecode-level approach.
- The performance overhead of ITS lies between the overhead of the Java source-level system (JavaGo) and the overhead of the bytecode-level systems (Brakes, JavaGoX). But in this case, the inverse proportion with latency is not present: ITS presents a higher serialization latency, compared to JavaGo, Brakes and JavaGoX. This is certainly due to the fact that ITS captures the complete thread state (see section 2.2) and thus performs more processing at serialization time.
- CTS does not impose any performance overhead but it presents the highest thread serialization latency because everything is done at serialization time. Furthermore, it



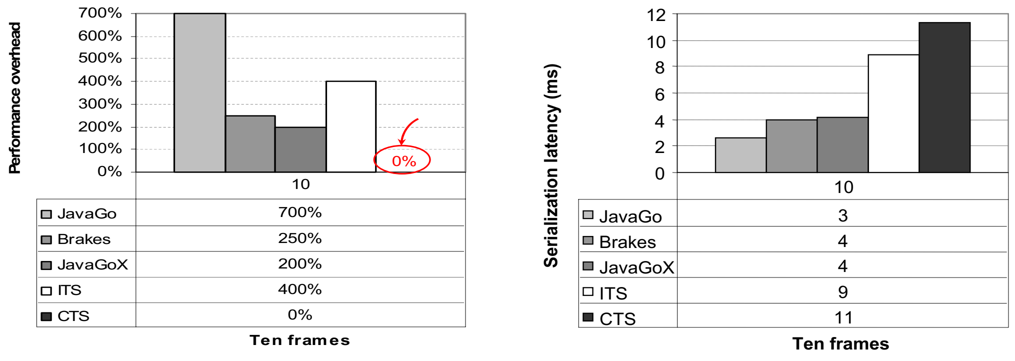
	5
□ JavaGo	686%
■ Brakes	186%
■ JavaGoX	114%
□ ITS	329%
■ CTS	0%

Five frames

	5
□ JavaGo	2
■ Brakes	4
■ JavaGoX	4
□ ITS	7
■ CTS	8

Five frames

Figure 26. Performance overhead vs. serialization latency, thread serialized with 5 frames (JIT disabled)



	10
□ JavaGo	700%
■ Brakes	250%
■ JavaGoX	200%
□ ITS	400%
■ CTS	0%

Ten frames

	10
□ JavaGo	3
■ Brakes	4
■ JavaGoX	4
□ ITS	9
■ CTS	11

Ten frames

Figure 27. Performance overhead vs. serialization latency, thread serialized with 10 frames (JIT disabled)

is important to notice that our current intermediary implementation of CTS does not yet include dynamic de-optimization. Thus, the final implementation would probably present a more significant latency due to the integration of dynamic de-optimization and re-optimization.

To summarize, our evaluation experiments show that:

- With JavaGo, Brakes and JavaGoX, a part of the cost of the provided functionality is added to the "normal" performance of the serialized thread, and the other part is put in the serialization latency. These two costs are inversely proportional.

- ITS behaves similarly to the above-mentioned systems, but it presents an important drawback: it is not compatible with JIT compilation. It is an interesting academic experiment but is probably not an effective solution for today's Java applications.
- With CTS, we show that it is possible to build a zero-overhead Java thread serialization facility. Indeed, apart from the thread serialization operation itself, the performance of the thread when it is running its own application code does not change. Here, the cancellation of the performance overhead is not magic; it is obtained by transferring all additional cost to the serialization latency.

As a result of this evaluation, we can identify two kinds of behaviors in the existing Java thread serialization systems:

(a) Serialization systems that provide a low serialization latency by adding an overhead on the serialized threads; this behavior is probably interesting for applications with frequent serialization operations, e.g., mobile agent based applications
(b) Serialization systems that do not modify the "normal" performance of the serialized threads and put all additional cost in the serialization operation; this kind of behavior targets applications where serialization is necessary but occurs rarely such as administration of distributed systems.

