Bypassing Portability Pitfalls of High-level Low-level Programming

Yi Lin, Stephen M. Blackburn
Australian National University
Yi.Lin@anu.edu.au, Steve.Blackburn@anu.edu.au

Abstract

Program portability is an important software engineering consideration. However, when high-level languages are extended to effectively implement system projects for software engineering gain and safety, portability is compromised—high-level code for low-level programming cannot execute on a stock runtime, and, conversely, a runtime with special support implemented will not be portable across different platforms.

We explore the portability pitfall of high-level low-level programming in the context of virtual machine implementation tasks. Our approach is designing a restricted high-level language called RJava, with a flexible restriction model and effective low-level extensions, which is suitable for different scopes of virtual machine implementation, and also suitable for a low-level language bypass for improved portability. Apart from designing such a language, another major outcome from this work is clearing up and sharpening the philosophy around language restriction in virtual machine design. In combination, our approach to solving portability pitfalls with RJava favors virtual machine design and implementation in terms of portability and robustness.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Languages

Keywords Virtual machine, Restricted language, Portability, High-level low-level programming

1. Introduction

Current hardware trends are increasingly exposing software developers to hardware complexity. Novel techniques such as multicore and heterogeneous architectures increase hardware capacity, but also leave programmers a list of challenges if they wish to fulfill the hardware's potential. Dealing with complex hardware increases the difficulty of systems programming. In the meantime, the complexity of system software grows in pace with hardware evolution. With the increasing software complexity, it is even harder to retain correctness, security and productivity.

Modern high-level languages are widely used for application programming for the assurance of correctness and security as well as boosting productivity. High-level languages provide type-safety, memory-safety, encapsulation, and strong abstraction over hardware [12], which are desirable goals for system programming as well. Thus, high-level languages are potential candidates for system programming.

Prior research has focused on the feasibility and performance of applying high-level languages to system programming [1, 7, 10, 15, 16, 21, 22, 26–28]. The results showed that, with proper extension and restriction, high-level languages are able to undertake the task of low-level programming, while preserving type-safety, memory-safety, encapsulation and abstraction. Notwithstanding the cost for dynamic compilation and garbage collection, the performance of high-level languages when used to implement a virtual machine is still competitive with using a low-level language [2].

Using high-level languages to architect large systems is beneficial because of their merits in software engineering and safety. However, high-level languages are not a perfect fit for system programming. In order to effectively undertake system programming tasks, *extensions* and *restrictions* are two essentials of high-level low-level programming — both leave unsolved challenges.

Extensions cause portability pitfalls. Portability pitfalls of high-level low-level programming include *poor hardware portability* (i.e., low-level code being unable to run on different processors) and *poor portability of programs between different runtimes*. High-level languages (HLLs) are designed to abstract over low-level details, and most of them do not provide necessary semantics for low-level operations, which is a key requirement in system programming projects. In order to undertake a low-level programming task, high-level languages need to be extended and require special support from the runtime for those extensions. This leads to the fact that VM components written in the extended HLL cannot execute on a stock runtime, and, conversely, a runtime with special support implemented will not be portable across different platforms. Both break portability.

These portability pitfalls limit code reusability of high-level low-level programming. Efficient implementation of modern language runtimes requires experts from different areas, such as memory management, concurrency, scheduling, JIT compilation. This leads to a trade-off between the high cost of hiring a group of specialists and the risk of failure for lacking expertise. One possible solution to this tension is to encourage reusability. However, when the implementing language cannot execute with a proper hosting runtime on the target platform, reusability is difficult to achieve — any given runtime that wishes to host high-level code for low-level programming needs to be modified to support the new semantics, otherwise the newly introduced low-level semantics need to be carefully dealt with in other ways [13, 14]. Both involve non-trivial work for each porting.

Low-level languages (LL languages) do not have such issues. Low-level languages such as C/C++ usually have available compilers across most target platforms. Thus, a bypass approach of

VMIL'12 October 21, 2012, Tucson, Arizona, USA

Reprinted from VMIL'12, [Unknown Proceedings], October 21, 2012, Tucson, Arizona, USA, pp. 1–8.

translating a HLL into a LL language could be one possible way to solve this pitfall. However, generally a LL language bypass is not easy to achieve — a HLL's precise exceptions and dependence on its standard libraries are hard to map into a LL language, and efficient dynamic dispatch requires non-trivial work when targeting a non-OO LL language such as C.

