Building a Flexible Java Runtime upon a Flexible Compiler

Final draft

G. Thomas[†], F. Ogel[‡], A. Galland[†,*], B. Folliot[†], I. Piumarta[†]

[‡]INRIA Rocquencourt, Domaine de Voluceau
78153 Le Chesnay, France
frederic.ogel@inria.fr

[†]Laboratoire d'Informatique de Paris VI, Université Pierre et Marie Curie,
4, place Jussieu, 75252 Paris Cedex 05, France
gael.thomas@lip6.fr bertil.folliot@lip6.fr ian.piumarta@lip6.fr

[*]Gemplus Research Labs, La Vigie, Avenue du Jujubier,
ZI Athelia IV, 13705 La Ciotat Cedex, FRANCE
antoine.galland@gemplus.com

Abstract

While Java has become a *de facto* standard for mobile code and distributed programming, it is still a rigid and closed execution environment. Not only does this lack of flexibility severely limit the deployment of innovations, but it imposes artificial constraints to application developers. Therefore, many extensions to the JVM have been proposed, each of them dealing with specific limitations, such as emerging devices (mobile phones, smart cards), or constraints (real-time, fault tolerance). It leads to a proliferation of *ad hoc* solutions requiring the design of new virtual machines. Furthermore, those solutions are still rigid, closed and poorly interoperable.

In response to this problem, we propose a flexible Java execution environment, called the JnJVM, that can be dynamically adapted to applications' needs as well as to available resources.

keywords: Interoperability, Flexibility, Virtual Machines.

1. Introduction

Virtual machines have become a widely-adopted solution — from smart objects to active routers — to handle emerging application domains, particularly mobility. A good illustration of this phenomenon is the emergence of Java as a *de facto* standard for mobile and distributed computing: it is used for embedded mobile applications in smart cards, for service and protocol deployment in Active Networks, for code mobility in Multi-Agent systems and as a general purpose programming language in Web applications.

The rapid growth of emerging application-domains has led to a proliferation of new dedicated execution environments. Indeed, as limitations of existing environments are identified, *ad hoc* solutions are proposed, to solve particular problems or limitations. Those solutions are thus as rigid and closed as the original environment they are based on. Most of the research projects focusing on solving a limitation in Java end up with a modified version of a standard Java Virtual Machine (JVM). For example, to support new devices, such as mobile phones with J2ME [21] and smart cards with JavaCard [7], or to introduce reflection, like MetaXa [24], objects persistence, as in PJama [1] or compilation optimizations, like Marmot [14], it was necessary to develop a new Java runtime, because the original was not flexible enough to be dynamically adapted. What about a Java runtime offering object persistence and compilation optimizations? It has to be another dedicated Java runtime.

Not only does this lack of flexibility limit the development and propagation of innovations, but it also imposes constraints on developers, by exposing rigid and frozen high-level abstractions. Thus, developers have to deal with the adaptation of their project to the semantics of their execution environment rather than focusing on the

computation problem they are trying to solve.

Even though software adaptation is a very active research area, most projects have focused on either system aspects (mainly resources management) or language aspects (through reification or partial evaluation) but without trying to merge them, in order to provide a fully dynamically-adaptable execution environment.

As stated in [19], the thin line between language and system is getting thinner and fuzzier. Based on the same observation, we have developed a systematic approach for software adaptation, based on a language and hardware independent platform called the Virtual Virtual Machine (VVM) [28]. In the context of the VVM project, we propose the JnJVM: a flexible Java runtime for smart devices that can be dynamically adapted to match applications' needs and available resources.

The remainder of this paper starts by presenting related work on flexible virtual machines and embedded execution environments in Section 2. Section 3 briefly presents the VVM architecture and its main components followed by its application to the construction of a dynamically adaptable Java runtime, described in Section 4. Section 5 illustrates the benefits of dynamic flexibility with some examples and some performance measurements. Finally, conclusions and perspectives are presented in Section 6.

2. Related work

In response to emerging application domains, many projects have focused on the evolution of existing environments to match new semantics, architectures and constraints, such as J2ME [21], JavaCard [7], JPS [6], RT-Java [5] and [27]. However, as stated before, the resulting environments remain as rigid as those from which they are derived.

