Live and Heterogeneous Migration of Execution Environments

Nicolas Geoffray, Gaël Thomas, and Bertil Folliot

Laboratoire d'Informatique de Paris 6
8 rue du Capitaine Scott, 75015 Paris France
firstname.lastname@lip6.fr

Abstract. Application migration and heterogeneity are inherent issues of pervasive systems. Each implementation of a pervasive system must provide its own migration framework which hides heterogeneity of the different resources. This leads to the development of many frameworks that perform the same functionality. We propose a minimal execution environment, the micro virtual machine, that factorizes process migration implementation and offers heterogeneity, transparency and performance. Systems implemented on top of this micro virtual machine, such as our own Java virtual machine, will therefore automatically inherit process migration capabilities.

1 Introduction

Process migration is the act of transferring a process from one system to another. When the two systems differ in hardware or operating system, migration is said to be heterogeneous. The main goals of process migration include [14]: accessing more processing power, exploitation of resource locality, resource sharing, fault resilience, system administration, and recently pervasive computing.

Most of the existing process migration implementations lack in heterogeneity. The base requirement of these systems is an homogeneous environment, which is unconceivable for a pervasive system. Furthermore, the use of static languages such as C or C++, and dynamic libraries of the operating system render the implementation of process migration difficult, allthough some seem to be operational but limited [4,25].

With the arrival of virtual machines such as the Java Virtual Machine (JVM) implementation of process migration in heterogeneous environment became easier. A JVM hides to applications the underlying operating system and hardware. Therefore, the execution environment of a process can be represented in a portable manner. Many different approaches have been developed for migrating threads in the JVM [21,28,18,3,32]. We think these approaches are in the right direction to achieve process migration in heterogeneous environments, but they by essence lack in genericity as they only target the JVM and the Java language.

In this paper, we propose a new execution environment that will enable process migration for every bytecode-targeted languages. This environment is a minimal virtual machine, called the *micro-virtual machine*, that contains the base of a virtual machine architecture. Virtual machines like JVM or .Net can thereafter be implemented on top of the micro-virtual machine and automatically benefit from process migration. We intend to apply process migration to our existing JVM implementation, JnJVM. The process migration system is an extension of the micro-virtual machine. The contributions of this paper are:

- A proposal for a minimal execution environment with process migration facilities. The environment is minimal enough to allow process migration for higher level virtual machines,
- An explanation of a standard algorithm for process migration,
- A reflexion on operating system migration on heterogeneous environments, when the operating system is implemented within a virtual machine.

Section 2 presents background and related work in the field of process migration. Section 3 presents the architecture of our minimal virtual machine and Section 4 explains how thread migration will be achieved in the micro-virtual machine. Section 5 presents perspectives, including our current ideas on how migrating an operating system provided that it is executed on a virtual machine. Operating system migration is a trend in ubiquitous computing, as it will allow users to migrate their entire working environment. Section 6 conludes the paper.

2 Related Work

Process migration is a popular field in systems research. Many implementations have been proposed. They differ in heterogeneity, performance, transparency and reusability. In this section, we give an overview of existing systems and compare them with respect to these four characteristics.

2.1 Binary Process Migration

In homogeneous systems: Binary process migration upon operating systems has long considered homogeneity to be the base requirement. Milojicic *et al.* [14] give a panel of process migration in operating systems. Process migration can be implemented either in the kernel, in system libraries (user-space), or in end-user applications. The three levels differ in complexity, transparency, performance and reusability. Kernel-space implementations typically regroup all these characteristics. User-space implementations are simpler than kernel-space implementations, but suffers from performance and transparency. Application-level implementations suffer from reusability, because modification of the application is most of the time required. Examples of kernel-level process migration are Locus [17], the V Kernel [26] or Amoeba [24]. Implementations in user-space include Condor [13] or MPVM [2]. Finally, Freedman [7] has implemented process migration in end-user applications.

In heterogeneous systems: Few attempts on process migration on heterogeneous systems have been accomplished. Most significant systems are Tui [23], SNOW [25] and recently MigThread [4].

The Tui system provides heterogeneous process migration by modifying the ACK (Amsterdam Compiler Kit). It generates an executable with process migration possibilities.

SNOW (Scalable Network Of Workstation) and MigThread consist of a pre-processor and a runtime system. The pre-processor performs a source-to-source transformation to provide an equivalent program with process migration facility. The program is compiled into all targeted architecture. When process migration is triggered, the runtime support of the three systems computes the type and value of each variable, the stack trace, the memory graph and the program counter. It represents them in an intermediate form and sends it to the destation host. The execution environment is then reconstructed at the destination host.

