VMKit: a Substrate for Managed Runtime Environments

Nicolas Geoffray
Université Pierre et Marie Curie
LIP6/INRIA/Regal
Paris, France
nicolas.geoffray@lip6.fr

Gaël Thomas
Université Pierre et Marie Curie
LIP6/INRIA/Regal
Paris, France
gael.thomas@lip6.fr

Julia Lawall
University of Copenhagen
Universitetsparken 1
Copenhagen, Denmark
julia@diku.dk

Gilles Muller
Université Pierre et Marie Curie
LIP6/INRIA/Regal
Paris, France
gilles.muller@lip6.fr

Bertil Folliot
Université Pierre et Marie Curie
LIP6/INRIA/Regal
Paris, France
bertil.folliot@lip6.fr

Abstract

Managed Runtime Environments (MREs), such as the JVM and the CLI, form an attractive environment for program execution, by providing portability and safety, via the use of a bytecode language and automatic memory management, as well as good performance, via just-in-time (JIT) compilation. Nevertheless, developing a fully featured MRE, including e.g. a garbage collector and JIT compiler, is a herculean task. As a result, new languages cannot easily take advantage of the benefits of MREs, and it is difficult to experiment with extensions of existing MRE based languages.

This paper describes and evaluates VMKit, a first attempt to build a common substrate that eases the development of high-level MREs. We have successfully used VMKit to build two MREs: a Java Virtual Machine and a Common Language Runtime. We provide an extensive study of the lessons learned in developing this infrastructure, and assess the ease of implementing new MREs or MRE extensions and the resulting performance. In particular, it took one of the authors only one month to develop a Common Language Runtime using VMKit. VMKit furthermore has performance comparable to the well established open source MREs Cacao, Apache Harmony and Mono, and is 1.2 to 3 times slower than JikesRVM on most of the DaCapo benchmarks.

Categories and Subject Descriptors D.3.4 [*Programing Languages*]: Runtime Environments - Compilers

General Terms Design, Performance, Experimentation

Keywords Virtual machine, Just in Time Compiler, VMKit, LLVM

VEE'10, March 17–19, 2010, Pittsburgh, Pennsylvania, USA.

1. Introduction

Managed runtime environments, such as the Java Virtual Machine (JVM) [28] or the Common Language Infrastructure (CLI) [20],[1] are becoming the norm for executing programs in environments ranging from web servers to desktops and even embedded systems. An MRE executes an intermediate representation of an application, which is portable, compact and, most important of all, can be checked to enforce safety. An MRE can in principle support many languages; for example, the CLI supports languages ranging from the imperative language C# to the functional language F#, and the scripting language IronPython. Nevertheless, an MRE inevitably imposes some design decisions on the languages that target it. For example, the JVM does not provide non-GC allocated structures or pointers, which are required for languages such as C or C++, and the CLI does not perform dynamic class unloading for a single application domain, which can introduce memory leaks for application servers such as OSGi [21] or Jonas [33] where classes are updated frequently. New managed runtime environments are thus potentially needed to support either new languages or MRE features [10, 16, 25, 34, 44, 47].

Implementing a new managed runtime environment, however, is a daunting task and may require implementing complex modules such as a Just In Time (JIT) compiler and an exact garbage collector (GC). Convincingly demonstrating the interest of a research prototype requires that it be competitive with other well established projects. Meeting such goals requires a huge development effort: (i) many functionalities that are uninteresting from a research point of view must be implemented to run standard benchmarks, and (ii) reaching the performance of current MREs requires spending a lot of time in finding, optimizing and removing performance bottlenecks. To allow the development of MREs for new languages and new language features, it is thus essential to reduce this development burden.

In this paper, we propose a solution to help experimentation in the domain of MREs based on a reusable substrate called VMKit. Our approach splits the implementation of an MRE into two independent layers. The first layer consists of VMKit, which provides basic functionalities: threads, GC-based memory management and a JIT compiler for a language-independent intermediate language.

[1] The CLI defines the executable code and the managed runtime environment of the Microsoft .NET Framework [43].

The second layer, called a high-level MRE, instantiates the substrate for a particular high-level language or class of high-level languages. It defines a complete and specific managed runtime environment.

The key challenge in building VMKit is to ensure that it does not impose any design decisions on the high-level MREs. VMKit thus defines the core of an MRE but does not impose any object model, type system or call semantics, which instead are defined by the high-level MRE. In particular, VMKit lets a high-level MRE control how memory is allocated (GC, non-GC, on stack), and how methods are dispatched (direct call, indirect call, single dispatch, multi dispatch, etc).

The contributions of this paper are the design and preliminary implementation of the VMKit substrate, a methodology for using this substrate in developing high-level MREs and a complete evaluation of the approach. Our implementation relies on the state of the art third-party projects LLVM [35] for the JIT compiler, MMTk [3, 23] for the garbage collector and the POSIX Thread library [9] for the thread manager. As these projects were not initially designed to work together, a challenge in the development of VMKit has been to define the necessary glue between them to construct an efficient and language-independent substrate, without modifying these projects. Technically, the glue allows the use of an exact GC integrated with the JIT compiler in a multi-threaded environment without imposing the object model, the type system or the call semantics. We also report our experience in building two high-level MREs to show the usability of VMKit: a Java Virtual Machine (J3) and a CLI implementation (N3). Although the JVM and CLI have many similarities, there are enough differences between them to highlight the reusability of VMKit.

The main lessons learned from our work are:

- One can build different high-level MREs on top of a shared substrate very easily. Out of the roughly 500,000 lines of code of J3 or N3,[2] only 4% (20,000 lines of code) are devoted to implementing the high level MRE, while the remaining 96% are provided by the substrate. Once we had gained experience with the J3 implementation, which was developed in parallel with VMKit, it took one of the authors one month to develop N3. In other work, we have also developed a variant of the Multitasking Virtual Machine [16] in J3, in around one month, without needing to change the substrate [25].
- By providing a substrate that includes a state-of-the-art JIT compiler and GC, VMKit supports the development of MREs that have performance similar to other research or open source MREs, such as Cacao, Harmony, and Mono, on *e.g.*, the DaCapo Benchmark suite [4]. Nevertheless, optimization opportunities remain. For example, our JVM does not yet provide an adaptive compiler [14], optimized array bound and null pointer check elimination [38], or copying collectors. J3 remains therefore 1.2 to 3 times slower than the state-of-the-art MRE JikesRVM.

The remainder of this paper is organized as follows. Section 2 presents the overall design of VMKit. Section 3 introduces our target high-level MREs 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

VMKit provides a JIT compiler, a memory manager, and a thread manager, as shown in Figure 1. We have selected these modules as the basis of VMKit because they are required by most existing high-level MREs. However, a high-level MRE is not obligated to use all of them. For example, if the MRE does not want to define threads or does not want to use the garbage collector, it can simply never call the corresponding modules.

[2] This number excludes the number of lines of code for the base libraries (rt.jar/glibj.jar, mscorlib.dll).