Restrictions lack of definitions. System programming with a HLL also relies on restrictions for performance-critical operations or avoidance of possible program failure. We observed that some VM components are written by following strict restrictions. Restrictions include omitting high-level language features that are unnecessary or problematic for certain contexts of low-level programming. These restrictions bring the HLL closer to a LL language. This justifies the possibility of translating from the restricted HLL to a lower-level language to solve portability pitfalls. Currently such restrictions do not have clear definitions in the program, and are achieved by careful ad hoc hand coding. Explicit definitions of the restrictions and automatic checking are more principled and more robust.

Thus, this paper focuses on two topics: *a*) high-level language restriction in system programming, and *b*) translation from restricted high-level language to LL language. These two topics are independent but quite coherent in our context: the natural existence of language restriction leads to the possibility of our HLL-to-LL bypassing, which further provides a solution to portability pitfalls of high-level low-level programming.

In this paper, we first discuss the important concerns in our design of RJava and the proper position of restriction within a whole system programming project, based on some observations we made on an existing high-level low-level programming project, Jikes RVM [1]. Then we propose an explicitly-restricted language called RJava with proper low-level extensions, and use MMTk [5] as an example to show how the elements of RJava are used in practice. We finish by discussing key elements of a low-level language bypass for RJava which is currently under development.

The contributions in this paper are three-fold: 1) identifying the motivation and requirement for a well-defined restricted high-level language for virtual machine implementation use, 2) sharpening the philosophy around language restriction in virtual machine implementation, and 3) designing a restricted language which inherits benefits from high-level languages, but supports flexible restriction model and also allows low-level language bypass for improved portability.

2. Design Concerns of RJava

In this section, we tidy up our approach to language restriction and the proper position of restriction within a whole system programming project, which needs to be thoroughly thought through.

2.1 HLL Restriction in System Programming

Examining the rationale for HLL restrictions in system programming helps proper definition of such languages. Our approach is based on some important observations. We made those observations on Jikes RVM as an example of system programming with a HLL. These observations further justify that language restriction naturally exists in high-level low-level programming and our approach of formalizing and exploiting existing restrictions to favor a low-level bypass for improved portability is reasonable and will not be a regression in the term of language benefits.

Restrictions exist for performance and correctness. A general understanding of programming language restriction is that languages are restricted by omitting features because they are too complex or because some programmers have used them to write bad code [8]. *Language restrictions for system programming exist for correctness and performance.*

Correctness is one challenge for engineering complex system projects. The use of a HLL introduces much better software engineering, such as abstraction, a strong type system and automatic memory management, which promotes correctness to a more manageable level. However, the correctness of a VM implemented using a HLL still needs careful consideration, especially in metacircular cases when using a HLL. When implementing a language in the same language, one pitfall threatening correctness is infinite regress. The code to support language feature X needs to avoid using the feature X itself, otherwise that code would recursively invoke itself and be unable to finish. For example, the scheduler code needs to generally avoid any language feature regarding threading and scheduling, but instead use more basic primitives such as locking to fulfill its function. Another example is the memory manager, whichh provided the imputus for RJava. The memory manager has to avoid triggering object allocation during allocation code, or triggering another garbage collection during garbage collection, so primitives to operate on raw memory have to be added to the HLL and used to implement the memory manager. The lessons here are that a HLL for VM implementation has different restrictions when applied to different scopes. Our initial focus for RJava was the scope of implementing a portable memory manager, however, we generalize our approach to be as flexible as possible so that RJava can be adapted and used in other scopes with trivial effort.

Performance is critical in systems programming. This is probably the most powerful argument as to why people stick with lower-level languages for such tasks. However, using a HLL to implement a VM can achieve a very compatible performance with proper restrictions. For example, dynamic dispatch is one important feature of object-oriented languages that incurs considerable overhead, and its cost is measured to be as high as 13% of instructions in extreme circumstances [9]. Thus, dynamic dispatch is carefully avoided by restricting the syntax in the fast path of MMTk, the most frequently executed code and the most performance-critical region.

Understanding the reasons why restrictions exist helps correctly define the restrictions. Both reasons suggest that in VM implementation, language restrictions depend on the context and scope where the restricted variant is applied. Clear language restriction relies on clear scope definition, as will be discussed later in this section.

Restrictions reduce benefits from HLLs. Generally, high-level languages provide type-safety, memory-safety, encapsulation, and abstraction. However, some of the benefits disappear when the level of restriction increases. For example, in the most restricted part of Jikes RVM, which is MMTk, garbage collection is forbidden. In this situation, the runtime is no longer able to ensure the memory safety of MMTk, but leaves it to the programmer and static analysis. Similarly, in MMTk, type safety is limited to static checks – dynamic loading is forbidden, virtual dispatch and type casting are carefully avoided in the fast path and its correctness can only be statically checked. This suggests that, under performance-critical and correctness-critical circumstances where very strict restrictions have to be applied, the benefits of a HLL are reduced, and the benefits are principally static, i.e. type-safety and memory-safety at the source code level, and encapsulation and abstraction as software engineering tools. A restricted HLL still has clear advantages over low-level languages.