For these three systems, the program has to be written in a type-safe subset of standard languages (Ansi-C, Pascal and so on). Type safety is required because the runtime support must compute location of each objects manipulated, and the values and type of each variable of the program. Migration can not occur anytime during execution: the pre-processor or compiler inserts adaptation points that will poll migration status. This leads to decrease performance.

2.2 Process Migration Upon Virtual Machines

Virtual machines such as JVM or .Net hide the underlying execution environment. The code is compiled into an intermediate bytecode that is either interpreted during execution or compiled dynamicaly by a just in time compiler (JIT). A virtual machine is therefore a convenient architecture for program portability. However, they do not provide in their specification thread or process migration facility. The difficulties to implement thread migration upon virtual machines include derivating the type of variables present on stack, selecting the thread's reachable object graph and restoring the thread at the destination from where it was interrupted. Three appraoches were taken by the research community to add process migration for the JVM: source-level transformation, bytecode-level transformation and extended virtual machines.

Source-level transformation: Funfroken implemented process migration by mean of source transformation [8]. It uses a specific compiler in order to add thread-state capturing and restoring in methods of the program. Migration is triggered by the thread itself by raising a Java exception. This code addition in the program decreases performance when no migration is triggered.

Bytecode-level transformation: Brakes [28] and the system of Sakamoto *et al.* [18] perform bytecode transformation. The algorithms taken for thread migration are basically the same than source-level transformation. Bytecode-level transformation has the advantage of enabling migration to applications only available in bytecode format.

Extended virtual machines: MOBA [21] or the system of Bouchenak *et al.* [3] extend the Sun JVM to provide thread migration. Both systems consist of a thread API, and the use of the Java debugging interface. The debugging interface allows to compute type and values of variables, and the thread's execution environment. The main issue faced by using the debugging interface was performance: the JIT had to be disabled. To bypass this issue, MOBA implemented type inference of the thread's stack, however the system could face collision of references with integer. Bouchenak *et al.* implemented a second run-time stack, that stores the type of variables stored on the memory stack.

Jessica2 [32] is a distributed JVM that runs on a cluster of workstations. It gives the illusion to applications of a single system image. It uses a distributed shared memory, the Global Object Space (GOS). Thread migration has been implemented on top of Jessica2 [31]. It performs dynamic recompilation of currently executing methods in order to compute the bytecode program counter, the emplacements of the methdods variable (memory or registers) and their type. This execution environment is then stored in an intermediate form and sent to the destination node. This algorithm does not face object accessibility because it is based on a distributed shared memory. Furthermore, the virtual machine is note migrated, it must be executing on the destination node. Squawk [22] is a JVM that does not need an operating system. It considers applications as isolates and is able to migrate them between different Squawk instances. In this context, the virtual machine is not migrated, only the application.

2.3 Summary on Existing Systems

Table 1 compares all presented systems with respect to heterogeneity, performance, transparency and reusability.

Table 1. Comparison between different process migration systems

System	Heterogeneity	Performance	Transparency	Reusability
Locus, V Kernel, Amoeba	no	***	***	***
Condor, MPVM	no	**	**	***
Freedman's	no	**	*	*
Tui, Snow, MigTthread	yes	**	**	**
VM Source transformation	yes	*	***	*
VM Bytecode transformation	yes	*	***	*
Moba and Bouchenak's	yes	***	*	*
Jessica2, Squawk	yes	***	***	*

All these systems have limitations. Binary process migration targets mostly homogeneous network of workstations. The few systems that allow heterogeneity can not trigger migration anytime during execution. Furthermore, it needs the source code of the application and requires that the application is implemented in a subset of standard languages such as Ansi-C or Pascal to ensure type safety.

Process migration upon a virtual machine is another approach. Virtual machines are usally type-safe, and hides the hardware and the operating system to the application. Extended virtual machines have less limitations than bytecode or source-level transformation. They do not add extra computations when migration is not triggered. However, the existent systems such as Moba or the one from Bouchenak *et al.* do not provide transparency, because the application must use a specific thread API. Furthermore, they both suffer from using the Sun JVM: they bypass its restriction on stack capturing by using a non-standard approach and does not benefit from the JVM evolution. The Jessica2 thread migration performs transparent migration anytime during execution, and does not face object graph migration. We think the algorithm taken by Jessica2 is the one that has the best advantages, because it is directly integrated in the JVMs, uses its JIT compiler and does not require to modify the application, nor use a specific API. However, their implementation is eased by the distributed shared memory.