Research in flexible operating systems has focused on extensibility of resource man-

agement components. Systems such as SPIN [2] or VINO [34] offer loading extensions directly into the kernel and therefore have to deal with security and consistency issues. Those issues are addressed with rigid and static policies, thus limiting the overall flexibility. On the other hand, EXOKERNELS [11] push extensions into the application-level, through libraries implementing traditional abstractions and services. Hence misbehaving extensions only affect the associated application. Although not limited, this flexibility is static: once an application has been launched with its extensions, nothing can be reconfigured. Furthermore, flexible operating systems do not consider language aspects and still try to enforce a "red line" between languages and system.

XVM [18] proposes a component-based extensible virtual machine that uses a dedicated language to describe a virtual machine's internals. Applications provide their extensions and the underlying virtual machine uses call-backs to replace an internal policy, such as garbage collector, by an application's extension. Although the virtual machine is extensible, its lack of reflexivity limits the flexibility to a predefined set of aspects.

VANILLA [10] uses a DSL-based[1] approach for the construction of dedicated virtual machines. Elements such as parsers, type checkers and interpreters are described using a dedicated language. A *Language Definition File* identifies the components that need to be combined to obtain the desired virtual machine. While being well adapted to the construction of dedicated/specialized virtual machines — like approaches to build JAVA environment for embedded system used in JITS [32] or JEPES [33] — it remains a static approach and the resulting environments are rigid and closed, thus lacking flexibility.

HARISSA [26] is a another DSL-based project. Its goal is JAVA program specialization through program transformation and especially partial evaluation, hence it focuses

[1]Domain Specific Language.

on the adaptation of an application to a specific context of utilization rather than on the definition of a flexible execution environment. A dedicated language is used to define specializations of "generic" classes and specialization parameters, which can be static (evaluated at compile-time) or dynamic (evaluated at run-time). The result is an equivalent specialized c-program that can be compiled with *gcc*. This approach is obviously orthogonal and may be applied to the construction a specialized execution environment, matching some given application domain semantics.

Embedded operating systems, such as PalmOS [29] or WindowsCE [25] are as rigid as their "traditional" counterparts. eCos [8] is structured as a set of independent components, so that it can be configured with specific memory allocators or scheduler, but its flexibility is still static and limited by the system/language separation.

Camille operating system [9] relies on the Exokernel [11] approach to obtain extensibility, without compromising security. It provides four basic characteristics for applications: security, extensibility, interoperability, and portability. Embedded code is expressed using a dedicated intermediate language called FAÇADE, which provides security ensured by a code-safety checking (based on PCC-like algorithm) and extensibility through a simple representation of the hardware. Due to the fact that the usual downside of extensibility is performance, it uses Just-in-Time (JIT) techniques to compile intermediate code into native code. From the architecture standpoint, its minimalist approach coupled with a JIT is comparable to the principles of our VVM.

3. The Virtual Virtual Machine Approach

Instead of developing a new dedicated virtual machine for each new application domain, the VVM approach defines a "virtualized" virtual machine that can be dynamically

extended and specialized by loading active specifications called *VMlets*. A VMLet specifies an environment by defining its abstractions and functional code. A VMLet is said to be active because it is a program executed by a minimal environment (*i.e.*, the MICRO-VM and not only a description of the environment).

Using a single generic/meta virtual machine (the VVM) allows a factorization of several low-level mechanisms[2] found in practically all virtual machines. This factorization results in a more efficient management of resources: (i) less memory is used, thanks to the elimination of many redundant components; (ii) resources are managed by a unique environment being shared between several VMLet in a more efficient way.

As illustrated in Figure 1, a VVM relies on a minimal execution environment, called the MICRO-VM [3], that is extended and specialized by VMLets.