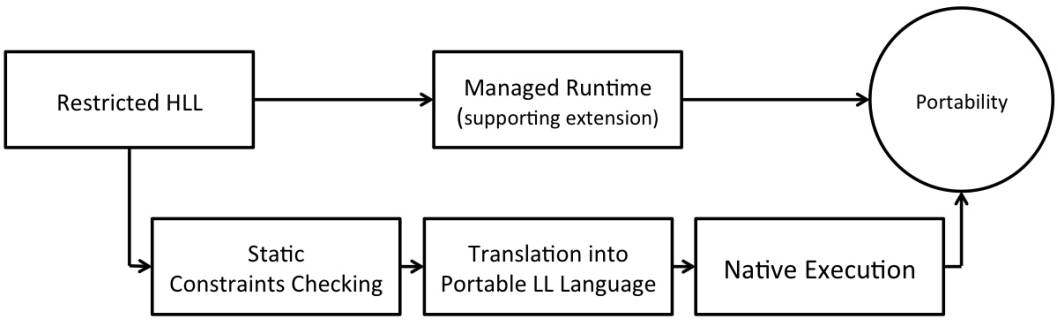

Figure 1: An illustration showing our bypass approach for portability issues.

Occurrences of	MMTk	Baseline Compiler	Rest of Jikes RVM	Eclipse (comparison)
'new' statements	0.59%	0.86%	2.40%	4.47%
'throws' declarations	0	0	0.21%	1.33%
Library imports	0	0.03%	0.40%	0.82%
Lines of Code	29933	17762	113359	-
Level of Restriction	From most restricted to not restricted.			

Table 1: Language restrictions in different scope of Jikes RVM (occurrences per LOC).

Restrictions are only applied to a limited scope. Restriction is essential for correctness and performance in system programming with HLLs, but different levels of restriction degrade high-level languages to different extents. Thus, one principle for system programming with HLLs is to minimize the scope where very strict restrictions are needed so to maximize the benefits from HLLs [11]. Table 1 reflects good design within Jikes RVM: the most restricted Java variant is used in a relatively small scope while the majority of the project is loosely restricted. Thus, heavier restrictions affect a small part of the system and do not detract the benefits of a HLL in other parts of the system. However, the strictly restricted scope (including MMTk and the baseline compiler) still has 47K LOC, which is important and large enough that is worth careful consideration.

2.2 Expressiveness vs. Restrictions

Higher-level languages are more expressive than low-level languages. Java, for example, is considered to have a 2.5× 'statement ratio' compared to the C language (i.e., on average, one Java statement needs 2.5 C statements to achieve the same function [17]). Restrictions to HLL syntax reduce expressiveness. In the limit, a restricted form of HLL that discards all features that C does not support will have a trivial mapping to C syntax. Such extreme restriction would favor our LL language bypass, but this is definitely not desirable. In contrast, if the language is minimally restricted, the expressiveness is maximally conserved, but the LL language bypass would be more difficult to achieve — Java's precise exceptions and dependence on its standard libraries are hard to map into a LL language, and efficient dynamic dispatch requires non-trivial work when targeting a non-OO LL language such as C. Thus, there is always a trade-off between expressiveness and how restricted the language is.

We resolve the trade-off with a simple principle: *we do not introduce more restrictions than necessary*. In addition to the two necessary reasons that restrict languages in VM implementation — correctness and performance — we have to put mappability to LL languages into consideration in our bypass approach. The language has to be restricted due to existing requirements on correctness and performance, and it also needs to be further restricted to adapt to LL language translation. We confine this set of restrictions to the minimum.

2.3 RJava in Different Scope

Ideally we want RJava to be a fixed language with constant restrictions so that we can use RJava to implement VM components where restrictions and portability are desired while we are able to use normal Java to implement the rest of a VM. However, this is not the case. 1) Restrictions are different among different VM components. This is mainly for a metacircular VM implementation. For example, a metacircular implementation of memory manager disallows object allocation during allocation and reclamation so that any language syntax that would introduce an object allocation is forbidden, including the use of libraries and the creation of exceptions. A scheduler, on the other hand, does not necessarily have any restriction regarding object allocation, but needs careful restriction around threading and synchronization. Thus, different components require different restrictions, and attempts to generalize restrictions among components would suppress expressiveness. 2) Restrictions are still different within one VM component. As explained before, a performance-critical scope needs more strict restrictions to remove any possible performance degradation.