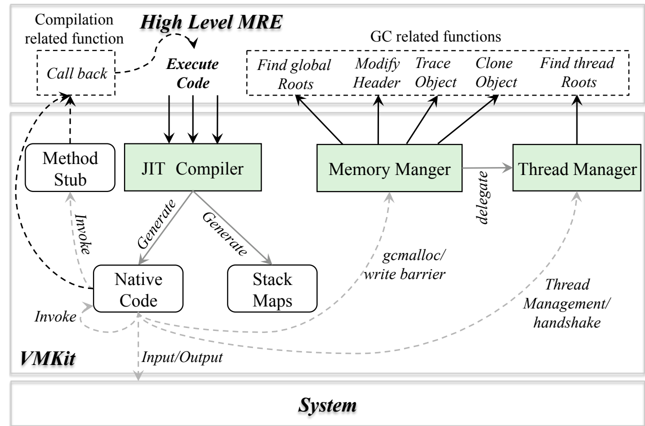

Figure 1. VMKit and a high level MRE. A grey box represents a module and a white box represents code generated by the JIT compiler. Black arrows represent calls between VMKit and the High Level MRE while grey arrows represent calls inside VMKit. Solid arrows are calls between modules and dashed arrows are calls between generated code (or an interpreted function) and a module.

To obtain good performance, however, it is not sufficient to provide these modules; they must also work together. In particular, the GC needs the help of the JIT compiler to understand the structure of the execution stack (or stacks, in the multi-threaded case) and the help of the thread manager to construct multi-threaded garbage collectors. Therefore, the implementation of VMKit adds glue between these modules: (i) glue between the GC and the JIT compiler to collect precise information about the layout of function frames and (ii) glue between the thread manager and the GC to build multi-threaded garbage collectors.

In addition to the glue, VMKit relies on a small interface that must be exported by the high-level MRE, as illustrated in Figure 1. This interface allows VMKit to be independent of the high-level MRE, by providing information that VMKit requires about various MRE design decisions. The functions in this interface are:

- Compilation-related functions: The high-level MRE must provide a callback function that is capable of generating code for an arbitrary function. This callback function allows the compiler to perform lazy compilation. Rather than compiling all functions during initialization, VMKit compiles a function only when it is first executed, by invoking this callback function.
- GC-related functions: The MRE must provide a collection of five functions that help the GC by reifying the structure of the root objects and the structure of the heap. These functions allow the GC to be independent of the object and type models used by the high-level MRE.

Concretely, our implementation of VMKit uses LLVM [35] as the JIT compiler, MMTk [3, 23] as the memory manager, and POSIX Threads [9] as the thread manager. We have made no modification to these projects to keep the code of VMKit as independent as possible. The rest of the section presents in detail these modules and their integration into VMKit.

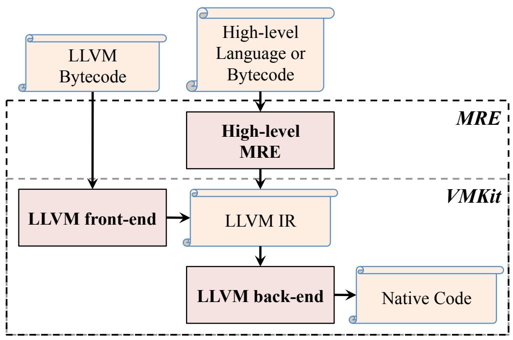

Figure 2. Compilation process in VMKit.

2.1 The JIT Compiler

The JIT compiler is the main entry point of VMKit for a high-level MRE. This compiler must support a general purpose instruction set to allow a high-level MRE to build the intermediate representation of arbitrary functions. This instruction set should be language-independent and it should produce efficient code. The JIT compiler must furthermore also generate stack maps for the GC to allow precise scanning of the heap, as described further in Section 2.2.

We have chosen to use LLVM [35] as the JIT compiler because it meets our requirements: (i) it does not impose any object model or type system, (ii) it does not impose any call semantics, (iii) it generates very efficient code and (iv) it can generate stack maps. The LLVM JIT compiler does, however, optimize all functions to the same degree, rather than focusing on functions that are frequently called (adaptive compilation), thus potentially introducing some unnecessary compilation overhead. Nevertheless, we believe that the benefits of LLVM outweigh this weakness for a first implementation of VMKit.

As shown in Figure 2, a high-level MRE uses the interface proposed by LLVM to build the intermediate representation of a function. A function generated by a high-level MRE may call: (i) functions provided by VMKit such as the memory allocator of the memory manager, (ii) functions provided by the high-level MRE and (iii) other generated functions. After the construction of the intermediate representation, the high-level MRE delegates the compilation to LLVM to generate the native code. LLVM also defines a bytecode language that allows serializing the intermediate representation to a file. VMKit only uses this bytecode to store and load a precompiled version of MMTk, as described in Section 2.2.

We now briefly present the instruction set of LLVM and two features proposed by LLVM that we exploit to build high-level MREs: compiler extension with intrinsics and lazy compilation.

Instruction set of LLVM LLVM defines an abstract register machine, i.e, LLVM supposes the existence of an infinite set of registers and all operations are executed on these registers. A high-level MRE projects each local variable or temporary of a function to an abstract register. During compilation, LLVM uses a linear scan register allocation algorithm [42] to select the abstract registers that will be implemented in a concrete register of the underlying CPU and the ones that will be implemented in the function frame (spilling).

The intermediate representation of LLVM contains five kinds of instructions: (i) arithmetic, copy and cast operations on registers, (ii) local control flow instructions (branch), (iii) method invocation (direct and indirect), (iv) memory reads and writes and (v) intrinsics, described below. This instruction set does not impose the object model. Nevertheless, a high-level MRE must give the definition of its data structures to LLVM so that LLVM can use it to optimize memory writes and the access modes of registers (float, integer, pointers, etc.).

Compiler extension with intrinsics During the transformation from a high-level language to the LLVM intermediate representation, many common sequences of instructions may be introduced for operations that are implicit in the high-level language, such as getting the length of an array in an array bounds check or dynamically checking the type of an object. Because LLVM can not predict that the accessed memory is immutable (in Java, the length of an array or the type of an object can not change), LLVM considers that the result of such sequences depends on the state of the memory, and thus the common subexpression elimination pass of LLVM does not remove redundant copies of such sequences when they are separated by writes or function calls. To solve this problem, LLVM proposes the notion of an intrinsic: a special function that is associated with information about any side effects it performs. An intrinsic is analysed by LLVM like a direct method invocation, however, if it is specified that an intrinsic does not have any side effects, LLVM can eliminate redundant invocations and lift such invocations outside loops.

Intrinsics are language-dependant and thus LLVM does not know how to handle them. To provide this information, the high-level MRE must define an external compilation pass for each intrinsic. A compilation pass is applied to the intermediate representation to optimize or modify it. It handles an intrinsic by transforming it into zero or more known instructions.

