A Serialisation Based Approach for Processes Strong Mobility

Soumaya Marzouk, Maher Ben Jemaa, and Mohamed Jmaiel

ReDCAD Laboratory
National School of Engineers of Sfax
BPW 3038 Sfax Tunisia
Soumarzouk@yahoo.fr, Maher.benjemaa@enis.rnu.tn, Mohamed.Jmaiel@enis.rnu.tn

Abstract. We present in this paper a generic approach for process transformation into strong mobile entity. Our approach is based on processes *Serialisation* using source code transformation, which generates the source code of a strong mobile process. Our approach is suitable for transforming distributed applications into mobile applications where every process can be migrated independently any time. We applied our approach to Java Thread by designing a grammar describing the generated mobile process code. The evaluation results of generated mobile Threads shows good performances.
key words : Strong Mobility, Source code transformation, Serialisation, Distributed systems, Java Thread.

1 Introduction

Process strong mobility represents an efficient mechanism for solving many problems like fault tolerance [GBB05] and load balancing [BSA05]. Moreover, process strong mobility contributes for managing pair to pair and grid based systems [CB06],[GYHP06].
In fact, process strong mobility allows the transfer of an executing process from a source site to a distant site, where it resumes its execution starting from the interruption point. Thus, strong mobility requires the capture of the process execution state which is a complicated task since programming languages do not allow direct access to the process execution stack.
Generally, there is a trade off between efficacity and portability in most works dealing with strong mobility. Indeed, solutions suggested to solve this problem are either non portable solutions but offering good performances like those which are implemented on operating system level [BSA05],[BHKP04],[DO91], and solutions operating on virtual machine level [BHKP04],[SBB+00],[ZWL02], or more portable solutions but not very powerful like those which operate on compiled code level [GBB05],[TRV+00],[SSY01], or solutions which operate on process source code level [BN01],[CWHB03],[Fun98],[CLG05].
In this paper, we present a solution for process strong mobility which is: generic, user transparent, offering a great portability, and rather powerful. Our solution is based on process serialisation. Object Serialisation consists in saving current

values of its attributes. In case of process serialisation, it consists in saving the process execution context. This makes it possible to have an image reflecting the instantaneous process execution state to resume the execution later by carrying out a deserialisation. Our solution consists in simulating the process execution context with an artificial stack saving a portable image of its execution state and which does not depend on the used programming language. This solution is made by a syntactic transformation of the source code, thus ensuring, the maintenance of the process execution state, while preserving its original semantics. Therefore, the process migration consists in (1) serialising the process on the source site, (2) transferring the serialized process towards the new execution site, (3) deserialising the process and resuming its execution. We applied our transformation approach for Java Thread by implementing a precompiler which transforms a Java Thread into a Mobile Thread. This transformation ensures that restarting Thread after migration is enough to continue its execution starting from the interruption point.
Our approach is distinguished from others in the way that it is completely transparent since it does not need any changes on the original process code, and programmer has not to fix interruption points prealably in the process code. In addition, our approach keeps the instantaneous process state value, so, there is no need to do periodic checkpointing and rollback to resume a suspended process. Moreover, the process migration is dynamic and can be repeated an arbitrary time. Our evaluation tests show that our approach minimizes the execution time overhead due to codes additions, and keeps a proportionally acceptable execution time compared to the initial one. Moreover, the generated Thread is totally portable, thanks to its artificial execution context structure, and it preserves the semantic of the original Thread.
This paper is organized as follows. We will present in the second section the related works. Next in the third section, we present a description of our code transformation approach. Then, in the fourth section, we will present the evaluation results of our approach applied to Java Threads. Finally, we conclude and present perspectives of our works.