As a result, instead of trying to define a universal set of restrictions that would be adaptable for general VM components, we define RJava with a set of fixed restrictions that favors easy mapping to a LL language which our frontend is able to translate to allow bypass. Also, RJava is designed to include a set of optional restriction rules that programmers can choose from to shape their own restriction ruleset for certain components. The RJava constraint checking tool processes each restriction rule in the ruleset and ensures code compliance.

3. Concrete RJava Language

The previous section discussed important concerns that affect our design of RJava. In this section, we present this restricted language

with its key elements and a concrete example of how RJava, the restricted language motivated by MMTk code, is re-adapted to MMTk and helps its robustness and portability.

3.1 Key Language Elements

RJava is a restricted subset from the Java language. It inherits the Java language syntax except that which is restricted. Extensions and restrictions are two major parts of high-level low-level programming, thus they are naturally two key elements in the RJava language.

The org.vmmagic Extension

The `org.vmmagic` extension is described in the paper "Demystifying Magic: High-level Low-level Programming" [12]. RJava takes the advantages of the existing `org.vmmagic` package for low-level semantics.

Most elements and ideas in `org.vmmagic` remain untainted when adopted by RJava. These include unboxed types and related intrinsic operations. However, some compiler 'pragmas' that are referred to as 'semantic regimes' are reconsidered and reconciled into RJava's restriction model. One example is `@Uninterruptible`. All the MMTk classes used to be described as `@Uninterruptible` to disallow garbage collection and thread switching. In RJava, `@Uninterruptible` is considered as a restriction rule, and can be integrated with other restriction rules to form a ruleset for MMTk. Another example is that some compiler intrinsics such as `@NoBoundsCheck` are categorized as restriction rules to coordinate with the content of this paper, since they restrict language run-time features. We now introduce the idea of restriction rules and rulesets.

Restriction Rule and Ruleset

Restriction rules and rulesets are fundamentals for RJava. We define restriction using Java's annotation syntax. Each restriction becomes a restriction rule, and is marked with the `@RestrictionRule` annotation for documentation. A restriction ruleset consists of different restriction rules or rulesets. This model brings some rigor to the definition to the restrictions and allows automatic checking. Figure 2 shows those elements with code examples.

`@RJavaCore` is a predefined ruleset that all RJava code should obey. The core ruleset contains restrictions to some language features that are infrequently used in VM implementation and also cannot be easily mapped to low-level languages. The ruleset suggests a minimum restriction to enable a feasible low-level language bypass while preserving expressiveness of the HLL. `@RJavaCore` also indicates language features that our frontend translator does not support, thus it must be contained by any user-defined ruleset for RJava.

We also provide different restriction rules with RJava. They are not included in `@RJavaCore` and their semantics are acceptable by the RJava frontend. Those restrictions can be used to aggregate user-defined rulesets and are essential to ensure correctness and performance for specific scopes. They can also be used solely to mark any code to indicate restrictions and also indicate a requirement for static constraint checks. However, defining a restriction ruleset specific to a certain scope is preferred than using scattered restrictions. It is best to have a 1-to-1 mapping between ruleset and scope wherever restrictions are needed. This design favors flexibility and allows clear definition of restricted scopes with certain rules.

In the next subsection, we give an example of how RJava restrictions are adapted in MMTk.

```
1 @RestrictionRule
2 public @interface NoDynamicLoading {
3 }
```

(a) An example of restriction rule.

```
1 @RestrictionRuleset
2
3 @NoDynamicLoading
4 @NoReflection
5 @NoException
6 @NoCastOnMagicType
7 ...
8 public @interface RJavaCore {
9 }
```

(b) RJava core restriction ruleset.

```
1 @RJavaCore
2 public class AnRJavaProgram {
3 ..
4 }
```

(c) RJavaCore clearly defines restrictions in a scope.

Figure 2: RJava restriction rules and rulesets.

3.2 A Concrete Example: MMTk in RJava

Though restrictions in RJava are inspired and motivated by the restricted coding patterns in MMTk, we design RJava to be a more flexible and general restricted language for implementing VM components. In this subsection, we show how RJava is applied to a specific scope (MMTk) to restrict its syntax and help with software engineering.

One important principle when coding with RJava is to map restriction rulesets to scopes. MMTk itself is a well-contained scope, thus we need a corresponding ruleset `@MMTk` to clarify the restrictions. Besides `@RJavaCore`, other restriction rules have to be carefully identified.

