Virtual Machine Design and High-level Implementation Languages

Yi Lin

A thesis submitted for the degree of Master of Philosophy
The Australian National University

March 2013
Except where otherwise indicated, this thesis is my own original work.

Yi Lin
7 March 2013
First I would like to thank my supervisor, Steve. His wisdom and great insights in computer system area helped me shape those projects and led me through without getting lost. Besides his academic/technical advice, I want to express my gratitude to Steve for creating a free, encouraging, and productive research atmosphere in our lab that I enjoy most during my study.

Then I would like to thank Daniel, my advisor. I learned a lot from him, especially technical details. He seems to know every part of the code that I need to deal with. We worked together to finish my first conference paper along with Steve in the fourth month after I had arrived here. This would not even be possible without him around and his great help in that work.

Then I would like to thank my parents. When I faced a choice of whether I should pursue a research degree and which research field I should choose, they encouraged me to pick whatever I am interested in and make my best effort to do it well. It was an important decision and they fully supported my choice. Besides, I would not be able to study abroad without their financial support.

I also would like to thank everyone in the lab, Ting, Vivek, Rifat, Tiejun and Xi. They are all friendly and smart. As the newest student in the lab, I knew little about anything that I was up to when I just arrived, but they patiently taught me everything. Special thanks to Rifat, who helped me review and proofread the thesis.
Abstract

System programming tasks such as implementing language virtual machines (VMs), are, by convention, associated with low-level programming languages, such as C/C++. Low-level programming languages provide efficient semantics to directly access hardware resources, thus are naturally a good choice for system programming.

However, there are two trends in system programming that are questioning this convention, challenging the ruling role of low-level languages in this field. On one hand, increasing hardware complexity requires languages to provide abstraction over complicated and diverse architectures. On the other hand, software size and complexity is also increasing as hardware evolves, leaving security, correctness and productivity as an even bigger challenge. Thus, efforts have been made to use higher-level languages to perform system programming in order to improve safety, productivity and correctness, as well as to maintain competitive performance.

Research has confirmed the feasibility and performance of implementing language virtual machines in high-level languages. The idea is novel and compelling. However, it leaves a trail of software engineering challenges, such as unclear VM/application context and poor portability, which may in return hamper the benefits that come with high-level languages.

My thesis is that imposing clearly defined constraints on code structure, context transition, and language features addresses important software engineering pitfalls that arise when using high-level languages to design and implement flexible and efficient virtual machines.

This thesis is divided into two parts, each addressing a major software engineering pitfall in the area of system programming with high-level languages: 1) clarifying VM/application interdependencies both statically (reflected by code structure and annotations) and dynamically (maintained as run-time information) with very low overhead; and 2) defining a restricted language called RJava, a restricted but still expressive subset of Java suitable for implementing VM components where portability and bootstrapping are major concerns.

Two key factors of system programming with high-level languages that former research paid great attention to are expressiveness of high-level languages and performance, while they paid less attention to software engineering concerns. This thesis focuses on two of the software engineering pitfalls, and shows that they can be fixed while preserving both expressiveness and performance. This thesis should help those designing new virtual machines and help improve existing ones, and so encourage the implementation of virtual machines in high-level languages.
## Contents

Acknowledgments .................................................. v

Abstract .................................................................. vii

1 Introduction ......................................................... 1
   1.1 Thesis Statement .............................................. 1
   1.2 Introduction .................................................. 1
   1.3 Thesis Outline ............................................... 4

2 VM/Application Interdependencies ............................. 5
   2.1 Introduction ................................................... 5
      2.1.1 Metacircularity .......................................... 6
      2.1.2 Unclear Contexts and Interdependencies .......... 7
   2.2 Tracking Execution Context ............................... 8
   2.3 Implementation .............................................. 11
      2.3.1 Supporting Mechanisms ............................... 11
      2.3.2 Finding Transition Points ............................ 12
   2.4 Design Implication and Lessons ......................... 13
   2.5 Results ....................................................... 17
      2.5.1 Methodology ............................................ 17
      2.5.2 Framework and Refactoring Overhead ............. 18
      2.5.3 Context Transition Analysis ......................... 18
      2.5.4 Memory Management Behavior ..................... 23
      2.5.5 Summary ................................................ 25
   2.6 Related Work ................................................ 25
      2.6.1 Metacircular VMs ...................................... 25
      2.6.2 Prior Work in VM Isolation ......................... 26
   2.7 Summary ...................................................... 27

3 Bypassing Portability Pitfalls of High-level Low-level Programming .................................. 29
   3.1 Introduction .................................................. 29
      3.1.1 Observations on System Programming with a HLL 31
      3.1.2 Bypass without Compromising ....................... 32
      3.1.3 Towards Restricted Language for More General Use 33
   3.2 Design Concerns of RJava ................................. 34
      3.2.1 Reasons for Restriction ............................... 34
      3.2.2 Expressiveness vs. Restrictions ................. 35
3.2.3 RJava in Different Scope .................................................. 35
3.3 Concrete RJava Language ......................................................... 36
  3.3.1 Language Key Elements ................................................. 36
  3.3.2 A Concrete Example: MMTk in RJava ............................... 38
3.4 Current Work: An RJava to LL Language Bypass ......................... 40
  3.4.1 Static Constraint Checking Tool ................................... 40
  3.4.2 Frontend: RJava to LL Language .................................. 41
  3.4.3 Backend: LL Language to Executable .............................. 42
3.5 Related Work ........................................................................ 42
3.6 Summary ................................................................................ 44

4 Conclusion ............................................................................... 45
  4.1 Future Work ........................................................................ 46
    4.1.1 Heap Policy on Context-Clear VM ................................ 46
    4.1.2 Bootstrapping Java VM with RJava ............................... 46
# List of Figures

1. An illustration showing the motivation and the outcome of this thesis.  
2. A simplified depiction of dynamic context transitions in Jikes RVM.  
3. An example transition point, Java's Object.wait().  
4. VMThread support for dynamic execution context.  
5. Inter-class refactoring of runtime services: BumpPointerAllocator.  
6. Intra-class refactoring of runtime services: RVMClass.  
7. Performance overhead of our refactoring and context tracking.  
8. Dynamic transition frequency.  
9. Fraction of total allocation by execution contexts.  
10. Nursery survival for objects allocated by execution contexts.  
11. Memory footprint by application context and runtime context.  
12. An illustration showing our bypass approach for portability issues.  
13. RJava restriction rules and rulesets.  
14. MMTk with RJava.  

2 \ 9 \ 12 \ 13 \ 15 \ 16 \ 17 \ 20 \ 21 \ 22 \ 25 \ 33 \ 37 \ 39
List of Tables

2.1 Transition Point classes and counts for Jikes RVM .......................... 14
2.2 Frequency distribution across individual transition points ............... 19
2.3 CPU cycles spent in each execution context ................................... 20

3.1 Language restrictions in different scope of Jikes RVM ....................... 32
Implementing language virtual machines in high-level languages is compelling, but introduces software engineering challenges. This thesis aims to identify two software engineering pitfalls: 1) unclear VM/application interdependencies and 2) poor portability, and provides in-depth discussions and solutions to them.

1.1 Thesis Statement

My thesis is that imposing clearly defined constraints on code structure, context transition, and language features addresses important software engineering pitfalls that arise when using high-level languages to design and implement flexible and efficient virtual machines.

1.2 Introduction

The complexity of computer hardware has increased dramatically. 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 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 the boost of productivity. High-level languages provide type-safety, memory-safety, encapsulation, and strong abstraction over hardware [Frampton, 2010], which are desirable goals for system programming as well. Thus, high-level languages are potential candidates for system programming.

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.