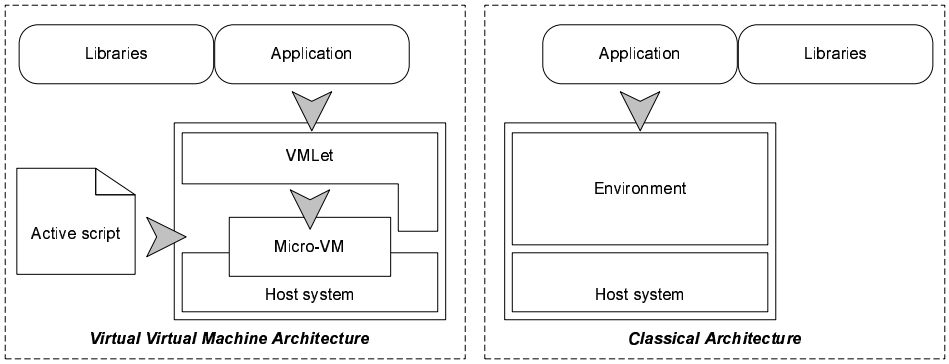

Figure 1: General Architecture

The MICRO-VM is dedicated to the construction of the execution environment . It is structured as a set of interfaces and components based on the ODP Reference Model [20]. The MICRO-VM is both a minimal virtual machine (since it can execute abstract instructions) and a dynamic compiler. It offers full flexibility because of two

[2]such as virtual processor or garbage collector.

[3]For a more detailed description of the MICRO-VM, also called YNVM, see [31].

properties: (i) the MICRO-VM is entirely open and reflexive, thus anything (from its internals to application code) can be adapted; (ii) the dynamic compilation inherent in the MICRO-VM allows "on-the-fly" reconfiguration.

This minimal execution environment is then extended or specialized according to a given application domain (as with a DSL-based approach). This adaptation consists in loading a VMLet, which describes a dedicated execution environment. This specification directly extends the MICRO-VM with new dedicated primitives, operators, abstractions for resource management, or language support for dynamic compilation. While remaining a generic virtual machine dedicated to execution environment construction, the MICRO-VM however becomes a native, domain-specific execution environment as well, offering the same semantics and performance as any other native, hand-coded, domain-specific execution environment.

Applications executed by the MICRO-VM are called *active applications*. They are composed of two distinct parts: the application code (for example, a JAVA class file) and an active script executed by the MICRO-VM. The active script is responsible for loading the appropriate VMlet and for adapting it to the application's needs. For example, an application based on a JVM with persistent objects is composed of a JAVA program (the application) and a script that loads a JAVA VMlet and adapts it with the extensions related to objects' storage.

The MICRO-VM runs on top of several host systems, such as LINUX, WINDOWS or MAC OS. It also runs as an "embedded" environment on bare hardware[4] using the THINK [35, 13] exokernel. Such a stand-alone MICRO-VM bootstrap has a 120 Ko memory footprint, including a reification of hardware resources, network, keyboard and framebuffer drivers, a complete MICRO-VM (dynamic compiler (JIT), garbage

[4]currently PowerPC processors.

collector,...) and a basic TFTP-like[5] protocol for incremental module loading from the network.

4. The JnJVM: a Java VMlet

The main goal of this VMlet is to define a standard implementation of Sun's specifications [23], while preserving a high degree of dynamic flexibility. The JnJVM is structured as a set of components, such as a JIT compiler based on the underlying Micro-VM dynamic compiler, a memory manager with garbage collection, a linker and an exception manager. Component interfaces are just a table of functions. Hence, adapting a functionality in a component is done by modifying the associated entry in the interface. An application can replace any component of the JnJVM by an arbitrary component. Thus, the JnJVM can define a wide range of Java runtimes, from a very minimal one to a complete implementation of Sun's specification, according to available resources on the target device. Moreover, additional components can be dynamically integrated, when (and only when) needed. The main limit of our approach is the Garbage Collector (GC): every component on the JnJVM is defined as collectable object. Changing the GC implies recompiling the JnJVM.

The JnJVM has been written using the Micro-VM's front-end language, which means the Java *virtual machine is an application in the* Micro-VM *environment*. The Micro-VM's dynamic compiler is used to introduce code and execute it. Moreover, since JnJVM's internal components are based on a simple yet flexible model[6], function pointers are re-bound *on-the-fly* to dynamically compiled code. Those two basic

[5]TFTP stands for Trivial File Transfer Protocol.