The memory manager fulfills two main tasks: object allocation and object reclamation. One obvious restriction for a metacircular implementation of a memory manager is to disallow object allocation in its own code during execution—otherwise, triggering object allocation in an allocating procedure would invoke another allocating procedure and triggering object allocation in a garbage collection would fail and invoke another garbage collection. We use the rule `@NoRunTimeAllocation` to describe this restriction. Object allocation in class static initializers and constructors (including methods used only by them) is allowed, since they can only be executed during the VM build process where object allocation is safe. The `@NoRunTimeAllocation` rule ensures that no **`new`** statements appear in places outside static initializers and constructors. This rule also implies two other restriction rules, `@NoException` (which is already included in `@RJavaCore`) and `@NoClassLibrary`. Using class libraries may introduce unexpected object allocation at run-time, since their implementation varies. It was possible to implement MMTk without using any class library classes[1], so we retain this restriction in the `@MMTk` ruleset.

Another restriction is `@Uninterruptible`. This annotation is inherited from the `org.vmmagic` package and we consider it to be a restriction rule. It informs the runtime to avoid triggering thread switching and garbage collection in certain scopes, and also to omit

[1] Use of Java's built in `String` and `Array` types is not restricted. However, the `@NoRunTimeAllocation` rule prohibits dynamic allocation of `Arrays` and `Strings`. This also implies a prohibition of `String` concatenation.

```
@RestrictionRuleset

@RJavaCore
@NoClassLibrary
@NoRunTimeAllocation
@Uninterruptible
public @interface MMTk {
}
```

(a) @MMTk restriction ruleset to map MMTk scope.

```
@RestrictionRuleset

@MMTk
@NoVirtualMethods
public @interface MMTkFastpath {
}
```

(b) @MMTkFastpath restriction ruleset to map fast path subscope of MMTk.

```
@RestrictionRuleset

@MMTkFastpath
@NoPutfield
@NoPutstatic
public @interface WriteBarrier {
}
```

(c) @WriteBarrier restriction ruleset to map write barrier code in the fast path.

```
@MMTkFastpath
public class GenMutator {
  ...

  public Address alloc() { ... }
  // no runtime alloc
  // virtual methods, has to be overridden

  @WriteBarrier
  public final void objectReferenceWriteBarrier() { ... }
  // no putfield, no putstatic on its own fields
  // non-virtual methods, thus no dynamic dispatching
}
```

(d) Restrictions ensure correctness and performance of the fast path.

Figure 3: MMTk with RJava.

emitting any code during code generation that would trigger thread switching or garbage collection.

The restriction ruleset for MMTk is showed in Figure 3a.

Subscope: MMTk Fast Path

The design of MMTk makes heavy use of the fast/slow path idiom. A fast/slow path idiom is a diamond-shaped control flow graph where the expected case is to do quick checks or operations to confirm a result. When some portion of the fast path fails, control transfers to the slow path that covers all remaining cases [19]. Take allocation in MMTk for example: The allocator's fast path tries to allocate space from its thread-local buffer. When the thread-local buffer is consumed, control is handed to the slow path where the allocator will acquire space from global memory and synchronization is needed. If the slow path still fails, a garbage collection will be triggered. The ratio that control falls into the slow path is typically 0.1% in MMTk allocation [6]. Thus, the fast path is the most performance-critical subscope in MMTk. MMTk forces all the fast path code to be inlined into its context to eliminate method invocation overhead, but also restricts syntax for performance improvement.

In the coding of the MMTk fast path, another restriction rule is carefully applied to minimize the performance overhead. The code avoids the possibility of dynamic dispatch by declaring all of its methods as non-virtual methods. In the fast path, all non-static non-private methods are either overridden or declared as 'final', thus there are no virtual methods and no dynamic dispatch in the fast path. We use `@NoVirtualMethods` to describe this restriction. We build the `@MMTkFastpath` ruleset based on `@MMTk`. Figure 3b shows the `@MMTkFastpath` ruleset.

Besides performance, correctness restrictions need to be reconsidered for the fast path. MMTk's fast paths include write/read barrier code. Barriers are a powerful tool to monitor mutator actions by tracking operations on objects. Take the write barrier for example: Because of metacircularity, the write barrier itself needs to avoid using `putfield` or `putstatic` on its own object fields, otherwise it leads to an infinite regress. We use `@NoPutfield` and `@NoPutstatic` to describe these restrictions. To avoid being overly restrictive, we form the `@WriteBarrier` ruleset that will be used only on write barrier code in the fast path. `@WriteBarrier` contains the `@MMTkFastpath` ruleset, and the two specific restriction rules stated above. Figure 3c shows this restriction ruleset.