2 Related Work

Many works dealt with the process strong mobility problem. We classify this works in four classes according to their action's level.
First, works which operate on **Operating System level** [BSA05], [DO91]. These techniques are characterized with a short response delay since all treatments are integrated into the operating system functionalities. However, these approaches have the disadvantage of forcing all participating nodes to use the particular operating system. Therefore, the use of this type of solutions can be done only in a network with homogeneous operating systems. Thus, such a solution will not be applicable on the grid, for example.
Second class of work act on **Virtual Machine level** [BHKP04], [SBB+00], [ZWL02]. Generally, these solutions were particularly proposed for the Java lan-

guage. They consist in JVM extension to support process strong mobility. These solutions grant the independence with the operating system layer. However, they reduce the application portability since they can be executed only on the extended JVM. This problem is not major if the user work on a local area network but it becomes significant if he wants to distribute the execution over the Internet.
Other solutions operate on **Compiled Code level** [GBB05], [TRV+00], [SSY01], [SSY00]. Most of them choose the Java language as target and transform process byte codes. These solutions increase the portability of a mobile application since it is independent from the operation system and the JVM. However, such techniques do not allow forced migration by external Thread but only the Thread itself can initiate its migration. Thus, this solution is not adapted to carry out load balancing or fault tolerance strategies.
Finally, other solutions act on **Source Code level** [BN01], [CWHB03], [CB06], [DR98], [GYHP06],[Fun98], [CLG05], [SMY99]. This approach has the advantage, to be independent of the operating system, to not modify neither the interpreter, nor the programming language. Thus it is more portable than the first three solutions. However, many works adopting this kind of solution reduce the application portability. In fact, many works use a specific platform [CWHB03], [CB06], or impose the use of a procedural language [BN01], or MPI based program [GYHP06], [CLG05]. Others do not allow a forced migration made by an external application [Fun98], [SMY99], or specifies static checkpoints in the process source code [DR98].

	References	**Specific Platform**	**Forced Migration**	**Language**
Operating System	[DO91][BSA05]	x	v	x
Virtual Machine	[SBB+00][BHKP04]	x	v	Java
	[ZWL02]	Multi Agent	v	
Compiled Code	[GBB05][TRV+00] [SSY00][SSY01]	x	x	Java
Source Code	[DR98]	x	Static checkpoint	C++
	[GYHP06][CLG05]	x	Static checkpoint	MPI
	[Fun98][SMY99]	x	x	Java
	[BN01]	x	v	procedural Language X-Klaim
	[CWHB03][CB06]	Aglet	v	C++

Table 1. Classification of Work Treating Strong Mobility

In *Table 1*[1] we summarize related works dealing with strong mobility. We consider in this classification many criterias like action level, use of a specific platform, the initiator of migration, etc.
Our approach can be classified under the source code modification class, but differs from the others in that it is a generic approach since it is independent of the programming language. It provides a portable solution since it does not depend on a specific platform. In addition, it is transparent because no manual changes must be done to the original process code.
We note that [CWHB03] is the most close solution to the present paper, but our work is distinguished by its portability. Indeed, our pre-processor is needed only at compilation time, and it doesn't introduce any restriction on the executing site configuration. However, in [CWHB03] the mobile agent can be executed only on a site lodging the agent platform, which restrict the mobility and do not motivate the use of this solution in a grid environment.
Moreover, our solution offer forced migration which is a very important characteristic of a process strong mobility approach. In fact, non forced migration signifies that only the process it self can initiate the mobility operation, which implies that migration is pre-programmed in the process code. For example, in a load balancing system, process migration is initiated when the execution host becomes overloaded, which cannot been known in advance. In this case, migration call cannot be written explicitly in the process code but must be initiated instantly by an extern application which is in this case the load balancing system.

3 Transformation Approach

Our approach consists in transforming a process into a strongly mobile entity. This transformation must guarantee

- Persistence: Allowing to save / restore the process execution state at any execution time,
- Repetitivity: The possibility of repeating the migration operation several times during process execution,
- Transparency: The original process code does not need any changes
- Portability: The generated mobile process can be run on any machine, whatever is its software or hardware configuration,
- Genericity: independency of the programming language.

