

VMKit: a Substrate for Virtual Machines

Nicolas Geoffray — Gaël Thomas — Charles Clément — Bertil Folliot — Gilles Muller

N° 6799

Janvier 2009

Thème COM

 ***Rapport
de recherche***

VMKit: a Substrate for Virtual Machines

Nicolas Geoffray, Gaël Thomas, Charles Clément , Bertil Folliot ,
Gilles Muller

Thème COM — Systèmes communicants
Équipe-Projet Regal

Rapport de recherche n° 6799 — Janvier 2009 — 25 pages

Abstract: Developing and optimizing a virtual machine (VM) is a tedious task that requires many years of development. Although VMs share some common principles, such as a Just In Time Compiler or a Garbage Collector, this opportunity for sharing has not been yet exploited in implementing VMs. This paper describes and evaluates VMKit, a first attempt to build a common substrate that eases the development of high-level VMs. VMKit has been successfully used to build three VMs: a Java Virtual Machine, a Common Language Runtime and a lisp-like language with type inference $\mu\text{vm-l}$. Our precise contribution is an extensive study of the lessons learnt in implementing such common infrastructure from a performance and an ease of development standpoint. Our performance study shows that VMKit does not degrade performance on CPU-intensive applications, but still requires engineering efforts to compete with other VMs on memory-intensive applications. Our high level VMs are only 20,000 lines of code, it took one of the author a month to develop a Common Language Runtime and implementing new ideas in the VMs was remarkably easy.

Key-words: Virtual Machines, Compilation, VMKit, LLVM

VMKit: un substrat pour machines virtuelles

Résumé : Le développement d'une machine virtuelle (MV) ainsi que ses optimisations est un travail complexe qui demande plusieurs années d'investissement. Alors que les MVs partagent des concepts communs comme les compilateur juste à temps ou les ramasse-miettes, les implémentations de MVs existantes n'ont pas exploité le partage d'implémentation de ces concepts. Dans ce rapport, nous décrivons et évaluons VMKit, un substrat commun pour le développement de MVs haut niveau. Nous avons implémenté trois MVs au-dessus de VMKit: une machine virtuelle Java, une machine virtuelle .Net et un langage á la Lisp avec inférence de type. Nous montrons que VMKit ne dégrade pas les performances d'applications gourmandes en processeur, mais demande encore des efforts d'ingénierie pour les applications gourmandes en mémoire. Nos MVs haut-niveau représentent 20,000 lignes de code, un des auteurs a mis un mois pour développer la machine virtuelle .Net et l'implémentation de nouvelles idées MVs s'est révélée très simple.

Mots-clés : Machines virtuelles, Compilation, VMKit, LLVM

1 Introduction

Virtual machines (VM) such as the Java Virtual Machine (JVM) and the Common Language Infrastructure (CLI) are becoming the norm for executing programs in many environments such as desktops, web servers and even embedded systems. This success fosters new researches in the domains of interpreters, (just in time) compilers and garbage collector (GC) so as to experiment new VM functionalities or to improve performance [8, 9, 34]. However, research prototypes must be evaluated against the industry state of the art VMs. Such goals induce a huge development effort: (i) many non interesting functionalities (from a research point) are required to run standard benchmarks, (ii) reaching the performance level of industry VMs requires spending a lot of time in finding, optimizing and removing bottlenecks. A common infrastructure for easing VM development is thus highly desirable.

Popular VMs such as the CLI [25] and the JVM [30] have been often promoted as universal runtime infrastructures. Still, they impose a fully defined behavior on the link, load phases and also on the type/object model that some languages may not need and/or want. For instance, the CLI allows to represent arbitrary code in its file format, but not in its type system and instruction set. It does not perform dynamic class loading which can have severe limitations for some execution environments [22]. The JVM does not provide non-GC allocated structures.

In this paper, we propose a solution for building VMs based on a two layer approach: (i) a common substrate providing basic functionalities such as threading, I/Os, GC based memory management, exceptions and a compilation engine (JIT, interpreter, dynamic optimizations) for a VM independent intermediate code, (ii) a high-level Virtual Machine providing the VM specific functionalities. More precisely, a high-level VM implements a runtime engine to find types, methods and fields, and a dynamic translator to translate from a VM specific code to the substrate intermediate code.

Our precise contribution is to describe VMKit, a first implementation of such a substrate using LLVM [28] as the compilation engine. We report our experience in building three high-level VMs using VMKit: a Java Virtual Machine (JnJVM), a CLI implementation (N3) and a lisp-like language with type inference ($\mu\text{vm-l}$). The reasons of these choices are that JVM and CLI are the most commonly used VMs, and even if they have many similarities, there are enough differences to show the advantages of a shared substrate. Finally, $\mu\text{vm-l}$ is a dynamic language with type inference, two features that Java and CLI do not currently have.

The lessons learnt from our work are:

- One can build different high-level VMs on top of a shared substrate very easily: we wrote around 20,000 lines of code for each high-level VM. Once we gained experience with our first JnJVM implementation, it took one of the author a month to develop N3.
- On CPU intensive applications, VMKit has similar performance than industrial (IBM, Sun) and top open-source (JikesRVM, Mono) VMs.
- On GC intensive applications, JnJVM is five times slower than the Sun OpenJDK, due to VMKit's memory management system limitations. We

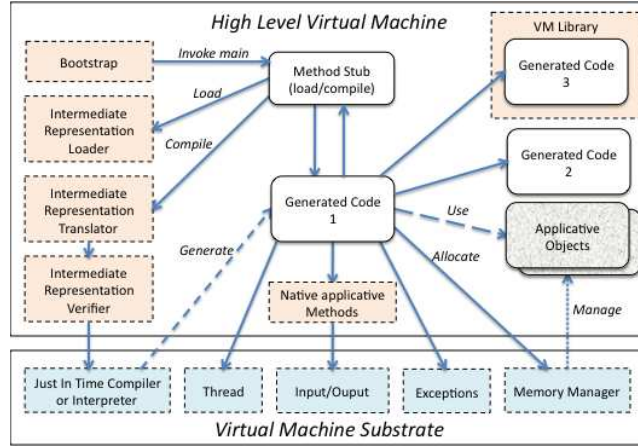


Figure 1: Virtual machine substrate and a high level VM. Solid arrows represent calls and dashed arrows represents semantic links between boxes. A grey box is a module and a white box is an object of the application, a generated code or a method stub.

expect that an implementation of latest GC techniques will yield better results.

The remainder of this paper is organized as follows. Section 2 presents the overall design of VMKit. Section 3 introduces our target high-level VMs and their implementation using VMKit. Section 4 analyses their performance. Section 5 discusses related work. Section 6 concludes the paper.