Figure 3d gives an outline of `GenMutator` as parent of all mutators for generational garbage collection algorithms to show how these rulesets are used to properly restrict language semantics in MMTk, and help ensure its correctness and performance.

4. Current Work: An RJava to LL Language Bypass

Formalizing the restriction rules is one significant aspect for designing the RJava language. In this section, we introduce our current work, the RJava to LL language translation toolchain that materializes the bypass approach (see Figure 1).

The toolchain to enable RJava to LL language bypass includes a static constraint checking tool that ensures compliance to the declared restriction rules, a frontend that takes RJava as source and produces code in LL language, such as C/C++, and a backend that compiles LL language to native code.

4.1 Static Constraint Checking Tool

The static constraint checking tool examines the compliance of code with restriction rules declared on them. It can be used as one part of our LL language bypass toolchain, and can also be used as an independent tool to check original RJava code to detect any violation of restrictions.

There are existing tools for Java syntax checking, such as PMD [20]. These tools parse Java syntax and perform rule-based style checking. But they do not fit our requirements. To be able to precisely examine restrictions defined in RJava, our static constraint checking tool needs to be able to process not only at the syntactic level but also at the more complex semantic level. For example, `@NoRunTimeAllocation` requires that no object allocation appear outside static initializers, constructors or any methods only called by them. Thus, this implies requirements at a semantic level, such as the relationship between methods (call graph) that existing syntax checking tools are not able to deal with.

We are building our static constraint checking tool based on the Soot framework [24]. Soot is a Java optimization and static analysis framework, and provides various forms of analyses. We have built a prototype that is able to validate the `@NoRunTimeAllocation` restriction. What remains to be done is expanding the rule/ruleset checking to cover all of the other RJava restrictions.

4.2 Frontend: RJava to LL Language

The frontend is the most critical part in our toolchain to translate RJava into a low-level language. There are several important tasks that the frontend has to complete, besides simple syntax mapping:

Implementing compiler intrinsics. Intrinsic methods such as `Address.loadByte()` and compiler pragmas such as the `@Inline` annotation do not have concrete implementations in RJava, but rely on support from the managed runtime. Since our bypass approach removes the existence of the runtime, compiler intrinsics need to be implemented in the frontend. We expect that the generated code is plain low-level language. For example, `loadByte()` would become a pointer dereference and `@Inline` would become an inline keyword in the target language.

Unboxing magic types. The `org.vmmagic` package we use in RJava introduces unboxed types, such as `Address` and `Offset`. Java types are by default 'boxed' with additional information such as header, virtual method table, etc. However, this package makes the assumption that those magic types are specially treated as unboxed types by the runtime, thus they are not real objects at the run-time. This assumption prevents a memory manager creating objects when it operates on addresses and object references during object allocation requests. It also makes retrieving actual values of such types significantly more efficient. This assumption is equally important when RJava is translated into a LL language for the same reasons. Unboxing is needed during translation to convert such magic types into pointers that the target language supports.

Removing dependencies on the Java class library. Even without explicit use of the Java class library, Java syntax is related to its class library, such as implicit support from `String`, `Array` and the common superclass `Object`. We require that the generated LL language has no dependency on the Java runtime. Thus, the frontend needs to remove all dependencies on the Java class library, and replace implicit uses with syntax and features from the target language.

Converting object-oriented syntax (optional). This task is only essential when our frontend targets a non-object-oriented LL language. In such cases, the OO syntax needs to be removed during the translation. Generally this is possible since RJava is restricted to forbid some dynamic features of object-oriented languages. But this still needs careful consideration regarding performance.

Those tasks are sensitive and specific to the source language, i.e. RJava, and the target LL language. Thus, we do not aim for our frontend to be a flexible framework that could produce code for different targets. The C language is a suitable LL language to target. It is the dominant language in system programming, and it is also portable. However, our first implementation (under development) does not target C. This is for two main reasons. First, we are not aware of any existing Java-to-C translator for general use that we can base our implementation on. Existing translators are too fragile and too specific to their own projects. If C were our target, we would have to build such a translator from scratch. Second, C is not object-oriented: translation from RJava to C would require more development effort, and naive object-oriented syntax conversion may result in inefficiency in the target performance.

We choose C++ as our target. C++ is portable and has a similar syntax to Java, thus mapping from Java into C++ is easier. We implement our frontend by modifying J2C [25]. J2C is a translator that converts Java code into C++. The tasks listed above need to be implemented in J2C, so it can properly handle semantics specific to RJava. This part of the work is our focus, and is under development.