3 The Micro Virtual Machine

Achieving heterogeneity in process migration with virtual machines is more straightforward and convenient than with binaries. However the proliferation of virtual machines (Smalltalk, JVM, .Net, Ocaml and so on) requires to implement thread migration for each one. The goal of the micro-virtual machine (MVM) is to resolve this type of issue [6].

The micro virtual machine is a minimal execution environment that can be specialized dynamically by applications. Early projects include specializing the MVM for active networks, web caches [15], or Java [27]. In this paper we propose an implementation for thread migration that will enable migration for all systems implemented on top of the MVM, and especially Java. For example, the work done to achieve a functionnal JVM can be applied for a .Net virtual machine.

Figure 1 represents the micro-virtual machine, specialized to become a Java virtual machine called JnJVM. The MVM is structured with components. It is composed of standard virtual machine components such as a thread manager, a memory manager and a JIT compiler. To enable dynamic adaptation, it also contains a Lisp front-end, an extensible compiler and a component system. It uses aspect technology [12] to modify and extend the components. JnJVM extends the MVM by adding new components that alltogether implement a Java virtual machine.

4 Thread Migration in the Micro Virtual Machine

The algorithm of our thread migration is close to the algorithm of the thread migration system of Jessica2. Because the MVM does not use distributed shared memory by default, our algorithm has to compute the object graph reachable from the migrating thread.

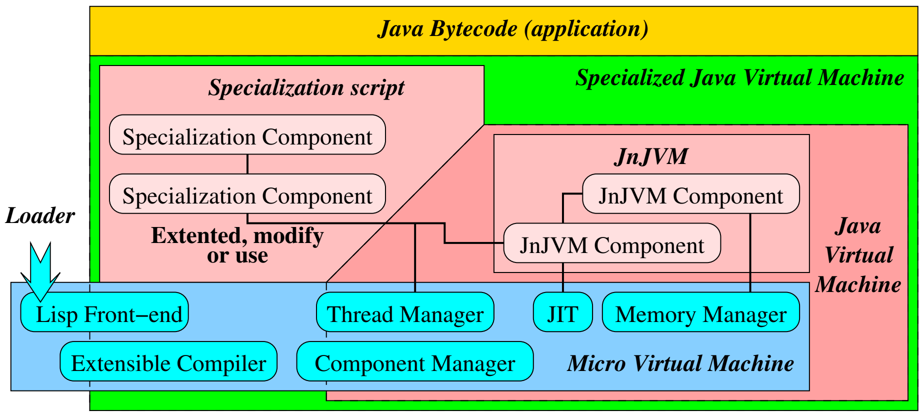

Fig. 1. Architecture of the MVM

4.1 Algorithm

The MVM components ease the implementation of our algorithm. It uses the just in time compiler to save the type of the variables on stack and the program counter relative to a MVM bytecode instruction. It uses the memory manager to locate the object graph of the thread.

Type information: The MVM reads by default a Lisp-like language. An application can however modify this language. The MVM compiles its input with a just in time compiler called the Virtual Processor Unit [16]. The VPU is a stack-based abstract machine that transforms a sequence of actions on a stack into an assembled form. Migration requires to know the type of each variable on the stack. Therefore the compiler makes a distinction between integers, float, and objects. After compiling, the VPU returns a method object. It contains the sequence of VPU actions, and the assembled code.

Recompilation: Migration is triggered by an external entity by using Posix signals. The thread interrupts and retrieves its stack trace. For each method on the stack, it analyses its sequence of VPU actions. This allows to found the frame pointer of each method, and the type of each variable on stack (register or memory). In order to compute the program counter and send it in a portable manner, the program counter must represent the start of a virtual instruction of the VPU. If it is not the case, execution is processed until it reaches the end of a virtual instruction.

Object Graph: Some variables on the execution stack are objects. These objects might reference other objects and the overall references give an object graph, all reachable from the thread. The memory manager allows to construct the graph and find the objects. A first naïve implementation would gather this graph with the execution environment in an intermediate form. The result is sent to the destination node. For efficiency, we may also implement a proxy system [20], where objects are not sent directly to the destination, but only proxies. The object will be sent when required by the destination node.