Actually, a process does not have the persistence character (it is not serialisable). To make it serialisable, we will use a source code precompiler which transforms a traditional process into a strong mobile one.
Our approach consists in designing transformation rules of process source code written in an object-oriented language while providing several functionalities. First, generated code simulates the instantaneous process execution state by an

[1] v : supported; x : unsupported

artificial structure. Second, it updates this structure while the process execution progresses. Finally, it ensures the resumption of the process execution while preserving its execution semantics.
In the following, we will present details of our process transformation approach including modelling process execution context, capturing and re-establishing process state and transformation rules of process code instructions.

3.1 Capturing and Reestablishing Process State

To ensure process strong mobility, we propose two mechanisms: Capturing and Re-establishing process state mechanisms.
Capturing process state mechanism serves to store an instantaneous image of process execution progress. It requires modelling and updating process execution context. Thus, we propose to add to the process source code an attribute simulating the process execution progress, and instructions updating the process execution state.
Explicitly, to model process execution state we propose a generic model called process artificial execution stack. This latter includes the execution progress state (method entry point) of each called method. Process artificial execution stack is build by pushing a method entry point for each called method. In fact, an entry point is an object storing method execution progress state. This object will include attributes saving the current values of method input data, local variables, and the position of the next instruction to be carried out by the method. Since the number and types of these attributes depend on methods data, we generate, for each called method (process methods or object methods), a class (method model class) having as attributes the method input data, local variables, and the position of the next instruction.
In addition, the capturing mechanism includes updating process execution state. Therefore, we propose to add, for each called method, instructions updating the method entry point with current method data values. Explicitly, we propose to add at the beginning of each called method (1) instructions which instantiate the method model class generated to create an entry point corresponding to the method call, (2) instructions which pop the entry point on the process artificial stack, and (3) instructions initializing the entry point attributes corresponding to the method input data and local method variables by their initial values. In addition, in the end of each method, it is necessary to add an instruction which pop the entry point from the artificial execution stack. Moreover, updating process execution state requires method instructions transformation which will be detailed in the next section. Thus, the current state of each method is stored in the artificial execution stack, so the capture of the current process state consists in suspending the process execution and serialising it.
The second mechanism involved in process strong mobility is Reestablishing process execution state mechanism. It serves to resume process execution after migration. Thus, reestablishing process execution state requires integrating the process execution state captured by the first mechanism in the new process execution instance, and resuming process execution starting from the interruption

point.
In order to integrate captured state in the new process execution, each method has to reference its captured entry point. In fact, we propose to add, in the beginning of each called method, instructions which refer to the captured method entry point if it is the reestablishing step.
In addition, reestablishing process state mechanism must ensure that execution resumption starts always from the interruption point. Thus, we modified process code by adding instructions ensuring that each method execution restart from the instruction having the position of the next instruction stored in method entry point attributes. Doing so, we propose to supervise every method instruction execution by a test on its position ensuring that the executing instruction is always the one which has the position of the next instruction.

3.2 Code Transformation

In order to achieve the process transformation into a strongly mobile entity, we define transformation rules which we will apply to code instructions. To do this, we classify code instructions into three categories:

- Simple instructions: they are elementary instructions, which include assignments, inputs/outputs instructions, calls of method belonging to the process, etc.
- Composed instructions: they are blocks of code containing loops or control structures.

In the following subsections, we will define for each type of instruction, corresponding transformation rules. We describe also code transformations of Shared object(remote object used by many process) and we propose optimizations for our transformation rules.