2 The Design of VMKit

The main motivation for VMKit is to split the design of a VM into two layers: a substrate that provides common functionalities that are often found in VMs and a high level VM that implements the VM specific functions. The benefit of sharing a low-level infrastructure is twofold. First, development time is reduced since many functions are provided by the common substrate. Second, high-level VMs benefit in performance from the optimized services provided by the substrate. In the rest of this section, we first detail the design of a VM using a high level VM and a common substrate, then we present VMKit, a first implementation of such a substrate.

2.1 Design Overview

Figure 1 presents the design of a VM as a high-level VM and a substrate. The substrate contains a compilation engine that defines a common intermediate representation (IR), a thread subsystem, an I/O subsystem, an exception manager and a memory manager. The substrate does not force the definition of a memory model ; This is done by the high-level VM that defines the structure

of the root object, how memory is allocated (GC, non-GC, on stack), and how methods are dispatched.

We have identified a common structure for high level VMs which are composed of:

- An bytecode/source loader: it loads an application (or a part of an application) to memory. It forwards the application to the IR translator.
- An IR translator: it transforms the IR of the high-level VM (e.g. bytecode) to the low-level IR of the substrate.
- An IR verifier: it verifies the IR (type verification or safety verification) and possibly infers types.
- A Virtual Machine Library: the application support libraries of the high-level VM.
- A set of native methods for linking between the VM library and the high-level VM. These are downcalls from the application to the high-level VM.

Since developing a VM library requires a major engineering effort (for instance, GNU Classpath contains 750k lines of code), we have tried in this study to reuse existing VM libraries developed by others (GNU Classpath [3] in the case of Java, Pnetlib [2] in the case of the CLI). Therefore, developing a high-level VM required only the implementation of the loader, the translator, the verifier and the native interfaces.

2.2 VMKit

VMKit is a first attempt at implementing a common VM substrate for multiple VMs. VMKit is originated from a previous work in modularizing the implementation of a JVM with exclusively existing projects [24]. VMKit contains a set of services that were either developed by us or comes from third party projects:

- LLVM as the compilation engine.
- A home made garbage collector as the memory manager.
- The GCC exception runtime for the exception manager.
- Posix OS threads as the thread subsystem.
- The local C library as the I/O subsystem.

At the current time, VMKit runs on Linux/x86 and MacOSX/x86. Ultimately, VMKit should be able to run on all architectures supported by LLVM. However, LLVM is still in major development, and only its x86 backend can be fully trusted.

2.2.1 Compation engine: the Low-Level Virtual Machine

The Low Level Virtual Machine (LLVM) [28] is a compiler framework. It defines a low-level code representation with a memory layout system. LLVM has many compiler optimizations, either per function, per module (e.g. file) or at link-time.

VMKit bases the CPU-related performance on LLVM. Since LLVM is the compilation engine of VMKit, execution time of such benchmarks is affected by the code LLVM generates.

2.2.2 Memory manager

VMKit implements a mark and sweep garbage collector (GC) [14] that contains a memory allocator and a threading system. The GC in its current implementation does not need any type information from the compiler. VMKit's GC has a list of object roots, and it traces the values on the execution stack of all threads. From these values, he finds out which data may be an object and which can not, based on the allocator metadata. Therefore our GC is not precise but conservative and an integer on the execution stack can be interpreted as an object. Then, the garbage collector calls the marking function on all of these objects. At the end of this step, the GC knows all reachable and unreachable objects and sweeps unreachable objects.

The performance of our GC is equivalent to the state of the art conservative Boehm GC [14] as shown in Section 4. However, our memory management system lacks many optimizations that modern VMs have, such as thread local heaps [21], stack allocation with escape analysis and a generational collector. These optimizations dependent strongly on the compiler and need a cooperation between the GC and the compiler.

The standard version of LLVM provides an API to implement garbage collectors and it can also perform accurate garbage collection with compiler support, i.e. LLVM can locate all pointers in the stack based on meta-information provided by the compiler, that ensures that there are no possible confusion with integers. We did not use these facilities because LLVM does not currently have GC implementations, and its API has not been tested.

We also changed VMKit's memory management system to use the Boehm Garbage Collector. The experience is reported in [24]. As section 4 reports, our memory management system yields similar performance.

2.2.3 Exceptions manager

VMKit uses the Dwarf [6] tables created by LLVM for dynamically generated code and registers the table to the GCC exception handling library [20]. This library will unwind the stack when an exception is thrown. Dwarf tables can also be used for debugging purposes.

2.2.4 Thread and I/O subsystem

The current version of VMKit relies on the pthread library of the underlying OS. Also, each thread of a high-level VM maps to one kernel thread. For the I/O subsystem, GNU Classpath implements all network and file input/output with POSIX functions.

3 Developing High-Level Virtual Machines on top of VMKit

We have implemented three high-level VMs on top of VMKit: a Java virtual machine [30] called JnJVM, an implementation of the Common Language Infrastructure (CLI) [25] called N3 (.Net is Microsoft's implementation of the CLI), and a high-level VM for a lisp-like language with type inference and C semantics. The three high-level VMs have different object models and behaviors for code translation and code loading.

3.1 JnJVM: a Java Virtual Machine

JnJVM is a JVM implementation that relies on VMKit for the execution engine. JnJVM defines a bytecode to LLVM IR translator, a class loader, and an interface to the Java support libraries.

Bytecode translation: Most of the JVM bytecodes translate into one, two or three LLVM instructions. The notable exceptions are: field accesses, method calls, exceptions, runtime checks, synchronizations, allocations and switch instructions.

For each Java class, a constant pool is associated, which contains references that will be dynamically linked with other classes. Therefore, when an entry in the constant pool is not resolved at compile-time, JnJVM inserts a callback function in the LLVM code. At run time, the function will do the resolution and return the appropriate information (e.g. a pointer to a field or a class). The callback function is only executed once.

For non-virtual calls, i.e. `invokespecial` or `invokestatic`, JnJVM relies on LLVM for callbacks. If the called method has not been resolved, LLVM generates a callback which will invoke a function provider. JnJVM implements its own function provider. The function provider loads the class if needed and translates the JVM bytecode to LLVM instructions. LLVM will patch the native call instruction to call the newly resolved method instead of the callback.

For virtual, non-interface calls, i.e. `invokevirtual`, JnJVM emits LLVM instructions that load the virtual table of the object, loads the function pointer from the virtual table and call the function. At run time, if the function has not been compiled, a stub is executed that will call the function provider of LLVM. The function provider lookups the offset of the method in the virtual table, compiles the method if not already compiled and updates the virtual table.