Lazy compilation To avoid the compilation of the whole application during initialization, LLVM provides a lazy compilation mechanism that relies on a callback function (the Compilation Related Function of Figure 1). When LLVM has to generate a direct invocation of a function that has not yet been compiled, it inserts a call to the callback function instead. The high-level MRE provides the callback function. It loads and generates the intermediate representation of the lazily compiled function, and finally delegates to LLVM the generation of its native representation. Once the function is generated, LLVM patches the calling site to call the newly generated function for subsequent calls.

In the same way, to allow lazy binding with other invocation semantics, LLVM defines stub functions. A stub function calls the above callback function. Invocation semantics such as single dispatch or multi-dispatch can then be decomposed into a sequence of memory reads to dynamically find the pointer to the receiver function, followed by an indirect method invocation through this pointer. For lazy binding, the indirect invocation first calls the stub, which calls the callback function. The callback function then generates the native code of the invoked method and updates the pointer to the stub to the native code.

2.2 The Memory Manager

The memory manager is in charge of memory allocation and reclamation. The efficiency of these operations has a critical impact on the overall performance of a high-level MRE. Current optimized state of the art GCs are exact and thus they must have accurate knowledge of the locations of references in memory. So that this constraint does not cause the GC to impose any object model or type system on the MRE, the MRE is required to provide functions to allow the GC to find references into the heap (see the box GC related functions in Figure 1).

We have chosen the Memory Manager Toolkit (MMTk) [3] as the memory manager for VMKit. MMTk is a state of the art toolkit for writing memory managers. It was initially used in the JikesRVM [32] Java Virtual Machine and provides a wide variety of powerful state of the art garbage collectors. MMTk is independent from the rest of JikesRVM and does not impose any

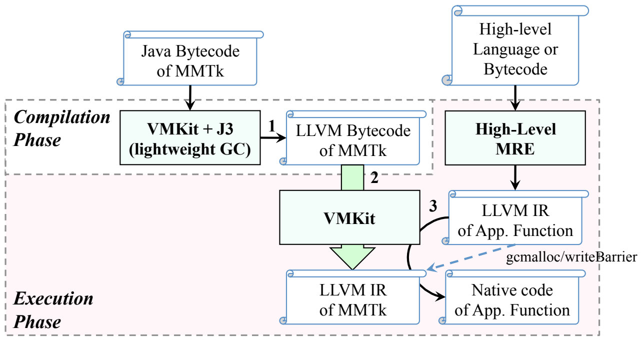

Figure 3. Integration of MMTk in VMKit. Solid arrows are compilations or transformations. Dashed arrows are invocations. The big solid arrow is the initialisation phase of VMKit.

object model or type system. It therefore meets our requirements. Nevertheless, using MMTk in VMKit raises two significant issues: (i) linking MMTk with VMKit, as MMTk is implemented in Java while LLVM and the rest of VMKit are implemented in C++, and (ii) providing MMTk's GCs with exact knowledge of the object locations in the heap and the threads execution stacks.

Linking MMTk with VMKit To link an MMTk function with VMKit, VMKit must have access to the MMTk function's LLVM code. To create this code, we exploit J3, our JVM built upon VMKit. We use J3 to load and compile MMTk during the compilation of VMKit (see Figure 3). Because LLVM includes a traditional ahead of time compiler, it can generate LLVM bytecode files. Thus, the compilation of VMKit is separated into two phases. During the first phase (arrow 1), J3 is executed with a lightweight memory manager and the rest of VMKit to generate the LLVM bytecode of MMTk. This bytecode of MMTk is not Java specific and VMKit can load it without any MRE. The second phase is performed during the initialisation of VMKit. Before launching a high-level MRE, VMKit loads and dynamically compiles the LLVM bytecode of MMTk (arrow 2). Once MMTk is compiled in VMKit, VMKit can launch a high-level MRE.

The LLVM bytecode of a function generated by a high-level MRE can call functions of MMTk, for example to allocate an object or to execute a GC write barrier. Object oriented languages intensively use these two functions and thus these calls must be inlined (arrow 3) by the high-level MRE to obtain good performance. Our strategy for linking MMTk with VMKit also allows inlining. Specifically, VMKit keeps the generated LLVM intermediate representation of the MMTk code. When a high-level MRE generates the intermediate representation of a function, it can thus inline the intermediate representation of the GC-related functions.

MMTk is normally linked with JikesRVM which provides about a hundred functions to compute statistics, to know the bounds of the heap, to allocate pages, to access the processor type, to allow debugging, to handle mutator threads, to create the collector thread and to understand the object layout. Most of these functions only depend on the operating system, underlying processor and Thread Manager. We have therefore directly implemented these functions in VMKit. VMKit delegates only five functions related to the object model or the type system to the high-level MRE, as described below.

Exact memory map To provide exact knowledge of the object locations, a high-level MRE must provide functions to allow the GC to accurately scan the heap. These functions depend on the MRE's heap organisation, object model, type system and execution stack layout. They are represented by the box GC related functions in Figure 1.

A high-level MRE must first provide a function to find the global roots of the object graph. These are the global variables (static variables in Java or C#), the constant objects and the reifications of internal structures (classes, fields, methods, assemblies, etc.). To find the roots in the execution stack of a thread, help from LLVM is needed. LLVM provides an intrinsic called `gcroot` that takes an abstract register as argument to indicate that the register contains a reference. LLVM uses this intrinsic to generate, at each collection point (handshake point), a stack map, i.e, a map that indicates where are the root objects in the frames. A high-level MRE can thus identify the references in the function frames by using these generated stack maps. In the execution stack, frames from the MRE are interspersed with frames from the application executed by the MRE. Stack maps of the former are generated during the compilation of a high-level MRE by LLVM with llvm-gcc, the C/C++ front-end of LLVM, and stack maps of the latter are generated on the fly during the compilation of a method with the LLVM's JIT.

To traverse the reachable object graph, MMTk must find the directly reachable object references from each object. Therefore, a high-level MRE must provide a function to *trace* an object, i.e, to find its directly reachable sub-objects. Furthermore, MMTk does not know the exact layout of a data structure; in particular, it does not know the locations of virtual tables or type descriptors. A high-level MRE must therefore provide functions to change the content of the header of an object (i.e, to assign a type to a data structure or to change its mark during a collection) and to clone an object, for copying collectors.

2.3 The Thread Manager

Multithreaded MREs need to be able to start threads and to synchronize them (lock, variable condition, join). For these thread-related operations, we have chosen to use the POSIX Thread Library implementation provided by the execution host. There is, however, one case in which the POSIX Thread Library is not sufficiently efficient: an MRE that supports multi-threaded garbage collection requires that each thread have its own memory area to store thread-private data such as the per-thread memory allocator data or the remembered set of inter-generational references [5, 30]. The latter set is accessed at each write to a reference and must therefore be accessible with the lowest possible overhead.

