SP&E

Designing highly flexible virtual machines: the JnJVM experience

Gaël Thomas, Nicolas Geoffray, Charles Clément, Bertil Folliot[*,†]

LIP6/CNRS - INRIA REGAL - University of Paris 6 - 104 avenue du Président Kennedy, F-75016 Paris

SUMMARY

Dynamic flexibility is a major challenge in modern system design to react to context or applicative requirements evolutions. Adapting behaviors may impose substantial code modification across the whole system, in the field, without service interruption, and without state loss. This paper presents the JnJVM, a full Java virtual machine (JVM) that satisfies these needs by using dynamic aspect weaving techniques and a component architecture. It supports adding or replacing its own code, while it is running, with no overhead on unmodified code execution. Our measurements reveal similar performance when compared to the monolithic JVM Kaffe. Three illustrative examples show different extension scenarios: (i) modifying the JVMs behavior; (ii) adding capabilities to the JVM; and (iii) modifying applications behavior.

KEY WORDS: Dynamic Adaptation, Aspect Weaving, Java Virtual Machine

1. Introduction

Dynamically modifying and extending a system is a critical need in modern systems design. Consider three examples. (1) Dynamic algorithm change: in an embedded device (e.g., a mobile phone) memory size limits the range of adequate systems algorithm available at any time. As resources or requirements evolve with the context, the current algorithm may become sub-optimal and must be replaced by another one. (2) Application specific optimization: the choices

[*]Correspondence to: LIP6/CNRS - INRIA REGAL - University of Paris 6 - 104 avenue du Président Kennedy, F-75016 Paris
[†]E-mail: firstname.lastname@lip6.fr

made during system design may not be optimal for all kinds of application [15]. For quality of service requirements, the system policies need adaptations at a fine granularity (such as a method). (3) Security: installing security patches typically requires a lengthy reboot, but in some applications, such as commercial application servers, a service interruption is not acceptable.

Simple plug-in mechanisms, such as loadable device drivers in Linux, allow an administrator to update a piece of code, but this technique is insufficient for changes that cut across previously loaded code, as in the above examples: the Linux kernel itself can not be changed and an adaptation implies unloading the plug-in and therefore the loss of its state. Published hot-swapping mechanisms as in K42 [28, 44] let the state transfer between components be handled by the developer. For changes that imply updates in many components, the state transfer can be an arduous task.

Our proposal is the addition of an aspect weaver to the component updating mechanism. The core of our system is the *Micro Virtual Machine* (MVM[†]) associated with a just in time compiler (JIT). The MVM is extended on the fly by adding new code or replacing its code dynamically. Components are used for code injection (see Sec. 2.1) but code replacement relies on our dynamic compiler coupled with a dynamic aspect weaver [27] (see Sec. 2.2). A predefined extension, called the JnJVM provides a full JVM. Applications can further extend and modify this base. The MVM is a common core for different virtual machines: a complete CLR is also developed on top of the MVM. The CLR is not presented in this paper[‡] because this paper focus on dynamic code evolution but not on VM interoperability.

System code is organized as a component architecture with associated adaptation points. Adaptation consists of compiling new source code and injecting it into the running components (either by extending or replacing methods) without losing their state and inserting new fields in components. This mechanism is inspired by aspect oriented programming and aspect weaving [27]. Of course, if the software architecture is modified and components are split or merged, the state of every obsolete component must also be transferred to the new architecture. Our adaptations focus essentially on binding new components without major modifications: oldest components are extended, but the software architecture of oldest components remains globally the same. When state transfer is furthermore required, we let the developer handle this transfer.

Compared to dynamic software updating [24], we focus on running code evolution instead of source evolution: we construct code evolution from a representation in memory of the running process independently from source code. MVM relies on runtime reflection whereas dynamic software updating relies on source code comparison. Especially, our adaptation system does not impose a unique language: applications developed in different languages are composed at runtime, thanks to the abstract representation. Original and adapted code are not necessarily developed in the same language and developing the new adapted code does not require access to the original code: a piece of Java code or Microsoft Intermediate Language (MSIL) code can replace native MVM code or a piece of Java code can replace MSIL code. Moreover,

[†]Our MVM is unrelated to the Multi-tasking Virtual Machine.
[‡]Another paper that describes VM interoperability is submitted at [19].

our adaptation system is developed only once for all languages, whereas semantic patches generation of dynamic software updating must be developed for all languages. On the other hand, dynamic software updating provides a general solution to state transfer, whereas we avoid this problem with AOP but we do not provide an automatic solution when state transfer is furthermore required.

The main contributions of this paper are the following:

- Design and implementation of a minimal, dynamically adaptable execution environment, the Micro Virtual Machine. It defines our component architecture and our aspect weaver. The Micro Virtual Machine is the main novelty of our research: it defines a common layout for building dynamically adaptable systems. As far as we know, there is no other work that tries to build a runtime dedicated to the construction of dynamically adaptable systems.
- Design and implementation of the JnJVM, a flexible Java Virtual Machine (JVM) which extends the Micro Virtual Machine. The JnJVM is our main realization of a flexible runtime based on the MVM. The goal of our work is not the construction of a flexible Java Virtual Machine, but the construction of an environment dedicated to the construction of flexible systems. The JnJVM is only a complete applicative example that demonstrates the flexibility of the MVM.
- Demonstrate different useful specialization scenarios: (i) modifying the behavior of the JVM, (ii) adding capabilities to the MVM and JVM, and (iii) modifying the execution behavior to applications above the JVM.
- Performance evaluation of the JnJVM. It shows that the JnJVM performs roughly the same as the existing open-source JVM Kaffe for an equivalent engineering effort. Most of all, JnJVM is compiled on the fly by the MVM when it is loaded. Our performance study shows that this level of virtualization has an acceptable cost in term of execution speed and memory consumption. Our evaluation also shows that our prototype is not optimized and performs 2 to 10 times slower than robust JVMs (IBM and Sun). Better performances of robust JVMs are explained by their Just In Time Compiler which optimizes the code. With a better JIT, we expect similar performances.

Our system is based on our minimal core, the Micro Virtual Machine, which contains a JIT, a dynamic loader, and a fine-grain dynamic binding mechanism. The MVM can run either atop a standard Unix or atop Linux. The JnJVM is itself a substantial extension of the core, and runs standard Java applications and benchmarks. We report here on a number of further specializations, both to the core and to the JnJVM. They demonstrate how applications can extend the core to accommodate to new requirements.

Our study focuses on a JVM for practical reasons but the lessons learned are applicable to execution environments in general. Indeed, a Java virtual machine acts as a small operating system for Java applications and provides a powerful experimental environment, which leaves aside hardware-related issues.

The main strength of the MVM is its independence from the targeted execution environment or language. We have also built a CLR over the MVM [19], a new middleware [35], a framework to adapt Jonas and OpenCCM [22] on the fly, a flexible Web cache [36], a flexible viewer for documents [45], and an execution environment for active network [12]. The MVM imposes only minimal abstractions by using a flexible language inspired from Lisp and gives all required abstractions to modify the execution environment on the fly. The technique used to modify execution environment on the fly relies on aspect weaving and is not new by itself, but its application in a generic execution environment that imposes only minimal abstraction coupled with a flexible language is an innovation. In this paper, we only present our Java virtual machine in order to study step by step the construction of a concrete execution environment with the MVM.

From a conception point of view, compiling the Java Virtual Machine in another virtual machine (the MVM) on the fly is original and our performance evaluation shows that this level of indirection does not seem to slow down the global performance of the JnJVM.

Some of the experiences reported in this paper were partially presented in other papers. A first version of the MVM without its aspect weaver was presented in [38], a first version of the JnJVM with partial performance evaluations and first results on escape analysis in [46], a partial implementation of the Java aspect weaver without technical algorithms in [37] and an extended version of the specialization of dynamic load balancing in [21]. The experience reported in this new paper completes the previous with new results and gives a global overview of our work on dynamic adaptability.

The outline of this paper is the following. Section 2 describes the MVM and the JnJVM. Section 3 presents our first example, optimizing allocation and synchronization for non-escaping objects. In Section 4 we present our experience extending the system with a Java aspect weaver. Section 5 presents our final example, a specialization for load balancing. In the next section we study the performance of the JnJVM (Section 6). Section 7 contains a discussion of the results, limitations and future work. Section 8 assesses related work. Section 9 concludes the paper.

2. MVM and JnJVM

The JnJVM is implemented on top of a minimal execution environment. This environment, the Micro Virtual Machine (or MVM), contains the usual core components of a virtual machine (see Section 2.1). We define a component in the MVM as a piece of code that implements an interface. A component is also our deployment unit. At runtime, a component is simply an object (a set of fields) which implements the interface.

The MVM loads and compiles code on the fly. We define specializations as the code loaded by the MVM. A specialization extends, modifies or removes components of the MVM dynamically for a particular need. The three kinds of specializations are discussed in Section 2.2. One of the specializations is the JnJVM, a complete Java Virtual Machine: the JnJVM specializes the MVM with a fully functional JVM. The runtime can accept further dynamic specializations. Sections 3, 4 and 5 presents three specializations for the JnJVM.

The primary loader of the Micro Virtual Machine is a small lexer/parser for a Lisp-like language. We discuss this choice in Section 2.1.1.

Figure 1 shows a snapshot of a specialized JnJVM. The MVM is extended with the components of the JnJVM. A standard Java Virtual Machine is therefore constructed in

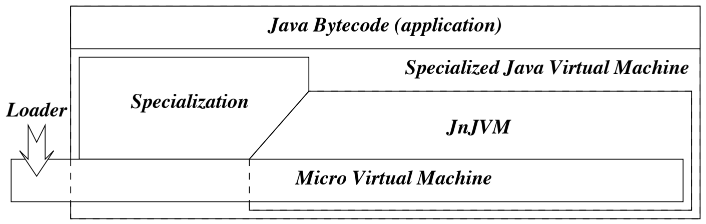


Figure 1. Global architecture of the MVM and the JnJVM

memory and executes standard Java classes. MVM can then load specializations and specializes the JnJVM on the fly, without interrupting the running Java application.

2.1. The Micro Virtual Machine

The MVM is a minimal execution environment. In addition to usual virtual machine components such as a memory manager, a thread manager, a just in time compiler (JIT), a lexer/parser for a Lisp-like language and an input/output layer (which relies on a standard libc library), the MVM defines components that will enable its specialization: an extensible compiler, a component manager and a code weaver.

Figure 2 summarizes our architecture. The code is loaded by the input/output component. The lisp front-end then constructs abstract syntax trees from the code. The abstract syntax trees are compiled by the extensible compiler, which uses the aspect weaver to construct adaptation points, the Just In Time Compiler to build native code and the component manager to build representations of loaded components. Adaptation points, native code and component descriptions are stored in components managed by the memory manager. All shared objects are synchronized thanks to the thread manager.

We choose a Lisp-like language as the primary language for the MVM for its simplicity and for its extensibility. The simplicity of the Lisp parser allows us to obtain good performance and the extensibility of Lisp-like languages permits the definition of new keywords by manipulating abstract syntax trees (see Section 2.1.1). This second point is essential in order to build the Java aspect weaver (see Section 4): new keywords are constructed on the fly so as to adapt assembled code. Two aspect weavers are defined: the first one is the MVM aspect weaver used to adapt the JnJVM, the second one is a Java aspect weaver used to adapt running Java applications.