[6]The JnJVM uses the RM-ODP based component model of the underlying Micro-VM.

mechanisms (dynamic compilation and adaptable components) allow any application to tailor the JnJVM to its needs. As a result, our architecture can be considered aspect-oriented [22]: applications dynamically weave aspects in the underlying Java virtual machine in order to adapt the execution environment to their needs. An application, from the JnJVM perspective, is split in two distincts parts: (i) a functional part, written in Java and (ii) a non-functional part (*i.e.*, aspects), written using the JnJVM language, that describes a Java virtual machine.

Our approach is completely dynamic: a Java virtual machine is build and adapted *on-the-fly* according to applications' needs. Thus architecture differs from JITS [32], which allows to build *a priori* a minimal Java virtual machine dedicated to an embedded application.

4.1 A dedicated language

In order to build the JnJVM, we exploited the Micro-VM's flexibility to extend it with new primitives, such as `(def-opcode name number explore compile)` which is used to define compilation functions associated with specific opcodes. Those new primitives hide the complexity of some operations not only from the VMlet, but from the applications too, which can in turn reuse work done at the VMlet level. This extension to the Micro-VM's font-end language is a DSL dedicated to the construction and adaptation of Java virtual machines, hence it encapsulates the semantics (and complexity) of this application-domain.

4.2 Interoperability

A major goal of the VVM architecture is to provide a common language substrate upon which interoperability at both application and execution environment levels can be achieved. Interoperability between execution environments means that an application Ax using a VMlet X can call functions from an Ay application using a VMlet Y. This can be far from trivial. The X VMlet has to know the exact semantics of VMlet Y calls (such as virtual calls), as well as the binding functions. In order to simplify JAVA calls from an external VMlet or script, we have defined an IDL (*Interface Definition Language*). This language is composed of a small subset of keywords, such as `(externalize-class cl)`, `(externalize-field cl name sign type)` and `(externalize-method cl name sign type)`[7]. Interface descriptions are compiled to the front-end language of the MICRO-VM: each JAVA symbol is associated with a MICRO-VM symbol. For example, a symbol `java.lang.Object.clone_sign`[8] is associated with the `clone()` method of the `java.lang.Object` class. Any call to `java.lang.clone_sign A` is transparently rewritten to a virtual call to JAVA method `clone`, through the bindings at the JNJVM level. Hence, any VMlet can transparently interoperate with the JNJVM.

To implement this feature, we define one *macro*[9] for each kind of JAVA call and field access. The only overhead introduced is therefore the memory footprint of the MICRO-VM symbols (*i.e.*, 16 bytes each). To help application adapting methods, MICRO-VM symbols keep a pointer to the meta-description of their associated method. This IDL can be bypassed by any application willing to do low-level manipulations.

[7] where `type` is either virtual, special or static.

[8] where `sign` is the method's signature.

[9] As described in [31], the Micro-VM defines a pre-processing mechanism that allows code transformation/rewriting before native code generation.

4.3 Introspection and reification

The introspection is handled at the Micro-VM level by dedicated functions, such as `(lookup-method cl name signature)`. Those functions allow the direct manipulation of data structures associated to any components of the JnJVM. Every part of those data structure can be accessed and modified from an application, especially the pointer to native code for the methods. This introspection layer can also be used to wrap **applicatives** aspects dynamically. The components manipulated by the non-functional part of the application are the internal components of the JnJVM.

4.4 Bytecodes Compilation

Java methods are not interpreted, but dynamically compiled to native code by the JIT components of the JnJVM. This component is based on the Virtual Processor of the Micro-VM (called the VPU [30]). Hence the JIT of the JnJVM is portable across any architecture with a Micro-VM.