Transformation of Elementary Instructions Simple instructions transformation serve to ensure execution state updating and execution resumption while preserving execution semantics. Process execution state updating is ensured by replacing all occurrences of local variables and input data of the method with references to the attribute of the corresponding entry point. Moreover, after each instruction execution, the value of the next instruction position to be executed must be updated. Therefore, after each instruction of the transformed code, we propose to increment the position value of the next code instruction to be carried out. For example, if the following instruction belongs to a method called `m_1` : `x = y ;`
Where `x` is a local variable and `y` a method input data, it will be then replaced with:

```
Entry_Point_m_1.x = Entry_Point_m_1.y ;
Entry_Point_m_1.position++ ;
```

In addition, to ensure resuming process execution after migration, every code instruction must be supervised by testing the value of its position. Consequently, the instruction `x = y ;` will be replaced with:

```
if(Entry_Point_m_1.position==current_position) {
       Entry_Point_m_1.x = Entry_Point_m_1.y ;
       Entry_Point_m_1.position++ ;  }
```

Besides, we must be sure that the execution interruption will not take place after the instruction execution and before the position update. Therefore, we propose to consider the transformation result of an instruction as an atomic operation which can't be interrupted by serialisation. The transformation of the instruction: `x = y ;` will be as follows :

```
Lock_Serialisation();
if(Entry_Point_m_1.position == current_position) {
       Entry_Point_m_1.x = Entry_Point_m_1.y ;
       Entry_Point_m_1.position++ ;
} Unlock_Serialisation();
```

Thus, these transformations applied for simple instructions, guarantee process execution state updating, as well as reestablishing after migration, while preserving its execution semantics.

Transformation of code with Loops and controls structures

```
while ((pc >= inPc(Bloc_transformed)) && (pc <= outPc(Bloc_transformed))) {
        if (pc == inPc(Bloc_transformed) && !cond) {
         // condition not verified
                pc = outPc(Bloc_transformed)+1;
                break;
        }
        Bloc_transformed;
        if (pc == outPc(Bloc_transformed))
                pc = inPc(Bloc_transformed);
}
```

Fig. 1. Transformation of while loop.

The difficulty which arises for the case of loops and control structures is the update of position of next instruction to be carried out.
In fact, the code transformation has to preserve the execution semantics, what-

ever the interruption position is, during loop or control structure execution. Next, we will study the case of the structure while (while(cond) Bloc;) *"Fig1"* and if-else (if(cond) Bloc1; else Bloc2;) *"Fig2"*.

```
if (((pc >= inPc(Bloc1_transformed)) && (pc <= outPc(Bloc1_transformed))) ||
(pc == Pc(If) && cond)) {
        Bloc1_transformed;
        if (pc == outPc(Bloc1_transformed)) {
        // end of the block if:  jump the block else.
                pc = outPc(Bloc2_transformed) + 1;
        }
} // if the condition is not verified: enter to the block else
else {
         if (pc== inPc(thisIf))
                  pc = outPc(Bloc1_transformed) + 1;
}
if ((pc>= inPc(Bloc2_transformé)) && (pc <= outPc(Bloc2_transformed))) {
        Bloc2_transformed;
}
```

Fig. 2. Transformation of if - else structure.

with:

- Bloc1_transformed represent the transformation result of Bloc1.
- outPC(Bloc_transformed) represent the first position in Bloc_transformed.
- inPc(Bloc_transformed)represent the last position in Bloc_transformed
- Pc(if) represent the position of the if instruction. Indeed, we attribute to the if instruction a position to ensure that the if condition will be evaluated only once.

The code given above preserves the initial semantics whatever the execution stop point in this code.

Transformation of Shared Objects In this step, we extend our transformation to support dependent process. Indeed, we propose to transform distributed

applications including dependent process using shared object on mobile application where every component can be moved from a site to another at any execution moment. All transformations presented above remain valid including transformation of shared object methods. Nerveless, if a process migrates while executing shared object method, the execution coherence may be lost. Thus, we propose to add an artificial lock to a shared object which interdicts the execution of a method belonging to a shared object used by a migrating process. That is, if a process migrate while executing a shared object method, it must lock the shared object until resuming the interrupted method. In addition, every process trying to execute a shared object method must verify if this object is unlocked before calling the method. Thus, every call of a shared object method in the process code must be supervised by a test on the shared object artificial lock.