On one hand, high-level languages are designed to highly abstract over low-level details, and most of them do not provide necessary semantics for low-level opera-
Introduction

Figure 1.1: An illustration showing the motivation and the outcome of this thesis.

tions, which is a key requirement in system programming projects. Former research has focused on the feasibility and performance of applying high-level languages to system programming [Alpern et al., 1999; Flack et al., 2003; Hallgren et al., 2005; Ungar et al., 2005; Prangsma, 2005; Hunt et al., 2005; Rigo and Pedroni, 2006; Simon et al., 2006; Blackburn et al., 2008a]. 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. Besides, regardless of cost for dynamic compilation and garbage collection, the performance of using high-level languages to implement a virtual machine is still competitive to using a low-level language [Alpern et al., 2002].

On the other hand, another significant concern for implementing virtual machines in high-level languages is the software engineering impact. Engineering virtual machines with high-level languages (HLLs) differs notably from the traditional way of using low-level languages, leaving unsolved challenges:

Unclear VM/Application Dependencies. Flexible and efficient runtime design requires an understanding of the dependencies between the application and the runtime. These dependencies are frequently unclear. This consequently affects resource management, security, and performance of the VM. However, implementing VMs in a HLL highlights the urgent need for clarifying VM/application dependencies. A VM in a HLL is a hosted application, but it also hosts applications, which complicates the hosting hierarchy. Vague boundaries between different levels of this hierarchy make it harder to guarantee the performance and correctness of the whole system. This motivation shaped one major part of my work on clarifying VM/application interdependencies.
Portability Pitfalls and Language Restrictions. Extensions and restrictions are two sides of system programming with HLLs. On one hand, in order to implement a VM, high-level languages need to efficiently access hardware resources and are extended to support low-level semantics. Low-level extensions to a HLL bind the program (in this case, VM) to a specific hosting runtime (which provides support for the extensions). This causes the portability pitfall of high-level low-level programming: components of a VM written in a HLL cannot be easily ported to other projects on other platform unless the target platform/project has an available hosting VM to support the required extensions, which limits code reusability. Low-level languages (LL languages) do not have such issues. A bypass approach of translating a HLL into a LL language could be one possible way to solve this problem. On the other hand, 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 issues. Besides, currently the 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. In combination, the possibility of bypassing and the requirements of explicit restrictions motivate my work on defining RJava, a restricted subset of Java. RJava provides mechanisms to explicitly declare and automatically check restrictions, and RJava is restricted in such a way that it can be translated into a LL language to bypass portability pitfalls.

Metacircularity. We are convinced by the benefits of using high-level languages in complex system programming tasks. For VM implementation, when the target language is also a good candidate, metacircularity (implementing a language in itself) is a natural outcome. Apart from its performance advantages by impedance matching and the satisfying idea of closure, however, metacircularity has its own challenges, such as the potential for infinite regress, VM re-entry, vague VM/application contexts, language feature restrictions, bootstrapping, etc. This thesis aims to solve software engineering challenges in general high-level low-level programming when building VMs. It also gives thoughts and considerations to problems related to metacircularity under the thesis context, since metacircularity is an important form of high-level low-level programming in practice.

Based on the motivations stated above, this thesis focuses on addressing two software engineering problems of implementing VMs in HLLs – VM/Application interdependencies and portability. These questions are explored on an existing high performance metacircular VM, Jikes RVM (a Java VM written in Java), thus how
metacircularity affects and complicates these problems is also included in this thesis.

**VM/Application inter-dependencies** How the VM and application interact with each other is an important aspect of understanding VMs. In this thesis, we propose a low-overhead framework to clarify the VM/application context. The contributions on this topic are: 1) implementing and measuring a framework that dynamically maintains execution context, 2) restructuring an existing VM to substantially reduce structural contextual ambiguity, and 3) analyzing the behavior and run-time characteristics based on our framework.

**Portability** Low-level extensions to a HLL compromise portability. This problem is explored in MMTk [Blackburn et al., 2004b], an efficient memory manager written in Java that suffers from portability issues. My contributions in addressing this problem are three-fold: 1) sharpening the philosophy around language restriction in system programming, 2) extracting code patterns used in MMTk to form a well-defined restricted language called RJava, 3) building a toolchain to check restriction compliance and translate from RJava to C++, to enable a bypass approach for improved portability.

### 1.3 Thesis Outline

Chapter 2 discusses the problem of unclear VM/application interdependencies, introduces a low-overhead framework to clarify VM/application context and presents results on VM characteristics based on different contexts. Chapter 3 introduces RJava, a restricted subset of Java, which is designed to maximize expressiveness, as a HLL, while allowing easy translation into LL languages for better portability and more convenient integration with legacy code. Finally, Chapter 4 concludes the thesis and describes future work.
Chapter 2

VM/Application Interdependencies

An understanding of the dependencies among the components internal to the runtime and those between the application and the runtime is an important aspect of high performance runtime design. This problem exists in all runtime design, but is complicated and made urgent when the VM is written in a high-level language. This chapter discusses this problem and our solution of imposing clear constraints on code structure and context transition to clarify dependencies among different VM components. Our approach is based on an existing metacircular VM, which is the most complex situation for this problem. However, our approach can be easily generalized and applied to general VM design.

This chapter is structured as follows: Section 2.1 states the problem and dangers of unclear VM/application interdependencies, along with a discussion about metacircularity. Sections 2.2 and 2.3 describe our solution of dynamically tracking contexts both in theory and with a concrete implementation. Section 2.4 further introduces our refactoring that brings maximum static context clarification. Section 2.5 shows the low performance overhead of our framework, and extensive profiling results of runtime characteristics, which can be easily acquired with our framework to help understand VMs written in high-level languages. Section 2.6 describes prior work on this topic.

This chapter describes work published in the paper "Unpicking the Knot: Teasing Apart VM/Application Interdependencies" [Lin et al., 2012].

2.1 Introduction

The engineering of a high performance managed runtime is a major undertaking. The competing goals of correctness, completeness, performance, robustness, agility and flexibility sit in tension. In this chapter, we explore the interdependencies among the components internal to a managed runtime and those between the application and the runtime. We do so in a metacircular runtime — a context where these interdependencies are particularly important and particularly hard to tease out. The challenge of cleanly and correctly identifying and designing for these dependencies is key to runtime design and perhaps the greatest impediment to wider use of metacircularity.
2.1.1 Metacircularity

Metacircular runtimes are implemented in the same language that they target, an approach which can bring software engineering and performance benefits, intriguing technical challenges, and a satisfying sense of closure. The intellectual appeal of metacircular managed runtimes is, therefore, unsurprising. We don’t see metacircularity as a meaningful end in runtime design and there are few, if any, examples of the approach being commercially successful. However, we are convinced of the value of using a high-level language to build a high performance runtime [Blackburn et al., 2004b; Frampton et al., 2009]. When the target language is also a good candidate for runtime implementation, metacircularity is the logical outcome. One outcome of our work is a teasing out of the benefits of metacircularity from its shortcomings. Another outcome is that our findings allow us to make significant improvements to an existing metacircular JVM and shed light on an element of runtime design relevant to implementers of future runtimes, metacircular or not. Because it acutely demonstrates the problem of identifying runtime interdependencies, the context throughout the remainder of this chapter is a metacircular runtime.

Metacircularity can have three obvious benefits. First, the strengths of the target language are brought to bear on the implementation. In the case where the target is a strongly-typed high-level language, the software engineering benefits may be clear, particularly where the high-level language is capable of efficiently expressing low-level semantics [Frampton et al., 2009]. Second, the impedance mismatch between implementation and target languages is removed. This problem is most clear at performance-critical boundaries between the runtime and the application, such as in the hot paths of barriers and allocation sequences [Apache]. Finally, a metacircular runtime ‘eats its own dogfood’. This means that the implementers are particularly motivated towards correctness and performance because their implementation has a circular dependency on those properties. Successful cases of metacircular runtimes include Singularity in C# [Hunt et al., 2005], Jikes RVM/Squawk/Maxine/JNode in Java [Alpern et al., 1999; Prangsma, 2005; Simon et al., 2006; Sun Microsystems, 2008], PyPy in Python [Rigo and Pedroni, 2006], Klein VM in Self [Ungar et al., 2005], etc.