The VPU uses a stack-based representation for the code. It does not define any bytecode for itself: compiling a function is done through stack manipulation functions. Since Java bytecodes are executed in a stack-based machine, the mapping is quite simple.

```
(def-opcode .goto 167                                        ;;; Define the opcode.
  (lambda(op code-input curseur env comp vpu)                ;;; The first pass
    (opinfo.def-label env (+ curseur (read-s2 code-input)));;; defines a label
    (+ curseur 3))                                           ;;; and consumes 3 bytes.
  (lambda(op code-input curseur env comp vpu)                ;;; The second pass compiles.
    ;;; 'lab' is the VPU reference of the label
    (let ([lab  (opinfo.label (opinfo.at (env.opinfo env)
                              (+ curseur (read-s2 code-input))))])
      (:compiler.vpu.br-int vpu lab)          ;;; asks the vpu to jump on lab
      (+ curseur 3))))                        ;;; and consumes 3 bytes.
```

Figure 2: The `goto` bytecode

For example, Figure 2 shows how to compile the `goto` opcode. The first pass defines a label in the VPU and the second one jumps to this label. All opcodes in the JnJVM are compiled in the same way.

While, the first pass is used to define branch point in the VPU, the second one is used to compile the bytecode itself. The VPU's functions are used to register the stack manipulation. After this second pass, the internal compilation function of the VPU is called to produce the assembly function directly into the memory.

Compiled functions are stored in collectable objects : the garbage collector frees the associated memory when a function is no longer used.[10] Thus, the the non-functional part of the application is freed from memory management.

5. Applications and performance measurements

We presents some example applications: a standard JVM, a remotely adaptable JVM and a JVM using escape analysis to optimize memory management.

5.1 The reference virtual machines

These examples of active applications instantiate:

- a standard JVM, fully compliant with Sun's specification [17, 23]. Basic classes required to bootstrap the JVM were taken from the GNU ClassPath project [16]. This active script is used for compatibility and performance tests and can be used to execute any standard Java application.

- the same JVM with the possibility of adapting on-the-fly the internal compo-

[10] Compiled functions keep references to other functions and variables they use.

nents of the JVM. As illustrated in Figure 3, such a JVM is remotely managed by launching a control process, responsible for receiving and executing serialized code[11] from the network, before starting the JAVA application itself. Reconfiguration commands are sent to the JNJVM while the application is running.

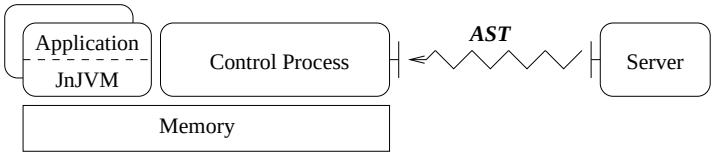

Figure 3: Remote reconfiguration of a JNJVM

Even if mechanisms to secure the remotely adaptable JVM exist, we did not investigate this aspect any further: functions and symbols used by and AST can be (read/write)-protected or isolated in a separate namespace in order to safely check remote-user rights. Especially the define or set! symbols can hidden/suppressed in this temporary namespace, hence disabling any possibilities of modification bypassing the strict set of abstractions exported by the namespace.

5.2 Escape Analysis and Stack Allocation

This section illustrates the addition and management of an application-specific attribute used to indicate the lifetime of objects allocated within a method, allowing the compiler to decide whether or not an object can be destroyed automatically when the method returns.

The goal of this example is to adapt on-the-fly a JVM to memory-stack allocation. It is based on code analysis called Escape Analysis [3]. This static analysis determines

[11] We use an intermediate representation: Abstract Syntax Trees (AST) for code migration.

whether the lifetime of the data exceeds its static scope (the scope of its declaration). In JAVA, the static scope is the method in which the object is allocated. When an object is created in a method, escape analysis aims to determine whether this object can be accessed or not after the method returns. If not, the object can be placed in a special region — in stack memory, called "stack allocation" — instead of the heap memory. Stack allocation can have several positive effects on the execution of a program. For example, it reduces the Garbage Collector workload, since it does not have to manage data allocated in the stack. This can lead to a decrease of the GC execution time and therefore improve the global execution time of the program.[12]

We use an off-board static tool to compute escape analysis, resulting in a list of allocations that can be "stack allocated". A JAVA code attribute, named `EscapedMap`, inside the class file marks each bytecode `new` that can be "stack allocated".

