A Framework for Efficient Method Specialisation in Java

A BSc joint mathematics and computing individual project report

Johan T Nyström-Persson

**Supervisor**
Andrew Cheadle

**Second marker**
Paul H J Kelly

Imperial College London
2006
Abstract

Object oriented languages are *virtual*, that is, they execute methods indirectly through method pointers. The extent to which code is virtualised depends on the language; a common paradigm is to devirtualise methods in order to gain performance. However, some languages, such as Haskell compiled with GHC, are fully virtual. It has been shown by Cheadle et al[2] that it is possible to exploit the virtualisation in GHC to construct a relatively cheap read barrier for an incremental garbage collector. This is done through the equivalent of method specialisation in Haskell.

The widespread programming language Java is only partly virtual, and indeed conventional wisdom for optimising it at run-time has been to devirtualise methods. We now ask: if Java were to be fully virtual, could the cost of the added virtualisation be traded for something beneficial that might potentially offset it, such incremental garbage collection similar to what was done with GHC?

This project makes several contributions. We explore some prior work in exploiting virtual dispatch and introduce Jikes RVM, an extremely popular platform for programming language research. We develop a general framework that allows method specialisation at low cost. This framework has three components: *transforms* that transform all classes loaded into the Java VM, *generic extensions* to the Jikes RVM, and *extensions to the inlining subsystem*. We evaluate this framework for the purposes of using it as a read barrier in an incremental garbage collector, though its use is by no means limited to garbage collection.
Acknowledgements

I would like to thank my supervisor Andrew Cheadle and Dr Tony Field for the plentiful and concise help, encouragement and guidance they have given me.

I also want to thank my friends in JMC for helping me to stay on track and for inspiration, particularly Gabriel Keeble, Hiromi Ozaki, Pierre Moschetta, Nuuman Chaudhry and Anders Aaltonen. For similar reasons I want to acknowledge Neil Dunn and John Ayres of MEng.

I would like to extend special thanks to Richard Hayden for providing the friendly competition that is such a strong motivational force.
Contents

Abstract i
Acknowledgements iii

1 Introduction 1
   1.1 Project objectives .................................. 3
   1.2 Report structure .................................. 4

2 Background 5
   2.1 The Jikes RVM ...................................... 5
      2.1.1 Jikes RVM structure ............................. 5
      2.1.2 The type information block (TIB) ............... 6
      2.1.3 Inlining in the optimising compiler .......... 8
   2.2 The BCEL Byte Code Engineering Library .......... 10
      2.2.1 JavaBean compliance .......................... 11
      2.2.2 The TIB specialisation patch .................. 11
      2.2.3 Class loading .................................. 11
   2.3 Garbage collection .................................. 12
      2.3.1 Some algorithms ............................... 14
      2.3.2 Barriers ...................................... 14
      2.3.3 Baker’s algorithm ............................. 14
   2.4 Assembling the framework .......................... 15

3 Byte code transformations 17
   3.1 Enforcing the JavaBean contract - BeanifierTransform .... 17
   3.2 Specialising methods - IncrGCMethodTransform ........ 19
   3.3 Managing inlining pragmas - NoInlineTransform ....... 22

4 General modifications to the VM 23
   4.1 Introducing method specialisations ................... 23
      4.1.1 Augmenting the TIB ............................ 23
      4.1.2 Handling method objects and references .......... 24
   4.2 Handling method recompilation ...................... 25
   4.3 Handling native methods (JNI) ....................... 26

5 Inlining 29
   5.1 A new inlining guard ............................... 29
   5.2 Allowing multiple inlining guards per target .......... 29
   5.3 Extending the inlining oracle ....................... 29
6 Evaluation ................................................. 33
   6.1 Methodology ........................................... 33
   6.2 Benchmarks ............................................ 33
       6.2.1 Baseline performance ............................... 33
       6.2.2 Performance of the enhanced VM ................... 33
       6.2.3 Cost of wrapping native methods .................. 34
       6.2.4 Cost of compilation .................................. 35
       6.2.5 Cost of the BCEL infrastructure ................... 36
       6.2.6 An explicit read barrier ............................. 36
   6.3 Performance overhead breakdown ....................... 37

7 Conclusion and future work ................................ 39
   7.1 Future work ............................................. 40
Chapter 1

Introduction

A defining feature of object oriented programming languages is polymorphism, which relies on methods being virtual. Some languages are more virtual than others; smalltalk is *purely virtual*, and no object can be accessed directly. Java, by contrast, is not since it allows direct field access. Virtual methods introduce overhead since method calls are indirect. One common strategy for managing this overhead is to let the compiler or runtime aggressively devirtualise methods whenever possible.

Virtual methods allow for polymorphism at the language level, but the virtualisation can also be exploited implicitly from a lower level, such as the language runtime. Once the price of full virtualisation has been paid, it can be used for other purposes than function calls. In the past various extensions, such as orthogonal persistence, fickle objects, and even a distributed JVM which migrates objects across machines, have been implemented in this way. One particular use of virtualisation is to provide a read barrier for incremental garbage collectors. This has been done successfully for languages such as Haskell, which is purely virtual.

A question that might be asked at this point is the following. Suppose that one were to take a programming language that is not purely virtual, but partly virtual, and make it purely virtual somehow. How much performance would be lost to the extra virtualisation, and is it possible to exchange this overhead for some other benefit? Might this benefit perhaps be so valuable that the whole venture becomes profitable?