The component manager of the MVM separates component interfaces (the definition) from their implementations (the code). Replacing a whole component is one mechanism for adaptation, but it implies the state transfer for this component. Most of the time, an adaptation does not imply the destruction of old components, but only their adaptation. Therefore, we introduce a *dynamic code weaver*. Our code weaver uses Aspect Oriented Programming (AOP,

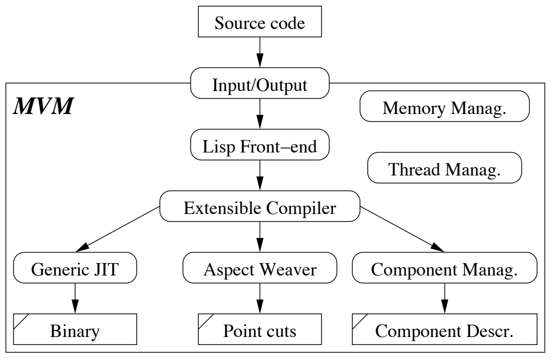


Figure 2. Internal architecture of the MVM

see Section 8) and aspect weaving mechanisms. It replaces code in well defined points of a program, called *join points* with the AOP terminology. Join points include calling methods, accessing fields and throwing exceptions.

The *Just in Time Compiler* of the MVM, called the Virtual Processor Unit (VPU) [40], assembles the code of components in memory. In order to interact with the MVM and more generally with C libraries, the VPU generates code following the local C ABI specification.

The *Aspect Weaver* of the MVM generates the join points for every methods. It is separated from the JIT to let the JIT independent from the aspect weaver.

The *extensible compiler* is used to define new keywords. A keyword is a symbol that compiles its s-expression itself (see Section 2.1.1). A specialization script can define its own language tuned to its domain. The keywords of this language are defined dynamically. For instance, it can be a high-level language that hides the internal implementation of specializations. The extensible compiler is used in Section 4 to define new keywords for the Java aspect weaver or in Section 2 to define new keywords for the JnJVM just in time compiler.

The MVM defines the basic components that are needed by the rest of the system: a *thread manager*, a *memory manager* and an *Input/Output manager*. The thread manager provides thread primitives such as thread creation, synchronization, condition variables and local thread memory. The memory manager allocates and collects objects. Its garbage collector uses the concurrent colored mark and sweep algorithm of Dijkstra et al's [14] based on Boehm's [11]. The Input/Output manager forwards I/O to the libc library. The MVM runs above any OS that provides the libc. Alternatively we provide a small subset of the libc to execute the MVM on top of Think [17].

In order to guarantee a stable system base, the component manager, the memory manager and the dynamic code weaver can not be modified after startup. However, nothing prevents us from replacing statically one of these component with another during compilation. All other internal MVM components may be subject to dynamic adaptation.

```
(define .inc−v1
  (lambda(x)
    (+ x 1)))

(define (syntax .inc−v2)
  (lambda(formal compiler)
    ‘(+ ,(:object.list.at formal 1) 1)))

(define (syntax .inc−v3)
  (lambda(formal compiler)
    (let ([vpu (:compiler.vpu compiler)])
      (:compiler.emit compiler (:object.list.at form 1))
      (:compiler.vpu.ld−int vpu 1)
      (:compiler.vpu.add vpu)
      0)))
```

Figure 3. Definition of new keywords

2.1.1. The Lisp-like Language

The MVM defines a new Lisp-like language. We choose this language for two reasons:

- Its simplicity. Only eight objects are used in abstract syntax trees (string, integer, long long, float, double, namespaces, symbols and list). A list is the abstract syntax tree of an s-expression.
- Its extensibility. Specializations can define new keywords on the fly. A keyword is a symbol called by the MVM compiler to compile an s-expression. A keyword can compile an s-expression by modifying it (i.e. macro) or by using the Just In Time Compiler of the MVM directly.

The Lisp-like language of the MVM is only used for its syntactic properties: by using quotation (quote, quasiquote and unquote), manipulating an s-expression is only the manipulation of a list. A list is directly used by a keyword. Thus, the simplicity of our Lisp-like language eases the development of new keywords. In contrast, our Lisp-like language is not a functional language: functions may have side effects.

Extensibility is used to define new keywords on the fly. For example, the MVM does not implement lazy binding whereas the JnJVM does. Thus, a new keyword is defined by the JnJVM on the fly to call Java methods. This new keyword generates a code that compiles on demand Java bytecode.

For the MVM, the semantic of an s-expression is purely imperative. A function is statically called with parameters. The object invocation semantic is defined in the Lisp-like language: new keywords are written in the Lisp-like language to generate virtual method invocations.

Figure 3 gives an example of our Lisp-like code. The first definition defines a function `inc-v1` with a traditional lambda expression that increments the parameter `x`. `Inc-v1` is compiled on the fly as a function. `Inc-v2` and `inc-v3` define new keywords. To define a new keyword, a symbol must be defined as a `syntactic` symbol. The first declaration is a macro: it replaces the abstract syntax tree `(inc-v2 value)` by `(+ value 1)`. The formal parameter is the original s-expression. `Inc-v3` uses the Just In Time compiler directly (see section 2.1.2). It emits the abstract syntax tree and increments it by using low level VPU functions. All predefined keywords of the Lisp-like language, such as `define`, `let` or `lambda`, are defined as syntactic symbols. Internal or applicative keywords use exactly the same mechanism. An extension can thus modify an internal keyword by redefining its syntactic symbol.

2.1.2. The Virtual Processor Unit

The Virtual Processor Unit (VPU) is the Just In Time Compiler of the MVM. It provides an interface to compile functions on the fly. The VPU is a stack-based machine: it transforms a sequence of actions on a stack to an equivalent machine code which uses stack and registers. The VPU compiles code for PowerPC and IA-32. Therefore, the MVM uses two main intermediate representations: abstract syntax trees and linearized sequences of action on the stack of the VPU.

The programming interface of the VPU is relatively standard. It defines loads and stores on stack, arithmetic operations (integer and float), flow operations, loads and stores on memory and loads and stores on local variables. The VPU interface is directly used from the Lisp-like language through syntactic symbols. This property is used to build the Java bytecode compiler of the JnJVM.

The new keyword `inc-v3` of Figure 3 shows the definition of a new keyword that uses the VPU directly. At line 12, the second element of the s-expression is emitted to the compiler. The compiler generates the VPU code of this abstract syntax tree (AST) and lets the result on top of the VPU stack. At line 13, the integer value 1 is pushed on the stack and at line 14, the two values on top of the stack are popped, added and the result is pushed on top of the stack. The VPU records all this sequence of actions. The MVM then calls the `compile` function of the VPU, which generates the equivalent machine code in memory.

The main weakness of our prototype comes from the VPU. The VPU does not optimize the code: all local variables live on stack, constants are not propagated, common sub-expressions are not eliminated, array bound checks are not eliminated, exceptions rely on setjmp/longjmp and so on. The weakness of the VPU explains why the JnJVM is slower than modern JVMs. As far as we know, there are only three generic JITs: the VPU, LibJIT [4] and LLVM [29]. LibJIT seems to have the same limitations as the VPU. We have replaced the VPU with LLVM and first results show that generated code with LLVM performs as fast as IBM and Sun for Java bytecode. The compilation time of LLVM is furthermore twice slower than VPU. A paper on this subject is submitted in [20].

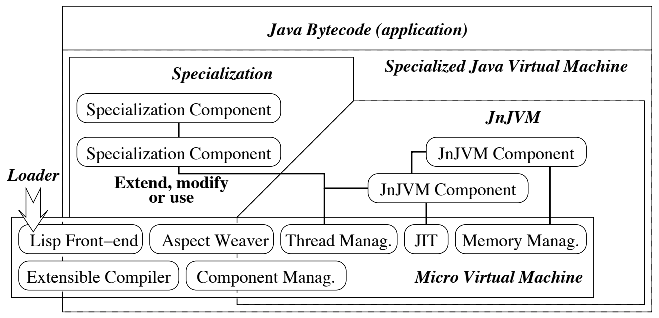


Figure 4. Specialization of the MVM

2.2. Adaptability of the Micro Virtual Machine

The component architecture and the dynamic code weaver provide performance and extensibility. The implementation of components uses virtual tables to hold pointer of methods: this structure is one of the fastest structures for virtual calls because only two memory reads are needed in order to find the method pointer. This structure is the same as the structure used by a C++ compiler for virtual calls and therefore it offers the same performance when call are not inlined. The extensibility mechanism relies on three mechanisms that we now detail.

2.2.1. The component structure: loading code extension.

New code is injected by loading new components in the MVM. An extension (or a specialization) is a set of components. They are loaded on the fly by the input/output component (see Figure 4)[§]. The input/output component is executed in the current thread. Loading a new specialization does not interfere with running threads and does not require synchronization: a new lexer instance, a new parser instance, a new compiler instance and a new VPU instance are allocated. Furthermore, accesses to the symbol table are synchronized.

A component holds methods, fields and implements an interface. The extensible compiler constructs a virtual table (a table of code pointers) filled with the implementations of the methods of its interface and the JIT compiles the code of the components. An instance of a component is thus represented in memory by its fields and a pointer to its virtual table. Figure 4 shows the loading of two specializations: the JnJVM and another specialization. These components extend the MVM and then the JnJVM and can use loaded components.

[§]Three input/output components are defined. The first one launches an interactive shell, the second one waits for incoming new specializations from a socket and the third one reads a specialization from a file.

```
 1  ;; ——- define an interface with two methods f and g —————
 2  (define .Ifce (def−interface "interface name" .f .g))
 3  ;; ——- define a component that implement Ifce —————————
 4  (define .Comp
 5    (implement "component name"
 6      ;; interface implemented
 7      .Ifce
 8      ;; methods implementation
 9      [(lambda(this x)      ;; implementation of f
10         (printf "; call f on %p/%d\n" this x))
11       (lambda(this)        ;; implementation of g
12         (printf "; the-field: %d\n" (the−field this)))]
13      ;; fields of the component
14      [word   the−field]))  ;; e.g. signed int the-field
15  ;; ——- allocate an instance of Comp ——————————————
16  (define .i (cnew Comp)) ;; i is an instance of Comp
17  ;; ——- call i.f(42) —————————————
18  (Ifce.f i 42)
```

Figure 5. Injection of a component

For example, the JnJVM defines a new Java bytecode compiler (extension) which uses the JIT of the MVM.

Figure 5 line 2 presents an example of an interface definition with methods `f` and `g`. A component `Comp` is defined from line 4 to 14. It implements the methods and holds one field, `the-field`. An instance of the component is then allocated (line 16) and `f` is called (line 18) on this instance. Previously loaded code does not call the newly-loaded extension, as it requires rebinding to the new code, which is explained in the next subsection.

Our component layer remains very simple: a component can only implement one interface or it can only inherit one component. Moreover, a component can not inherit a component and implement an interface simultaneously. We made this choice for two reasons:

- Because we want to keep our component layer as fast as possible. With this restriction, virtual tables can directly be used for interface method invocations.
- Because we also want to use the component layer for MVM internal components. However, the MVM is written in C++, which does not permit dynamic adaptation of the code. An internal component of the MVM is defined as a C structure with a virtual table. Managing this structure is relatively simple in C++. Building a more complex component layer requires many C++ lines of code for each component access. We estimate that the engineering cost of rewriting the MVM is prohibitive, whereas our component layer is sufficient for our work.