Table 1 shows some static Escape Analysis benchmarks using our off-board tool. The benchmarks we selected (the sequential benchmarks) are a subset of the applications developed by the JAVAGRANDE Forum [12] using the JAVA GNU CLASSPATH library [16]. These static results are comparable with those presented in prior work [15, 3]. Furthermore, thanks to stack management, runtime execution decreases up to 30%.

In order to respect the size limitation of processes stack, we use a pseudo-stack (in the heap) to store "stack allocated" objects. When a method returns, a function is called to update stack pointer and finalize objects in the frame. Nonetheless, this solution raises a problem: when an exception occurs, we must be able to finalize the "stack allocated" objects, in particular those which were allocated in "sub-functions". Therefore, we slightly modified the `throw(object)` function so that it finalizes allocated

[12]Of course, escape analysis has to respect the pointer safety policy, thus the result of escape analysis belongs to the Trusted Computing Base (TCB) of the platform. We do not discuss this part in this paper.

Program	Stack Allocation considering that native methods escape		Stack Allocation considering that native methods *do not* escape		Total Allocation
GNU ClassPath	543	(4%)	800	(6%)	13226
JavaGrande Forum	363	(46%)	386	(49%)	774
⟶ Euler	18	(31%)	18	(31%)	57
⟶ MolDyn	3	(33%)	3	(33%)	9
⟶ Montecarlo	4	(4%)	6	(6%)	104
⟶ Raytracer	11	(19%)	11	(19%)	57
⟶ Search	7	(24%)	7	(24%)	29

Table 1: Static Escape Analysis benchmarks

objects[13].

This example demonstrates how an active application can enhance the JVM with relatively low development cost (about a hundred lines of code).

5.3 Performance measurements

In order to evaluate our Java VMlet we compared it with IBM JVM (with a JIT) and Blackdown JVM (no JIT) using the JavaGrande benchmark [12] on a 466 MHz PowerPC running Linux[14]. Results are summarized in Table 2. The JnJVM appears to be 3 times slower than its IBM counterpart, but 12 times faster than Blackdown JVM. The gap between our Java VMlet and Blackdown comes from the JIT compilation: most operations from the benchmark are embedded in loops, which are faster when compiled into native code than when interpreted.

The difference between the JnJVM and IBM mainly comes from the optimizations performed during the dynamic compilation, especially the use of registers on stack to

[13]As a result, management of `try/catch` blocks has also been slightly modified.

[14]Since IBM and Blackdown JVM do not run on bare hardware, we used Linux as a common host system.

[15]4 fields.

	AddInt 10^6add/s	AddLong 10^6add/s	Set 10^6set/s	Create[15] 10^6objects/s	Throw 10^3excp/s	Call 10^6call/s
IBM1.3.1	456	215	148.9	1.38	313	55.4
JnJVM	141	1.8	37.2	0.61	107	17.6
Blackdown	2.0	1.6	2.0	0.35	1753	0.7

Table 2: Evaluation of the JnJVM

store local variables. Our VPU does not evaluate local variables lifetime, hence they are stored in the stack instead of recycling registers. As a consequence, most assignments involve a memory access, which slows down the execution. Nevertheless, the VPU is still a research prototype, not a highly-optimized commercial product as its IBM counterpart. Another weakness of our prototype explaining the performance overhead lies in objects allocation. We used a standard "mark-and-trace" algorithm, based on Boehm's garbage collector [4]. Thus, Micro-VM's internal objects (around 46.000), which have a greater lifetime than Java objects lifetime, are uselessly considered each time a garbage collection is issued. Then, a generational approach for memory management would significantly increase performance. Implementing a distinct garbage collector for the JnJVM could be another solution, but it goes in opposition to VVM's objectives.

6. Conclusions and Perspectives

This paper presented the JnJVM, a flexible Java runtime, based on the VVM architecture, that can be dynamically adapted to applications' needs as well as to resources available on the target device. Since it is entirely expressed in terms of the Micro-VM and its VPU, the resulting JVM will run on any platform that they support (currently Linux, THINK, MacOS). Such a platform is THINK, meaning that the JnJVM can

execute standard JAVA programs on "bare" hardware.