For each interface call site, i.e. `invokeinterface`, JnJVM creates a linked-list cache that records which object classes were called last. At run time, when the object class is not available in the cache, JnJVM performs a full method lookup. The resulting class is then placed at the front of the list. The following calls compare the object class with the front of the list. If they are equal, the call proceeds. This implementation of interface calls has the advantages of being compiler independent but has low performance. Alpern *et al.* [12] generate optimized native stubs to implement efficiently the `invokeinterface` bytecode. Such an optimization is difficult to achieve in VMKit because VMKit uses LLVM as-is and does not extend it with new calling conventions for native stubs.

Other implementation issues are as follows. Synchronization bytecodes (i.e. `monitorenter` and `monitorexit`) are translated to a thin lock implementation [13]. Switch instructions (i.e. `tableswitch` and `lookupswitch`) are decomposed in many LLVM instructions that perform the comparisons and branches. Allocation bytecodes (i.e. `new`, `newarray`, `anewarray` and `multianewarray`) are translated to calls to the memory manager.

Class loading: JnJVM follows the Java mechanism of class loading [30, 29]. Classes are resolved just in time, which means that at compilation time the compiler may not know the layout of a class. The compiler inserts runtime calls to dynamically resolve entries in the constant pool. The resolution may imply invoking Java code to dynamically load a `.class` file. Once the class is resolved, JnJVM is able to resolve methods and fields with lookups.

Library Interface: JnJVM interfaces with the GNU Classpath implementation of the Java libraries, e.g. `java.lang` or `java.net`. Moving to another library implementation should be straightforward, as the JnJVM code base only makes a few upcalls to the library.

Compiling a Java method from GNU Classpath follows the same model as compiling a method from a Java application. The only difference lies in JnJVM having to actually find the method's class file on the file system.

3.2 N3: a CLI virtual machine

N3 implements the CLI specification [25] on VMKit. By design, CLI and Java share some common high-level mechanisms: method and field lookups on the internal representation of classes, arrays, synchronizations, virtual and interface calls, threads and exceptions. N3 and JnJVM have similar implementations of these mechanisms. There are however notable differences between the two VMs that highlight the contribution of N3 on top of VMKit:

- Structures: in the CLI, everything is not an object. A CLI implementation must support the allocation of structures on stack or in an object. Hence the compiler and the garbage collector of a CLI implementation must deal with this kind of memory, that does not exist in Java.
- Static class resolution: contrary to Java, the CLI requires that a class is resolved when the compiler references it. Accessing a field, or calling a method of a class triggers its resolution. In Java, the resolution of a class happens during execution of Java methods, i.e. when the field is actually accessed or the method is called.
- Generics: in the CLI, the virtual machine handles generics. In Java, the source to bytecode compiler handles generics and produces a generics-free class file. In VMKit, the LLVM compiler does not know about generics, therefore N3 must instantiate itself the generics and produce the LLVM representation of the instantiated method.

As for JnJVM, the main functionalities of N3 are a CLI opcode translator, a CLI assembly loader and an interface to the support libraries.

Opcode translation: CLI bytecodes are almost all translated to LLVM instructions. Contrary to the JVM, the CLI does static resolution of classes, structures and fields. Therefore, N3 does not insert runtime calls for dynamic resolution. N3 follows the same translation model as JnJVM for non-virtual calls, virtual calls and interface calls. Similarly, bytecodes for exceptions such as `leave`, `leave-s` or `endfinally` are compiled following the JnJVM exception model.

The CLI defines overflow instructions, which are instructions that perform normal arithmetic operations but raise an exception if the operation overflows. LLVM does not currently handle this kind of instructions. Therefore overflow instructions are translated to normal operations with runtime checks for overflow.

Finally, the CLI handles structures, non GC-allocated memory that can be boxed to an object and unboxed from an object. N3 translates the `box` instruction to a GC-allocation and the `unbox` instruction to a native stack allocation.

Class resolution: The actual process of class resolution is the same as in JnJVM. The CLI however performs class resolution at compile time, whereas the JVM does it at execution time. One of the reasons for this difference is that the presence of structs requires to know at compile time where structs lie in the object for field accesses.

Library Interface: N3 uses the Mono library [9] for the VM library. As well as in JnJVM, methods in the VM library are compiled like application methods.

3.3 $\mu\text{vm-l}$: a Lisp-like language with type inference

The CLI and the JVM are very close in their object model and instruction set. They are also both currently targeting static languages. On the opposite, $\mu\text{vm-l}$ is a dynamic language with a syntax close to Lisp, and with C-like abstract syntax trees. $\mu\text{vm-l}$ does not have a bytecode instruction set. In $\mu\text{vm-l}$, types are not declared: the developer declares symbols whose types are dynamically found and set. Examples of this lisp-like language can be found in [36]. While $\mu\text{vm-l}$ is a minimal implementation of a dynamic language, its contributions on top of VMKit are nevertheless twofold:

- A dynamic language: our architecture does not prevent the implementation of a virtual machine whose input is a dynamic programming language. Dynamic languages can modify methods, type definitions and objects. Moreover, types are often known just in time: the execution engine knows the actual type of a function parameter or a variable on its first use.
- Type inference in the compiler: a Java bytecode or CLI compiler does not perform type inference because the languages they target are static and all types are known at class loading time. On the contrary, dynamic languages, as in $\mu\text{vm-l}$, may define methods without types. As an optimization, the execution engine can infer the types before the method is actually used. $\mu\text{vm-l}$ is a proof-of-concept implementation of type inference on top of VMKit.

Source language translation: It is not possible to compile $\mu\text{vm-l}$ source code directly to LLVM instructions, because the developer does not explicitly type function arguments and return values. Therefore the compiler needs to perform an additional pass in order to infer these types. The LLVM type system only handles type layouts. Each LLVM instruction type checks its arguments, so that the program in the end is well checked, with respect to type layouts. For example, an integer addition operation checks that the two operands have the same type and a call instruction checks that its real arguments have the same layouts as its formal arguments.

Type inference for dynamic languages in LLVM is currently under discussion [26]. The infrastructure of LLVM falls short for having type inference because of the type name loss conversion between the source language and the bytecode. While we currently are investigating adding type inference in the LLVM compiler framework, our first implementation adds a new type layer before generating LLVM instructions.

The $\mu\text{vm-l}$ compiler works in two phases: the first pass infers types, the second pass generates LLVM instructions. In the first pass, each untyped parameter of the function starts with an anonymous type that will be inferred if possible when used. If one or more arguments types can not be inferred (we call this function "untyped"), LLVM instructions generation is delayed until the function is called. A hash table is associated for each "untyped" function so that for a same function type, the function is not compiled to native code twice. This functionality is similar to the implementation of generics in N3.

Type resolution: The VM of $\mu\text{vm-l}$ follows a very simple read-eval-print loop. Whenever the application wants to load new files, it explicitly requires it. When a method does not have all of its arguments inferred, a call with concrete arguments will do an instantiation of the method.