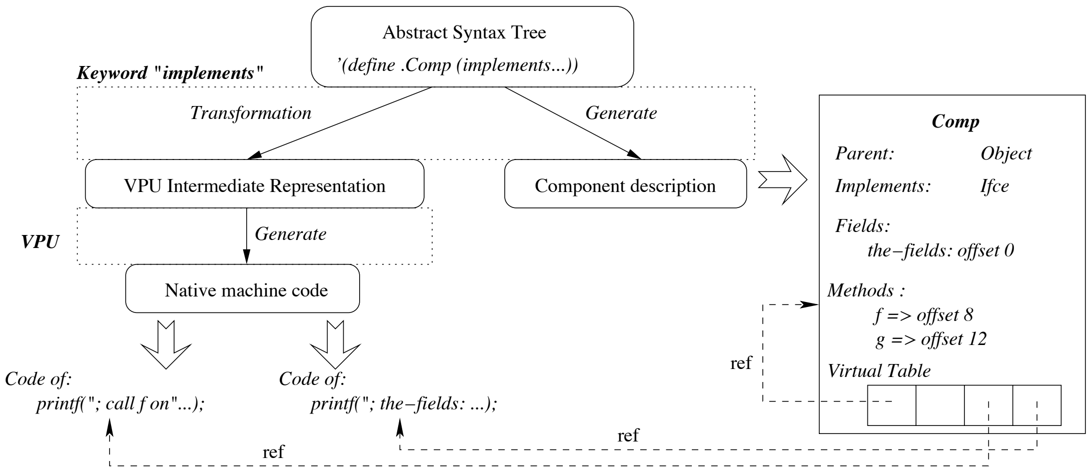


Figure 6. Compilation and reflection.

2.2.2. *The code weaver: binding extension code.*

Modifying existing components of the MVM on the fly relies on a code weaver. For each method of an interface, the aspect weaver defines a join point, i.e. an adaptation point in the AOP terminology. In Figure 4, binding an extension code is symbolized by the term *modify*.

Figure 6 summarizes the compilation of the abstract syntax tree of Figure 5. The keyword `implements` takes the abstract syntax tree of the component definition. It generates the code of methods `f` and `g` thanks to the VPU and generates a representation of the component `Comp` in memory. This representation is used to introspect and to adapt the component on the fly.

Moreover, at line 2 of Figure 5, two join points (`Ifce.joinpoint.f` and `Ifce.joinpoint.g`) are generated during the definition of the interface. A join point is defined by a new symbol used to modify a method of the virtual table of a component: this symbol is used to bind and therefore replace the existing code with the new code. A join point is a keyword that hides the direct manipulation of the component description. In Figure 7, we use the join point `Ifce.joinpoint.f` to modify the virtual table of the component `Comp` on the fly: the compiler inserts a new code in the entry `f` of the virtual table of the component definition `Comp`. A specialization can also modify the content of a field to bind a new component to an existing one.

A method modification does not interrupt the execution of the old code. If a method is executing while it is changed, the old code is still executed. Future calls will execute the new method. However, the application might be in an inconsistent state if the old and

```
;; ---- modify on the fly f ----------------------------------------
(set! (Ifce.joinpoint.f Comp)
  (lambda(this x)
    (printf "; Hi!, it's the new f!\n")))
```

Figure 7. Modification of a component

the new methods are executing concurrently. We do not believe that a common pattern to ensure consistent state transition is the right solution: we let the developer ensure that his synchronization is functional. If synchronization is required during adaptation, MVM defines a function that interrupts temporarily all executing threads (by issuing a signal) except the thread that initiates the adaptation.

Adding new fields to components is also possible, but with restrictions. This task is much more complex, as all instances of the component must be found in memory and reallocated. The system must then update all references to this component. To find all these references, the MVM relies on the garbage collector, which has a list of all live objects. Two passes on objects are done. The first one finds all the instances of components that implement or inherit the modified component. All these old instances are stored in a list and a new version is allocated with the new size. The second pass updates all references from the old instances to the new ones. This pass uses the meta description of components, which is stored in the virtual table of components. If a component B references a modified component A, after this pass, the component B references the new version of A. Finally, all compiled functions that access a field of an inherited component of the modified component must be recompiled. Indeed, if a component B inherits a component A, fields of B are stored after the fields of A in an instance. If A grows, offsets of fields of B change. Because these offsets are directly used in the machine code to access fields, functions that access fields of B must be recompiled. Maintaining the list of these functions for each field is relatively costly in terms of memory and an option enables or disables the capability of adding new fields.

Figure 8 gives an example of field insertion. In this example, we add the new field `a-new-field` in the component that describes a Java class in the JnJVM. This field is initialized by the lambda-expression to 42. We have tested this example on a running JnJVM with an object memory of 5MB. On a Linux on a Core 2 Duo at 2.33GHz executed in the Parallels virtual machine and 768MB of memory, adding this field takes around 3 seconds. This first result shows that adding fields is relatively costly. The capability of adding fields do not cause overhead during execution but requires memory to store field accesses.

We have two restrictions on field addition:

- If a modified component is directly referenced from the compiled code, this reference is not updated because offsets of these references from the code are lost during the compilation.

```
(add-field :jnjvm.class.class-def
           [word .a-new-field]
           (lambda(cl)
             (:system.printf "; add field on: %p (%s)\n" cl (:object.print-string cl))
             (set! (.a-new-field cl) 22)))
```

Figure 8. Adding a field in a component

- If a modified component is referenced from the stack, this reference is not updated because the type of local variables is lost during compilation.

Avoiding the first restriction requires modifications directly in our VPU and does not require a lot of work. Avoiding the second restriction requires also modifications in our VPU to maintain a description of the stack, but also in all the MVM code. Indeed, if a component is referenced from MVM code, which is written in C++, we can not have the description of the stack. We are actually working on a new prototype of the MVM written in the Lisp-like language of the MVM to solve this problem. By using our own compiler we will maintain a description of the stack for all functions.

2.2.3. Deleting unneeded code.

After an adaptation, component definitions or methods may become unneeded. In this case, the old component or the old method will become unreachable. The unused memory will be reclaimed by the garbage collector. Even if a method is executing but is not referenced by a component after an adaptation, the garbage collector reaches the old code because it finds a reference to it on the stack or in the program counter. The reclamation of the unused code will therefore be delayed and the execution of the method will finish normally.

2.3. Internal architecture of JnJVM

The JnJVM is a complete Java Virtual Machine that conforms to the second version of the Java Virtual Machine specification [31]. It is based on the bootstrap classes of GNU Classpath [1], which follows version 1.2 to 1.4 of the Java API Specification.

The JnJVM is developed on top of the MVM and uses its component architecture. Implementing the JnJVM atop the MVM allows specializations to adapt JnJVM components on the fly: the JnJVM is extensible since it is built upon the MVM. The JnJVM is thus written in the Lisp-like language of the MVM: the MVM compiles the JnJVM on the fly when it is loaded. Executing a complete virtual machine (JnJVM) in another (MVM) and obtaining good performance is one of the challenges of our work. From a reflective point of view, the MVM executes at the base level, the JnJVM at the meta level and Java applications at the meta-meta level.

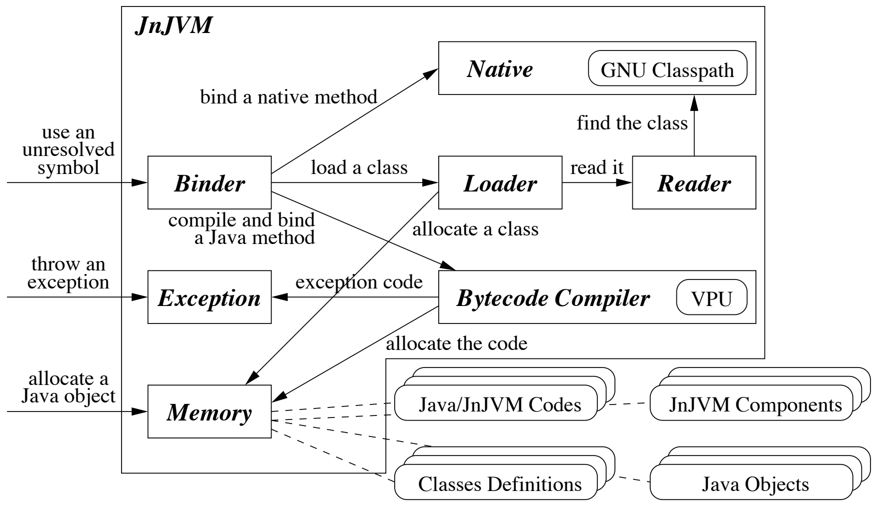


Figure 9. Relationships between JnJVM components

The JnJVM adds an entirely new functionality to the MVM, therefore loading it does not require any code weaving. Because the JnJVM is written in the Lisp-like language of the MVM, it is portable for different architectures and systems (the MVM itself has to be adapted for different architectures and systems).

Figure 9 summarizes the relationships between the components of the JnJVM. The three incoming arrows symbolize the calls generated by the Java code during the execution of the application. Each component of the JnJVM implements one of the standard mechanisms of a JVM. The reader and loader components read, resolve and initialize Java classes in memory.

Each element of a class file (a field definition, a method definition, a class definition or an attribute), is represented in memory by a MVM component instance. These components are collected by the garbage collector.

2.3.1. The Java Just In Time Compiler

The bytecode compiler component compiles the bytecode of a Java method: this component, the Just In Time Compiler (JIT) of the JnJVM, reuses the JIT (the VPU) of the MVM (see Section 2.1). The bytecode compiler of the JnJVM is therefore portable because it uses only the VPU interface. We use directly the API of the VPU to compile Java bytecode on the fly. By using the same technique, we can compile other bytecoded languages. A first version of the CLR built over the MVM is functional and reuses also the VPU [19].

```
 1  (def−opcode .irem 112
 2    (lambda(op code−reader position compile−env compiler vpu)
 3      (:compiler.vpu.imod vpu)
 4      (+ position 1)))
 5
 6  (def−opcode .invokevirtual 182
 7    (lambda(op code−reader position compile−env compiler vpu)
 8      (let ([index     (read−u2 code−reader)]
 9            [jnjvm     (env.vm compile−env)]
10            [cl        (env.resolve−class jnjvm compile−env)]
11            [signature (generate−invoke−interface compile−env vpu (:jnjvm.class.constant−pool cl) index)])
12        (+ position 3))))
```

Figure 10. Mapping Java bytecode to VPU API

Figure 10 gives two examples of mapping. A new keyword is defined in the MVM to map Java bytecode to VPU intermediate representation: `def-opcode`. This keyword compiles the translator method (the lambda-expression), fills the appropriate entry of the dispatch array of Java opcode translator (for the first example, the instruction `irem` has the opcode 112) and preserves a symbol for debugging (`irem` or `invoke-interface` in examples).

The first example is the opcode `irem` which computes the remainder of the division of the first argument on the Java compilation stack by the second one. The mapping is direct: the VPU provides an integer remainder function and also uses a stack. The second example translates the `invoke-interface` Java opcode. At line 8, the index in the constant pool of the invoked method is extracted from the bytecode. At line 9, the current instance of the JnJVM is retrieved from the compile environment component. At line 10, the class is resolved by the JnJVM, i.e. a class component is found from the String which represents the class. At line 11 the code to invoke a Java interface is generated. The mapping is not direct because the VPU provides only two function call semantics: direct and indirect call. The function `generate-invoke-interface` generates the code with the VPU to invoke an interface by generating dispatch code.