To address the constraints of efficient multi-threaded garbage collection, we associate each thread with thread-local storage implemented in the first few pages of the thread's execution stack. Because each stack is aligned on a power of 2 boundary, finding a pointer to the local storage of a thread requires only masking the thread's stack pointer with the boundary complemented by two. This approach gives access to the thread local storage without performing a function call, as would be required with POSIX. Alternatively, we could have reserved a register to hold a pointer to the thread-local storage, but this approach would have required modifications to LLVM.

As shown in Figure 1, a high-level MRE must provide to the thread manager a function for finding the root objects in the execution stack of each thread. When a garbage collection is triggered, all threads must join a handshake, so that the garbage collector knows the objects referenced on the stacks. To implement a handshake, a thread-local boolean variable is regularly polled by the threads (on return calls and backward branches) to verify if a collection is happening. Once all threads have joined the handshake, the thread manager invokes the function provided by the high-level MRE to find the root objects in the threads' execution stacks.

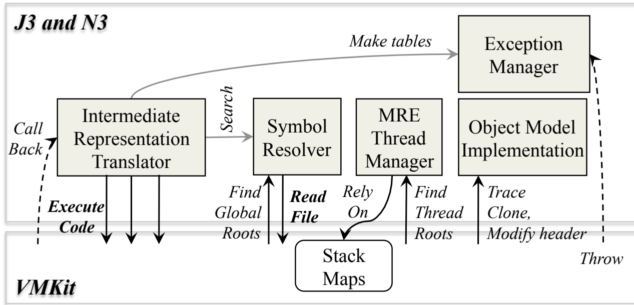

Figure 4. Implementation of J3 and N3.

3. Developing the J3 and N3 High-Level MREs using VMKit

To illustrate the process of building a high-level MRE and to show that VMKit is well-suited for this purpose, we have implemented two high-level MREs using VMKit: a Java virtual machine [37] called J3 and a Common Language Infrastructure (CLI) [20] called N3. In our implementation of J3, the goal has been to demonstrate the efficiency of VMKit. On the other hand, in our implementation of N3, the goal has been to demonstrate that by using VMKit it is possible to rapidly prototype an MRE, while still obtaining an implementation that is reasonably efficient.

The Java specification and the CLI specification both require an object-oriented high-level MRE with threads and exceptions. Therefore, as presented in Figure 4, we have chosen to design both high-level MREs according to the same architecture with the same set of modules. Within these modules, J3 and N3 implement the Compiler-related and GC-related functions required by VMKit, as described in Section 2.

The rest of this section presents the implementation of these modules for J3 and N3 in detail.

3.1 J3 implementation

J3 is a JVM implementation that interfaces with the GNU Classpath implementation of the Java libraries [26]. The rest of this section presents our choices in the J3 implementation.

The symbol resolver The symbol resolver is in charge of class, field and method resolution. In Java, symbols are managed by class loaders, which both load classes and provide a namespace for classes that have been loaded [36, 37].

A Java class loader groups together a set of classes to enforce isolation and to permit unloading: a class and all its associated native code are unloaded by the GC when its class loader is no longer referenced. When a class loader is reached during garbage collection, it traces its constant objects, its global (static) variables, the reifications of its classes, and its own reification.

J3 has a direct reference to the initial class loader (the class loader of the Java library) to build the class reifications, the arrays, the strings and the internal exceptions. The GC related function to find the global root objects traces only this initial class loader. Other class loaders are not directly referenced by J3, but may be reachable through the instances of their classes and through the methods of their classes that are on the execution stack.

The object model implementation Building a high-level MRE requires defining the layout of data structures, which determines the object header access and clone functions required by the VMKit memory manager. For J3, we use a header of two words. The first one holds a thin lock [2] (a fast lock implementation that avoids atomic operations when the lock is not in contention), information for the memory manager (bits for the GC write barrier and bits to store the GC mark), and a 10-bits hashcode. The second word holds a pointer to a virtual table. The virtual table holds virtual function pointers, a pointer to the class descriptor and information to improve the efficiency of dynamic type checking instructions, following the Java HotSpot algorithm [15].

The object model implementation also provides the function to trace a Java object required by the memory manager of VMKit. This function finds the class descriptor of the object in its virtual table. The class descriptor describes the layout of the object and is used to mark and trace the directly reachable subobjects. Moreover, the class descriptor contains a reference to its class loader which is also marked and traced.

The exception manager To manage exceptions, J3 reserves a word for the pending exception in the local storage of each thread. After each method invocation, the word is tested. If an exception has been raised and the function is able to trap the exception, a branch to the exception handler is executed. Otherwise, the function returns to the caller. This implementation is not optimal because it requires a test and branch after each method invocation. A more efficient implementation would use exception tables. However, at time of writing the paper, the exception tables generated by LLVM rely on the GCC runtime library [19] which is not optimized for dynamically generated tables, at least on Linux.

The intermediate representation translator The bytecode translator relies on the LLVM interface to build the LLVM intermediate representation of the Java bytecode. A Java Virtual Machine is, however, an abstract stack machine while LLVM is an abstract register machine. Thus, the bytecode translator of J3 defines an abstract LLVM register for each entry of the Java stack, and for each parameter and local variable of a function definition. This is possible because LLVM provides an infinite set of abstract registers. To build stack maps, J3 also marks the abstract registers that hold references with the `gcroot` intrinsic. The resulting stack map is stored in the method descriptor of the compiled function.

The JVM provides 10 kinds of instructions: (i) arithmetic operations, (ii) branch and flow operations, (iii) local variable accesses, (iv) lock operations, (v) memory allocations, (vi) class comparisons, (vii) constant loading operations, (viii) array accesses, (ix) field accesses, (x) method invocations. These instructions are translated as follows:

An arithmetic, branch, or local variable operation is directly translated into the corresponding LLVM instruction, except for the Java switch instructions (i.e., `tableswitch` and `lookupswitch`) which do not exist in LLVM. These instructions are decomposed into multiple LLVM instructions that perform the same comparisons and branches.

A lock operation (i.e., `monitorenter` or `monitorexit`) is translated into a thin lock implementation [2].

A memory allocation operation (i.e., `new`, `newarray`, `anewarray` or `multianewarray`) is translated to a call to the memory manager.

A class comparison is translated into a sequence of LLVM instructions that perform the Java HotSpot algorithm for dynamic type checking [15]. J3 defines an intrinsic for this operation to allow LLVM to eliminate redundant checks (see Section 2.1).

A constant loading operation `ldc` is directly translated to a register assignment if the constant is not an object. Otherwise, the constant object is stored in the class descriptor and used as a root object, and the constant loading operation is translated to a load from memory. This solution avoids introducing the need for the GC to search the native code for references to root objects.

An array access is translated into a sequence of LLVM instructions that perform the operation. For an array length operation (i.e., arraylength), J3 defines an intrinsic to allow LLVM to eliminate redundant loads of the length. For an array read or write, J3 generates LLVM instructions to check that the access is inside the bounds, evaluate the address of the access and perform the access itself. Array bounds checks also involve accessing the length of the array. Because the access to the array length is implemented using an intrinsic and does not have side effects, LLVM is able to remove some redundant bounds checks.