This approach introduces dynamic adaptability at the lowest level of the runtime architecture. In contrast with more traditional solutions (such as adapting an existing VM to a specific application domain), it offers more formalized support for VMlet descriptions, with a higher level of expressivity achieved through specialization of the MICRO-VM compiler's semantics to fit the needs of a given VMlet. To illustrate the benefits of this dynamic flexibility we show how a JAVA runtime is transparently adapted when loading an application modified by Escape Analysis to use this extra-information for performance purpose.

The evaluations of the JNJVM show that the VVM approach can bring dynamic flexibility and interoperability without sacrificing performance. However, an optimization of some mechanisms is still necessary to compete with industrial products such as IBM JVM.

Future directions for the JNJVM include the construction of a flexible, real-time JAVAOS for smart devices based on the MICRO-VM and THINK. Other bytecoded programming languages will be targeted to the MICRO-VM, in order to improve its architecture and abstractions.

References

[1] ATKINSON, M., DAYNES, L., JORDAN, M., PRINTEZIS, T., AND SPENCE, S. An Orthogonally Persistent Java. *ACM SIGMOD Record 25*, 4 (December 1996).

[2] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, Safety and Performance in the SPIN Operating System. In *the 15th ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1995).

[3] BLANCHET, B. Escape Analysis for Java : Theory and Practice. *ACM Transactions on Programming Languages and Systems 25*, 6 (Nov. 2003).

[4] Boehm, H.-J., Demers, A. J., and Shenker, S. Mostly parallel garbage collection. *Conference on Programming Language Design and Implementation (PLDI) — ACM SIGPLAN Notices 26*, 6 (1991).

[5] Bollella, G., Gosling, J., Brosgol, B., Gosling, J., Dibble, P., Furr, S., and Turnbull, M. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[6] Boyapati, C. JPS: A Distributed Persistent Java System. Master's thesis, Massachussetts Institute of Technology (MIT), September 1998.

[7] Chen, Z. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. The Java Series. Addison-Wesley, 2000.

[8] Cygnus. eCos. `http://sources.redhat.com/eCos/`.

[9] Deville, D., Galland, A., Grimaud, G., and Jean, S. Smart Card Operating Systems: Past, Present and Future. In *the 5th USENIX/NordU Conference* (Västerås, Sweden, February 2003).

[10] Dobson, S., Nixon, P., Wade, V., Terzis, S., and Fuller, J. Vanilla: An Open Language Framework. In *Generative and Component-Based Software Engineering (GCSE)* (Denver, Colorado, USA, September 28-30 1999), vol. 1799 of *LNCS*, Springer-Verlag.

[11] Engler, D. R., Kaashoek, M. F., and O'Toole, J. W. Exokernel: an operating system architecture for application-level ressource management. In *the 15th ACM Symposium on Operating Systems Principles (SOSP)* (Copper Mountain Resort, Colorado, USA, December 1995).

[12] EPCC. Edinburgh Parallel Computing Centre. Java Grande Forum Benchmark Suite - Version 2.0, 2003. `http://www.epcc.ed.ac.uk/javagrande/`.

[13] Fassino, J.-P., Stefani, J.-B., Lawall, J., and Muller, G. THINK: A Software Framework for Component-based Operating System Kernels. In *the USENIX Annual Technical Conference* (Monterey, CA, USA, June 2002).

[14] Fitzgerald, R., Knoblock, T. B., Ruf, E., Steensgaard, B., and Tarditi, D. Marmot: an optimizing compiler for Java. *Software Practices and Experiences 30*, 3 (March 2000).

[15] Franz, M., Haldar, V., Krintz, C., and Stork, C. H. Online Verification of Offline Escape Analysis. Tech. Rep. 26-2002, Department of Computer Science University of California, Santa Barbara (UCSB), September 2002.

[16] GNU. ClassPath 0.10. `http://www.gnu.org/software/classpath/`.