2.3.2. Other components

The binder associates a Java symbol to an internal representation of this symbol. Binding is lazy: methods, fields and classes are bound when they are used during execution. The exception component manages exceptions. The memory component uses the garbage collector of the MVM, but other garbage collectors can be implemented. The native component implements the interactions between the JnJVM and GNU Classpath.

Once the JnJVM is loaded, Java applications can run. Applications may replace components of the JnJVM or invoke the MVM code weaver to introduce or modify components.

There can be several instances of the JnJVM; they can be isolated from one another or they can share components. A first implementation of isolates [3] is under development and preliminary work on isolation based on different instances of JnJVM in the same address space is submitted in [19]. In section 4, we use multiple JnJVMs to load a new version of a bytecode. The original JnJVM instance has the old version of the class and the new instance loads the new version, compiles the new code with its constant pool and replaces the old method with the new one.

The main originality of the JnJVM is the use of MVM components which enable their dynamic adaptation. Internal components of the JnJVM are relatively standard and algorithms used are well known in the virtual machine developer community, hence we do not detail them further in this paper.

To demonstrate the extensibility of the JnJVM, the next three sections present adaptation experiments that show three different levels of adaptability. The experiment with escape analysis illustrates *modification* of JVM internals such as bytecode compilation and thread synchronization. The second experiment illustrates *extension* of the virtual machine with a Java aspect weaver. The third experiment *extends* the JnJVM with system services that are not application specific and *modifies* the application with distribution and monitoring.

3. Specializing the JnJVM for non-escaping objects

The first experiment modifies the behavior of the JVM. It consists of specializing allocation and synchronization for non escaping-objects [13, 9, 46]. Escape analysis calculates the lifetime of Java objects. A captured object is defined as an object whose lifetime does not exceed the lifetime's method in which it was allocated. Otherwise, the object escapes from the method. Captured objects do not need garbage collection nor synchronization.

A static lifetime analyser finds captured objects (e.g. objects whose lifetime does not exceed the method). The static analysis was done by A. Galland [18] with GemPlus[¶] to optimize Java applications for embedded devices. Class files are annotated by the analyzer to list captured objects. This information is then exploited by the escape specialization script.

The escape specialization modifies on the fly the JnJVM to allocate captured objects on a stack called *captured stack*[‖] instead of putting them in the heap. The script also removes synchronizations on captured objects. An uninstallation script restores the old behavior on demand.

The main goal of this experience is to demonstrate the dynamic adaptability of the JnJVM. We choose escape analysis because it modifies the behavior of the garbage collector and the synchronization model. It requires modification in the core of a Java Virtual Machine: in the memory allocator, the garbage collector and the code generator. Adding this new behavior on the fly does not slow down the JnJVM: JnJVM is launched normally and adapted later.

[¶]The french smart cards constructor.
[‖]We do not use the execution stack to avoid stack overflow.

A standard Java Virtual Machine might be statically modified to provide escape analysis facilities. However, escape analysis is one possible system mechanism required by an application (such as persistence, migration, specific resource control...) and an application server can not predict at startup which applications will be executed and what possible specific system mechanisms will be required by these applications. Choosing specific system behaviors at startup becomes therefore impossible and dynamic adaptation is a solution to inject new system mechanisms on the fly, when it is required by an application.

3.1. The escape specialization script

The escape specialization script modifies methods of the JnJVM by using extension and modification mechanisms provided by the MVM. The escape specialization script weaves an aspect in the JnJVM to take into account escape information given by the analyzer. This aspect is defined by the join points (the methods of the JnJVM) modified by the script. Three kinds of join points have been identified and are modified by the escape specialization:

Allocation The compilation of bytecodes `new, newarray, anewarray` and `multianewarray`. For captured objects, they specialize instances with synchronizations removed and by allocating them on stack.

Method compilation The bytecode compilation of the prologue and the epilogue of methods. Captured objects are stored on the captured stack which must be updated at the end of the method that holds captured object. Figure 11 presents a simplified version of the code advice woven for the prologue and the epilogue. First (line 3 to 6), the state of the stack is preserved, then the original code is called (line 8 and 9) and finally (line 10 to 13) the state of the stack is restored before the return. The code advice is woven at lines 15 and 16, e.g. the compilation of the prologue and epilogue method is updated.

Exception throw When an exception is thrown, the captured stack must be updated to deallocate all unused captured objects.

3.2. Performance measurements

Figures 12 and 13 summarize measures observed with the escape specialization script. The evaluation was done on a PowerPC at 1GHz with a Gentoo Linux. To compare performances with and without escape analysis, we evaluated the execution time of four representative applications of the JavaGrande Benchmark suite [16]. The heap is limited to 512 MB and the stack for captured objects to 4 MB.

With the Search test, the number of results per second is increased by 25%. Indeed, most of the collections are avoided because most of allocations are captured. This test shows the potential benefit of escape analysis. The Euler test gives an intermediate result where the half of the allocations are captured. MonteCarlo test shows that the impact of escape analysis may be insignificant when most of the main allocations are not captured.

```
(define .java−compiler−escape ; define the code advice
  (lambda(code−input code−len opinfo env comp vpu)
    (push−locals vpu 1) ; create a new local variable
    (call vpu 0 get−captured−stack−pointer) ; get sp
    (st−tmp vpu 0)     ; preserve it in new local
    (drop vpu 1)       ; clean up VPU stack
    ; call the original method
    (java−compiler−orig code−input
                        code−len opinfo env comp vpu)
    (ld−tmp vpu 0)     ; get new local
    (call vpu 1 set−captured−stack−pointer)
    (drop vpu 1)       ; clean up VPU stack
    (pop−locals vpu 1))) ; delete the local variable

(set! (java−compiler (compiler (jnjvm)))
      java−compiler−escape) ; weave the aspect
```

Figure 11. The code advice woven at the prologue and epilogue

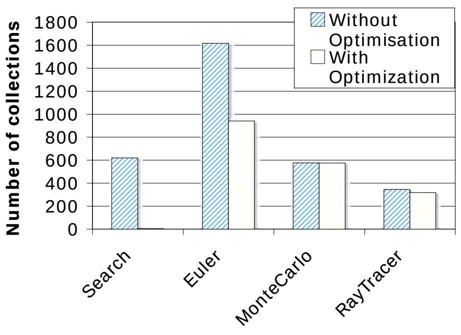


Figure 12. Number of collections

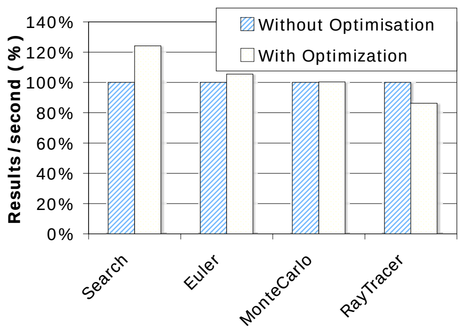


Figure 13. Acceleration of escape analysis

The last test, RayTracer, shows that escape analysis can decrease the performance. Most of the allocations are captured in this test and captured allocations are performed in a loop. The stack is filled rapidly and objects are then allocated with the MVM memory manager. The $4MB$ of the stack contain only unreachable objects that are followed during collections because these objects can reference gc-allocated objects. Our garbage collector is not generational and the 318 collections traverse the stack. Approximately 1.2 useless GB of memory is traversed by the garbage collector during the test: escape analysis degrade performance.

The four tests show the different behaviors under a new system policy: escape analysis is an optimization for some applications such as the Search and the Euler test but can degrade performance for inadequate applications: this result confirms that specialized system policies are not suitable for all applications.

3.3. Summary on escape analysis

Escape analysis illustrates the flexibility of the JnJVM. The specialization script remains small (approximately 1000 lines of code). It can be reused with another specialization of the JnJVM, (provided that they do not adapt the same behavior). The above measures of escape analysis show that our architecture make possible application-specific optimizations and improves performance for suitable applications.

The escape specialization script modifies three components of the JnJVM: Just in Time Compiler, memory management and thread synchronization. Because every behavior of the JnJVM is exposed at the specialization script level, the technique presented in this section can be used to deploy other technologies, such as Proof Carrying Code [34, 41] to accelerate type verification during bytecode compilation.

4. Extending the JnJVM with a Java-level aspect weaver

The second experiment extends the capabilities of the JnJVM. It consists of adding a Java aspect Weaver (see Section 8) that can adapt the application. This experience shows the extensibility of the JnJVM and illustrates potential of the Micro Virtual Machine for language extension.

To adapt the Java application, we developed a second dynamic aspect weaver [27, 39] ** at the Java level, similar to Steamloom [10] (see Section 8). The MVM aspect weaver is not reused for Java applications because it only weaves Lisp-like code to running components: Java classes have specific data that do not exist in MVM components. Weaving Java code also requires loading of new bytecode and thus new mechanisms.

Moreover, MVM components do not support multiple interface inheritance and simple virtual tables were not pertinent (see Section 2.2). Modifying the component layer of the MVM could be a judicious choice, but we preferred building a new object layer to avoid rewriting the MVM internal components, which would be a considerable engineering effort. Another reason is more pragmatic. We did not design the MVM to execute Java bytecode, but to adapt on the fly the MVM and its running components. At the MVM level, we can not predict all specificity of all possible virtual machine. Therefore, we do not want to impose a fixed object layer. Adding a new object layer in the MVM on the fly is thus the right solution: if we impose the Java object layer at the MVM level, we are designing the MVM as an execution environment only for JVMs, whereas adding the new object layer on the fly shows the extensibility of the MVM.

**The first aspect weaver is the MVM aspect weaver.

By transitivity, the aspect weaver of the MVM can not manage Java objects and we decided to extend the MVM aspect layer to support this new object layer. Moreover, adding an aspect weaver on the fly through our MVM aspect weaver shows the extensibility of the MVM and provides an experimental demonstration of the MVM extensibility.

We could also integrate the Java aspect weaver directly in the JnJVM, but it is not the point of the JnJVM: a dynamic aspect weaver is a specific system mechanism and must thus be external. Integrating all possible specific system mechanisms directly in a Java virtual machine is not a good solution because it complicates the code and consumes memory. Moreover, some system mechanisms can be incompatible and can not be present simultaneously. We argue in this paper that specific system mechanisms must be provided as tier components. We want to load the aspect weaver dynamically because we let the application choose its specific system mechanisms. The Java application requires a Java aspect weaver and thus loads this system component in its JVM: the end user does not have to choose which system mechanism will be needed by his application.

The specialization script, called the aspect specialization script, is loaded dynamically in the JnJVM. It modifies and extends the JnJVM to provide (i) a simple aspect language and (ii) a way to weave aspects in Java applications. The aspect weaver permits the weaving of Lisp-like code in Java code, but also the weaving of Java code in Java code. Weaving Java-code in the Lisp-like code is already possible with the JnJVM to invoke the `main` method of the Java application or to find the applicative class loader.