4 Evaluations

To evaluate the interest of having a common substrate for implementing high-level VMs, the following criteria must be considered:

1. Code base: we want VMKit to ease the development of VMs, hence requiring as little effort as possible from the developer.
2. Ease of experimentation: with high-level VMs having a small code base, it is conceivable to implement new ideas very easily.
3. Startup time: there is always an overhead when running an application on top of a VM, even for a simple HelloWorld program.
4. Memory footprint: a VM has a non-negligible memory footprint when running applications.
5. Performance: once the application is loaded and compiled, do high-level VMs achieve good performance?

For benchmarking, we use an Athlon XP 1800+ processor with 512MB of memory running Gentoo Linux 2.6.23.

	JnJVM	N3	$\mu\text{vmm-l}$
Total lines of code	23200	16200	7400
LLVM translation	5500	5500	2400
Runtime engine	15000	9200	5000
Library interface	2700	1500	-

Table 1: Lines of code of the high level VMs

4.1 Code base

Table 1 sums up the lines of code for each subsystem of the high-level VMs. VMKit is composed of 10k lines of code, which are basically the memory manager and the interfaces with the compiler engine and the thread and exception libraries.

JnJVM is a full implementation of the JVM specification [30] and should be able to run any Java applications that GNU Classpath can support. JnJVM and GNU Classpath are robust enough to execute large scale applications such as the Tomcat servlet container [4] or the Felix OSGi implementation [1].

In comparison, Sun’s implementation of the JVM, OpenJDK, is 6.5M lines of code, which contains the compilation engine, GCs and class libraries. On a smaller scale, JikesRVM [8] which contains the compilation engine and GCs, has a code base of 270k lines of code.

N3 is able to run simple applications as well as the pnetmark benchmark [2]. After implementing JnJVM, it took a month of the first author of the paper¹ to implement most of N3, without generics.

4.2 Virtual Machine Prototyping

To evaluate the development cost of implementing a new research feature using VMKit, we implemented parts of the MultiTasking Virtual Machine (MVM) [18] by modifying the JnJVM high-level VM. MVM provides the ability to run multiple Java applications in the same execution environment, but with full isolation at the application-level. To improve scalability, MVM enables the sharing of class metadata and dynamically generated native code.

We implemented class metadata, bytecode, constant pool and native methods sharing between applications, including sharing across class loaders [19]. It took the first author of this paper one month to implement these functionalities. Implementing MVM into JnJVM required the modification of 1000 lines of code and the addition of 2000 lines. No modification of VMKit was needed.

4.3 Startup time

VMKit only uses the compiler of LLVM, but not its interpreter. Thus all bytecode must be compiled before being executed. A Java HelloWorld program takes 8 seconds to run with the basic configuration of JnJVM. When turning all optimizations on, the program runs in 12 seconds. In comparison, other JVMs boot in less than one second.

¹Nicolas Geoffray is a third year PhD student.

Time (ms)	Time %	Name of the Pass
5872	71.8	X86 Instruction Selection
629	7.7	Live Variable Analysis
386	4.7	Local Register Allocator
199	2.4	X86 Machine Code Emitter
8168		Total time

Table 2: Compilation time without optimization of a Java HelloWorld program.

Time (ms)	Time %	Name of the Pass
4431	35.8	X86 Instruction Selection
807	6.5	Global Value Numbering
633	5.1	Predicate Simplifier
547	4.4	Live Interval Analysis
439	3.5	Live Variable Analysis
405	3.2	Linear Scan Register Allocator
174	1.4	X86 Machine Code Emitter
12420		Total time

Table 3: Compilation time with optimizations of a Java HelloWorld program.

Execution of the `HelloWorld` program involves compiling many methods. The methods create the class loader, load the file and print "HelloWorld". With GNU Classpath version 0.97.2, 536 methods have to be compiled, with a total of 26952 bytecodes. Other JVMs do not compile but interpret these methods (or apply a simple baseline compiler) in order to achieve a better responsiveness.

Table 2 shows the time of compiling an `HelloWorld` Java class program without any optimization and Table 3 shows the time with the optimization passes turned on. We only show the most time-consuming passes of LLVM.

When optimizations are off, most of compilation time is spent in instruction selection. The local register allocator (which is one of the simplest in LLVM) and the live analysis of variables do not cost much. With optimizations and the linear scan register allocator (the most efficient register allocator of LLVM), instruction selection is also the most time-consuming pass.

In comparison, the CLI does not need to execute many startup methods. We execute N3 with a simple `HelloWorld` program. Table 4 gives the compilation times when optimizations are off, and Table 5 gives compilation times with all optimizations. Overall, for a `HelloWorld` program, N3 compiles 91 methods and 1273 bytecodes to execute the program.

4.4 Memory footprint

The memory footprint of our VMs is larger than other VMs. We did not perform any memory footprint tuning in JnJVM and N3. Moreover, LLVM is a large piece of software which is currently not concerned on its footprint. Tables 6 and 7 compare the memory footprint of JnJVM and N3 with other virtual machines. JVMs have a larger footprint than CLI implementations because many classes

Time (ms)	Time %	Name of the Pass
283	66.2	X86 Instruction Selection
32	7.5	Live Variable Analysis
24	5.6	Local Register Allocator
23	5.4	X86 Machine Code Emitter
428		Total time

Table 4: Compilation time without optimization of a CLI HelloWorld program.

Time (ms)	Time %	Name of the Pass
254	43.3	X86 Instruction Selection
42	7.2	Live Interval Analysis
41	7	Linear Scan Register Allocator
31	5.2	Live Variable Analysis
22	3.8	X86 Machine Code Emitter
16	2.7	Predicate Simplifier
12	2.0	Global Value Numbering
540		Total time

Table 5: Compilation time with optimizations of a CLI HelloWorld program.

JVM	JnJVM	IBM	Sun	Jikes	Cacao
Memory (MB)	24	12	7	38	8

Table 6: JVM memory footprint (smaller is better)

CLI	N3	Mono
Memory (MB)	9.1	3

Table 7: CLI memory footprint (smaller is better)

are loaded and initialized at startup, which involves more class metadata objects, JIT compiled code, and static instances.

4.5 Execution time

We now analyze the steady-state performance of JnJVM and N3 in order to isolate the effects of compilation startup (for VMKit) and dynamic optimizations (for the other VMs).

$\mu\text{vm-l}$ compiles the lisp-like language to C-like abstract syntax trees (AST). Hence the performance of $\mu\text{vm-l}$ is the same as performance of C, augmented with a type-based alias analysis. Therefore, we focus on JnJVM and N3 in the rest of this analysis.