A field access is translated using a lazy resolution technique, because at compilation time the compiler might not know the layout of a class nor its methods. If the class is already resolved, J3 translates a Java field access directly into the memory access instructions provided by LLVM. Otherwise, J3 inserts a runtime call to first dynamically resolve the entry of the class.

A non-virtual call (i.e., `invokespecial` or `invokestatic`) is translated into a direct LLVM call to the method when the method is already loaded, resolved and compiled. Otherwise, J3 inserts a call to the callback function provided by J3 (see Section 2.1) that resolves, loads and compiles the method, as needed. Then LLVM patches the calling site to call the real receiving method for subsequent calls.

A virtual, non-interface call, (i.e., `invokevirtual`) is translated into a sequence of LLVM instructions that load the virtual table of the object (defined as an intrinsic to avoid redundant evaluations), load 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 callback. The callback looks up the offset of the method in the virtual table, compiles the method if it is not already compiled and updates the virtual table with the real receiver method for subsequent calls.

An interface method invocation is translated following the interface method table algorithm, described in [1].

The MRE thread manager The MRE thread manager defines the MRE representation of a thread. In J3, a thread is materialized by an instance of the class `java.lang.Thread`. The MRE thread manager uses the thread manager of VMKit for thread creation, synchronisation and thread-local storage. Threads are therefore scheduled by the POSIX library or by the kernel depending on the POSIX thread implementation.

The MRE thread manager is also in charge of finding the root objects of a thread in the thread's local storage and in the thread's execution stack. The objects referenced from the thread-local storage are the pending exception and the JNI references, i.e., the Java references used by the currently executing native functions. J3 finds the root objects in the stack by scanning it frame by frame. For each frame, J3 finds the method descriptor associated with the frame in a global table. If the method is a Java method, J3 adds the class loader of the class that defines the method to the root set.[3] In all cases, J3 finds the root objects in the frame by using the stack map generated during the compilation of the function.

3.2 N3 implementation

N3 is a CLI implementation that interfaces either with the PNetLib [18] or with the Mono [39] implementation of the standard library (e.g., the System name space). Because the goal of the N3 experiment has been to show that VMKit permits rapid prototyping of an MRE, N3 is not as optimized as J3. In particular, some mechanisms that could be the same are implemented differently to decrease the development burden (principally, N3 does not use virtual tables, thin-locks, fast dynamic type checking and any of the J3-specific intrinsics). This drawback of N3, however, further highlights the genericity of VMKit: VMKit does not impose a specific implementation of a virtual machine functionality.

There are also notable differences between the JVM and the CLI specifications that further highlight the contribution of the N3 experiment:

- Structures: In the CLI specification, not everything is an object. A CLI implementation must thus support the allocation of unboxed structures, which are allocated on the stack or within 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 the JVM specification.
- Static class resolution: Contrary to the JVM specification, the CLI specification requires that a class be resolved when the compiler references it. Generating code that accesses a field, or calls a method of a class triggers its resolution. The JVM specification, permits the resolution of a class during the execution of methods, i.e., when a field is accessed or a method is called.
- Generics: In the CLI specification, the managed runtime environment handles generics, while there are no generics in the JVM specification. Instead, the source to bytecode Java compiler handles generics and produces a generic-free class file. In VMKit, the LLVM compiler does not know about generics, and therefore N3 must itself instantiate the generics and produce the LLVM representation of the instantiated method.
- Out arguments: In the CLI, the `byref` keyword indicates that an argument is passed by reference, i.e., that the callee can modify the value of the caller. No comparable operator exists in Java

The Exception Manager and the MRE Thread Manager of N3 and J3 are totally equivalent. We now describe the differences in the other modules.

The object model implementation A N3 object header consists of three words. The first one holds information for the memory manager (GC write barrier and GC mark), the second one holds a pointer to a fat lock (N3 does not use the thin-lock algorithm) and the third one holds a pointer to the class descriptor (N3 does not use virtual tables). The hashcode of an object is encoded in the first word.

While the object model of J3 permits the use of more sophisticated and efficient algorithms, by using another object model in N3, we show that the object model is independent of VMKit.

To trace an object, N3 finds the class descriptor of the object in the object's header and uses it to find the objects it references. An N3 unboxed structure, however, is not an object and does not have an object header. If an object contains unboxed structures, N3 follows them by using the descriptors of structures found in the class descriptor.

Bytecode translation Like J3, N3 is an abstract stack machine and defines the same ten kinds of bytecode operations. The only significant differences between the implementations of the instruction sets of the MREs concern the construction of the stack maps and the implementation of virtual method invocation.

The construction of stack maps in N3 must take into account unboxed structures. N3 also relies on the `gcroot` intrinsic of LLVM. If a local variable is a reference, it is marked `gcroot` and if a local variable is an unboxed structure, each object referenced by the structure is marked `gcroot`.

For virtual method invocation, N3 does not use virtual tables and instead relies on the linked list of caches based algorithm that is used for interface method invocation in J3.

The symbol resolver N3 loads assemblies while J3 loads classes. An assembly defines a set of classes. In N3, the bytecode is therefore associated with the assembly descriptor, while it is associated

[3] To avoid the destruction of the function during a collection, the class loader that defines the function must not be freed, even if there are not any instances of any class managed by this class loader.

with the class in J3. Moreover, contrarily to J3, an assembly is not unloaded when it is no longer referenced. N3 can therefore maintain a list of the loaded assemblies and, to find the global root objects, N3 explores all the assemblies to collect the global references (static), the constant objects and the reification of assemblies, properties, classes, methods and fields.

4. Evaluation

Currently, VMKit runs on Linux/x86 and MacOSX/x86. Ultimately, VMKit should be able to run on all architectures supported by LLVM. However, the JIT compiler of LLVM is still in major development, and only its x86 backend is sufficiently robust.

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

1. Code base: the development of a high-level MRE should require writing as little code as possible.
2. Ease of experimentation: the common substrate should ease the extension of high-level MREs with new features.
3. Startup time: there is always an overhead when running an application on top of an MRE, even for a simple HelloWorld program. However, this overhead should be as small as possible.
4. Memory footprint: an MRE should use as little memory as possible, beyond the memory usage of the application.
5. Steady state performance: an application should run as fast as possible, once it is loaded and compiled.

For benchmarking J3, we use a Pentium D 3GHz with 3GB of memory, running Mandrake with a Linux 2.6.23 kernel, while for N3 we use a Core 2 Duo 2.5GHz with 1.5GB of memory, running Gentoo with a Linux 2.6.27 kernel.

4.1 Code base

Table 1 summarizes the lines of code for each subsystem of the high-level MREs. VMKit consists of 12K lines of code, which are basically the glue between the JIT compiler, the thread manager and the memory manager. LLVM is about 450K lines of code and MMTk 50K.