Optimizing the transformed code In order to optimize the transformed code, we propose to affect a position number for blocks containing more than one instruction. Thus, an instructions block, with the update instruction of its corresponding position, will form an atomic operation during which a serialisation is not authorized. This makes it possible to reduce the size of the code added compared to the initial code, and consequently to reduce the execution time of the transformed process. This modification requires several rules for the choice of blocks.
First, blocks should not contain the headings of controls structures or of loops of the original code. This case can generate compilation errors, since it causes crossed loops.
Second, the method call must be an elementary block or in extreme cases, must be at the beginning of a block. Otherwise, if a serialisation starts during the method execution, block instructions which are before the method call will be re-executed after migration.
Third, the block size must be quite selected not to be, neither too large causing the delay of the serialisation operation, nor too small causing the increase of the size of the generated code compared to the original code.
We propose also another optimization, which consists in not applying transformations concerning loops and control structures in all cases. Indeed, if the code carried out by a loop or a control structure is simple (without imbricated structures, without call of object methods), we propose to assign to this structure only one position number, and thus to authorize the serialisation only at the end of the execution of all the structure code. For the case of loops, this solution remains valid if the total number of instructions to be carried out by the loop is not very large. Otherwise, in general case, we propose to allow the serialisation at the end of each iteration.

4 Performance Evaluation

In order to evaluate performances of the generated mobile process, we apply our transformation rules to Java Threads. Thus, we designed a grammar describing

the Java syntax of the mobile Thread transformed code, and we implement a source code transformer which takes a java Thread as entry and generates the equivalent mobile Thread.
In order to evaluate our solution performance, we present the evaluation results of our transformed process execution times, compared to the original processes execution time. We used a mobile computer equivalent processor Centrino 1,7 GHz and having a 1Go size of RAM.
We evaluate the execution time increase, due to the code portions added by our transformer. Moreover, the evaluation of our solution will be based on several criteria:

- Criteria related to the original process: code complexity, code size.
- Criteria related to the transformation: maximum size of the elementary instruction block.
- Criteria related to the execution: data size.

We can notice that the transformation overhead is relatively big for an execution with small data size *"Fig3"* This can be explained by the fact that the added code can be classified in two classes. First, the initialization code which has a constant size and which is carried out only once at the beginning of each method. Second, the updating code which has a variable size according to the original code size, and which can be carried out several times, according to the size of the input data.

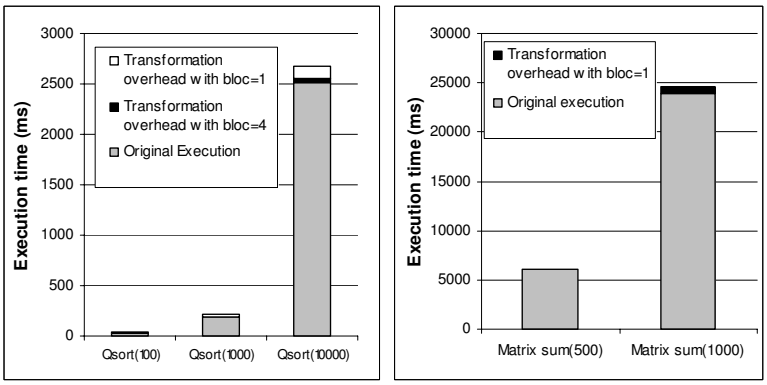

Fig. 3. Transformed Thread Execution time compared to the original Thread execution time.