9. 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 and resume its execution, or be checkpointed on disk and then recovered.

Recently, several projects attempted to provide thread serialization in the Java environment; but the proposed solutions are limited in terms of performance: they impose a significant overhead on threads performance. The objective of this paper was twofold:

- to detail the implementation techniques that are necessary to build a zero-overhead Java thread serialization system (the CTS - Capture-time Thread Serialization - system), and
- to exhibit the benefits of this system in term of performance via a comparative evaluation of several Java thread serialization approaches.

We implemented the CTS 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.

The second result of our work comes from the performance evaluation that we reported on for the comparison of our thread serialization prototype with other prototypes implementing different approaches. This evaluation shows that:

- Java thread serialization based on an extension of the Java interpreter is non-compliant with JIT compilation, and is therefore not a realistic solution for today's Java applications.
- Application-level Java thread serialization is probably an interesting solution for applications that require frequent thread serialization operations, e.g., mobile agent based applications.
- CTS is an effective solution for applications where thread serialization is necessary but occurs rarely and for which the "normal" behavior of applications must be kept unchanged, e.g., distributed system administration.

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 [10], 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.

10. SOFTWARE AVAILABILITY

The CTS implementation of Java thread serialization is available from
http://sardes.inrialpes.fr/research/JavaThread

ACKNOWLEDGEMENTS

Java thread serialization is the brainchild of Xavier Rousset de Pina. We would like to thank Jacques

Mossire and Christian Jensen for their comments on earlier drafts of this paper. We are also grateful to Fabrice Dutron for his contributions to the CTS prototype, to Tatsurou Sekiguchi, Eddy Truyen, Takahiro Sakamoto and Torsten Illmann for providing useful information regarding their thread migration systems, and to Timothy Lindholm for his insightful information on JVM internals.

REFERENCES

1. Acharya A, Ranganathan M, Salz J. Sumatra: A Language for Resource-aware Mobile Programs. *2nd International Workshop on Mobile Object Systems (MOS'96)*; Linz, Austria, Jul. 1996. http://www.cs.umd.edu/∼acha/publications.html
2. Artsy Y, Finkel R. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer, 1989*; **22**(9).
3. Bagrodia R, Chu W W, Kleinrock L, Popek G. Vision, Issues and Architecture for Nomadic Computing. *IEEE Personal Communications Magazine, 1995*; **2**(6). http://pcl.cs.ucla.edu/papers/
4. Baker M, Carpenter B, Hoon Ko S, Li X. mpiJava: A Java interface to MPI. *1st UK Workshop on Java for High Performance Network Computing (Euro-Par'98)*; Southampton, UK, Sep. 1998. http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html
5. Bershad B N, Zekauskas M J, Sawdom W A. The Midway Distributed Shared Memory System. *IEEE International Computer Conference (COMPCON'93)*; Feb. 1993. http://www.cs.washington.edu/homes/bershad/Papers/
6. Bouchenak S, Hagimont D. Pickling Threads State in the Java System. *Technology of Object-Oriented Languages and Systems Europe (TOOLS Europe'2000)*; Mont-Saint-Michel / Saint-Malo, France, Jun. 2000. http://sardes.inrialpes.fr/∼bouchena/publications/
7. Bouchenak S. Making Java Applications Mobile or Persistent. *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*; San Antonio, TX, USA, Jan. 2001. http://sardes.inrialpes.fr/∼bouchena/publications/
8. Bouchenak S. *Mobility and Persistence of Applications in the Java Environment*. Ph. D. Thesis, French National Polytechnic Institute of Grenoble (INPG), France, Oct. 2001. http://sardes.inrialpes.fr/∼bouchena/publications/
9. Bouchenak S, Hagimont D. Zero Overhead Java Thread Migration. *INRIA Technical Report No. TR-0261*; France, May 2002. http://sardes.inrialpes.fr/∼bouchena/publications/
10. Cardinale Y, Hernández E. Checkpointing Facility on a Metasystem. *European Conference on Parallel Computing (Euro-Par'2001)*; Manchester, UK, Jan. 2001. http://suma.ldc.usb.ve/
11. Chess D, Harrison C, Kershenbaum A. Mobile Agents: Are They a Good Idea? *T.J. Watson Research Center White Paper, IBM Research Division*; Mar. 1995. http://www.research.ibm.com/iagents/publications.html
12. Czajkowski G. Application Isolation in the Java Virtual Machine. *17th Annual ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA'00)*; Minneapolis, MN, USA, Oct. 2000.
13. Deconinck G, Vounckx J, Cuyvers R, Lauwereins R. Survey of Checkpointing and Rollback Techniques. *Technical Report, Katholieke Universiteit Leuven, Belgium*; Jun. 1993.
14. Dornan A. *The Essential Guide to Wireless Communications Applications, From Cellular Systems to WAP and M-Commerce*. Prentice Hall, Dec. 2000.
15. Douglis F, Ousterhout J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience, 1991*; **21**(8). http://www.douglis.org/fred
16. Douglis F, Marsh B. The Workstation as a Waystation: Integrating Mobility into Computing Environment. *3rd Workshop on Workstation Operating System (IEEE)*; Key Biscayne, Florida, USA, Apr. 1992. http://www.douglis.org/fred
17. Engel J. *Programming for the Java Virtual Machine*. Addison Wesley, 1999.
18. Fuggetta A, Picco G P, Vigna G. Understanding Code Mobility. *IEEE Transactions on Software Engineering, 1998*; **24**(5). http://www.cs.ucsb.edu/∼vigna/listpub.html
19. 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. http://www.informatik.tu-darmstadt.de/∼fuenf
20. Chou T C K, Abraham J A. Load Redistribution under Failure in Distributed Systems. *IEEE Transactions on Computers, 1983*; **32**(9).