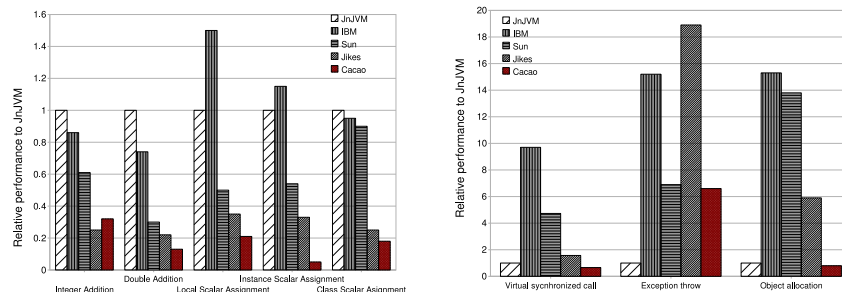


Figure 2: Section 1 of JGF (higher is better)

4.5.1 JnJVM

To compare Java implementations, we have used the Java Grande Forum (JGF) benchmark [15], the Spec JVM98 [11] and JBB05 [10] benchmarks. We have run all the VMs with a minimum and maximum heap size of 512MB. We compare JnJVM with industrial and open-source JVMs. Industrial JVMs are the Sun's OpenJDK (Java version 1.7) in server mode and IBM J9 version 1.6.0.1. The open-source JVMs are JikesRVM version 2.9.2 with the optimizing compiler as the default compiler and Cacao version 0.98. JikesRVM is the most popular open-source JVM lead by IBM, and has many contributors. Cacao is an open-source JVM focused on just in time compilation. Its developer base is equivalent to ours. JikesRVM, Cacao and JnJVM use GNU classpath as their base library.

JGF has three types of benchmarks. Section 1 contains low-level operations such as mathematical operations or memory assignments. Section 2 contains CPU-intensive applications. Section 3 contains large-scale applications. We ran the benchmarks multiple times inside the same JVM in order to exclude JIT compilation time and class loading time. On all figures, higher is better and numbers are relative to JnJVM: a number of two means two times faster than JnJVM.

Figure 2 shows a subset of the micro-benchmarks (Section 1 of JGF). We see that JnJVM has similar performance for micro-operations as industrial JVMs or JikesRVM on field accesses and arithmetic operations. However, it has considerably lower performance than these JVMs for synchronized calls, exceptions and object allocations.

Figure 3 shows the results of Section 2 of JGF, applied on different input sizes. On most benchmarks JnJVM has similar performance (from 0.7 to 1.2, higher is better) than Sun JVM and JikesRVM. IBM JVM outperforms JnJVM as well as the other JVMs. This shows that IBM developers have spent many efforts on optimizing their JVM. JnJVM also outperforms Cacao on all benchmarks.

The poor performance of JnJVM for the SOR benchmark is mainly due to our non-optimized array check removal implementation, which is based on an incomplete LLVM optimization pass. To illustrate our assumption, we compare the execution of the SOR benchmark between JnJVM without array bounds check, and JikesRVM with the `-X:opt:no_bounds_check=true` option. Figure 4

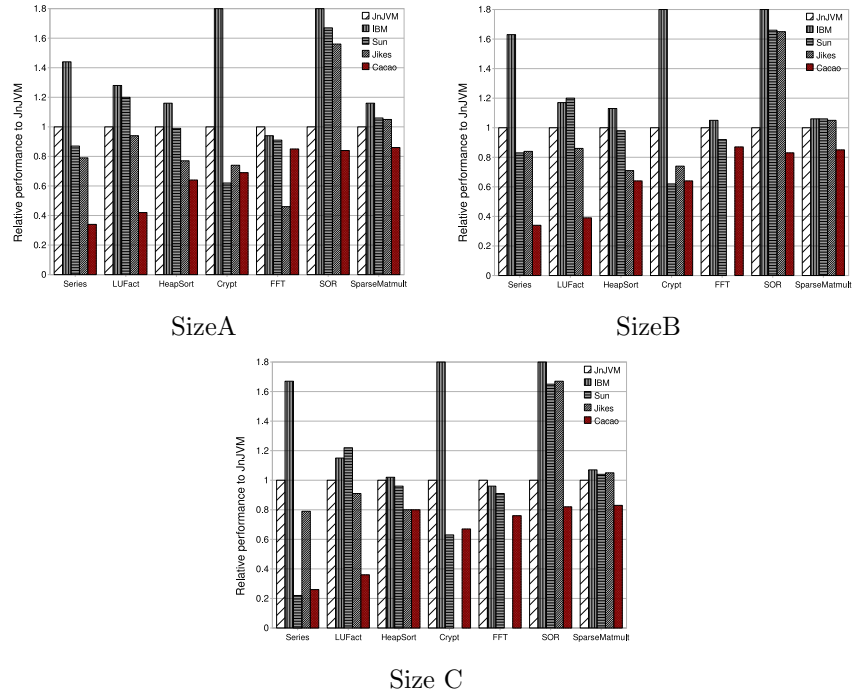


Figure 3: Section 2 of JGF (higher is better)

shows that without array bounds check, JnJVM executes the SOR benchmark with similar performance than JikesRVM.

Figure 5 show the results of Section 3 of JGF on different input sizes. A score of 0 means an out of memory error.

JnJVM has poor performance with the Euler and AlphaBetaSearch benchmarks. This is due to our lack of escape analysis. In previous work [36], we have shown that these two benchmarks are very sensitive to a good escape analysis. An optimized escape analysis will yield better performance, as the garbage collector is less invoked. In [36], we show that with escape analysis applied to our previous JVM prototype, the Euler benchmark goes from 1616 collections to 963 and the AlphaBetaSearch benchmark goes from 619 to 1 collection.

On the MolDyn, MonteCarlo and RayTracer benchmarks with small inputs, JnJVM is similar in performance to JikesRVM (from 1.0 to 1.2, higher is better). IBM and Sun JVMs outperform the other JVMs. With large inputs, JnJVM and Sun have similar performance on these benchmarks and outperform the other JVMs.