[17] Gosling, J., Joy, B., Steele, G. L., and Bracha, G. *The Java Language Specification*, 2nd ed. The Java Series. Addison-Wesley, 1996.

[18] Harris, T. L. An Extensible Virtual Machine Architecture. In *Simplicity, Performance and Portability in Virtual Machine Design Workshop (OOPSLA Conference)* (Denver, Colorado, USA, November 1999).

[19] Howell, J., Montague, M., and College, D. Hey, You Got Your Compiler In My Operating System! In *7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)* (Rio Rico, Arizona, USA, March 1999).

[20] ISO International Organization for Standardization. *Information Technology – Open Distributed Processing – Reference Model*, ISO/IEC 10746-(1-4) ed., 1996 — 1998.

[21] Java 2 Platform Micro Edition (J2ME). Connected Limited Device Configuration (CLDC) Specification. `http://www.java.sun.com/products/cldc/`.

[22] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., and and John Irwin, J.-M. L. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*, vol. 1241. Springer-Verlag, Jyväskylä, Finnland, 1997.

[23] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*, 2nd ed. The Java Series. Addison-Wesley, September 1996.

[24] Michael Golm. Design and implementation of a meta architecture for Java. Master's thesis, University of Erlangen, January 1997.

[25] Microsoft. WindowsCE. `http://msdn.microsoft.com/embedded/`.

[26] Muller, G., and Schultz, U. P. Harissa: A Hybrid Approach to Java Execution. *IEEE Software 16*, 2 (March/April 1999).

[27] Myers, A. C., Bank, J. A., and Liskov, B. Parameterized types for Java. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (Paris, France, January 1997).

[28] Ogel, F., Thomas, G., Piumarta, I., Galland, A., Folliot, B., and Baillarguet, C. Towards Active Applications: the Virtual Virtual Machine Approach. In *New Trends in Computer Science and Engineering*. A92 Publishing House, POLIROM Press, 2003.

[29] Palm. PalmOS. `http://www.palmos.com/`.

[30] Piumarta, I. The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation. In *the 3rd USENIX symposium on Virtual Machine Research and Technology* (San Jose, California, USA, May 2004).

[31] Piumarta, I., Ogel, F., and Folliot, B. YNVM: dynamic compilation in support of software evolution. In *Engineering Complex Object Oriented System for Evolution Workshop (OOPSLA Conference)* (Tampa Bay, Florida, USA, October 2001).

[32] Rippert, C., Courbot, A., and Grimaud, G. A Low-Footprint Class Loading Mechanism for Embedded Java Virtual Machines. In *3rd Principles and Practice of Programming in Java (PPPJ)* (Las Vegas (USA), 2004), ACM Press.

[33] Schultz, U. P., Burgaard, K., Christensen, F. G., and Knudsen, J. L. Compiling java for low-end embedded systems. *Language, compiler, and tool support for embedded systems (LCTES) — ACM SIGPLAN Notices 38*, 7 (2003).

[34] Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (1996).

[35] Senart, A., Charra, O., and Stefani, J.-B. Developing dynamically reconfigurable operating system kernels with the think component architecture. In *Engineering Context-aware Object-Oriented Systems and Environments Workshop (ECOOSE), in association with OOPSLA* (Seattle, USA, November 2002).

Frédéric Ogel *was a PhD student at INRIA. Its research activities concern dynamic flexibility in execution environments. Now he's a research engineer at France Télécom R&D.*

Gaël Thomas *is a PhD student at Pierre & Marie Curie University, LIP6 Laboratory. Its research activities concern mechanisms for dynamic adaption in middlewares.*

Antoine Galland *is a research engineer at Gemplus and PhD student at Pierre & Marie Curie University, LIP6 Laboratory. Its research activities concern resources management in smart cards.*

Bertil Folliot *is Professor at Pierre & Marie Curie University, LIP6 Laboratory. Its research activities concern flexibility and interoperability in distributed systems. He leads the Virtual Virtual Machine project.*

Ian Piumarta *is a research engineer at Pierre & Marie Curie University, LIP6 Laboratory. Its research activities concern reflexive and flexible virtual machines.*