Despite this academic success, we know of no commercially successful metacircular runtimes. One explanation might be the performance overheads that one associates with high-level languages, due to garbage collection, dynamic compilation, etc. However, Alpern et al. note that Jikes RVM achieved 95% the performance of its then commercial counterpart, the IBM 1.3.0 DK JVM, on the SPECjvm98 benchmarks on Linux/IA32 [Alpern et al., 2002]. Indeed, metacircularity has the attraction of potential performance benefits:

- **Aggressive inlining and optimization:** Runtime code can be directly inlined into the application context, and optimizations applied across the resulting integrated code.

- **Seamless library downcalls into runtime:** No cross-language bridge is needed for library downcalls, which provides more inlining and optimization opportunities.
These benefits address the target-host impedance mismatch by eliminating the overhead of language transition barriers. However, they also blur the relationship between the application and runtime, and result in unclear execution contexts.

### 2.1.2 Unclear Contexts and Interdependencies

In a metacircular runtime, the application, language library and runtime components do not have a crisply defined boundary within the runtime, and are heavily mixed together. So it is generally not possible to tell, dynamically, whether a thread is executing code on behalf of an application or the runtime. This absence of runtime/application isolation is the metacircular runtime’s dirty laundry, manifest in issues such as security and resource management.

Resource management is one conspicuous manifestation of the lack of isolation that comes with metacircularity. In general, it is impossible to know whether an object is allocated by the runtime or by the application when they both allocate to the same heap. A single allocation site (within a standard library, for example), may allocate objects on behalf of the application and the runtime without distinction. Thus, class objects, class metadata, code, compiler detritus, and other generic objects used by the runtime are intermixed with application allocations. As a consequence, it is hard to know what the real footprint of the application is, and it is hard to know what the adverse locality effect is of this intermingling. Ogata et al. reported that in the IBM J9 JVM, a conventional JVM written in C/C++, non-Java memory usage can account for more than the Java heap in over half of the DaCapo benchmarks [Ogata et al., 2010]. We use our framework to conduct a similar experiment with Jikes RVM, and find that the runtime/application breakdown in our system is similar. As an average of ten DaCapo benchmarks, 57% of memory is used by runtime allocated objects, and only 43% of memory is consumed by application objects. Thus, the Java heap footprint of a benchmark running on a runtime such as Jikes RVM is significantly overstated.

Security can also suffer when isolation is diminished and boundaries are blurred. Consider, by contrast, the boundary between the system and user spaces in an operating system, which are crisply and rigorously defined. Generally within a runtime there are many instances of code that should be considered as privileged. When the execution context of such privileged code is ill-defined security is compromised. This downside to metacircular runtimes is a significant inhibitor to commercial uptake. During the development of the Moxie project, leading developers were polled, and the issue of application/runtime isolation was rated as one of the most pressing issues confronting metacircular runtime design [Blackburn et al., 2008a].

In this chapter we first propose an efficient framework for creating efficient and clean runtime/application isolation and extend it to distinguish among multiple runtime sub-components such as the compiler, garbage collector, classloader, etc. We examine the Jikes RVM code base and identify 41 calls into the runtime. We take the model further by introducing a well-defined concept of runtime services code; stati-
cally identifiable code that is executed by the application on behalf of the runtime, and is typically inlined and optimized by the compiler. This approach minimizes points where the execution context cannot be inferred statically. We have implemented this framework in Jikes RVM, and present preliminary performance numbers that show that it can be implemented at virtually no overhead. We give examples of the mechanism we use to analyze allocation patterns across the application and runtime components. Finally, we argue that the isolation and clarity from using this framework carries with it software engineering benefits, including enhanced modularity within the runtime.

2.2 Tracking Execution Context

Our goal is to better understand the dependencies and dynamic execution behavior of a managed runtime. In particular, we want to remove contextual ambiguity, which arises because of the interactions and dependencies between components. This problem is particularly acute in a metacircular setting, because metacircular runtimes are reentrant: while servicing a request from the application the runtime may need to re-enter itself. For example, while performing a classloading operation, the runtime may need support from the memory manager to allocate objects. In that example, what the memory manager sees as the ‘application’ is in fact the classloader, another part of the runtime. In the absence of dynamic context to inform it, the memory manager is not aware of the fact that it is being called by another part of the runtime.

We attack the issue of contextual ambiguity directly by dynamically maintaining an explicit execution context for each thread in the runtime. We define execution contexts as holding the current runtime request being executed (e.g., classloading or compilation). This state allows us to:

1. Realize context-specific policies, such as using different heaps according to the dynamic context of the allocation. For example, the compiler could use a separate, dedicated, memory management policy.

2. Explicitly assert that certain code may only be executed from within certain dynamic context(s), improving isolation and security, and making it easier to understand the code.

3. Relate behavior to contexts. For example, establishing how often a given library function is dynamically called from within different runtime components or by the user application. Such information could guide optimizations or structural improvements to the runtime design.

Transition Points In order to track execution context, we identify and annotate transition points. These points are comprised of runtime downcalls (DCs): transitions from the ‘application’ to the runtime; and runtime service calls (SCs): intra-runtime requests where the responsibility for execution is transferred to another service. In
§2.2 Tracking Execution Context

(a) Transitions from the application.

(b) All context transitions, including metacircular edges. Moving from left to right, subsystems exhibit fewer and fewer dependencies.

Figure 2.1: A simplified depiction of dynamic context transitions in Jikes RVM.
a metacircular runtime, the ‘application’ may in fact be the runtime itself, so intra-runtime transitions may be due to either downcalls or service calls. We analyzed the Jikes RVM code base and annotated all calls that constitute a transition. A simplified depiction of the context transition graph for Jikes RVM is illustrated in Figure 2.1(a).

Limiting Interdependencies While dependencies between runtime components are an essential property of a managed runtime (in particular in the metacircular case), infinite regress is not. It follows that certain transitions are permissible whilst others are not. As an example, it is fine for the compiler to trigger a garbage collection, but it is not correct for the garbage collector to recursively trigger a garbage collection. So while intra-runtime transitions are permissible, we require that the graph of these context transitions be acyclic. This helps to curtail dependencies within the runtime, and is of particular value in dealing with metacircularity. In general, this corresponds to disallowing individual features or services from (transitively) requiring access to the feature or service they implement.

Selecting Contexts We broaden the definition of execution contexts from the binary application/runtime divide, to include a number of key components of the runtime as discrete contexts, such as memory management, class loading, etc. This refinement allows us to differentiate an acyclic downcall from the classloader to the memory manager from a cyclic call from the classloader to itself or from the memory manager to itself. Figure 2.1(b) exposes the intra-runtime dependencies that were elided from Figure 2.1(a). Only substantive transitions are tracked. There also exist ‘lightweight’, superficial, transitions which are not tracked for pragmatic reasons and therefore do not register as transitions, as we discuss below.

Upcalls Managed runtimes often require upcalls from the runtime back to the application. For example, a downcall to the classloader may eventually lead to an upcall to the application to execute a class initializer (illustrated in Figure 2.1 as a dashed line). The upcall may in turn lead to a downcall, creating a cycle. The possibility of generating cycles this way is a property of the runtime specification, independent of the runtime implementation, and is therefore outside the scope of this thesis. However, our approach must accommodate upcalls. To achieve this, we include an explicit transition of execution context when an upcall is made, and we discount upcall edges when checking for cycles in the graph of context transitions. Upcalls in effect amount to a break in the graph of metacircular context transitions.

Lightweight Transitions In addition to the ‘substantive’ transitions discussed above, there also exist lightweight transitions which we pragmatically do not track. Examples of these may include SCs such as accessor methods within the classloader which are used by the compiler or garbage collector to identify field offsets, or DCs such as the fast path of an allocation sequence or write barrier. In the absence of a context
transition, the dynamic context remains that of the calling context (e.g. the compiler) although the code falls within the static domain of the callee context (e.g. the classloader). In a non-metacircular case, the code for a lightweight DC is easy to identify because it will be written in the target language (e.g. Java) or perhaps in inlined assembler. However, in a metacircular case there is no language transition so the transition point for the lightweight DC is ambiguous. To remove this ambiguity, we explicitly identify and annotate such code with a @RuntimeService annotation, which facilitates static checking whilst still supporting this common optimization.