Figure 6 reports the results of the Spec JVM98 and JBB05 benchmarks applied to the JVMs (we could not include the mpeg benchmark because JnJVM is not able to run it). Except for `compress` and `db`, JnJVM does not compare with other JVMs. We profiled JnJVM on these benchmarks, using the `gprof` tool. On `jess`, `raytrace`, `javac` and `mtrt`, JnJVM spends 30% of its time in the memory manager, whether it is allocation or collection. On the `jack` benchmark,

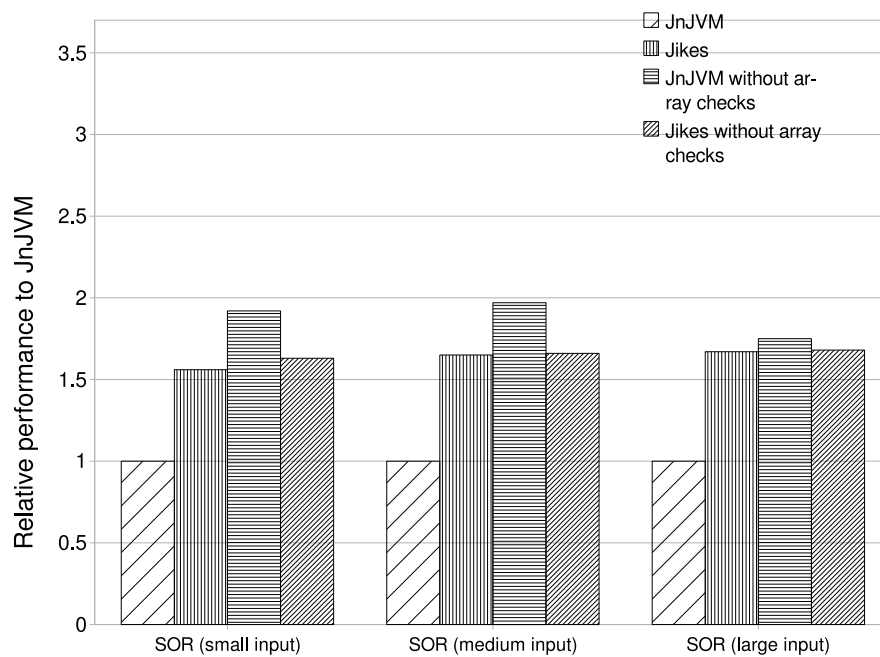


Figure 4: The SOR benchmark with and without array bounds checks (higher is better)

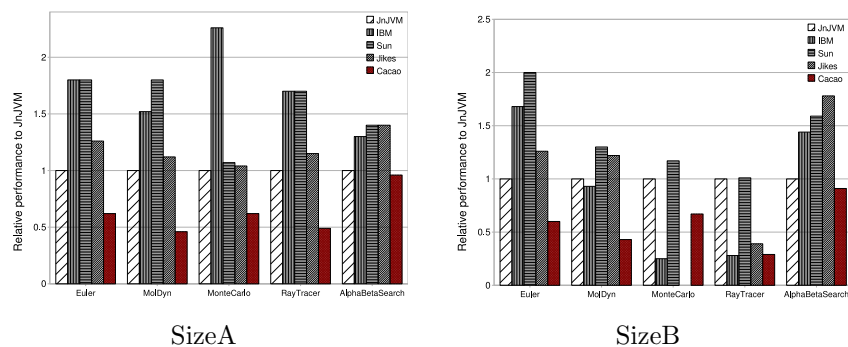


Figure 5: Section 3 of JGF (higher is better)

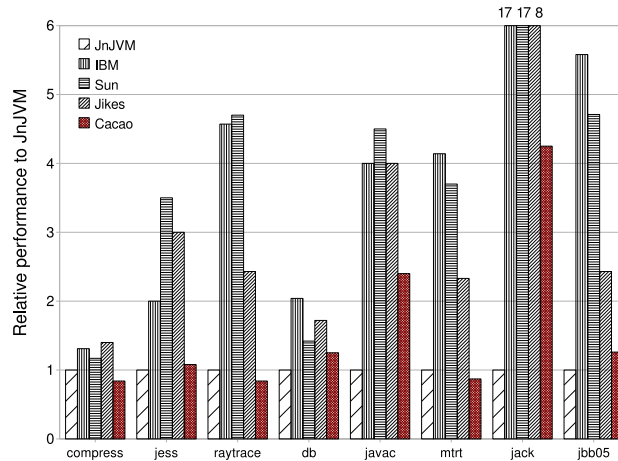


Figure 6: Spec JVM98 and JBB05 results (higher is better)

it goes to 40%. The profiler also indicated that 30% of the execution time on the *jack* benchmark is spent in the exception library.

The Spec JBB05 benchmark emulates a three-tier client/-server system and outputs the transaction rate of the server. The memory manager behavior of JnJVM in this benchmark is worse than with the JVM98 benchmarks: 62% is spent in allocation/collection routines.

4.5.2 N3

Unlike for Java, there are currently no real standard industry benchmarks available for CLI implementations. We use the *pnemark* benchmark [2]. It provides *Scimark* and *Linpack* applications which are CPU-intensive benchmarks. We compare N3 with Mono [9] version 1.2.6, Mono's developer base is comparable to JikesRVM. We did not compare with Microsoft's .Net implementation for legal reasons. Also N3 runs on Linux and Darwin and .Net only runs on Windows.

Figure 7 gives the results of the *pnemark* benchmark, relative to N3. The results show that N3 is very close to Mono in performance and yields better results on most benchmarks. Mono performs best on the *SOR* benchmark, mainly due to the non-optimized array checks removal of N3. This was also the case with the Java *SOR* benchmark. To illustrate our assumptions, we execute the benchmarks with modified Mono and N3 runtimes that remove (unsafely) all array checks. Figure 8 reports the results.

4.6 Assessments and lessons learnt

There are many areas in which VMKit could be improved. Most of the performance of VMKit is directly related to LLVM. High-level VMs implemented on VMKit achieve good performance on CPU-intensive applications because LLVM generates highly optimized code. On the other hand, the high-level VMs have a

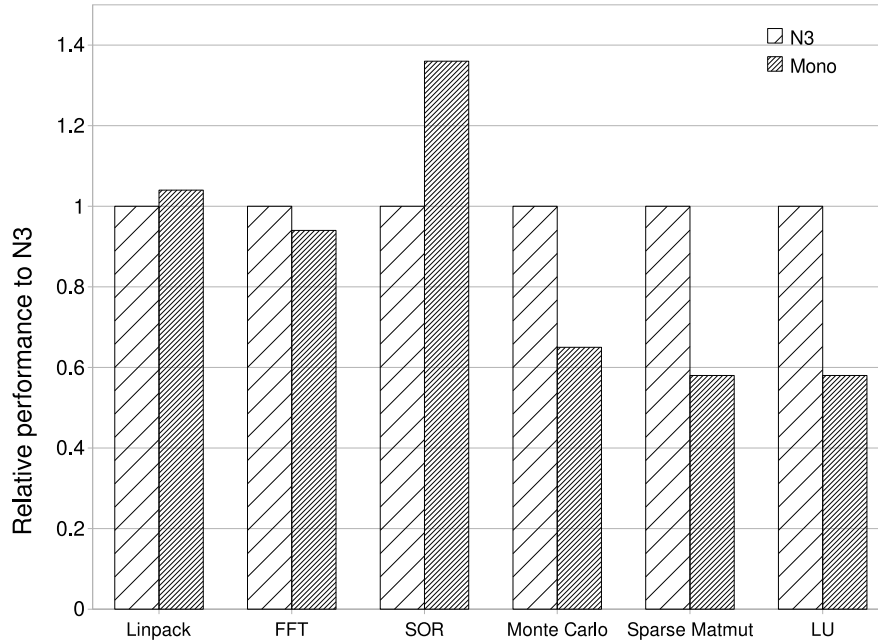


Figure 7: PNetMark results (higher is better)

large startup time because LLVM does not have a mixed mode execution. Also, the memory manager, which LLVM does not provide but VMKit implements needs to be rewritten. The GC needs a strong collaboration with the compiler, hence LLVM. The major missing features in VMKit/LLVM are:

Mixed-mode execution: Startup time as well as execution time for dynamic link resolutions in Java definitely benefit from a mixed mode of execution, when methods are first interpreted and compiled if they achieve a hotness threshold. Most industrial JVMs have this behavior [35].