Recovering the execution: When the destination node receives the intermediate form of the thread's execution environment, it compiles all methods of the stack trace and records emplacement of variables (register or memory). The memory stack is reconstructed depending on these informations, and object pointers are changed depending on their location in the new workstation. The thread is then started with the given stack.

4.2 Limitations

The migration system will have limitations: some limitations are inherent to process migration, others are particular to our architecture. In the first case, process migration must face dependances with kernel structures such as open files, or sockets. In the second case, with our architecture, process migration can not occur while the process is compiling a method (due to the architecture-dependant code generation) or while it is collecting memory (because it manipulates structures dependant of the memory graph).

In order to allow migration allmost anytime during execution, the MVM can not rely on a C/C++ implementation as it does in the present. We will have to introduce a metacircular virtual machine, i.e. a virtual machine implemented in itself. It will enable migration of any of its methods. Recent metacircual virtual machine implementations such as Klein for the Self language [29] or Squawk for the Java language [22] show many interest of this approach. For example, it inherits features of the virtual machine such as possible type safety, garbage collection or exception handling. It also generally eases porting and debugging the virtual machine. Squawk executes without an operating system on top of small devices. It defines isolates which are basically applications, that can be migrated between different Squawk virtual machines. Our proposed architecture follows this direction, but enables the virtual machine to migrate as well, therefore the entire system itself.

5 Perspective: Heterogeneous OS Migration

Live OS migration has become realistic with the arrival of virtual machine monitors (VMM) such as Xen [1], or VMWare [30]. Virtual machine monitors enable execution of many operating systems simultaneously on the same hardware. In this context, a virtual machine is an operating system executing on top of the VMM. Clark *et al.* have implemented OS migration with Xen [5], and Sapuntzki *et al.* with VMWare [19]. Migration is based on memory page transfer. The base requirement for these systems is homogeneity: the processor has to be the same between nodes. The harware dependances are mostly dealt by the virtual machine monitor.

In order to reconcile OS migration with heterogeneity, we propose to combine our minimal virtual machine approach with emergent OS implemented in a virtual machine such as JVM. There are two main systems that propose OS abstractions in virtual machines. KaffeOS [9] is an extension of the Kaffe virtual machine, a free JVM. KaffeOS enables execution and protection of many processes inside one virtual machine. It requires to execute on an existing operating system. On the other hand, Jnode (Java New Operating system Desing Effort) [11], or JX [10] are complete operating systems implemented in Java that execute Java-only applications. They are composed of a nano-kernel for basic hardware communications, a JVM implemented in Java, and device drivers implemented in Java.

Combining the micro virtual machine with the concepts of KaffeOS would allow migration of processes and the virtual machine (therefore the OS, as seen by the KaffeOS authors). A wider perspective is to combine the Jnode operating system with the micro-virtual machine. The entire operating sytem could be migrated allmost anytime during execution, limitation being access to device drivers, compilation and garbage collection. Kernel structures can be migrated because they are implemented in Java. We can not however deal with local hardware access such as local files, but OS migration with virtual machine monitor faces the same issue.

6 Conclusion

In this paper we have presented a thread /process /OS migration architecture in heterogeneous environments. The architecture is a proposal and OS migration is more of a reflexion than a proposal. Much effort has to be provided to enable this architecture in the micro virtual machine framework. Moreover, the combination of a virtual machine oriented operating system and the micro virtual machine remains to be discussed.

Thread and process migration have long time proved their efficiency for load-balancing, mobile computing, resource sharing, fault resilience, etc. OS migration is recently in current investigation thanks to the arrival of virtual machine monitors. It inherits all advantages of process migration at coarse grain. One of the main benefit of OS migration presented by Clark *et al.* [5] is for cluster administration. System and hardware are clearely separated, and the removal of a node does not lead to loosing its operating system. Furthermore, mobile computing is allways an interesting target for migration; OS migration will allow a user to carry and use its operating system wether it is on a Portable Digital Assistant, a laptop or a personal computer.

References

1. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Symposium on Operating Systems Principles*, pages 164–177, New-York, USA, October 2003.

2. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam. PVM and HeNCE: Tools for Heterogeneous Network Computing. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106. Springer-Verlag, 1993.
3. S. Bouchenak and D. Hagimont. Zero Overhead Java Thread Migration. Technical Report 0261, INRIA, 2002.
4. V. Chaudhary and H. Jiang. Techniques for Migrating Computations on the Grid. *Engineering the Grid: Status and Perspective*, January 2006.
5. C. Clark, K. Fraser, S. Hand, J. Gorm Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, Boston, USA, May 2005.
6. B. Folliot, I. Piumarta, and F. Riccardi. A Dynamically Configurable, Multi-Language Execution Platform. In *8th ACM SIGOPS European Workshop*, 1998.
7. D. Freedman. Experience Building a Process Migration Subsystem for UNIX. In *Proceedings of the Winter Usenix Conference*, pages 349–356, 1991.
8. S. Funfrocken. Transparent Migration of Java-Based Mobile Agents. In *Mobile Agents*, pages 26–37, 1998.
9. G. Back and W. H. Hsieh and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, San Diego, USA, October 2000.
10. M. Golm, M. Felsera, C. Wawersich, , and J. Kleinoeder. The JX Operating System. In *Proceedings of the Usenix Annual Technical Conference*, pages 45–58, Monterey, USA, June 2002.
11. JNode: Java New Operating System Design Effort. http://www.jnode.org.
12. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, Jyväskylä, Finland, June 1997.
13. M. Litzkow. Remote UNIX. Turning Idle Workstations into Cycle Servers. In *Proceedings of the Summer Usenix Conference*, pages 381–384, June 1987.
14. D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computer Survey*, 32(3):241–299, 2000.
15. F. Ogel, S. Patarin, I. Piumarta, and B. Folliot. C/SPAN: A Self-Adapting Web Proxy Cache. In *Proceedings of the Autonomic Computing Workshop*, pages 178–186, Seattle, USA, June 2003.
16. I. Piumarta. The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation. In *Proceedings of the Virtual Machine Research and Technology Symposium*, pages 97–110, San Jose, USA, May 2004.
17. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS a Network Transparent, High Reliability Distributed System. In *Proceedings of the Symposium on Operating Systems Principles*, pages 169–177, Pacific Grove, USA, December 1981.
18. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in java. In *Proceedings of the International Symposium on Agent Systems and Applications/Mobile Agents*, pages 16–28, Zurich, Suisse, 2000.
19. C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M.Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Boston, USA, December 2002.
20. M. Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the International Conference on Distributed Systems*, pages 198–204, Cambridge, USA, May 1986.

21. K. Shudo and Y. Muraoka. Asynchronous Migration of Execution Context in Java Virtual Machines. *Future Generation Computer Systems*, 18(2):225–233, October 2001.
22. D. Simon and C. Cifuentes. The Squawk Virtual Machine: Java on the Bare Metal. In *Proceedings of the Companion to the Object-Oriented Programming, Systems, Languages, and Applications Conference*, pages 150–151, San Diego, USA, October 2005.
23. P. Smith and N. Hutchinson. Heterogeneous Process Migration: The Tui System. *Software Practice and Experience*, 28(6):611–639, 1998.
24. C. Steketee, W. Zhu, and P. Moseley. Implementation of Process Migration in Amoeba. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 194–201, June 1994.
25. X. Sun, V. Niak, and K. Chanchio. A Coordinated Approach for Process Migration in Heterogeneous Environments. In *Proceedings of the SIAM Parallel Processing Conference*, March 1999.
26. M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the Symposium on Operating Systems Principles*, pages 2–12, Orcas Island, USA, December 1985.
27. G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta. Building a Flexible Java Runtime upon a Flexible Compiler. In Jean-Jacques Vandewalle David Simplot-Ryl and Gilles Grimaud, editors, *Special Issue on 'System & Networking for Smart Objects' of IASTED International Journal on Computers and Applications*, volume 27, pages 28–47. ACTA Press, 2005.
28. E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings of the International Symposium on Agent Systems and Applications/Mobile Agents*, pages 29–43, Zurich, Suisse, 2000.
29. D. Ungar, A. Spitz, and A. Ausch. Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment. In *Proceedings of the Companion to the Object-Oriented Programming, Systems, Languages, and Applications Conference*, pages 11–20, San Diego, USA, October 2005.
30. VMWare, Inc. VMWare VirtualCenter Version 1.2 User's Manual, 2004.
31. W. Zhu, W. Fang, C. Wang, and F. Lau. A New Transparent Java Thread Migration System Using Just-in-Time Recompilation. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, pages 766–771, Cambridge, USA, November 2004.
32. W. Zhu, C. Wang, and F. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *Proceedings of the International Conference on Cluster Computing*, Chicago, USA, September 2002.