21. Gosling J, McGilton H. The Java Language Environment. *Sun Microsystems White Paper*; May 1996. http://java.sun.com/docs/white
22. Hagimont D, Boyer F. A Configurable RMI Mechanism for Sharing Distributed Java Objects. *IEEE Internet Computing, 2001*; **5**(1). http://sardes.inrialpes.fr/∼hagimont/publications/
23. Hernández E, Cardinale Y, Figueira C, Teruel A. Suma: A Scientific Meta-computer. *Parallel Computing (ParCo'99)*; Delft, The Netherlands, Aug. 1999. http://suma.ldc.usb.ve
24. Hofmeister C, Purtilo J M. Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement. *13th International Conference on Distributed Computing Systems (ICDCS'93)*; Pittsburgh, PA, USA, May 1993.
25. 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, CA, USA, Jun. 1992. http://www.cs.ucsb.edu/labs/oocsb/papers.shtml
26. Huang Y, Kintala C, Wang Y M. Software Tools and Libraries for Fault-Tolerance. *IEEE Technical Committee on Operating Systems and Application Environments (TCOS), 1995*; **7**(4). http://www.tcos.org/Bulletin/winter95/winter95.html
27. IBM Tokyo Research Labs. Aglets Workbench: Programming Mobile Agents in Java, 1996. http://www.trl.ibm.co.jp/aglets
28. Illmann T, Krueger T, Kargl F, Weber M. Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture. *5th IEEE Int. Conference on Mobile Agents (MA'2001)*; Atlanta, GA, USA, Dec. 2001. http://cia.informatik.uni-ulm.de/papers
29. Izatt M, Chan P, Brecht T. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. *Concurrency: Parctice and Experience, 2000*; **12**(8).
30. Jul E, Levy H, Hutchinson N, Black A. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Science, 1988*; **6**(1). http://ftp.diku.dk/pub/dists/emerald/
31. Lindholm T, Yellin F. *The Java Virtual Machine Specification (2nd edn)*. Addison Wesley, 1999. http://java.sun.com/docs/books/vmspec
32. Litzkow M J, Solomo M. Supporting Checkpointing and Process Migration outside the UNIX Kernel. *USENIX Winter Conference*; San Francisco, CA, USA, Jan. 1992. http://www.cs.wisc.edu/condor/publications.html
33. Mandelbrot B. *Les Objets fractals : forme, hasard et dimension*. Flammarion, 1975.
34. Meloan S. The Java HotSpot Performance Engine: An In-Depth Look. Sun Microsystems, Jun. 1999. http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/
35. Milojičić D, Douglis F, Wheeler R. *Mobility: Processes, Computers and Agents*. Addison Wesley, Feb. 1999.
36. Mostefaoui A, Raynal M. Efficient Message Logging for Uncoordinated Checkpointing Protocols. *IRISA Technical Report No. RR-2972*; France, Jun. 1996. http://www.inria.fr/rrrt/rr-2972.html