4.1. The aspect specialization script

The aspect specialization script is loaded in the JnJVM by the MVM. It defines a language to describe and weave aspects in Java applications. This language manipulates Java join points and associates code advice with these join points. Basically, the script encapsulates the internal component methods of the JnJVM that define and manipulate the state of class, method and field definitions in order to provide a higher abstraction to the developer. Figure 14 presents a Java application with its aspects and its aspect specialization script:

- The *Java binary application* is a standard Java application on which aspects will be woven;
- The aspect specialization script extends the language of the MVM with the *Java aspect language* that defines, manipulates and weaves aspects;
- The *Java aspect description* defines the Java join points and defines code advice that will be woven on these join points. The Java aspect description uses the Java aspect language.
- The *Java code advice* and *MVM code advice* are the code advice that will be woven by the Java aspect description. The aspect specialization script permits the use of both kinds of code advice. A MVM code advice interacts directly with the MVM from the JnJVM because the code advice can manipulate the internal elements of the MVM. A Java code advice replaces a Java methods by another, provided that they have the same signature and are members of the same class.

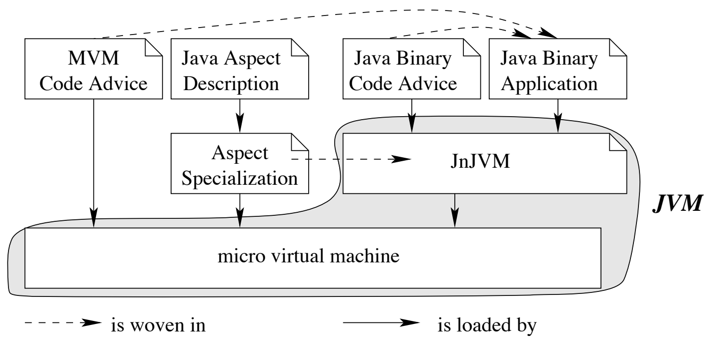


Figure 14. The Java aspect weaver of the JnJVM

The code advice, the Java aspect description but also the aspect specialization can be loaded dynamically during the execution of the Java application.

The aspect language defines new keywords to search classes, fields, methods, to duplicate methods, and to weave aspects on these methods and fields. In this experience, we have not developed the addition of new fields in existing objects. The solution provided for field addition in MVM components could be reused in our Java aspect weaver with the same limitations (see Section 2.1). To weave an aspect on a method, the aspect specialization script modifies the code pointer of the method and updates the different calling sites of this method.

4.2. Aspect weaving example

Figure 15 presents a complete sample of a simple aspect weaving. This code weaves a MVM code advice that prints which method is executed on the Java instance method `void Hello.m(int)`. First, the instance method `void m(int)` of the class `Hello` is stored in the symbol `m` (lines 1 to 5). At line 7, a join point named `cut` is defined for the method in the symbol `m`. The keyword `def-joinpoint-on` duplicates the method `m` with a new name and reallocates the original method definition so that it holds a reference to the original method, reachable with the keyword `get-original`. The function `m-advice` (line 9 to 15) is the code advice. It prints the name of the method and calls the original method. Finally, the aspect is woven (line 17).

The keyword `weave-lisp` at line 17 updates the code pointer of the method and updates all the potential calling sites for the method. This sample with Lisp-like code is not type safe because MVM does not yet provide a complete type model. Defining a complete type model is planned for the next version of the MVM.

```
;; ---- find a method (Aspect description) -----------
(define .m                 ;; store in the symbol m
  (get-method              ;; the method
   (get-class (jvm) "Hello") ;; in the class "Hello"
   "m" "(I)V" ACC_VIRTUAL)) ;; named "void m(int)"
;; ---- define a joinpoint (Aspect description) -------
(def-joinpoint-on cut m) ;; define a joinpoint named cut on m
;; ---- define the code advice (Microvm code advice) ---
(define .m-advice
  (lambda(cache this the-integer-arg)
    (let ([meth (deref-cache cache)]) ;; find the method
      (printf "; enter in: %s\n" (:object.print-string meth))
      ;; *** invoke the original method ***
      (invoke-virtual (get-original meth) value this)
      (printf "; get out: %s\n" (:object.print-string meth)))))
;; ---- weave the code advice (Aspect description) ----
(weave-lisp cut m-advice)
```

Figure 15. An aspect weaving example

4.3. Weaving Java code

Another keyword (`weave-bytecode`) weaves directly a new Java bytecode to a join point. To load a new Java bytecode, we must have two versions of the same class in the same class loader: the two classes must have the same fields, but the new class contains the new methods. If a new method is present in the oldest version of the class, the oldest version is replaced and if a method is not present, it is added to the oldest. The old and the new versions of the class do not share the same constant pool and the two versions must be present in memory. The two versions share the same fields and methods. The methods array is thus a mixed array with old methods and new methods. Old methods reference the old class and thus the old bytecode, whereas new methods reference the new class and thus the new bytecode. The JnJVM does not support the definition of multiple versions of the same class in the same class loader and can not manage mixed classes. To solve this problem, we allocate a new instance of the JnJVM for the new version of the class, which shares all JnJVM components with the original JnJVM. This new JnJVM can store multiple versions of the same class. When this version loads a new class, it updates the original one in the original JnJVM.

Figure 16 describes a class with old and new methods. Original methods use the old constant pool (Ctp) while new methods use the new constant pool. If a second new bytecode must be loaded, a third JnJVM is allocated and three versions of the class co-exist. The new JnJVM is small because it shares all the original components.

4.4. Summary on the aspect specialization script

The specialization script enables:

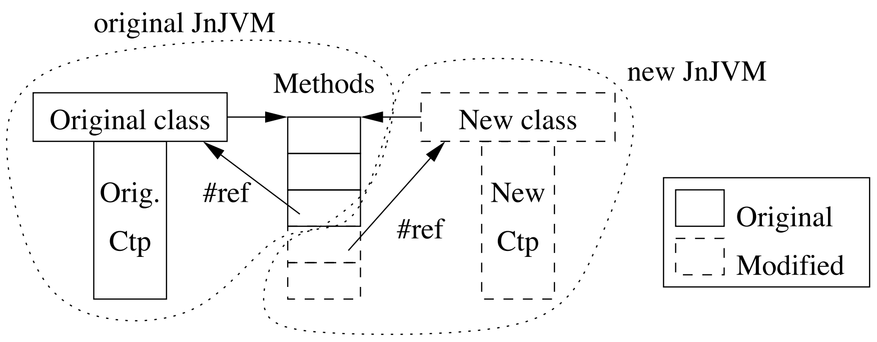


Figure 16. Weaving Java code

- Flexibility, as in Steamloom, by defining on the fly new join points.
- Good performance because a join point does not require an indirection to reach a code advice. Indeed, the internal structure of a Java method (the method descriptions, see Section 2.3) is used directly. Moreover, to weave an aspect dynamically, we do not require to define an unused join point like an aspect weaver at language or loading level would (see Section 8) .

The MVM aspect weaver allows an application to adapt the JnJVM for a specific need, whereas the Java aspect weaver allows an application to adapt the running Java code. The Java aspect weaver relies on the MVM aspect weaver, but also on the extensibility of the language of the MVM to define new keywords.

The purpose of the experience on Java Aspect presented in this section is not to design of a new complete Java Aspect weaver but to put forth the dynamic adaptability of the JnJVM. The Java aspect specialization script enriches the JnJVM with new keywords (`weave-on-*`) and adds a new functionality on the fly (Java code replacement). New keywords, such as equivalents of AspectJ keywords `after`, `around` or `before` (see Section 8) could be defined in the same way.

The aspect specialization script uses the internal mechanisms of the MVM to generate and replace the Java code on the fly. Our Java aspect weaver updates directly code pointer and there is no indirection to reach code advice. Therefore, executing a Java application with and without the Java aspect weaver does not change the execution time.

The aspect specialization script is fully functional and has been used to realize the integration of our aspect weaver with the application modeling tool Objecteering [5]. It uses the Java aspect weaver to adapt dynamically the execution of an application when its model (representation) in Objecteering changes.

5. Extending the JnJVM for transparent load-balancing

Our final experiment adds distribution capabilities to the MVM and the JnJVM on the fly. Some applications evolve in environments where their load or available resources can change at runtime. Pervasive computing and Web servers belong to these kinds of systems: pervasive computing aims at bringing software into all kinds of devices, which do not provide lots of resources, and web servers are prone to overload because the number of requests can not be predicted. A solution to exploit available resources is to offload entities of the application dynamically.

We have implemented method offloading on top of the JnJVM. The application is not modified but enriched with meta data. The application is not aware of the offloading system. It can be added dynamically during the execution of the application, when needed.

This experiment shows how adapting the JnJVM and the application. The JnJVM is enriched with the load balancing system which modifies the application to distribute the computation on a cluster. The load balancing system relies on the JnJVM aspect weaver to modify a local call to a remote call. With this last experiment, we highlight two interested points. First, our specializations can be composed: the load balancing system requires the Java aspect weaver. Second, this experience executes a complete real Java applications, Tomcat, and evaluate the completeness of our JVM.

5.1. System overview

Method offloading relies on (i) an offline analysis of which methods can be offloaded, (ii) the weaving of a monitoring aspect and (iii) the weaving of a distribution aspect. The offline analysis gives a set of methods that do not manipulate system-dependent objects. Currently, the analysis is done by the developer but we plan to automate this task. The monitoring aspect collects information on methods at run-time: execution time, number of times executed and size of the arguments. This information is used to choose which methods should be offloaded. The monitoring aspect is woven when an overload is detected and unwoven when the distribution aspect is woven.

The distribution aspect sends the execution of methods chosen by the monitoring aspect to underloaded computers by using the RMI framework: on every computer where computations will be sent, an RMI object is executed. This object can receive, load and execute bytecode. To update locally modified objects on remote peers, we have implemented a copy-restore algorithm [47] at the JVM level.

The peers are not dedicated to one application and can be shared with other applications. They can be personal computers or clusters. When the end of the overload is detected, the distribution aspect is unwoven and the execution continues normally. Because the Java aspect weaver does not introduce performance loss (see Section 4), the execution performance without aspects is not affected. On the other hand, when the distribution aspect is woven, a remote method invocation is executed and performance decreases. However, when the distribution aspect is woven, the load decreases and the response time increases.

5.2. Case study: load-balancing a multi-threaded Web server

We did a case study of our system with the TPC-W benchmark.TPC-W is composed of Java servlets that implement an interactive bookstore. It is executed on the Tomcat servlet container, version 5.0.28. The application contains 29 classes, 15 of which are servlets, 49 methods and 5409 code lines. A manual analysis of the code detects 21 methods whose execution can be migrated.

At first, the application is launched on the JnJVM without the offloading specialization. When the administrator decides that the application would need offloading, he can inject the specialization. The offloading system starts to monitor the application, and waits for computational peers to record themselves. When many users of the application are connected, the system disables monitoring and enables distribution of Java methods.

A first evaluation of the system shows that responsiveness and availability of the server are improved. The Web server is a $1GHz$ PowerBook G4 with $1GB$ of memory. In the first experiment, a dual-processor $G5$ $2GHz$ and a $3GHz$ Pentium register to the offloading system. In the second experiment, a cluster of 20 dual-processors Xeon $2GHz$ dynamically registers to the offloading system. We could not use a machine of the cluster to be the Web server because the JnJVM does not compile on x86_64. Figure 17 shows the average response time of the Web server proportionately to the number of clients. When 200 clients are connected to the server, the average response time is 1.4 times better with two surrogates, and two times better with the cluster. The small improvement between two surrogates and a cluster of surrogates is mostly due to the fact that the server still has to dispatch the requests. Hence, the performance of the server is the bottleneck. Note that the goal of this experiment is to show how the JnJVM can be easily adapted for an application-specific need; in this case load balancing. The performance results, while necessary, are still very limited to the JnJVM performance.