**Precise Transition Placement**  In practice, a transition between components, in particular from the application to the runtime, may manifest as a relatively lengthy call chain, inviting the question of where to draw the line and declare the transition point. For example, a request for memory when allocating a new object will typically start with a frequently taken fast path, and only fall through to a slow path in the uncommon case. In both metacircular and non-metacircular runtimes, the fast path is typically inlined into the application code by the compiler, while the call to the slow path is kept out of line. In this case, we can argue that the fast-path constitutes a lightweight DC with the substantive DC only beginning at the call to the slow path. In a non-metacircular runtime, a natural choice may be the inevitable language boundary between the supported language (Java) and implementation language (C++), which most likely occurs at the call to the slow path. In a metacircular runtime the language transition does not exist, forcing the implementer to make a decision in each case. Our approach, when given a choice, was to select the first out-of-line call as the point of the DC. We reason that this allows us to trigger transitions as close to the application as possible whilst ensuring transitions do not occur too frequently, which would lead to unacceptable overheads.

### 2.3 Implementation

We now describe an implementation of our framework for efficiently tracking execution context. Concretely, the implementation involves two steps: 1) additions to the runtime that provide the mechanisms for efficiently identifying and tracking context transitions, and 2) modifications to the runtime to identify all transition points (DCs and SCs).

#### 2.3.1 Supporting Mechanisms

We create two annotations, @RuntimeDownCall and @RuntimeServiceCall, to allow us to cleanly identify and differentiate DCs and SCs within the runtime. Figure 2.2(a) illustrates how @RuntimeDownCall annotation is used to identify `Object.wait()` as a downcall. The compiler then expands the annotated method to include context switching calls, as illustrated in Figure 2.2(b). Figure 2.3 illustrates context switch methods that track the execution context for each thread, and may optionally implement assertions to ensure only legal context transitions occur.
VM/Application Interdependencies

```java
@RuntimeDownCall(Context.Scheduler)
public void Object.wait() {
    /* Original code */
}
```

(a) Annotating Object.wait() as a DC.

```java
public void Object.wait() {
    int old = switchContextTo(Context.Scheduler);
    try {
        /* Original code */
    } finally {
        switchContextBack(old);
    }
}
```

(b) The same code after expansion by the compiler.

Figure 2.2: An example transition point, Java’s Object.wait().

**Assertion of execution context.** We use two basic techniques to assert the current execution state, helping with both finding transition points and making runtime development easier. First, we implement an `@AssertExecutionState` annotation that may be applied at the granularity of a class or method, which asserts that the code covered by the scope of the assertion is only executed within a specified set of execution contexts. Thus, if there exists a class that may only ever be called within the context of garbage collection, such an assertion will ensure that this restriction is enforced. Second, we automatically inject assertions into user code at run time to ensure that such code is only executed in an ‘application’ context.

Newly created threads are assigned an initial execution context based on the type of thread being created. For threads that exist within the runtime, such as the compiler thread, we set their initial execution context appropriately.

### 2.3.2 Finding Transition Points

We identified transition points iteratively. First we identified major runtime contexts such as the garbage collector, scheduler, etc., and found all transitions into each context. We resolved whether a transition was a DC (from the ‘application’ to the runtime) or an SC (intra-runtime requests), and marked it with corresponding annotation. For example, Figure 2.2(a) illustrates a downcall to the Scheduler context. We then enforced the acyclic limitation on interdependencies, and further refined context selection. We chose the set of contexts to reflect the structure of the runtime and to maintain the invariant that no transition point is called from within the context it targets.

We also made extensive use of assertions, which we added to code whose context
class VMThread {

    /* Current execution context */
    int executionContext;

    /* Change context and return previous context */
    int switchContextTo(int newContext) {
        int oldContext = executionContext;

        /* Assert that transition is valid */
        executionContext = newContext;
        return oldContext;
    }

    /* Change back the context */
    void switchContextBack(int oldContext) {
        executionContext = oldContext;
    }

    ...

Figure 2.3: VMThread support for dynamic execution context.

we knew unambiguously. For example, an assertion of garbage collection state that is placed unambiguously within the garbage collector would fail if a transition point into the garbage collector had been overlooked. By following the stack trace from the failed assertion, we were able to identify the missing transition point. We repeated this process until we found all transition points. During this process we identified small sections of code within the gray area where execution context is ambiguous. Our identification of these ambiguous areas led to us refactoring the runtime to remove such ambiguities, as we discuss in Section 2.4.

Table 2.1 summarizes the transition points we found in Jikes RVM, and their groupings. In all, we identified 41 transition points grouped into 10 classes. We implemented a simple tool that generates a state transition diagram, which allows us to visualize all state transitions. This tool was used to automatically generate Figure 2.1. Note from this figure that the memory manager can be reached directly from the application, and also via a chain through the classloader, exceptions, and locking, but never from itself. This graph highlights the fact that the memory manager is written in a restricted style which ensures that it does not generate downcalls to any of the other subsystems of the runtime.

2.4 Design Implication and Lessons

The act of making context and context transitions crisp and explicit encouraged us to rethink elements of the runtime design. In our case this resulted in some minor refactoring and explicit annotations with the consequence that the static code context
Table 2.1: Transition Point classes and counts for Jikes RVM.

<table>
<thead>
<tr>
<th>Transition Point Class</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemoryManager</td>
<td>8</td>
</tr>
<tr>
<td>Scheduler</td>
<td>8</td>
</tr>
<tr>
<td>Classloader</td>
<td>6</td>
</tr>
<tr>
<td>StackTrace</td>
<td>6</td>
</tr>
<tr>
<td>Compiler</td>
<td>4</td>
</tr>
<tr>
<td>Exception</td>
<td>3</td>
</tr>
<tr>
<td>Locking</td>
<td>2</td>
</tr>
<tr>
<td>Reflection</td>
<td>2</td>
</tr>
<tr>
<td>YieldPoint</td>
<td>1</td>
</tr>
<tr>
<td>NewArrayArray</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>41</strong></td>
</tr>
</tbody>
</table>

more closely matches the dynamic execution context. Such changes improved the VM we target and we imagine that the patterns we applied will be relevant to other VMs, whether or not they are metacircular.

In Section 2.2 we introduced an annotation, @RuntimeService, to statically identify contexts where lightweight service calls are executed by the caller on behalf of the callee’s context. When we systematically applied this annotation throughout Jikes RVM, it highlighted the fact that ambiguity between contexts, including those between the application and runtime code are re-enforced by the structure and software engineering of the runtime. We now describe how we refactored code to better reflect execution context boundaries.

**Inter-class refactoring.** Where possible, we moved all lightweight SCs out of their original context into distinctly named packages. By better reflecting the dynamic contexts in the structure of the runtime, this restructuring avoided a large number of unnecessary transition points. It also became straightforward to assert the correct context in the majority of the codebase, since it was largely cleansed of ambiguous code, making the static to dynamic context mapping straightforward in many cases.

Figure 2.4 illustrates our refactoring with the example of a bump pointer allocator. Figure 2.4(a) shows the fast path of the allocator, which is executed by the application on behalf of the memory manager. The code is in a distinct package for runtime services and the class is annotated with @RuntimeService. When a local pool of memory is exhausted (in line 14), the fast path falls through to a slow path. This call is a transition point, a fact reflected by the call into the memory manager proper to a method annotated with a @RuntimeDownCall annotation (Figure 2.4(b)).