In comparison, Sun's JVM, OpenJDK, is 6.5M lines of code, and contains multiple JIT compilers and interpreters, multiple GCs and class libraries. On a smaller scale, JikesRVM [32], which contains a JIT compiler and multiple GCs, has a code base of 270K lines of code, and Cacao, which contains multiple compilers and a GC has a code base of 230K lines of code.

J3 is a full implementation of the JVM specification [37] and should be able to run any Java application that GNU Classpath can support. These include the applications of the DaCapo benchmark suite [4]. J3 and VMKit were developed in parallel and the development time was between six and nine months for one person.

N3 is able to run simple applications as well as the PNetMark benchmark [18]. After implementing J3, it took one month for one of the authors of the paper to implement most of N3, without generics, and another month to implement them.

4.2 Ease of experimentation

To evaluate the development cost of implementing a new research feature using VMKit, we implemented parts of the MultiTasking Virtual Machine (MVM) [16] by modifying the J3 high-level MRE. MVM provides the ability to run multiple Java applications in the same execution environment, 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 method sharing between applications, including sharing across class loaders [17]. It took one of the authors of this paper one month to implement these functionalities. Integrating MVM into J3 required modifying 1000 lines of code and adding 2000 lines. No modification of VMKit was needed.

	J3	N3
Total lines of code	23200 (4.5%)	16200 (3.1%)
LLVM translation	5500	5500
Runtime engine	15000	9200
Library interface	2700	1500

Table 1. Lines of code of the high level MREs. The percentage is relative to the total number of lines of code of the MRE.

JVM	J3	IBM	Sun	Jikes	Cacao
Memory (MB)	24	12	7	38	8

Table 2. JVM memory footprint (smaller is better)

Based on this first work on MVM, we have implemented a Java virtual machine, called I-JVM, that enforces the isolation between components in OSGi [25]. OSGi is a Java-based, centralized, component oriented framework. Current implementations of OSGi are unable to isolate components from each other, and thus, for instance, a malicious component can freeze the platform by allocating too much memory or alter the behavior of other components by modifying shared variables. It took one of the authors of this paper one month to implement I-JVM, which required modifying 650 lines in the implementation of J3.

4.3 Startup time

VMKit only uses LLVM for code execution, and does not provide an interpreter or baseline compiler. Although LLVM has the advantage of being language-independent and generating efficient code, it only performs aggressive optimizations. A Java HelloWorld program thus takes 8 seconds to run without any optimization activated. When turning all optimizations on, the program runs on J3 in 12 seconds. In comparison, other JVMs execute the same program in less than one second.

Execution of the `HelloWorld` program involves compiling many methods. These methods create the class loader, load the file and print `"HelloWorld"`. With GNU Classpath version 0.97.2, 536 methods have to be compiled, resulting in a total of 26952 bytecode instructions. Other JVMs interpret these methods or apply a simple baseline compiler in order to achieve a better responsiveness.

To reduce the startup time, we plan on using the ahead of time compiler (AOT) of VMKit for Java. However, the scan of the Java objects emitted in the ahead of time compiled code, e.g. String and Class objects, by the garbage collector remains to be implemented. Notice that the compilation of MMTk by the AOT does not suffer from this missing functionality because it does not emit Java objects that the collector must scan at runtime.

4.4 Memory footprint

Tables 2 and 3 compare the memory footprint of J3 and N3 with that of other MREs. JVMs have a larger footprint than CLI implementations because they load and initialize many classes at startup, which involves more class metadata objects, JIT compiled code, and static instances. The memory footprint of our MREs is larger than that of the other MREs. We have, however, not yet performed any memory footprint tuning of J3, N3, or VMKit.

CLI	N3	Mono
Memory (MB)	9.1	3

Table 3. CLI memory footprint (smaller is better)

4.5 Steady state performance

We now analyze the steady-state performance of J3 and N3 in order to factor out the effects of compilation startup (for VMKit) and dynamic optimizations (for the MREs).

Evaluation of J3 Figure 5 shows the results of running the DaCapo benchmark release 2006-10-MR2 [4] on the following JVMs:

- J3: VMKit's JVM with MMTk's MarkSweep garbage collector.
- Cacao 0.99.4: An open-source JVM with a similar number of developers as VMKit.
- Harmony 5.0M11: Apache's JVM.
- JikesRVM/Immix 3.1: JikesRVM with the default garbage collector (Immix).
- JikesRVM/MarkSweep 3.1: JikesRVM with MMTk's MarkSweep garbage collector, which is the garbage collector used in our evaluation of J3.
- Sun JVM 1.6.0_13: The Sun JVM server implementation.

Out of the eleven benchmarks of DaCapo, we have been able to run nine on J3. Due to multi-threading bugs, the *eclipse* benchmark could not run on J3. The *chart* benchmark requires a graphics library (gtkpeer from GNU Classpath) that we could not compile on our test machine. Also, J3 uses the local register allocator of LLVM for the *fop* benchmark instead of the default linear scan register allocator because of a bug in LLVM. This change results in less optimized JIT-generated code. Finally, with Harmony version 5.0M11, the *antlr* benchmark did not run to completion because Harmony depends on a version of the antlr library incompatible with the benchmark.

We have conducted all the benchmarks with a minimum heap of 128MB and a maximum heap of 1GB. For each benchmark, we ran 31 iterations, and computed the mean of the last 30 iterations, so that the initial compilation time of the benchmark is excluded.

The benchmarks show that, while J3 has performance similar to other well-established open-source JVMs (Cacao, Harmony), it is still 1.2 to 3 times slower than JikesRVM and 1.5 to 5 times slower than OpenJDK on most DaCapo benchmarks. The main missing optimizations of our current J3 prototype are described below.

No copying collector: J3 does not yet use the copying or generational collectors of MMTk such as the Immix collector [6]. Using a copying collector of MMTk in J3 is a matter of engineering. J3 is exact and can already update the root objects on the stack. However, at time of writing this paper, we have not yet implemented the interface between MMTk and VMKit for copying objects.

No inlining of *System.arraycopy*: The *System.arraycopy* method of the Java class libraries is critical in most applications [29] and most JVMs optimize the call by inlining a fast path. Inlining a fast path in J3 should not raise particular difficulties but has not been implemented yet.

No array bounds check elimination: J3 does not provide a complete array bounds check elimination pass such as that of Lujàn et al. [38] in Sun or ABCD [7] in JikesRVM. Implementing this compilation pass in LLVM requires an engineering effort but does not raise particular difficulties.

No stack map optimization: LLVM reduces the level of optimization when generating functions with stack maps: most compiler optimizations do not apply because LLVM forces object references to be allocated on the stack and never in registers. Solving this problem would require modifying the implementation of the LLVM optimizations to take the `gcroot` intrinsics into account.