4.3 Backend: LL Language to Executable

Our frontend translates RJava into plain C++ syntax. RJava's bypass approach does not make any particular assumption about the backend. A general compiler that takes our frontend target (i.e., C++) as the source language can fit well in our bypass toolchain.

5. Future Work: Bootstrapping Java VM with RJava

The flexible design of RJava encourages its use for different components in VM design. Besides the memory manager, our first chosen component, the interpreter could be another candidate for implementation in RJava. Table 1 showed that the interpreter/baseline compiler has a similar restriction pattern in Jikes RVM. This suggests that it should be straightforward to adapt the baseline compiler into RJava, and is therefore suitable for LL language bypass.

Being able to implement a Java compiler/interpreter in RJava could introduce a full bootstrapping model to metacircular Java VMs. Most current metacircular VMs use a half bootstrapping model, i.e. the metacircular VM A requires another available Java VM B on the target machine so A's compiler can be executed on B and the executable will further execute its own code and the whole VM code. Half bootstrapping is blamed for bad portability, since it relies on the availability of another Java VM/compiler B. However, a Java compiler/interpreter written in RJava can execute as native code without the need of another available Java VM/compiler on the target platform. This would greatly enhance the portability of a metacircular VM.

There are difficulties lying in this direction that we will have to resolve in the future. One obvious point is whether we can implement an interpreter with RJava's restricted syntax. Though results showed that the baseline compiler in Jikes RVM uses a very similar pattern, more investigation is needed to ensure that, with acceptable refactoring/rework, an interpreter can strictly follow RJava restrictions. Furthermore, Java interpreter/baseline compiler is not an isolated component that can easily be decoupled from the rest of the VM. It requires support from other parts of the VM. This suggests code from other parts of the VM may be involved and have to be restricted with RJava syntax. The amount of code that has to be restricted is another concern. However, we believe that these difficulties can be overcome (the interpreter in the PyPy VM is written in RPython, which proves this is a feasible bootstrapping option for high-level language metacircular VMs) and RJava can be also used to benefit bootstrapping of metacircular Java VMs.

6. Related Work

Our design of the restricted language RJava is related to prior work in two main aspects: extending high-level languages for low-level programming, and the need for language restrictions in VM implementation.

Extending High-level Languages for Low-level Programming

The work [2009] by Frampton et al. (referred as the vmmagic work in the following discussion) described general concepts around extending high-level languages for low-level programming and the `org.vmmagic` framework. The `org.vmmagic` framework is solidly grounded in real world experience, including three Java-in-Java virtual machines [1, 7, 18], a Java operating system [21], and a C/C++ JVM [4]. This concrete framework introduced type-system extensions (raw storage and unboxed types) and semantic extensions (intrinsic functions and semantic regimes), and it was well

designed to resolve the tension between efficient low-level access and the encapsulation of low-level semantics. Our RJava also takes advantages of `org.vmmagic`. However, our work differs. The vmmagic work aimed at an efficient high-level low-level approach, and focused on extensions that would enable such an approach. As we explained earlier, extensions cause portability issues. Our work aims for a bypassing approach to solve portability issues of high-level low-level programming and along its way, we also examine, clarify and enforce language restrictions in VM implementation. There are three principal advances we make on the vmmagic work: 1) formalizing the restricted language RJava with clear extensions and scope-specific restrictions, 2) introducing a flexible design of restriction rules/rulesets and their compliance checking tool, and 3) implementing a translation toolchain that produces portable low-level language code from RJava and enables our bypass approach.

Language Restriction in VM Implementation

Our work is highly inspired by the work of RPython [3]. RPython is a restricted subset of Python and is used to write an interpreter in the PyPy virtual machine [22]. However, it is also an independent language that can be used for general use. RPython inherits its most features from Python. It is restricted: for example, it is statically typed and does not allow dynamic modification of class or method definitions. The RPython backend supports code generation for different languages, such as LLVM code, C code, and even JVM and CLI code (work in progress). We have learned from RPython, from its success and also its imperfections. The design of RPython does not support flexible restrictions for different scopes (in Section 2.3, we explained its necessity in VM implementation). Besides, its restrictions are not clearly defined [23], so the restriction compliance cannot be automatically checked and programmers may not be able to realize restriction rule violations unless their code meets a translation error or a run-time error. We took those considerations into RJava's design, and addressed them with a flexible restriction rule/ruleset model that can be automatically checked. Currently RPython has a more reliable supporting framework than what RJava has, such as support for accurate garbage collection and precise exceptions, and RPython's backend can target several different languages. But we believe that with future development, RJava could be equally reliable, while being more flexible for different scopes of VM implementation.