Our refactoring also brings the metacircular runtime closer to a non-metacircular runtime, where the impedance matching service code is explicitly identified and exposed. The change also makes it easier to consider hosting another language.
§2.4  Design Implication and Lessons

```java
package org.jikesrvm.runtime.services.mm

@RuntimeService
public class BumpPointer {
  Address cursor;
  Address limit;
  BumpPointerSpace global;

  @Inline
  public Address alloc(int bytes) {
    Address start = cursor;
    Address end = start.plus(bytes);
    if (end.GT(limit))
      return global.allocSlow(this, bytes);
    cursor = end;
    return start;
  }
}
```

(a) Fast path of bump pointer allocator; an example of a runtime service.

```java
package org.mmtk.util.alloc

public class BumpPointerSpace {

  @NoInline
  @RuntimeDownCall(Context.MemoryManager)
  public Address allocSlow(BumpPointer allocator,
                            int bytes) {
    Extent blockSize = getBlockSize(bytes);
    Address start = acquire(blockSize);
    if (start.isZero()) {
      return Address.zero();
    }
    allocator.cursor = start;
    allocator.limit = start.plus(blockSize);
    return allocator.alloc(bytes);
  }
}
```

(b) Slow path of bump pointer allocator in the memory manager.

Figure 2.4: Inter-class refactoring of runtime services: BumpPointerAllocator.
```java
public class RVMClass {
    private byte state; //current class-loading stage

    ... 

    @RuntimeService
    @Inline
    public boolean isResolved() {
        return state >= CLASS_RESOLVED;
    }

    @RuntimeService
    @Inline
    public synchronized void resolve() {
        if (isResolved()) return;
        if (superClass != null) superClass.resolve();
        for (RVMClass interface : declaredInterfaces)
            interface.resolve();
        resolveInternal();
    }

    @RunnimeDownCall(Context.Classloader)
    public synchronized void resolveInternal() {
        /* heavyweight resolving code */
        state = CLASS_RESOLVED;
    }
}
```

Figure 2.5: Intra-class refactoring of runtime services: RVMClass.
§2.5 Results

Intra-class refactoring. We found that it was not always practical or desirable to factor runtime services into separate classes and packages. Figure 2.5 contains an example of the intra-class refactoring that was applied to the classloader. The classloader we use represents classes with $\text{RVMClass}$ instances. These contain metadata and simple methods to inspect that data in addition to heavy-weight operations such as those to support loading and resolving classes. A number of services, such as $\text{isResolved()}$, are typically inlined by the compiler into the application code. These cases are lightweight DCs because they are executed by the application on behalf of the runtime. In this circumstance, insisting on separating them into separate classes would not achieve our software engineering goals, and would only serve to complicate the structure of the runtime. Instead, we place $\text{RuntimeService}$ annotations on these inspection methods. This solution does not provide all the benefits of separate classes, but allows stronger isolation and clarity.

2.5 Results

We now present results for several experiments we conducted using our framework. In the first experiment we measure the performance impact of our refactoring and our framework for tracking dynamic execution context, showing that the overhead of each is negligible. In the second experiment we use the framework to analyze the behavior of Jikes RVM, measuring the distribution of transitions, as well as a breakdown of the time spent in each execution context. In the third experiment we analyze Jikes RVM’s memory management behavior, showing that different contexts exhibit different behavior, and that on average around half the memory footprint is due to the runtime.

2.5.1 Methodology

Our implementation is based on Jikes RVM [Alpern et al., 1999] release 3.1.1 +hg r10393. All experiments use a production build: the default high-performance configuration which has debugging assertions turned off, builds code with an aggressive optimizing compiler, and utilizes a high-performance generational Immix garbage collector [Blackburn and McKinley, 2008]. For performance measurements we run
each benchmark 10 times (10 invocations) and report the average time for the 5th iteration (steady state performance). We also report 95% confidence intervals for the average using Student’s t-distribution.

We draw our benchmarks from the DaCapo [Blackburn et al., 2006] and SPEC-jvm98 [Standard Performance Evaluation Corporation] benchmark suites. We use benchmarks from both the 2006-10-MR2 and 9.12 Bach releases of DaCapo to enlarge our suite and because some 9.12 benchmarks do not run on Jikes RVM.

Performance measurements were conducted using a six-core AMD Phenom II X6 1055T with 3MB of L2 cache running at 2.8GHz, and 4GB of RAM. The machine was running the Ubuntu 10.04.01 LTS server distribution with a 64-bit (x86_64) 2.6.32-24 Linux kernel.

2.5.2 Framework and Refactoring Overhead

We start by examining the performance overhead of the design changes we made to support the framework (as described in Section 2.4) and the overhead of dynamically tracking execution context in a moderate (4x minimum) heap. Figure 2.6 shows the performance impact due to the refactoring (red) and then the refactoring with dynamic context tracing (green), both relative to an unmodified Jikes RVM configuration (blue).

**RuntimeServices refactoring.** We did not anticipate any significant overhead due to our refactoring, and Figure 2.6 shows that this is generally true, with an average overhead of 0.1%, which is well within the noise. Individual benchmarks range from 2.4% speedup to a 1.6% slowdown.

**Tracking execution context.** Our TrackContext configuration performs on average 0.6% worse than the Base system. Again, these values are very small compared to the noise, suggesting that on average tracking context incurs a negligible overhead. There are two benchmarks, bloo and xalan, that exhibit slightly higher overheads (4.5 and 3.2% respectively) although it is worth noting that the confidence intervals for these benchmarks are also among the largest. In summary, these results show that across a wide selection of benchmarks our execution context tracking incurs minimal overhead.

2.5.3 Context Transition Analysis

We now use our framework to analyze dynamic context transitions using instrumented versions of our framework. We count both the overall frequency of transitions, and the distribution across individual transition points for each benchmark. We also count the CPU time executing each context to help understand where time is spent during execution.
Table 2.2: Frequency distribution across individual transition points, showing the 20 most frequently executed transition points on average. Each number indicates the percentage of total transitions for that benchmark for a given transition point.
Table 2.3: CPU cycles spent in each execution context.
§2.5 Results

Execution frequency The frequency of transitions will have a direct impact on the overhead of our system. Figure 2.7 shows the transition frequency for each benchmark, measured in invocations per millisecond. We see a wide range of values, from 850 calls per millisecond for bloat, to less than 10 per millisecond for compress, mpegaudio and sunflow. Note that these low-frequency benchmarks are kernel-based benchmarks, and that two of the three are from the older SPECjvm98 benchmark suite. On average, the frequency is 205 transitions/ms.

In general, we expect benchmarks with more frequent transitions to incur a greater overhead. However, comparing Figures 2.7 and 2.6 shows no clear relationship. As an example, avrora has an invocation frequency more than 30× that of db, but avrora speeds up, while db slows down. This indicates that the overhead of changing execution state is small, and being obscured by other factors. We believe these effects are due to interactions with the optimizing compiler. Injecting code may affect inlining decisions, and may more significantly affect methods which have greater register pressure.
Figure 2.9: Nursery survival for objects allocated by execution contexts.
Distribution of executed transition points. Table 2.2 shows the most frequently executed transition points, both on average and for individual benchmarks. By far the most frequently executed transitions occur at `Lock.lock` and `Lock.unlock`, which support the implementation of Java's `synchronized` keyword. Note that the frequency of lock and unlock calls in Table 2.2 do not match. This is primarily due to the use of a biased locking protocol. When an object is first locked, a DC is required for the first thread to obtain the bias for the lock, which then allows that thread to continue to lock and unlock that object within a lightweight DC, assuming no contention occurs.

Although allocations and generational write barrier invocations are very frequent, heavyweight DCs related to these operations are uncommon because they only occur during (rarer) slow path invocations. This shows the benefit of using lightweight DCs for the fast path. If heavyweight DCs were used on the write barrier fast path, they would account for over a third of all transitions on average.

Time spent in execution contexts. Table 2.3 shows the CPU cycles spent in each execution context when running a single iteration for each benchmark (i.e. startup, rather than steady state). Each result is the fraction of total CPU time spent in each context (including the application context and runtime contexts). We measure cycles for all threads in the system, which includes applications threads as well as runtime threads (e.g., garbage collection and adaptive compilation threads) which never directly run application code but still contribute to overall execution time.