Thus for small size data or for processes having small methods size, and for the same complexity, the overhead of the transformed process execution time compared to that of the original process is proportionally big, since the initialization code is of constant size. This also explains, the overhead increase, for

the same code, when the data size increases *"Fig3"*. Indeed, since the data size increases, the iteration number also increases, and consequently the iteration execution number of added code increases too. Moreover, the increase in the maximum block size of atomic instructions causes the decreases of the transformation overhead. This phenomenon happen because the atomic instructions blocks number decrease induced that the added code became smaller than the original one.

We also stress that the overhead is increasingly big, when the number of overlapping loops increases, and especially when the loop code is of small size, a typical example is the multiplication of two matrices. In this case, the added code size becomes large compared to the original code, and considering the great iteration number and the code complexity, the transformed code execution becomes very heavy. To cure this type of behaviour, we presented an optimization in section 3.2.4. The results relating to this optimization are presented in *"Fig4"*.

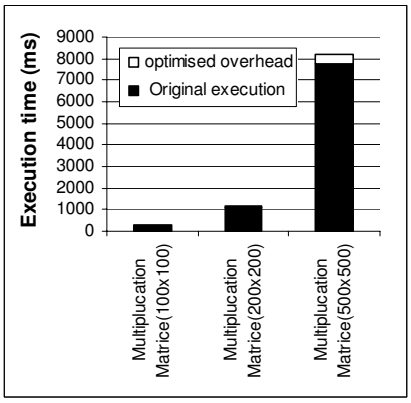

Fig. 4. Execution time of Mobile Thread having an Optimized code.

Following, we aim to evaluate the serialisation/deserialisation operation. Thus, we will use Matrice Multiplication 500X500 Thread, without taking into account the process transfer cost, which depends on the network conditions. Presented results in *"Fig5"* correspond to the Thread execution time stopped at the instant "interruption time", serialized, deserialized, and resumed on the same execution site. These results show that the serialisation/deserialisation operation of process has a weak cost. Consequently, the integration of the execution context operation which requires a partial re-execution of the process code is not an expensive operation.

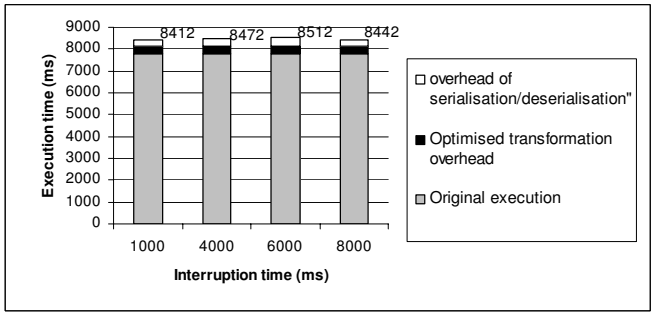

Fig. 5. Execution time of Multiplication matrix Thread (500X500) with serialisation / deserialisation operation

Next, we aim to evaluate the cost of migration of a process belonging to a distributed application. In this context, we use a producer/consumer application. This application involves a producer mobile process, a consumer mobile process and a Remote Object representing the Buffer.

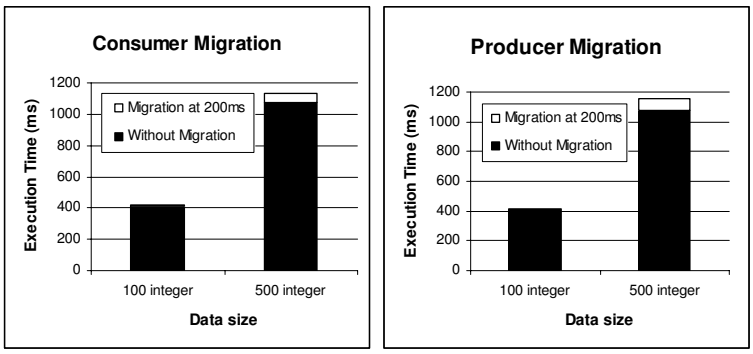

Fig. 6. Overhead introduced by process migration of producer/consumer application

In *"Fig.6"* we represent the overhead caused by the migration of the producer process and the consumer process at different execution moment and for different data size. We notice that the overhead introduced by the migration of producer or consumer process is very small.