7. Conclusion

We see a trend of applying high-level languages to systems programming to cope with the growing complexity of hardware and software. The security, productivity boost and software engineering advantages introduced by using a high-level language has benefited virtual machine's design and implementation. However, portability pitfalls of high-level low-level programming limit the reusability of VM components and the portability of VMs written in a high-level language.

Our approach is to formalize language restriction to define RJava, a restricted high-level language, for the implementation of virtual machine components. The design of RJava allows a low-level language bypass for improved portability, which promotes the reusability of VM components written in a high-level language, and provides better integration with legacy code.

We argue that restrictions are prevalent in virtual machine implementations for performance and correctness reasons, however, they are typically implicit, unprincipled, and ad hoc. The flexible and explicit restriction model of RJava requires virtual machine designers to consider scopes in a virtual machine along with restrictions to different scopes at an early stage. This explicit declaration not only benefits the design of virtual machines, but also favors automatic restriction compliance checks to enhance the robustness of the virtual machine and ease the implementors' work.

We hope that our insights and ideas will draw attention to the principled use of language restriction, and further encourage the implementation of virtual machines in high-level languages.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 314–324. ACM, 1999.

[2] B. Alpern, M. Butrico, A. Cocchi, J. Dolby, S. Fink, D. Grove, and T. Ngo. Experiences Porting the Jikes RVM to Linux/IA32. In *Proceedings of the 2nd Java(TM) Virtual Machine Research and Technology Symposium*, JVM '02, pages 51–64. USENIX Association, 2002.

[3] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8. doi: `10.1145/1297081.1297091`. URL `http://doi.acm.org/10.1145/1297081.1297091`.

[4] Apache. DRLVM – Dynamic Runtime Layer Virtual Machine. `http://harmony.apache.org/subcomponents/drlvm/`, 2009.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146. IEEE Computer Society, 2004.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: the Performance Impact of Garbage Collection. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: `10.1145/1005686.1005693`. URL `http://doi.acm.org/10.1145/1005686.1005693`.

[7] S. M. Blackburn, S. I. Salishev, M. Danilov, O. A. Mokhovikov, A. A. Nashatyrev, P. A. Novodvorsky, V. I. Bogdanov, X. F. Li, and D. Ushakov. The Moxie JVM experience. Technical Report TR-CS-08-01, Australian National University, Department of Computer Science, May 2008.

[8] C2 Wiki. Restricted Programming Language. `http://c2.com/cgi/wiki?RestrictedProgrammingLanguage`.

[9] K. Driesen and U. Hölzle. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 306–323, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: `10.1145/236337.236369`. URL `http://doi.acm.org/10.1145/236337.236369`.

[10] C. Flack, T. Hosking, and J. Vitek. Idioms in Ovm. Technical Report CSD-TR-03-017, Purdue University, 2003.

[11] D. Frampton. *Garbage Collection and the Case for High-level Low-level Programming*. PhD thesis, Australian National University, 2010.

[12] 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 *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90. ACM, 2009.

[13] R. Garner. *JMTK: A Portabl Memory Management Toolkit*. PhD thesis, Australian National University, 2003.

[14] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 51–62. ACM, 2010.

[15] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A Principled Approach to Operating System Construction in Haskell. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005.

ACM. ISBN 1-59593-064-7. doi: 10.1145/1086365.1086380. URL http://doi.acm.org/10.1145/1086365.1086380.

[16] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[17] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[18] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a Common Intermediate Representation for the Ovm Framework. *Science of Computer Programming*, 57:357–378, September 2005.

[19] M. Paleczny, C. Vick, and C. Click. The Java HotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267847.1267848.

[20] PMD Project. Pmd. http://pmd.sourceforge.net/pmd-5.0.0/.

[21] E. Prangsma. Why Java is practical for modern operating systems, 2005. Presentation only. See http://www.jnode.org.

[22] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953. ACM, 2006.

[23] RPython Coding Guide. Rpython. http://doc.pypy.org/en/latest/coding-guide.html#id1.

[24] Sable Research Group, McGill. Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[25] J. Sieka. J2c project. http://code.google.com/a/eclipselabs.org/p/j2c/, 2012.

[26] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 78–88. ACM, 2006.

[27] D. Ungar, A. Spitz, and A. Ausch. Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment. In *Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 11–20. ACM, 2005.

[28] C. Wimmer, M. Haupt, M. L. V. D. Vanter, M. Jordan, L. Daynes, and D. Simon. Maxine: An Approachable Virtual Machine For, and In, Java. Technical report, Oracle Labs, 2012.