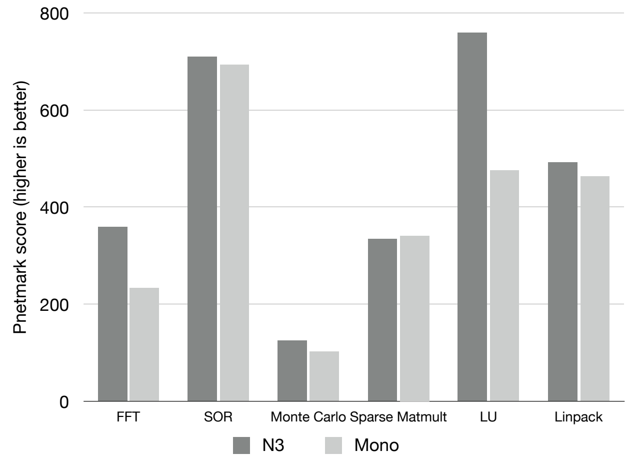

Figure 6. Mean performance of N3 on the PNetMark Benchmark. Higher is better.

No adaptive compiler: LLVM does not yet provide an adaptive optimization system, where native code is dynamically optimized on hotspots [14]. All the other JVMs have an adaptive compiler. Constructing an adaptive compiler would require significant modifications to the LLVM JIT compiler.

Evaluation of N3 Unlike for Java, there are currently no real standard industry benchmarks available for CLI implementations. We use the PNetMark benchmark [18]. It provides the Scimark and Linpack applications, which are classical scientific computations. Scimark includes the programs FFT, SOR, Monte Carlo, and Sparse Matmult. We compare N3 with Mono [39] version 2.4.2.3, an open-source CLI implementation. The number of developers of Mono is comparable to that of JikesRVM. We did not compare with Microsoft's .Net implementation because it only runs on Windows while N3 only runs on Linux or MacOS.

Figure 6 gives the results of the PNetMark benchmark relative to N3. For this benchmark, we use a Core 2 Duo 2.5GHz with 1.5GB of memory, running Gentoo with a Linux 2.6.27 kernel. The results show that N3 is very close to Mono in performance and yields better results on all benchmarks except one. However, these benchmarks only stress the JIT compiler and do not stress other aspects of the MREs.

5. Related Work

The inspiration for this work comes from the Flux OSKit project [22] and from the availability of several low-level services that can be used in implementing MREs, such as compiler infrastructures [18, 35, 41, 45] and garbage collection libraries [8, 40].

OSKit has shown that a common substrate can be used for implementing kernel OSes, including an OS integrating the Kaffe JVM [22]. VMKit applies the substrate idea to the design of MREs.

The idea of modularizing a managed runtime environment and using third-party components originates in our work on LadyVM [24], which implemented a JVM exclusively using the existing projects LLVM and Boehm GC [8]. VMKit goes much further by demonstrating that a common substrate can be used to implement MREs differing in their functionalities. Moreover, for better performance, VMKit relies on MMTk which provides a selection

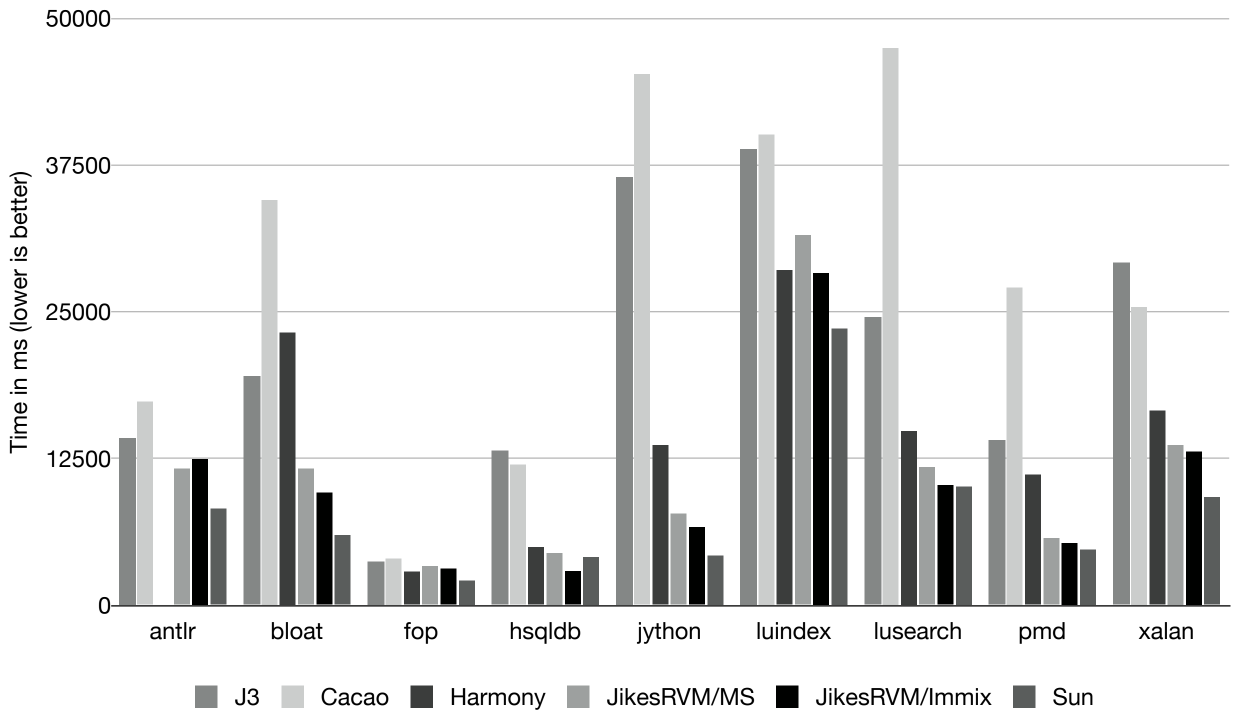

Figure 5. Mean steady-state performance of JVMs on the DaCapo Benchmark. 128MB Heap min, 1GB Heap max. Lower is better.

of exact GCs. Adding support for exact GCs entailed a significant reorganization of the design.

The GNU Compiler Collection (GCC) [27] is the most representative example of a compiler shared between different languages. However, GCC does not have a JIT, therefore MREs such as JVM and CLI cannot be implemented with it. The `gcj` project of GCC statically compiles Java bytecode to native code, and `gij` interprets Java classes. Neither `gcj` nor `gij` provides an exact map of the memory and they can therefore only rely on conservative garbage collection [8]. On the other hand, the integration of the compiler and the garbage collector in VMKit permits the use of exact garbage collectors.

VPU [41] is a JIT that exposes its intermediate representation (IR) as an application programming interface. A MRE developer can use this interface to translate a high-level language or bytecode to native code. The JVM JnJVM [46] has been implemented on top of VPU. VPU has shown that an intermediate representation between the bytecode and the processor can be the target of high-level MREs. While the performance of the JnJVM prototype was equivalent to the open-source Kaffe JVM, it was ten times slower than well established JVMs. VPU does not have all of the aggressive optimizations of modern MRE compilers.