37. Netzer R, Helary J M, Mostefaoui A, Raynal M. Communication-Based Prevention of Useless Checkpoints in Distributed Computation. *Distributed Computing, 2000*; **13**(1).
38. Nichols D A. Using Idle Workstations in a Shared Computing Environment. *11th Symposium on Operating Systems Principles (SOSP'11)*; Austin, TX, USA, Nov. 1987.
39. Oueichek I. *Conception et Ralisation d'un Noyau d'Administration pour un Systme Rparti Objets Persistants*. Ph. D. Thesis, INPG (Institut National Polytechnique de Grenoble), France, Oct. 1996.
40. Peine H, Stolpmann T. The Architecture of the Ara Platform for Mobile Agents. *1st International Workshop on Mobile Agents (MA'97)*; Berlin, Germany, Apr. 1997. http://wwwagss.informatik.uni-kl.de/Projekte/Ara/
41. 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. http://www.cs.dartmouth.edu/MA2001/
42. Sakamoto T, Sekiguchi T, Yonezawa A. Bytecode Transformation for Portable Thread Migration in Java. *4th International Symposium on Mobile Agents 2000 (MA'2000)*; Zürich, Switzerland, Sep. 2000. http://web.yl.is.s.u-tokyo.ac.jp/∼takas/
43. Sekiguchi T, Masuhara H, Yonezawa A. A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation. *3rd International Conference on Coordination Models and Languages*; Amsterdam, The Netherlands, Apr. 1999. http://liang.peng.free.fr/people-mobile.html
44. Suezawa T. Persistent Execution State of a Java Virtual Machine. *ACM Java Grande 2000 Conference*; San Francisco, CA, USA, Jun. 2000. http://www.ifi.unizh.ch/staff/suezawa/
45. Sun Microsystems. Java JIT Compiler Overview. http://www.sun.com/solaris/jit/ [October 2002]

46. Sun Microsystems. Improving Serialization Performance with Externalizable. http://developer.java.sun.com/developer/TechTips/2000/tt0425.html [November 2002]
47. Truyen E, Robben B, Vanhaute B, Coninx T, Joosen W, Verbaeten P. Portable Support for Transparent Thread Migration in Java. *4th International Symposium on Mobile Agents 2000 (MA'2000)*; Zürich, Switzerland, Sep. 2000. http://www.cs.kuleuven.ac.be/∼eddy/research.html
48. Wojcik Z M, Wojcik B E. Optimal Algorithm for Real-Time Fault Tolerant Distributed Processing Using Checkpoints. *Informatica, 1995*; **19**(1). http://ai.ijs.si/informatica/
49. Zayas E R. Attacking the Process Migration Bottleneck. *11th ACM Symposium on Operating System Principles (SOSP'11)*; Austin, TX, USA, Nov. 1987.
50. Zhu W, Wang C-L, Lau F C M. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. *IEEE Fourth International Conference on Cluster Computing (CLUSTER 2002)*; Chicago, USA, Sep. 2002. http://www.srg.csis.hku.hk/homepage/srg2002/jessica2.htm