Table 2.3 shows that for most benchmarks the vast majority of time is spent performing application work. However, for some benchmarks, time spent in the runtime can be significant, in particular for `avrora` which spends more than half of all of its cycles executing slow paths for synchronization primitives (e.g., `wait`, `notify`, `lock`, and `unlock`). On average, nearly 13% of cycles are spent performing compilation, and nearly 10% of cycles can be attributed to synchronization/scheduling operations. Garbage collection and other memory management operations (such as slow path allocation) account for a further 10% of execution on average.

2.5.4 Memory Management Behavior

One of the motivations for this work was isolating application and runtime behaviors, allowing new analyses and optimizations. The behavior of different contexts are typically conflated, in particular when analyzing a metacircular runtime such as Jikes RVM. In this experiment we examine the memory management behavior of the runtime and applications separately. Specifically, we explore allocation volume, nursery survival rate, and heap footprint. These experiments were run using a $2.5 \times$ minimum heap size.

Object Allocation To understand object allocation patterns, we collect allocation information by instrumenting the allocation fast path and tagging each object as it is allocated. Because many execution contexts allocate very little (or not at all) we use a
logarithmic scale, and only show those contexts that make a significant contribution, aggregating the rest into an ‘Other Runtime’ category.

Figure 2.8 shows the fraction of total allocation attributed to each execution context. It is clear that application allocation dominates, although it does vary significantly between benchmarks (not shown in the figure). The most active context is the compiler, followed by booting and classloader. In all the benchmarks, the compiler allocates more than all other runtime contexts combined, and in 13 of the 17 benchmarks, more than 10× that of the rest of the runtime.

Survival Ratios Object lifetime patterns help guide memory management policies. If different runtime contexts exhibit significantly different lifetime patterns, there may be an opportunity to apply context-specific memory management optimizations. Here, we measure the fraction of objects that survive the first nursery collection after allocation. To measure survival rate for each context, we tag the current execution context in the header of each object at allocation time and count the total number of bytes allocated in each context. Then during collection we count the bytes that survive the collection, and express this as a ratio of the total bytes allocated in that context. The experiment is imperfect because a nursery collection is triggered based on total allocation, not allocation per context, but the results still provide interesting insights. Figure 2.9 shows the result of this experiment. The booting and classloader contexts show consistently high survival rates. In comparison, the survival rate for the compiler is consistently low, with less than 20% of data surviving the first nursery collection in 12 of the 17 benchmarks.

 These experiments show that while allocation behavior between applications varies, allocation in runtime contexts follows a more regular pattern. The compiler context allocates more memory than all other runtime contexts combined for every benchmark, and also shows a consistently low survival rate, averaging just 16.4%. The other allocation-intensive contexts are the classloader and runtime booting procedure, which have stable and high survival ratios.

Heap Footprint Our framework allows us to easily determine the objects allocated by the runtime. This is generally difficult in a metacircular runtime because application and runtime objects are mixed together in a single heap. In our preliminary study we allocate runtime objects into a different region of memory based on the dynamic execution state. Then, at garbage collection time, we can measure the live size of the respective heaps to determine the memory footprint for the application and the runtime separately.

Figure 2.10 shows that in more than half of the benchmarks the memory footprint of the runtime is larger than that of the application. Ogata et al. make a similar observation for the IBM J9 runtime, which is not metacircular [Ogata et al., 2010].
§2.6 Related Work

2.5.5 Summary

These results show that a framework such as ours can disambiguate dynamic execution context at very low overhead, and can also be used to both understand and improve the implementation of managed runtimes.

2.6 Related Work

We categorize related work in two parts: 1) the state of the art for metacircular runtimes, for which ambiguous context is one of the major problems, and 2) approaches to improve isolation within runtimes more broadly.

2.6.1 Metacircular VMs

There is a long history of metacircular runtimes, tracing back to LISP in the 1960s. In some languages, metacircular implementations are in the mainstream (e.g., LISP and Smalltalk). For others, in particular Java, metacircular implementations have been more of a sideshow to commonly used production runtimes, despite a long thread of interest: the first Java-in-Java runtime was created not long after Java’s initial introduction.

Jikes RVM (formerly known as Jalapeño) was the first high-performance metacircular runtime written in Java [Alpern et al., 1999]. Besides a small amount of C code used to boot the runtime and act as an OS interface layer, Jikes RVM is entirely implemented in Java, and once built can execute without the support of a host JVM. Jikes RVM largely ignores matters of application/runtime isolation in its design.
Jikes RVM generally identifies methods that have calls injected by the compiler (one type of downcall from the application into runtime) this convention is not enforced by the runtime. Our implementation is based on the most recent release of Jikes RVM.

**Moxie** was designed to be a platform for developing production-quality JVMs and performing research into new JVM designs and technologies [Blackburn et al., 2008b]. Moxie is metacircular, and targets portability, clean bootstrap, debugging and systems programming. The importance of VM/application isolation was recognized early in the Moxie design process. Moxie set a goal of having all transitions from application to JVM services explicit (similar to an operating system trap). As part of this process, Moxie developed a quite different bootstrap model to that used in Jikes RVM, and its *hosted-target* execution model was an important step towards strong VM/Application isolation [Blackburn et al., 2008b]. Unfortunately, the Moxie VM was never publicly released.

**Maxine** is, like Moxie, a research platform for next generation Java runtimes [Sun Microsystems, 2008]. Maxine is metacircular, and is built on principles of modularity and configurability. Maxine provides dedicated replacement mechanisms for important components, including the compiler, garbage collector, and object model. To our knowledge, Maxine does not particularly address the issue of application/runtime ambiguity that arises due to metacircularity, which is the focus of our work.

**Ovm** is a research JVM that particularly targets realtime applications. [Palacz et al., 2005]. Ovm provides the basic components of a managed runtime, each of which is written almost entirely in Java. Ovm is designed as a general framework to support different object models. Ovm's intermediate intermediate representation, OvmIR reduces some of the ambiguity between the runtime and target language. However, this approach does not serve as a common solution to the context ambiguity and isolation problem that arises from metacircularity, which is our focus in this chapter.

### 2.6.2 Prior Work in VM Isolation

Other work has focused on the issue of isolation in the context of managed runtimes more broadly, motivated by other concerns such as security and inter-application isolation.

**KaffeOS** [Back et al., 2000] is a Java runtime system designed to implement isolation and resource management boundaries between different applications in a single runtime based on the Kaffe VM. KaffeOS is not metacircular, but is closely related to our work because isolation between different contexts is a primary goal of KaffeOS. In KaffeOS, to support application isolation, limited VM/application isolation is also required. KaffeOS uses a *red line* metaphor [Cheriton and Duda, 1994] to divide 'user mode' and 'kernel mode' contexts. This 'red line' metaphor is similar to our dynamic switching, but there are also some important differences, due to the non-metacircular nature of KaffeOS. First, the kernel in KaffeOS is written in C while the other parts are written in Java, thus a clear cross-language boundary indicates parts of the 'red
line' between the runtime and application. Also, the metaphor does not translate well to the metacircular setting, because the metacircular runtime is *reentrant*: during execution a 'red line' could be crossed multiple times. We analyze this problem in detail and propose a clean approach for maintaining a more appropriate runtime state for a metacircular runtime. In addition, our approach to maintaining context has a low overhead, while the isolation used by KaffeOS is more heavyweight, with an 11% overhead.

### 2.7 Summary

The dependencies between the application and the runtime that supports it—and between the components of the runtime itself—are critical elements of runtime design, yet are often unclear. We examine the problem of these dependencies in the context of a metacircular runtime, where the boundaries between the application and individual runtime components are particularly unclear. We demonstrate a low overhead framework that allows us to clearly identify the dynamic execution context within the runtime, and show how this has led to software engineering improvements in the JVM we use.