VMKit relies on LLVM for the compilation engine. Since the just in time compiler of LLVM is still under development (most of the current improvements in LLVM target static compilation), there has been little to no optimization on compilation time and no implementation of a mixed-mode engine. There is however recent work of a fast instruction selector [27] and LLVM already has profiling mechanisms to measure the hotness of a method or a loop. Therefore having a mixed-mode execution is a matter of engineering efforts and should not contradict with the architecture of VMKit. We think that even so the writing of a mixed-mode engine in LLVM requires many man/year, our work shows that it is worth the price.

Cooperation between the GC and the compiler: GC time is significant in most Spec JVM98 and JBB05 benchmarks, mostly because our garbage collector (as well as the Boehm GC) traces all reachable objects on each collection.

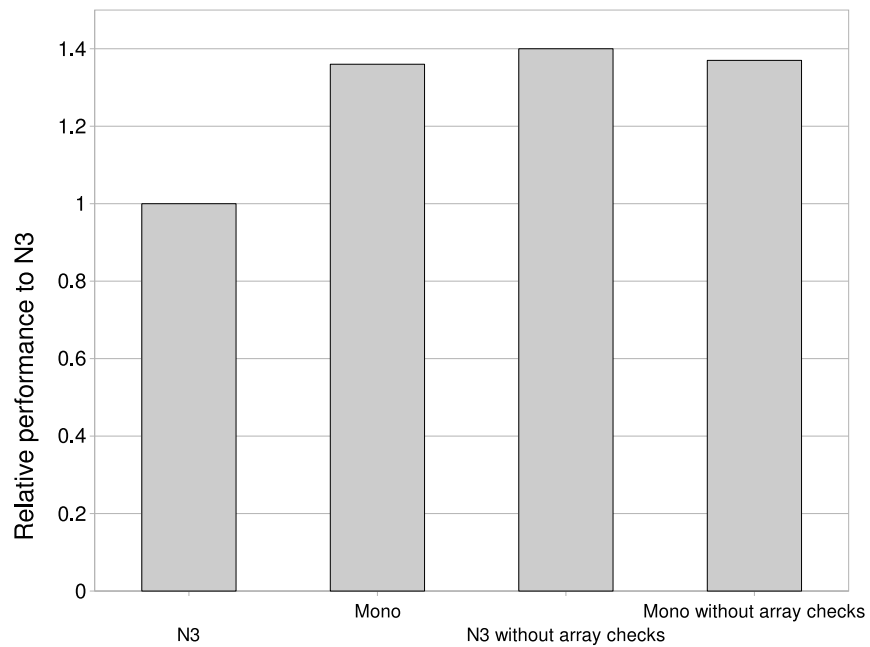


Figure 8: SOR benchmark with and (unsafe) without array checks (higher is better)

We used these GCs because they are language/VM agnostic. However, as our performance study shows, such GCs have poor performance compared to state of the art GCs. Implementing a generational GC with memory optimizations such as thread local heaps [21] or escape analysis will yield better performance on applications and benchmarks like Spec JVM 98 [33].

This kind of GC is strongly dependent on the compiler. For example the development of a generational GC requires the compiler to:

- Generate stack information on method code generation. The information allows the GC to know the stack location of objects.
- Generate write barriers, so that the GC can control the generations of objects.

LLVM already implements these functionalities, but only for static compilation. As for a mixed-mode execution, porting the functionalities to the just in time compiler is a matter of engineering efforts, rather than architectural changes. We think that LLVM getting more and more users and in the light of projects like VMKit, it will have its set of GC implementations.

5 Related Work

The inspiration for this work partly comes from the Flux OSKit project [23] and partly from the availability of several low-level services that can be used in implementing VMs, such as compiler infrastructures [31, 28, 34] or garbage collection libraries [14, 5].

OSKit has shown that a common substrate can be used for implementing kernel OSes, and even a language specialized OS integrating the Kaffe JVM [23]. VMKit applies the substrate idea to the design of VMs.

Some of the ideas behind VMKit come from a previous study in the development of a modular JVM [24]. However, sharing in this work was limited to the implementation of variants of JVM. VMKit goes much further by demonstrating that this common substrate can be successfully used to implement VMs differing in their functionalities.

The GNU Compiler Collection (GCC) [7] is the most representative example of a compiler shared between different languages. However, GCC does not have a JIT, therefore VMs such as JVM and CLI can not be implemented with it. The `gcj` project of GCC compiles statically Java bytecode to native code, and `gij` interprets Java classes.

VPU [31] is a JIT that exposes its intermediate representation (IR) as an application programming interface. Virtual machine developers use this interface to translate their bytecode or language to native code. Our first prototype of JnJVM [36] is an example of a JVM on top of VPU. VPU has shown that an intermediate representation between the processor and the bytecode can be the target of high-level VMs. While the performance of the JnJVM prototype was equivalent to the open-source Kaffe JVM, it was one third of commercial JVMs. VPU does not have all aggressive optimizations of modern VM compilers.

The Common Language Infrastructure (CLI) [25] defines an instruction set, called the Common Intermediate Language (CIL). The CLI targets many different languages with different needs, like C#, Java, Smalltalk, Python, etc.

Each language (or its bytecode) is compiled to the CIL, the only language that the CLI understands natively. The difference between an implementation of a JVM on top of the CLI (such as IKVM.Net on top of Mono [9]) versus on top of LLVM is that with LLVM there is no need to perform a translation between a high-level bytecode (JVM bytecodes) representation to another (CIL). With LLVM, JVM bytecodes and CIL bytecodes are directly translated to an IR targeted for compilation. Moreover, there are some VM mechanisms (for example class loading) that can not be implemented efficiently with the CLI [22].