The Common Language Infrastructure (CLI) [20] supports many languages with different needs, including C#, Java, Smalltalk, and Python. Each language (or its bytecode) is compiled to the CIL, the bytecode language of the CLI. IKVM.Net [31] is an implementation of the JVM on top of a CLI implementation. The difference between IKVM.Net and a JVM on top of VMKit is that with VMKit there is no need to translate from one high-level bytecode (JVM bytecode) representation to another (CIL). With VMKit, JVM bytecodes and CIL bytecodes are translated to the LLVM intermediate representation, which is directly targeted for compilation. Moreover, some JVM mechanisms, such as class loading, cannot be implemented efficiently with the CLI [21].

The JikesRVM [32] provides a JVM that allows experimentation with new MRE technologies. So far, no high-level languages other than Java and the closely related experimental language X10 [11] have been implemented with the JikesRVM compilation system. Parley [12] is an interoperability layer for MREs. Its goal is to provide efficient communication between programs targeted for different existing MREs. It thus does not provide any common MRE components.

The Open Runtime Platform (ORP) [13] provides an MRE infrastructure with a JIT compiler and GC framework. The goal is that new GCs or JIT compilers can be implemented independently from each other. ORP has thus focused on componentizing the substrate while we focus on the ease of development of high-level MREs.

6. Conclusion

In this paper, we have presented an approach for designing MREs using a common substrate that can be efficiently reused. Our current implementation of VMKit has been successfully used to build the two complete high-level MREs, J3 and N3. Our experiments furthermore show that VMKit eases the development of MREs and experiments with new ideas.

Our performance evaluation shows that VMKit has performance close to that of the well established research or open source prototypes Cacao, Mono and Apache Harmony. However, J3 does not yet have the same level of performance as state of the art Java virtual machines such as JikesRVM and Sun's JVM. The difference in performance comes principally from the lack of advanced optimizations that we are currently implementing in VMKit (the use of the MMTk's copying collector Immix and inlining *System.arraycopy*) and that should be implemented in the core of LLVM (array bounds check elimination and adaptive compilation). The first two optimizations appear to be relatively easy to implement. To reduce startup time and dynamically optimize hotspots, an adaptive compiler is needed. However, developing a generic adaptive compiler, that can be specialized by high-level MREs, merits a full research project.

7. Availability

VMKit, J3 and N3 are publicly available via an open-source license at the URL: `http://vmkit.llvm.org`

Acknowledgments

We would first like to thank the LLVM and MMTk communities for their enormous help and early feedback on this work. We would also like to thank Vikram Adve for his insightful suggestions on the paper. Finally, we thank the anonymous reviewers for their thoughtful comments.

References

[1] 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, FL, USA, 2001.

[2] 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.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *International Conference on Software Engineering*, pages 137–146, Edinburgh, Scotland, May 2004. IEEE Computer Society.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA'06)*, pages 169–190, Portland, OR, USA, Oct. 2006.

[5] S. M. Blackburn and A. L. Hosking. Barriers: friend or foe? In *4th international symposium on Memory management (ISMM'04)*, pages 143–151, Vancouver, BC, Canada, Oct. 2004.

[6] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 22–32, Tucson, AZ, USA, 2008. ACM.

[7] R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of the Programming Language Design and Implementation Conference*, pages 321–333, Vancouver, Canada, June 2000. ACM.

[8] 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.

[9] D. Buttlar, J. Farrell, and B. Nichols. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996.

[10] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80, New York, NY, USA, 2009. ACM.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the Object-oriented programming, systems, languages, and applications Conference*, pages 519–538, San Diego, USA, October 2005. ACM.

[12] 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*, Sept. 2004.

[13] 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.

[14] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of the Programming Language Design and Implementation Conference*, pages 13–26, Vancouver, BC, Canada, 2000. ACM.

[15] C. Click and J. Rose. Fast subtype checking in the HotSpot JVM. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 96–107, Seattle, Washington, USA, 2002. ACM.

[16] G. Czajkowski and L. Daynès. Multitasking without compromise: a virtual machine evolution. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 125–138, October 2001.

[17] L. Daynès 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.

[18] DotGNU portable.NET. dotgnu.org/pnet.html.

[19] The Dwarf Debugging Standard. http://dwarfstd.org.

[20] ECMA International. Common Language Infrastructure (CLI), 4th Edition.

[21] 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, NV, USA, Jan. 2006. IEEE Computer Society.

[22] 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, Oct. 1997.

[23] D. Frampton, S. M. Blackburn., P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 81–90, New York, NY, USA, 2009. ACM.

[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] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN 2009)*, pages 544–553, Estoril, Portugal, June 2009. IEEE Computer Society.

[26] The GNU Classpath Project. http://www.gnu.org/software/classpath.

[27] The GNU Compiler Collection. http://gcc.gnu.org/.

[28] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, USA, 2000.

[29] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd*

conference on Virtual Machine Research And Technology Symposium, pages 12–24, San Jose, USA, May 2004. USENIX Association.

[30] A. L. Hosking, J. E. B. Moss, and D. Stefanovic. A comparative performance evaluation of write barrier implementations. In *OOPSLA*, pages 92–109, Vancouver, BC, Canada, Oct. 1992.

[31] The ikvm.net project. http://www.ikvm.net/.

[32] The Jikes Research Virtual Machine. http://www-124.ibm.com/developerworks/oss/jikesrvm.

[33] Jonas web site. http://jonas.ow2.org.

[34] G. Kliot, E. Petrank, and B. Steensgaard. A lock-free, concurrent, and incremental stack scanning for garbage collectors. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 11–20, New York, NY, USA, 2009. ACM.

[35] 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, Mar. 2004. IEEE Computer Society.

[36] 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, Oct. 1998.

[37] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley, 1997.

[38] M. Lujàn, J. R. Gurd, T. L. Freeman, and J. Miguel. Elimination of Java array bounds checks in the presence of indirection. In *Proceedings of the Conference on Java Grande*, pages 76–85, Seattle, USA, Nov. 2002.

[39] The Mono Project. www.mono-project.org.

[40] Objective-C 2.0 overview. http://developer.apple.com/leopard/overview/objectivec2.html.

[41] 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, CA, USA, May 2004.

[42] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[43] D. Stutz. The Microsoft shared source CLI implementation. Technical report, Microsoft, Mar. 2002. http://msdn.microsoft.com/en-us/library/ms973879.aspx.

[44] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a VM-centric approach. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 1–12, Dublin, Ireland, 2009. ACM.

[45] D. Sugalski. Building a multi-language interpreter engine. In *International Python Conference*, Feb. 2002.

[46] G. Thomas, N. Geoffray, C. Clément, and B. Folliot. Designing highly flexible virtual machines: the JnJVM experience. *Software: Practice and Experience*, 38(15):1643–1675, 2008.

[47] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for Java. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 129–154, Paphos, Cypress, 2008. Springer-Verlag.