5.3. Summary on load balancing

Dynamic load balancing can improve performance of a Web server during overload and there is no performance loss when monitoring and distribution are not enabled. For a well-written application, which separates platform dependent entities from business entities, many of its methods can be migrated. Therefore no effort on rewriting the application has to be done.

Achieving dynamic offloading inside the JnJVM does not impose modifying the JnJVM code. The code is composed of 300 lines for manipulating the JnJVM components, and 2000 lines for the resource manager, manipulating the set of methods, and the call by copy-restore implementation.

6. Performance measurement of the JnJVM

In order to evaluate our prototype, we used the JavaGrande Test Suite [16]. It measures the performance of basic operations such as arithmetic operations, collection and exceptions, and it measures global performance with representative applications. The JnJVM performs as fast as the Kaffe virtual machine with a Just In Time compiler. The Kaffe virtual machine is an

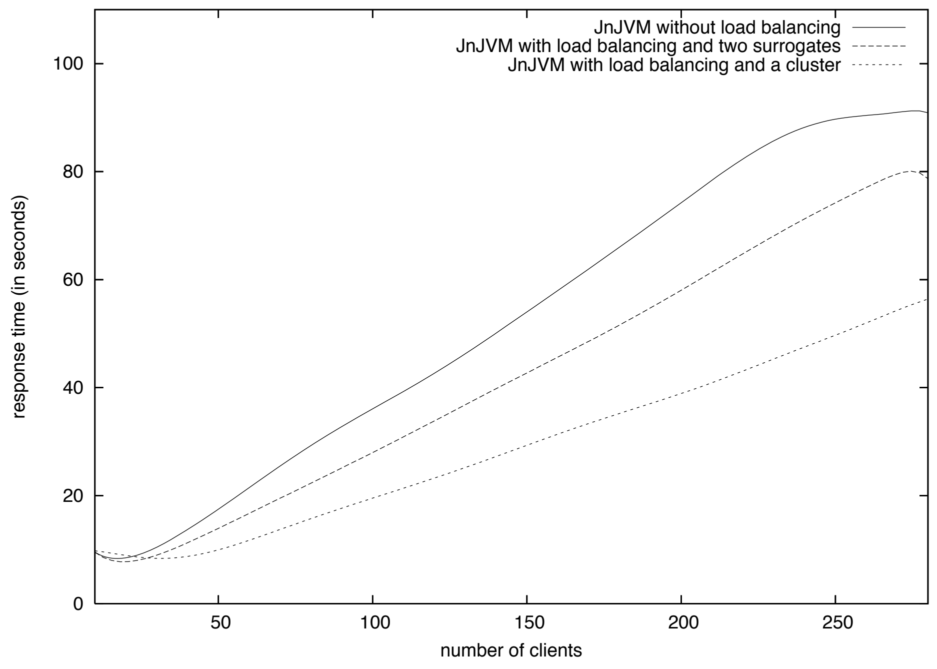


Figure 17. Average response time of the Web server (smaller is better)

open source virtual machine distributed under the GNU public licence. Kaffe was popular five years ago and was the default virtual machine shipped with most Linux distributions. In opposition to the IBM virtual machine or Sun virtual machine, Kaffe is not developed by an industrial. Kaffe is equivalent in terms of engineering effort to the JnJVM. However, JnJVM and Kaffe perform 2 to 10 times slower than the robust virtual machines from IBM and Sun. The remainder of this section describes and explains our results.

Our tests were performed on a PowerPC with a Gentoo Linux, kernel revision 2.6.15 at 1 GHz with 1 GB of RAM, and an Athlon XP 1800+ with Gentoo Linux 2.6.15 with 512 MB of RAM. On the IA-32 architecture, the JVMs we evaluated were: IBM JVM version 1.4.2 with a Just In Time bytecode compiler; this Java Virtual Machine is one of the fastest JVM [6]; Sun JVM version 1.4.2.10 with JIT; Kaffe 1.1.6 with JIT and Cacao 0.95, an open-source research JVM, which concentrates on JIT compilation.

On the PowerPC architecture, the JVMs we evaluated were IBM JVM version 1.4.2.03 with JIT, and Cacao version 0.95 with JIT. We have not reported the result of Kaffe and Sun on PowerPC because they do not have a JIT for this processor: comparisons with these JVMs only show that a JIT is better than an interpreter.

JVM	JnJVM	IBM	Sun	Kaffe	Cacao
Memory on IA-32 (MB)	25.6	12.7	20.2	4.6	5.3
Memory on PPC (MB)	26.6	11.7	-	-	5.6
Startup on IA-32 (s)	1.2	0.2	0.1	0.0	0.0
Startup on PPC (s)	2.1	0.3	-	-	0.0

Table I. JVM Bootstrap (smaller is better)

JnJVM comprises 15000 lines of MVM code and MVM comprises 30000 lines of C++ code. The amount of code is comparable with that of other JVMs.

We present a relatively complete set of results because the JnJVM is itself compiled on the fly by the MVM and we want to show that this property appears not to degrade performance.

6.1. Bootstrap performance

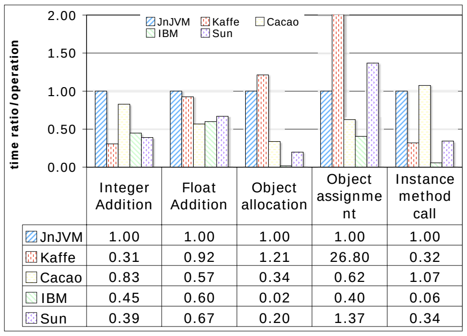
Because our system compiles the JnJVM when it is loaded, the bootstrap of the JnJVM is slower than other JVMs. Nevertheless, the loading time of the JnJVM remains reasonable: 1.2 seconds on IA-32 and 2.1 seconds on PowerPC.

The memory footprint of the JnJVM is larger than other JVMs. JnJVM keeps the meta components and interfaces in memory and the MVM has a compiler for the Lisp-like language that is not present in other JVMs. Table I indicates the total amount of memory used by the JVMs on IA-32. The JnJVM consumes between 10 and 20 MB. For modern desktop machines, this occupied memory is not a limitation, but for embedded devices, the memory footprint of the JnJVM has to be downsized.

6.2. Micro-benchmarks

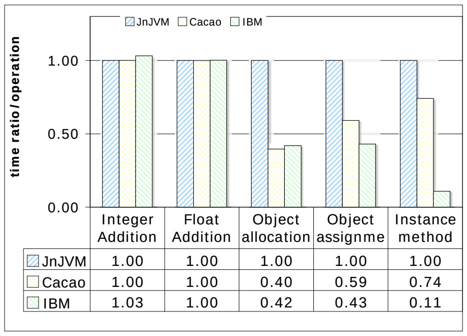
Micro benchmarks give information on micro operations such as addition or object allocation. We present these results because they explain the difference observed in the later macro benchmark (see Section 6.3) and show some limitations of our prototype. Figures 18 and 19 give a set of results of the JavaGrande section 1 benchmark. All results are normalized to 1 which is the constant result of the JnJVM. The result given is the average time of one operation. We concentrate on arithmetic operations, object allocation, object assignment and method calls because they are the most significant to explain the difference observed between the JnJVM and others for complete applications.

The Kaffe and Cacao virtual machines give the most significant results. Indeed, these virtual machines and the JnJVM are comparable in terms of engineering, but are not comparable with optimized Java Virtual Machines (IBM and SUN).



	Integer Addition	Float Addition	Object allocation	Object assignment	Instance method call
▨ JnJVM	1.00	1.00	1.00	1.00	1.00
▨ Kaffe	0.31	0.92	1.21	26.80	0.32
□ Cacao	0.83	0.57	0.34	0.62	1.07
▨ IBM	0.45	0.60	0.02	0.40	0.06
▨ Sun	0.39	0.67	0.20	1.37	0.34

Figure 18. Micro Benchmark on IA-32



	Integer Addition	Float Addition	Object allocation	Object assignme	Instance method
▨ JnJVM	1.00	1.00	1.00	1.00	1.00
□ Cacao	1.00	1.00	0.40	0.59	0.74
▨ IBM	1.03	1.00	0.42	0.43	0.11

Figure 19. Micro Benchmark on PowerPC

6.2.1. *Arithmetics.*

Integer and floating-point addition benchmarks are the same with the JnJVM and with other JVMs on PowerPC. However, performance differs on the IA-32 architecture. This is explained by the lack of any register allocation algorithm of the VPU. On PowerPC, VPU uses r14 to r31 for the first variables and leaves the others in stack. In the arithmetic tests on PowerPC, these registers are sufficient for local variables and JnJVM performs as fast as other JVMs. But on IA-32, the VPU keeps all variables on stack: when a variable is accessed, it is loaded from the stack, the operation is done and the result is stored on stack. Modern JITs try to maintain most frequently used variable on register. In other JVMs, intensively used local variables are maintained in registers, and in contrast, the VPU can not move variables and suffers when temporaries are on the stack. This limitation requires a complete rewriting of the VPU because live analysis of locals is necessary. This test explains the differences observed between the JnJVM and other JVMs: when the register allocator is stressed, the JnJVM is not competitive.

6.2.2. *Object allocation.*

The object allocation benchmark of the Javagrande Test Suite measures both the garbage collection algorithm and the allocation performance. Our garbage collector is the colored mark and sweep of the MVM (see section 2.1). JnJVM outperforms Kaffe, but is slower than others on PowerPC and IA-32. IBM and Sun are using a modern generational GC, which explains the ratio of 2.5 to 5 with the JnJVM (except for IBM in IA-32). The result of IBM on IA-32 is spectacular: 50 times faster than JnJVM and 10 times faster than Sun. IBM version 1.4.2.03 probably uses a single object for this allocation because an instance of an object does not have fields, whereas version 1.4.2 does not.

6.2.3. Object assignment.

Object assignment in the JnJVM is far from optimized because of the VPU register allocation algorithm. Most of the time, parameters are taken from the stack in the JnJVM, while they are in register in other JVMs.

6.2.4. Method invocation

The JnJVM is slower than other JVMs because the VPU allocates most of its local variables on the stack. The JnJVM uses caches for instance method invocations. A virtual table could improve performance by 5%, but does not suffice to explain the differences.