5 Conclusion

In this paper, we proposed a generic solution for the processes strong mobility, with great portability, completely transparent and rather powerful. Indeed, our

approach consists in transforming process into a serialisable object. Throughout its execution, our mobile process could migrate several times from a site to another, at any execution time, without losing its execution state, nor the semantics of the original process. Our approach is novel in that it was designed to be completely transparent to the programmer, requiring no changes to the original application code. Moreover, our approach makes it possible to generate completely portable mobile processes. Indeed, our approach is independent of the used platform and there are no software or material constraints on the migration participating sites. In addition, our approach makes it possible to force the process migration starting from an external application, which allows its use to implement load balancing, fault-tolerance, peer to peer or grid based systems. We apply our transformation approach for Java Thread. Indeed, to achieve the process migration, it suffies to apply the transformation to the original process code, to compile the generated classes and to launch the Mobile Thread execution from any host lodging the JVM. Thread can be stopped and migrated towards any host lodging the JVM, at any moment of its execution and an ar number of times for the same execution.
Our work perspectives consist in providing solutions to the problem of resource sharing (file, socket) between mobile processes. Indeed until this stage, the Thread migration using a shared resource does not preserve execution semantics. We aims also to validate our code transformer, in order to affirm that the transformation is purely syntactic and that the mobile process always preserves the original process semantics. Another prospect consists in using this approach of mobility for the implementation of a load balancing system or fault tolerant grid based applications. Doing so, an execution environment should be developed.

References

[BHKP04] S. Bouchenak, D. Hagimont, S. Krakowiak, and N. Palma. Experiences implementing efficient java thread serialization, 2004.

[BN01] Lorenzo Bettini and Rocco De Nicola. Translating strong mobility into weak mobility. *Lecture Notes in Computer Science*, 2240:182–197, 2001.

[BSA05] A. Barak, A. Shiloh, and L. Amar. An organizational grid of federated mosix clusters. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 1*, pages 350–357, Washington, DC, USA, 2005. IEEE Computer Society.

[CB06] Arjav J. Chakravarti and Gerald Baumgartner. Self-organizing scheduling on the organic grid. *International Journal of High Performance Computing Applications*, 20(1):115–130, 2006.

[CLG05] Jiannong Cao, Yinghao Li, and Minyi Guo. Process migration for mpi applications based on coordinated checkpoint. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 306–312, Washington, DC, USA, 2005. IEEE Computer Society.

[CWHB03] Arjav J. Chakravarti, Xiaojin Wang, Jason O. Hallstrom, and Gerald Baumgartner. Implementation of strong mobility for multi-threaded agents in java. *icpp*, 00:321, 2003.

[DO91] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[DR98] B. Dimitrov and V. Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):459–??, 1998.

[Fun98] Stefan Funfrocken. Transparent migration of java-based mobile agents. In *Mobile Agents*, pages 26–37, 1998.

[GBB05] Pawel Garbacki, Bartosz Biskupski, and Henri E. Bal. Transparent fault tolerance for grid applications. In *EGC*, pages 671–680, 2005.

[GYHP06] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 471–478, Washington, DC, USA, 2006. IEEE Computer Society.

[SBB+00] Niranjan Suri, Jeffrey Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Renia Jeffers. Strong mobility and fine-grained resource control in nomads. In *ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 2–15, London, UK, 2000. Springer-Verlag.

[SMY99] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *Coordination Models and Languages*, pages 211–226, 1999.

[SSY00] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in java. In *ASA/MA*, pages 16–28, 2000.

[SSY01] Tatsurou Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling. *Lecture Notes in Computer Science*, 2022:217+, 2001.

[TRV+00] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in java. In *ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 29–43, London, UK, 2000. Springer-Verlag.

[ZWL02] Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.