We imposed clear constraints on runtime code structure and context transition to clarify VM/application interdependencies. 1) We introduce transition point annotations and require all runtime methods that would cause context transitions to be marked with such annotations. This allows the runtime to maintain a clear dynamic execution context within the metacircular runtime with very low overhead. 2) We introduce the service annotation and require all runtime code with ambiguous context to be marked with such annotation. The explicit identification of context-clear/ambiguous code drives a restructuring of an existing runtime to substantially reduce structural context ambiguity.

The contributions of our work are: 1) a critique of metacircularity that identifies lack of isolation and blurred boundaries as blockers to more widespread uptake of metacircularity, 2) a concrete implementation of a new very low overhead framework that reintroduces isolation without forsaking the benefits of metacircularity, 3) a quantitative analysis of Jikes RVM's behavior that was not possible without our framework, and 4) insight into how these lessons can be applied to existing and future runtime implementations, whether they are metacircular or not.

Although our implementation is developed in the context of a particular runtime, the principles we apply are general in VM design, especially VM design using high-level languages. We hope that our insights and analysis may help runtime developers think more clearly about the various contexts that constitute a VM and their relationship. In the next chapter, we will discuss another software engineering problem, portability pitfalls of high-level low-level programming.
VM/Application Interdependencies
Chapter 3

Bypassing Portability Pitfalls of High-level Low-level Programming

The previous chapter presented work to address VM/application interdependencies and showed how constraints can be introduced to define execution contexts and their transitions. This chapter looks at a separate, but equally important software engineering problem: portability pitfalls of high-level low-level programming. In our thesis context, portability pitfalls 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 have to be extended and restricted to effectively undertake low-level programming. The extensions are specific to certain projects, VMs, and platforms, thus good portability, as one of the important features of modern high-level languages, is compromised. As a consequence, VM components cannot easily be reused in other projects. This chapter describes our approach to bypassing portability issues of high-level low-level programming.

This chapter is structured as follows: Section 3.1 discusses the problem of poor portability for VM components written in a HLL and argues that a restricted language can be introduced to implement VM components and that allows a low-level language bypass which improves portability. Section 3.2 tidies up our approach to language restriction and the proper position of restriction within a system programming project. Section 3.3 proposes a restricted language called RJava and uses MMTk as an example to show how the elements of RJava are used in practice. Section 3.4 discusses key parts of the low-level language bypass of RJava, which is currently under development.

3.1 Introduction

Implementing an efficient modern language runtime is always a hard task, and requires experts from different areas, such as memory management, thread management and 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. This includes reusing existing VM components and allowing reusability of newly implemented components.
There are two prerequisites for reusability.

Decoupling of VM Components. Language VMs tend to be monolithic for performance reasons. However, modular design means less dependency and cleaner interfaces, which favors reusability. Modular design has also been successful. A JIT compiler, as one of the most complex and important components, has least dependencies with the rest of the VM, and is highly reusable [Alpern et al., 1999; David Stutz, 2003; Cierniak et al., 2005]. A memory manager with careful modular design is also a potential reusable candidate without hampering performance [Blackburn et al., 2004b]. Chapter 2 in this thesis focuses on dependencies between the VM and application, and also provides insight into modular design by clarifying dependencies between contexts internal to the VM (similar to components).

Portability of the language used to implement VM components. To facilitate the reusability of VM components, the implementing language needs to execute with a proper hosting runtime on the target platform. Some low-level languages, such as C, and most high-level languages are portable, since there are usually available compilers or language VMs for such languages across different platforms. However, high-level low-level programming, as an intermediate form between high-level languages and low-level languages, is less portable than both of its counterparts. 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.

Efficient high-level low-level programming relies on three key elements: compiler intrinsics, semantic restriction and unboxing of user-defined types. Those require support from the runtime. These extensions and restrictions may be very specific to a certain project and are unlikely to receive wide support from a common VM of the given language. Thus, any specific 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 [Garner, 2003; Geoffray et al., 2010]. Both involve non-trivial work for each porting.

One way to improve portability is to turn the repeated work during porting into a one-time work that eases all subsequent porting. The org.vmmagic package [Frumpston et al., 2009], a low-level extension package of Java adopted by many high-level low-level programming projects [Alpern et al., 1999; Prangsma, 2005; Sun Microsystems, 2008], takes a similar idea and uses an IntrinsicGenerator to separate the semantics of intrinsic functions (extensions) from the hosting runtime so that the IntrinsicGenerator carries most information needed for intrinsic functions. However, this does not solve the portability issue. On one hand, an IntrinsicGenerator needs intimate information from the runtime, such as types and object layout, thus
§3.1 Introduction

it has to be tightly coupled with the runtime, or modularized with a bloated interface. On the other hand, unboxing, another extension that org.vmmagic introduces (e.g. Address is treated as a plain pointer instead of a Java object), still has to be implemented and deeply integrated into the runtime’s type system.

In this chapter, we describe our bypass approach via lower-level languages, which resolves the portability pitfalls of high-level low-level programming and still preserves most benefits of a HLL.

3.1.1 Observations on System Programming with a HLL

Our approach is based on some important observations. We made those observations on Jikes RVM as an example of system programming with a HLL.

Restrictions omit features of HLLs. Implementing a high performance VM using a HLL relies on not only extensions, but also restrictions. Language restriction in the context of this thesis means that some language features are omitted in exchange for correctness and performance. Restriction includes semantic restriction to forbid parts of the language semantics. For example, in a metacircular implementation of a memory manager, the semantics of allocation have to be forbidden for correctness. Table 3.1 shows some semantic restrictions to the Java language in different parts of Jikes RVM, including MMTk as the memory manager and the baseline compiler as an equivalent to an interpreter in most VMs, and this table conveys a clear idea that the code of a HLL in system programming follows a quite different pattern. Restriction may also be motivated by performance. For example, in Jikes RVM, @NonNullCheck omits run-time null checks for object access in certain methods and @NoBoundsCheck omits run-time array bound checks for array element access. These kinds of restriction are important in performance-critical code.

Restrictions reduce benefits from HLLs. Generally high-level languages provide typesafety, 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 cannot appear 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.

Restrictions are only applied to a limited scope. Restriction is essential for correctness and performance in system programming with HLLs, but different levels
Bypassing Portability Pitfalls of High-level Low-level Programming

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

Table 3.1: Language restrictions in different scope of Jikes RVM (occurrences per LOC).

...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 [Frampton, 2010]. Table 3.1 also shows good design of 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 baseline compiler) still has 47K LOC, which is important and large enough that is worth careful consideration.

3.1.2 Bypass without Compromising

Based on those observations, it is clear that restriction of HLL is essential, and strict restrictions can be allowed in small scopes, even though restriction reduces benefits at run-time.

We extend this idea, and propose an approach to improve portability of restricted high-level languages. We introduce heavier restrictions to HLLs to form a well-defined restricted language. We clarify the restriction rules and apply constraint checks so as to ensure the compliance of its programs and the static benefits from HLLs. Also, the language variant is restricted in such a way that it can be translated to a lower-level language. Thus, for portability, programs coded in the restricted language can either be executed within a managed runtime or translated into a portable low-level language and executed natively. Figure 3.1 illustrates this idea.

We chose MMTk, a high-performance memory manager written in Java [Blackburn et al., 2004b], as our porting candidate. One of MMTk’s design goals is portability, thus it is independent of any hosting VM and provides a clean interface for the hosting VM. However, its portability is still constrained by the portability of high-level low-level programming and the porting work was non-trivial [Garner, 2003; Blackburn et al., 2005; Geoffray et al., 2010].

We extracted the restrictions and org.vmmagic extensions used in MMTk and define a restricted variant of Java called RJava. VM components written RJava can be ported and executed in a Java runtime if the hosting VM supports org.vmmagic
3.1 Introduction

Figure 3.1: An illustration showing our bypass approach for portability issues.

extensions. If there is no available runtime to support its extensions on the target platform, RJava can be translated into a more portable lower-level language and be integrated and executed as native code. A toolchain to enable such a bypass is under development. The toolchain consists of a static constraint checking tool that ensures restriction compliance, a frontend to translate RJava into C++ and a C++ backend.