6.3. Macro-benchmarks

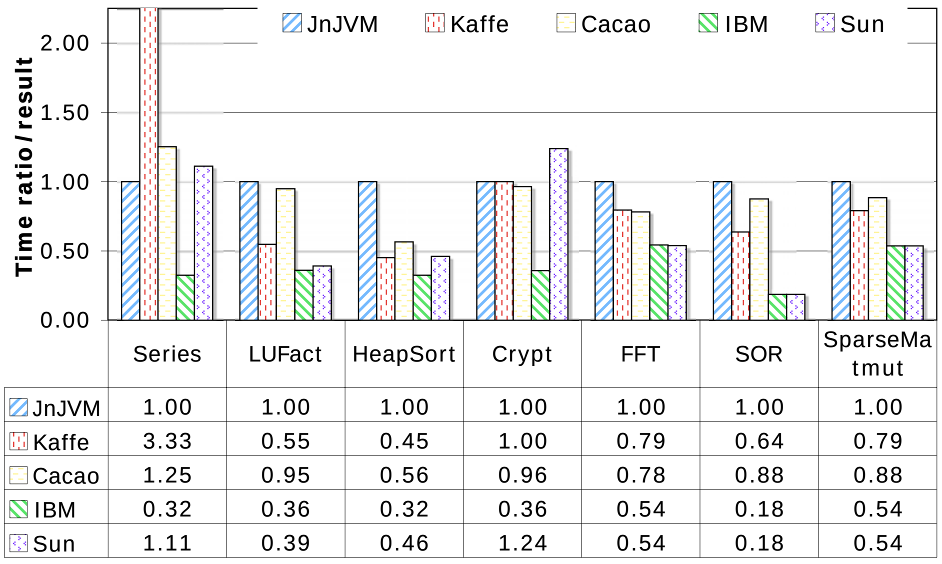
The JavaGrande Test Suite also measures performance of JVMs with representative applications. The results are presented in Figure 20 and 21. Only measures on IA-32 are reported here because results on PowerPC are equivalent. These tests show that the JnJVM is on average equivalent to Kaffe for section 2 and better than Kaffe for section 3. The JnJVM is at the same level as Cacao, but most of the time slower. The limitations in JnJVM presented in the micro-benchmarks explain the main differences between the JnJVM and the other JVMs: we did not observe a big difference between micro and macro benchmarks (the ratio remains globally the same between the two series of test). The comparison with Sun and IBM shows that JnJVM is 2 to 10 times slower (in the worst case).

The set of results given by the JavaGrande test is very encouraging considering the level of immaturity of the JnJVM, whereas Sun Microsystem and IBM have spent considerable engineering effort on optimizations. Only three individuals developed the JnJVM and the MVM, and we can not expect the same performance as with industrial JVMs. Results reported in this paper show that the architecture of the JnJVM gives admissible performance. We expect that the bad performance of the JnJVM compared to IBM and Sun are explained by some unoptimized algorithms of the JnJVM and by the limitations of the VPU (see Section 2.1), but not by our flexible architecture. However, without an optimized prototype, we can not prove yet that our architecture has no impact on performance.

6.4. Summary on performance evaluation

Executing the JVM on the MVM introduces a new level of virtualization: the JnJVM is compiled by the MVM. Our performance study shows that the JnJVM is still competitive and we show how the limitation of the VPU impacts these results (the VPU does not perform any optimizations and does not have an optimized register allocator).

We note that with the best JVMs, despite virtualization, performance is comparable to C++ [26, 32], at the cost of substantial engineering effort. We expect that similar efforts applied to the JnJVM would yield similar results.

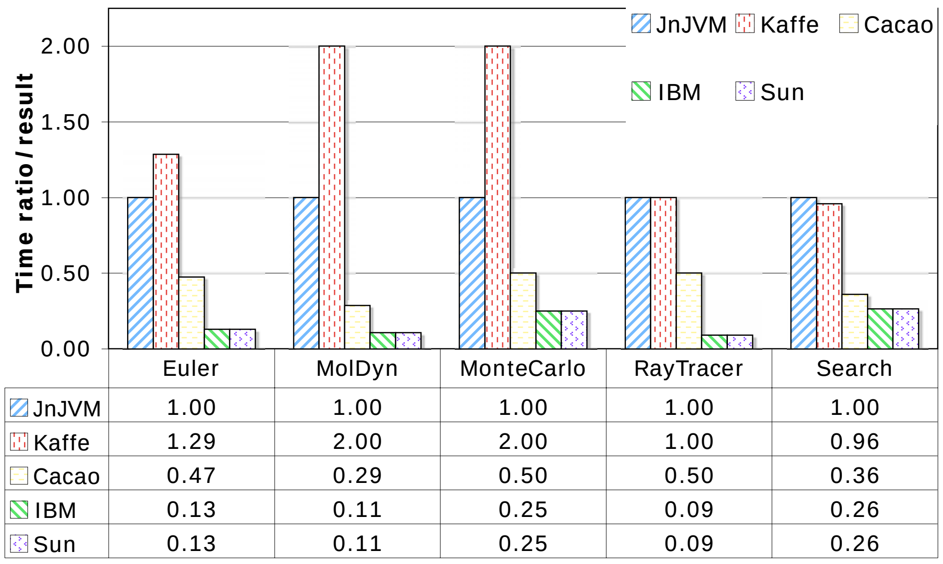


	Series	LUFact	HeapSort	Crypt	FFT	SOR	SparseMatmut
JnJVM	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Kaffe	3.33	0.55	0.45	1.00	0.79	0.64	0.79
Cacao	1.25	0.95	0.56	0.96	0.78	0.88	0.88
IBM	0.32	0.36	0.32	0.36	0.54	0.18	0.54
Sun	1.11	0.39	0.46	1.24	0.54	0.18	0.54

Figure 20. Macro Benchmark of section 2 on IA-32.

7. Discussion and future work

By modifying method pointers directly in memory, adaptations are done at a finer granularity than component replacement. Furthermore, the method replacement does not imply the loss of the component state: the developer of a specialization does not have to manage state transfer. Nevertheless, our adaptation mechanisms are based on aspect technology and have the same limitations: to modify the JnJVM, a specialization directly uses and modifies the internal structures of the JnJVM. A specialization script is therefore highly dependent on the code of the JnJVM. If the JnJVM evolves, reconfiguration scripts may have to be changed. This is because specialization scripts weave code by using the names of the methods directly. To solve this problem, we plan to express the reconfigurations by manipulating higher level abstractions via well defined reconfiguration interfaces: a reconfiguration script will be independent from the JnJVM internals.



	Euler	MolDyn	MonteCarlo	RayTracer	Search
JnJVM	1.00	1.00	1.00	1.00	1.00
Kaffe	1.29	2.00	2.00	1.00	0.96
Cacao	0.47	0.29	0.50	0.50	0.36
IBM	0.13	0.11	0.25	0.09	0.26
Sun	0.13	0.11	0.25	0.09	0.26

Figure 21. Macro Benchmark of section 3 on IA-32.

The second direction of our research is security of adaptation that we have ignored for the current JnJVM experiments. This could be solved by authenticating the application or the user who wants to modify the JnJVM at the reconfiguration interface. Indeed, by letting an application modify the methods of the internal components of the JnJVM, we can not ensure security, but by using a reconfiguration interface, we can restrict the possibility of such applications.

The third perspective of our work concerns optimizations and portability. The speed and the memory footprint of the JnJVM will be improved. The VPU currently generates code for PowerPC and x86, thanks to a clean separation of its interface and its processor description; other processor targets can easily be accommodated.

The last perspective of our work concerns the language. MVM is developed in C++ and we can not know its internal structures at the MVM level. This limitation implies limitations on field addition. We are actually developing a new prototype of MVM in the Lisp-like language of the MVM. This prototype will solve this limitation.

8. Related Work

This work draws on three different research areas: (i) Flexible operating systems, (ii) Aspect oriented programming and (iii) Virtual Machines/Languages.

Flexible operating systems. The design of our flexible system is based on previous works on flexibility issues on operating systems design. The component architecture of the JnJVM is similar to simple plug-in mechanisms, such as loadable device drivers in Linux. Defining specific policies is also possible in micro-kernel using servers (such as L4 [30] or Mach [7]) and exo-kernels (such as Aegis [15]) in user-space libraries. The problem with loadable modules is that the state is lost: when a dynamic update is needed, the state of the module (the loadable device driver, the server or the library) is lost. Kernels defined with component architecture, such as the OSKit Think [17, 42] or K42 [28, 44] with its hot-swapping mechanism raise the same problem. The dynamic replacement of modules is permitted, but the developer has to manage the transfer from old components to new ones.

Our MVM defines the core concepts of a flexible operating system and the JnJVM is implemented with components (equivalent of libraries or servers). Because our experiment is at the application level, the same address space is used by the MVM, JnJVM and the application. Our component architecture could also be used for hot-swapping, but the main originality of our work is the use of adaptation points to adapt a component without losing its state.

Aspect oriented programming. Aspect oriented programming (AOP) defines language constructions to (i) declare sets of points in the code and to (ii) associate entities with these sets of points. From an engineering point of view, AOP increases the reusability of the code and from a system point of view, dynamic AOP is a technique to modify the code on the fly. In the AOP terminology, a *join point* is a point in the code (a method calling site, a field access site, an exception throwing site). A set of join points is a *pointcut* and weaving an aspect is the act of inserting some code (an aspect, or a set of *code advice*) across a pointcut.

Many aspect tools exist for Java (AspectJ, JBoss AOP, Jac...) and are used to modify applications during compilation or loading time. These tools can only adapt predefined join points (all join points must be defined before the execution of the application). They are relatively slow in case of dynamic adaptation because an intermediate method is called on each join point to select the current code advice. Steamloom [10] is a modified JVM that defines a dynamic Java aspect weaver. Steamloom does not introduce indirection to reach code advice because the JVM can modify directly code pointers in memory to adapt the application. Therefore, Steamloom does not degrade performances. The Java aspect weaver of the JnJVM presented in Section 4 is similar to Steamloom, but we do not use a modified JVM: the JnJVM is adapted on the fly with a new system abstraction, the Java aspect weaver.

Aspects are principally used to adapt applications on the fly, but are rarely used to adapt execution environments such as the operating system or the virtual machine. To our knowledge, the JnJVM is the only JVM that can adapt itself on the fly with aspect technology.

Extended Virtual Machines and languages. The Distributed Virtual Machine [43] (DVM) is a Java virtual machine distributed on a network that factorizes common tasks, such as bytecode verification, optimization or compilation, on a centralized server. DVM distributes internal components in a network: DVM does not permit dynamic updates of components, but it uses an architecture close from ours. XVM [23] is an extensible virtual machine in which internal components of the virtual machine can be updated independently. XVM does not address the problem of state transfer between components during an update. JnJVM separates the virtual machine in components as well, but also provides an aspect weaver that allows an application to modify its execution without losing its state.

JavaOS [33] was a project of Sun microsystem and IBM research to construct a system based on Java where all drivers were written in Java. KaffeOS [8], which is based on the open source Kaffe Java virtual machine, is a Java system that runs on Linux and provides process abstractions. KaffeOS provides also isolation between processes based on Java type verifications. JavaOS and KaffeOS do not address the problem of flexibility: their runtime are monolithic JVMs that can not be adapted. Singularity [25] is a Microsoft research prototype in operating system based on the Microsoft Intermediate Language (MSIL). Singularity allows an application to define its own system policies for garbage collection algorithm and its own memory layout, but the goal of Singularity is dependability: extensibility in Singularity is achieved via communication channels. Singularity isolates the kernel from the drivers and does not permits adaptations of kernel system policies.

The Common Language Runtime (CLR) of the .Net framework is a monolithic virtual machine able to execute MSIL code. The CLR allows a user to execute different language, but it can only executes MSIL. Compared to the CLR, the MVM does not impose a unique bytecode: we are executing Lisp-like source, Java bytecode or MSIL with the same JIT. Moreover, the goal purchased by the MVM is dynamic flexibility while the goal purchased by the CLR is the construction of a common virtual machine. The CLR does not provide any abstraction to modify the virtual machine or the application on the fly.

JikesRVM [2] or Joeq [48] are prototypes of Java Virtual Machine to experiment new virtual machines algorithms. They allows a developer to change one of their main components at compilation time, but the goal of these virtual machines is not dynamic adaptation: they does not provide tools to modify theirs internal components on the fly. JikesRVM or Joeq could be good platforms to experiment dynamic flexibility, but it would require the same engineering effort as the development of the MVM.