The JikesRVM [8] provides a JVM on which new VM technologies can be experimented. There is currently no other VM implemented with the JikesRVM compilation system, only a dynamic binary translator [32]. Parley [16] is an interoperability layer for VMs. Its goal is to provide efficient communication between programs targeted for different VMs. By comparison to VMKit, it does not build VMs from scratch but modifies existing VMs for communication, thus it does not share common VM components.

The Open Runtime Platform (ORP) [17] provides a VM infrastructure with a JIT compiler and GC frameworks. The goal is that garbage collection research advances (or JIT compilation) can be implemented independently from JIT compilation (or garbage collection respectively). The difference between ORP and our work is that ORP focused on componentizing the substrate while we focus on the ease of development of high-level VMs.

6 Conclusion

In this paper, we have presented an approach for designing VMs using a common substrate which can be efficiently reused between VMs. Our current implementation of this substrate is sufficiently mature so that high-level VMs implemented using VMKit demonstrate performance close to industrial VMs on CPU-intensive applications.

The main lesson learnt is while we think LLVM has reached maturity for being used as a compilation engine, there are definitely research opportunities to provide a generic garbage collector. We have shown that current generic GCs, which do not cooperate with the compiler and thus can only be conservative, do not compete with state of the art GCs in industrial VMs.

Sharing a common substrate between VMs has proved to offer many facilities. We hope that this will motivate the required engineering efforts to maintain and improve VMKit.

7 Availability

VMKit, JnJVM and N3 are publicly available via an open-source license at the URL:

<http://vmkit.llvm.org>

References

- [1] Apache felix. <http://felix.apache.org/site/index.html>.

- [2] DotGNU portable.NET. dotgnu.org/pnet.html.
- [3] The GNU Classpath Project. www.gnu.org/software/classpath/classpath.html.
- [4] Jakarta tomcat. <http://jakarta.apache.org/tomcat/>.
- [5] Objective-c 2.0 overview. <http://developer.apple.com/leopard/overview/objectivec2.html>.
- [6] The Dwarf Debugging Standard. dwarfstd.org.
- [7] *The GNU Compiler Collection*. <http://gcc.gnu.org/>.
- [8] The Jikes Research Virtual Machine. <http://www-124.ibm.com/developerworks/oss/jikesrvm>.
- [9] The Mono Project. www.mono-project.org.
- [10] The Spec JBB05 Benchmark. <http://www.spec.org/jbb05/>.
- [11] The Spec JVM98 Benchmark. <http://www.spec.org/jvm98/>.
- [12] B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications Conference*, pages 108–124, Tampa Bay, USA, 2001.
- [13] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the Programming Language Design and Implementation Conference*, pages 258–268, Montreal, Canada, 1998.
- [14] H. Boehm, A. 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.
- [15] Edinbough Parallel Computing Center. Java grande forum benchmark suite – version 2.0. <http://www.epcc.ed.ac.uk/javagrande/>, 2003.
- [16] P. Cheng, D. Grove, M. Hirzel, R. O’Callahan, and N. Swamy. Parley: Federated Virtual Machines. In *Workshop on the Future of Virtual Execution Environments*, September 2004.
- [17] M. Cierniak, B. T. Lewis, and J. M. Stichnoth. Open Runtime Platform: Flexibility with Performance using Interfaces. In *Proceedings of the Java Grande Conference*, pages 156–164, Seattle, USA, 2002.
- [18] G. Czajkowski and L. Daynés. Multitasking without Compromise: a Virtual Machine Evolution. *SIGPLAN Not.*, 36(11):125–138, 2001.
- [19] L. Daynés and G. Czajkowski. Sharing the Runtime Representation of Classes Across Class Loaders. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP’05)*, pages 97–120, Glasgow, UK, 2005. Springer-Verlag.

- [20] C. de Dinechin. C++ Exception Handling. *IEEE Concurrency*, 8(4):72–79, 2000.
- [21] T. Domani, G. Goldshtein, E. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local Heaps for Java. In *SIGPLAN Notices*, pages 76–87. ACM Press, 2002.
- [22] C. Escoffier, D. Donsez, and R. S. Hall. Developing an OSGi-like Service Platform for .NET. In *Proceedings of the Consumer Communications and Networking Conference*, Las Vegas, USA, January 2006. IEEE Computer Society.
- [23] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for Kernel and Language Research. In *In Proceedings of the Symposium on Operating Systems Principles (SOSP'97)*, pages 38–51, Saint-Malo, France, October 1997.
- [24] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. A Lazy Developer Approach: Building a JVM with Third Party Software. In *International Conference on Principles and Practice of Programming In Java (PPPJ 2008)*, Modena, Italy, September 2008.
- [25] ECMA International. Common Language Infrastructure (CLI), 4th Edition. Technical Report ECMA-335.
- [26] C. Lattner. The LLVM Compiler System. In *Bossa Conference on Open Source, Mobile Internet and Multimedia*, Recife, Brazil, March 2007.
- [27] C. Lattner. Private conversation, 2008.
- [28] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, USA, March 2004. IEEE Computer Society.
- [29] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 36–44, 1998.
- [30] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley, 1997.
- [31] 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.
- [32] I. Rogers and C. Kirkham. JikesNode and PearColator: A Jikes RVM Operating System and Legacy Code Execution Environment. In *Proceedings of the Programming Languages and Operating Systems Workshop*, Glasgow, UK, July 2005. ACM.
- [33] S. Dieckmann and U. Hlzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*, pages 92–115, Lisboa, Portugal, 1999. Springer Verlag.

- [34] D. Sugalski. Building a Multi-Language Interpreter Engine. In *International Python Conference*, Februray 2002.
- [35] T. Sukanuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-in-time Compiler. *SIGPLAN Not.*, 36(11):180–195, 2001.
- [36] G. Thomas, N. Geoffray, C. Clément, and B. Folliot. Designing Highly Flexible Virtual Machines: the JnJVM Experience. John Wiley & Sons, Ltd., 2008.

Contents

1	Introduction	3
2	The Design of VMKit	4
2.1	Design Overview	4
2.2	VMKit	5
2.2.1	Computation engine: the Low-Level Virtual Machine	6
2.2.2	Memory manager	6
2.2.3	Exceptions manager	6
2.2.4	Thread and I/O subsystem	6
3	Developing High-Level Virtual Machines on top of VMKit	7
3.1	JnJVM: a Java Virtual Machine	7
3.2	N3: a CLI virtual machine	8
3.3	μ vm-1 : a Lisp-like language with type inference	9
4	Evaluations	10
4.1	Code base	11
4.2	Virtual Machine Prototyping	11
4.3	Startup time	11
4.4	Memory footprint	12
4.5	Execution time	13
4.5.1	JnJVM	14
4.5.2	N3	17
4.6	Assessments and lessons learnt	17
5	Related Work	20
6	Conclusion	21
7	Availability	21



Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399