One important step in the low-level language bypass of RJava is the clear definition of constraints/restrictions and static checking. In Chapter 2, we were convinced by the power of using constraints to bring context clarification to source code. Here, we rely heavily on annotations to express constraints/restrictions in RJava, and impose them with static checking. This approach helps maximize correctness, including type safety and memory safety, and ensures that our bypass to a LL language will not regress.

3.1.3 Towards Restricted Language for More General Use

According to the discussions above, it is reasonable to use a heavily restricted language variant for the implementation of VM components in order to achieve portability. We now elevate this idea and design RJava as a relatively independent restricted language for more general use. Apart from improving portability, its potential use for other purposes is encouraged by our design. One example is to implement an interpreter or compiler to allow bootstrapping [Rigo and Pedroni, 2006]. If the interpreter or compiler written in the restricted language can support full language syntax and be executed natively, the interpreter (native) could support execution of its own code in a managed runtime and thus get the rest of the VM bootstrapped. This would introduce a clean and interesting bootstrapping option for metacircular VMs.

The next two sections detail the development of RJava, including our design concerns for RJava as well as the constraints we introduce and their compliance checking.
3.2 Design Concerns of RJava

In the last section, we discussed the motivation for RJava, as a language that has clear restrictions and allows a low-level bypass for improved portability. 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.

3.2.1 Reasons for Restriction

Examining the rationale for HLL restrictions in system programming helps proper definition of such languages. 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 [C2 Wiki]. However, these are not the exact reasons for system programming. 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 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, around which RJava is developed. 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 focus for RJava is the scope of implementing a portable memory manager. Thus, we define restrictions to that particular scope, but we generalize our approach to be as flexible as possible so that RJava can be used in other scopes without major change.

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. 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 [Driesen and Hölzle, 1996]. 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. Besides, 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.

3.2.2 Expressiveness vs. Restrictions

Higher-level languages are more expressive than low-level languages. Java, for example, is considered to have a $2.5 \times$ 'statement ratio' compared to the C language (i.e., on average, one Java statement needs 2.5 C statements to achieve the same function [McConnell, 1993]). 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.

3.2.3 RJava in Different Scope

Ideally we want RJava to be a fixed language with constant restrictions so we can use RJava to implement some 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 scattered 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.

3.3 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.

3.3.1 Language Key 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” [Frampton et al., 2009]. 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. @Uninterruptible is now 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 thesis. 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 3.2 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.

Figure 3.2: RJava restriction rules and rulesets.
3.3.2 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 emitting any code during code generation that would trigger thread switching or garbage collection.

The restriction ruleset for MMTk is showed in Figure 3.3(a).

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 [Paleczny et al., 2001]. 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

\(^1\)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 3.3: MMTk with RJava.
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 [Blackburn et al., 2004a]. 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 3.3(b) 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 3.3(c) shows this restriction ruleset.

Figure 3.3(d) 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.

3.4 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 3.1).

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.

3.4.1 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.
§3.4 Current Work: An RJava to LL Language Bypass

There are existing tools for Java syntax checking, such as PMD [PMD Project]. 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 [Sable Research Group, McGill]. 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 @NoRuntimeAlloc restriction. What remains to be done is expanding the rule/ruleset checking to cover all of the other RJava restrictions.

3.4.2 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 in the runtime. 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 [Sieka]. 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.

3.4.3 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.

3.5 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 [Alpern et al., 1999; Palacz et al., 2005; Blackburn et al., 2008], a Java operating system [Prangsma, 2005], and a C/C++ JVM [Apache]. 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 [Ancona et al., 2007]. RPython is a restricted subset of Python and is used to write an interpreter in the PyPy virtual machine [Rigo and Pedroni, 2006]. However, it is also an independent language that can be used for general use. RPython inherits 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 3.2.3, we explained its necessity in VM implementation). Besides, its restrictions are not clearly defined [RPython Coding Guide], 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 runtime 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.
3.6 Summary

This chapter discussed the portability pitfalls of high-level low-level programming, including the problem statement and its causes. With this motivation, this chapter introduced a restricted language called RJava and its low-level language bypass. One key element in RJava is constraints, appearing as restriction rules and rulesets. Clarifying and imposing the constraints ensures the performance and correctness of RJava programs and also allows a bypass to a LL language for better portability.
Conclusion

Due to the growing complexity of hardware and software, high-level languages that provide type-safety, memory-safety, and better software engineering are encouraged and widely accepted for application programming. For the same reasons, we see a trend of applying high-level languages to systems programming. The feasibility and performance of implementing language virtual machines in a HLL has been verified in several research projects, and the security, productivity boost and software engineering advantages introduced by using a HLL has benefited those projects.

However, despite the benefits brought by using a HLL for VM implementation, using a HLL for system programming also brings with it software engineering pitfalls, affecting the robustness, performance and correctness of the whole system. We argue that the state-of-the-art approach to VM implementation with HLLs needs to be further improved.

This thesis discusses two aspects of software engineering challenges in VM implementation brought in by using a HLL: 1) unclear VM/application interdependencies and 2) poor portability. We test the idea of introducing clear constraints to carry extra semantics that we want to impose on the code. In both scenarios, imposing clear constraints proves to be a valuable solution that not only brings clarification to static code but also carries information that can be processed at run-time.

Unclear VM/application interdependencies are one reason that confound the understanding of VM behavior. This issue are more urgent when a HLL is used to implement the VM. We verified the power of constraints in solving this issue. We defined constraints to clarify code contexts and the transitions between them. Constraints play a significant role in our approach. We aim for a low-overhead approach that does not significantly affect performance but is able to clarify dependencies. Constraints fulfill our goal: we are able to leave most of the original implementation unmodified while we only need to add constraints on context and transitions, thus we can manage and carefully eliminate introduced overhead. Our approach turned out to be a low-overhead approach that brings both static and dynamic clarification of dependencies between VM/application and among components internal to the VM.

Bypassing portability issues of high-level low-level programming is another case addressed by this thesis. Portability pitfalls of high-level low-level programming limit the reusability of VM components and the portability of VMs written in a HLL.
Conclusion

This part of our work focuses on defining language restrictions to allow a low-level language bypass for improved portability. The definition of the restricted language relies on constraints and static compliance test: constraints deliver restrictions to language features and syntax within particular scopes, which not only ensures performance and correctness of the program, but also provides the possibility of enabling translation from a restricted language to a portable low-level language.

In combination, this thesis discusses two software engineering challenges in VM design using a HLL, and both demonstrate that using constraints and enforcing compliance is a good approach to tackling those flaws for the design and implementation of flexible and efficient VMs in high-level languages.

4.1 Future Work

Each main chapter described in the thesis has a future of possible extension.

4.1.1 Heap Policy on Context-Clear VM

In most JVMs written in C, there is no confusion between memory allocated for the application (Java objects created with new) and memory allocated by the VM itself (memory allocated by malloc(), for example). However, in Jikes RVM, a state-of-the-art metacircular VM, both VM and application allocate Java objects in the same way. Besides, due to lack of clear context, objects from both sides have to share the same heap. This is usually not desirable, and leads to ambiguous accounting and fewer optimization opportunities.

With our work to clarify context including dynamic context, it is possible to construct separate heaps for the application and VM. It is also possible to use several heaps, each serving one important context such as application, JIT, classloader, etc. As we reported in Figure 2.8 and Figure 2.9, objects from different contexts have different allocation volume and survival ratios. There are possibilities for optimizing specific heap policies and garbage collection strategies for each VM context.

4.1.2 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 3.1 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.
Conclusion


SIEKA, J. J2c project. http://code.google.com/a/eclipselabs.org/p/j2c/. (cited on page 42)