Works on dynamic software updating [24] show how to adapt a software during its execution by generating dynamic patches between two versions of the source and by loading them to adapt the software on the fly. The state of the software is also preserved and dynamic addition of fields is permitted. Our work does not address evolution of source code but evolution of the runtime system: whereas dynamic software updating needs to access the source code of the application, JnJVM constructs evolution directly from the running application in memory. In our system, patches are not necessarily written in the same language. The innovative concepts presented in [24] to adapt running code and to construct patches are reusable in MVM. A new interesting research project could be the construction of dynamic patch from the representation of the running application in memory (instead of using source code) to automatize adaptation in JnJVM and MVM.

The Low-Level Virtual Machine (LLVM) [29] and LibJIT [4] define also a generic virtual machine with a common intermediate representation. However, our experiment explores flexibility of execution environment while these virtual machines offer a framework to build other virtual machines. The VPU acts as LLVM, but we propose also a flexible language and an aspect oriented system with the MVM. We are actually investigating the replacement of the VPU by LLVM [20].

9. Conclusion

This paper presents a flexible Java Virtual Machine based on a generic flexible system, the MVM. Flexibility relies on language extensibility and aspect weaving. The use of dynamic code weaving mechanisms is an alternative to traditional component replacement used to bring flexibility in operating systems. It avoids the problem of state transfer between components during an adaptation. Our performance measures show that adaptation points do not imply a decrease in performance during normal operations.

The Micro Virtual Machine is the execution environment of the JnJVM. It is a dynamic compiler that enables the construction of adaptable components by using aspect techniques. It is independent from the JnJVM and can be reused for another execution environment, such as an operating system or another virtual machine.

The JnJVM is entirely written in the MVM Lisp-like language and compiled on the fly by the MVM at bootstrap. The cost of compiling a Java Virtual Machine is reasonable and our performance study shows that our JVM is equivalent to the Kaffe Java Virtual Machine.

The Java virtual machine was the base experiment for our work, but our results are generalizable to other systems, like operating systems, other virtual machines, interpreters or middlewares. The weaknesses and strengths of our architecture are:

Engineering cost Our approach has a startup engineering cost: the Java virtual machine had to be wholly reconstructed for this work. JnJVM is equivalent in size to other Java virtual machines with 45000 lines of code (JnJVM and MVM). For complete operating systems with millions of lines of code, such as Linux, research has to be conducted to automatically re-engineer existing code to match our component architecture.

Performance Our adaptable architecture does not introduce indirections in the code. Replacing whole components, methods or adding fields does not require extra indirections during execution, due to runtime meta-data. However, the meta-data information increases the amount of memory required to execute the JnJVM and slows down the garbage collector. In term of performance, the JnJVM is approximately equivalent to Kaffe and remains 2 to 10 times slower than recent and robust JVMs by IBM and Sun. This result is encouraging for a first prototype and lets us expect that our architecture does not slow down an application. Nevertheless, we can not state that this is the case after our evaluation.

Our prototype shows however that admissible performance is not in contradiction with dynamic adaptability: JnJVM performs roughly the same as the Kaffe virtual machine, which was a standard 5 years ago.

Adaptability The experiments presented in this paper show different levels of adaptability of the JnJVM. Escape analysis shows that traditional hidden components, like the memory manager and the thread manager, can be adapted. Java aspects show the extensibility of our Lisp-like language. It allows an application to build simultaneously a specialized behavior and an associated language. The load-balancing specialization shows that system behaviors can be added dynamically and transparently for the application.

We have constructed a generic flexible environment, the MVM, to build other flexible environments. Flexibility relies on an extensible language and an aspect weaver. The result is a highly flexible virtual machine, whose internal components can be modified and adapted dynamically by the application or a system administrator. We presented three adaptation experiments, that show dynamic extensions and modifications of the virtual machine. We believe our system can be a framework for the construction of many other dynamic flexible systems.

Acknowledgements

We wish to thank our shepherds Emin Gun Sirer and Marc Shapiro for their enormous contributions to the organization of this paper. We are also grateful to Yolande Berbers for her insight suggestions. We thank the authoring Eurosys Workshop committee, especially Rodrigo Rodrigues and John Wilkes for their interest and comments. We wish also to thank the anonymous reviewers of SPE for all their comments and suggestions to improve the quality of the paper. Finally, we wish to thank Ian Piumarta for his enormous contribution, the VPU, which was the first generic JIT when we begun this work.

REFERENCES

1. The GNU Classpath Project. www.gnu.org/software/classpath/classpath.html.
2. Jikes – research virtual machine home (rvm) page. http://www-124.ibm.com/developerworks/oss/jikesrvm.
3. Jsr 121: Application isolation api specification. http://jcp.org/en/jsr/detail?id=121.
4. Libjit. http://demakov.com/projects/index.html#libjit.
5. Objecteering home page. http://www.objecteering.com/.
6. Performance comparison of jits, Jan. 2002. http://www.shudo.net/jit/perf/.
7. M.J. Accetta, R.B. Golub, R.F. Rashid, Jr A. Tevanian, and M.W. Young. Mach: A new kernel fundation for unix development. In *Proceedings of the USENIX Technical Conference and Exhibition*, june 1986.
8. G. Back, W.C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, USA, sept. 2000.

9. B. Blanchet. Escape Analysis for Java: Theory and Practice. *Transactions on Programming Languages and Systems*, 25(6):713–775, January 2003.
10. C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the Aspect-Oriented Software Development Conference*, pages 83–92, Lancaster, UK, March 2004.
11. H-J. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the Programming Language Design and Implementation Conference*, pages 157–164, Toronto, Canada, June 1991.
12. I. Piumarta C. Khoury, B. Folliot. Aan: A highly reflective active network. In *20st IASTED Applied Informatics Conference*, Innsbruck, Autriche, Fev 2002.
13. J. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications Conference*, pages 1–19, Denver, USA, November 1999.
14. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly Garbage Collection: an Exercise in Cooperation. *Communication of the ACM*, 21(11):966–975, September 1978.
15. D.R. Engler, M.F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, USA, December 1995.
16. Edinbugh Parallel Computing Center EPPC. Java grande forum benchmark suite – version 2.0. http://www.epcc.ed.ac.uk/javagrande/, 2003.
17. J-P. Fassino, J-B. Stefani, J. Lawall, and G. Muller. Think: A Software Framework for Component-based Operating System Kernels. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, USA, June 2002.
18. A. Galland. *Contrôle de ressources dans les cartes à microprocesseur.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 2005.
19. N. Geoffray, C. Clement, G. Thomas, and B. Folliot. Extracting common components of virtual machines. In **Submitted** *at the 7th conference on Software Composition (SC'07)*, Budapest, Hungary, March. 2008.
20. N. Geoffray, C. Clement, G. Thomas, and B. Folliot. Implementing virtual machines on a common framework. In **Submitted** *at the 22th Europeen Conference on Object-Oriented Programming (ECOOP'08)*, Paphos, Cyprus, jul. 2008.
21. Nicolas Geoffray, Gal Thomas, and Bertil Folliot. Transparent and dynamic code offloading for java application. In *Proceedings of International Symposium on Distributed Objects and Applications 2006 (doa 2006)*, pages 1790–1806, Montpellier, France, October 2006.
22. Assia Hachichi, Gal Thomas, Cyril Martin, Simon Patarin, and Bertil Folliot. A generic language for dynamic adaptation. In *European Conference on Parallel Processing (EuroPar 2005)*, Lisboa, Portugal, August 2005.
23. T. L. Harris. An eXtensible Virtual Machine Architecture. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, page 135, Denver, USA, November 1999.
24. Michael Hicks and Scott M. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2005.
25. G.C. Hunt, J.R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and Brian Zill. An overview of the singularity project. Technical report, Microsoft Research, Oct. 2005.
26. J.P.Lewis and U. Neumann. Performance of Java versus C++, January 2004. http://www.idiom.com/ zilla/Computer/javaCbenchmark.html.
27. 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.
28. O. Krieger, M. Auslander, B. Rosenburg, R.W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. In *Proceedings of EuroSys2006*, pages 133–145, Leuven, Belgium, 2006.
29. Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
30. J. Liedtke. On Micro-Kernel Construction. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, Colorado, United States, 1995.

31. T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley Publishing Company, Inc., 1997.
32. C. Mangione. Performance tests show Java as fast as C++, 1998. http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf_p.html.
33. Sun Microsystems. Javaos for business: Technical summary. http://java.sun.com/developer/products/JavaOS/OverView/index.html.
34. G.C. Necula. Proof-Carrying Code. In *Proceedings of the Principles of Programming Languages Conference*, pages 106–119, Paris, France, January 1997.
35. F. Ogel, B. Folliot, and G. Thomas. A step toward ubiquitous computing: An efficient flexible micro-orb. In *proceedings of the 11th ACM SIGOPS European Workshop*, pages 176–181, Leuven, Belgium, sept. 2004.
36. 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.
37. F. Ogel, G. Thomas, and B. Folliot. Support efficient dynamic aspects through reflection and dynamic compilation. In *20th Annual ACM Symposium on Applied Computing*, Santa Fe, New Mexico, USA, March 2005.
38. F. Ogel, G. Thomas, I. Piumarta, A. Galland, B. Folliot, and C. Baillarguet. Towards active applications: the virtual virtual machine approach. In Mitica Craus, Dan Glea, and Alexandru Valachi, editors, *New Trends in Computer Science and Engineering*. A92 Publishing House, polirom press edition, 2003. ISBN 973-9476-40-6.
39. R. Pawlak, J-P. Retaillé, and L. Seinturier. *Foundations of AOP for J2EE Development*. Apress, Berkely, USA, 2005.
40. 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.
41. E. Rose and K.H. Rose. Lightweight Bytecode Verification. *Formal Underpinnings of Java*, 1998. Vancouver, Canada.
42. A. Senart, O. Charra, and J-B. Stefani. Developing Dynamically Reconfigurable Operating System Kernels with the Think Component Architecture. In *Proceedings ot the Engineering Context-aware Object-Oriented Systems and Environments Workshop*, Seattle, USA, November 2002.
43. E.G Sirer, R. Grimm, A.J Gregory, and B.N Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computer. In *Proceedings of the Symposium on Operating Systems Principles*, pages 202–216, Kiawah Island Resort, USA, December 1999.
44. C.A.N. Soules, J. Appavoo, K. Hui, R.W. Wisniewski, D. Da Silva, G.R. Ganger, O. Krieger, M. Stumm, M.A. Auslander, M. Ostrowski, B.S. Rosenburg, and J. Xenidis. System Support for Online Reconfiguration. In *Proceedings of USENIX Technical Conference*, pages 141–154, San Antonio, USA, june 2003.
45. G. Thomas, B. Folliot, and I. Piumarta. Les documents actifs bass sur une machine virtuelle. In *Actes de la conf. Journes des Jeunes Chercheurs en Systmes, Chapitre franais de l'ACM-SIGOPS*, pages 441–447, Hammamet, Tunisie, April 2002.
46. 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.
47. E. Tilevich and Y. Smaragdakis. NRMI: Natural and Efficient Middleware. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 252–264, Washington, USA, May 2003. IEEE Computer Society.
48. J. Whaley. Joeq: A virtual machine and compiler infrastructure, 2003.