This is no humble question. It challenges one of the basic design principles behind enormously successful languages such as Java. This project will ask that question. In particular, we will investigate ways of exploiting Java’s dynamic dispatch while retaining as much performance as possible. Although this work has been carried out with garbage collection in mind, it is in no way tied to this particular application. The approach described may be suitable for other uses.

Research on the implementation of object oriented programming languages and in particular on the Java programming language has been abundant in recent years. One of the most important research environments is the *Jikes RVM*, developed and maintained by IBM, and fully open source. While being a full-ledged Java VM with performance comparable to many production VMs, the Jikes RVM provides a convenient “cleanroom” environment in which to do research, since it is written in Java and relatively easy to modify and extend. In providing the foundation for a read barrier, we will be specialising methods, that is, for specialised methods, one specialisation might contain the read barrier action while the other would not. In particular, we will specialise the virtual method tables. In fact, we will go further and specialise the entire type information block\(^1\) in the Jikes RVM since this is more

---

\(^1\)The type information block, or TIB, in the Jikes RVM uniquely defines a type, such as a class,
The Jakarta BCEL byte code engineering library is a library for creating and manipulating Java byte code. We will use this to achieve full virtualisation by transforming classes as they are loaded into the VM.

The simplest compilation cycle in the RVM is illustrated in Figure 1.1. Classes are loaded on-demand and are compiled by a baseline compiler that translates the byte code for the class into native code. If the adaptive subsystem is being used, runtime information is gathered during program execution. In this case, subsequent to baseline compilation, the optimising compiler may recompile dominant “hot” methods to improve performance. The adaptive subsystem is a requirement to get any sort of decent performance as the baseline compiler does no optimisation whatsoever.

Code to implement method specialisation has been developed prior to this project by Andrew Cheadle. This uses the BCEL patch (see section 2.2) to rewrite class byte code as it is loaded, introducing method variants as specified by a user-supplied transformation class (see Section 2.2.2). This is shown in Figure 1.2. One transformation that has already been defined implements beanification, as described in section 3.1. This is the key to the specialisation, as it ensures that all object accesses jump through the TIB associated with the object’s class.

With the optimisation phase of the RVM turned off the code generated by this transformation behaves identically to the original code, but runs much slower, because of the additional method calls that ‘wrap’ each object field access. However, turning the optimiser on immediately introduces two problems:

1. Inlining has been turned off, because otherwise the code added to implement and contains its virtual method tables, among other things.

![Diagram of method compilation processes](image)
1.1 Project objectives

This project aims to

- give an account of prior work in this field, including some incremental garbage collectors, inlining strategies and extensions to the Jikes RVM.
investigate the possibility of providing a framework for method specialisation in the Jikes RVM, in which arbitrary actions can optionally be taken before each object is accessed, by exploiting dynamic dispatch. This will be done specifically with a garbage collector in mind.

- minimize the performance impact of the above by developing a suitable inlining protection mechanism and inlining heuristic.

- evaluate the results through the use of suitable benchmarks.

The framework that will be provided will have three major components:

- Byte code transformations to enforce full virtualisation and create specialised methods

- Various modifications to the Jikes RVM to accommodate method specialisations

- A new inlining guard and an updated inlining heuristic

1.2 Report structure

The rest of this project is structured as follows.

- Chapter 2 gives an overview of the software and prior research that forms the foundation of this project.

- Chapter 3 details the byte code transformations being used.

- Chapter 4 describes general modifications to the Jikes RVM.

- Chapter 5 describes the new inlining guard and the extended inlining oracle.

- Chapter 6 presents a detailed evaluation of the implementation.

- Chapter 7 presents the conclusions drawn from this project, and suggests possible future directions.
Chapter 2

Background

This chapter will introduce software and prior work that this project builds on.

2.1 The Jikes RVM

The Jikes Research VM (RVM), originally the Jalapeno Virtual machine, is a Java Virtual Machine developed by IBM. Although it is written almost entirely in Java, its performance is comparable to IBM’s product VM. The Jikes RVM was developed specifically for the purposes of research, as its name suggests, and it is completely open source. Since development of the Jikes RVM started in 2001, a great deal of research based on it has been done and resulted in numerous publications. It is a de facto cleanroom environment for research in programming language implementation, garbage collectors, memory management and many other areas. The history of Jikes RVM is outlined in detail in [3].

2.1.1 Jikes RVM structure

Since the Jikes RVM is the primary research environment of this project we will here give an overview of its components and how they interact. We will use the terms “Jikes RVM”, “the RVM”, and “the VM” interchangeably. We will use the term “the vanilla VM” to mean the unmodified CVS version of Jikes RVM from the 16th of January 2006, which this project is founded on. We will routinely contrast our enhanced VM and the vanilla VM to highlight functionality or assess performance.

- **Core runtime**
  The core runtime contains fundamental components such as thread scheduling, class loading, the type system, support for booting and loading libraries, etc.

- **Compilers**
  The Jikes RVM runs Java byte code in a fully compiled fashion, meaning all byte code is compiled to machine code before being executed. Since the VM itself is written largely in Java, one important side effect is that application code can be inlined into VM code before being compiled, or vice versa. Several different compilers exist. The fastest, simplest compiler is the baseline compiler which only applies the most basic optimisations to quickly produce runnable code. The most sophisticated (and complex) compiler is the optimising compiler, which applies a full range of aggressive optimisations. In the fastest configurations of the Jikes RVM, the VM itself is compiled using the optimising compiler, which first is compiled using some external Java compiler. We will look at specific components of the optimising compiler in
greater detail later on. For now, some of the most important differences are that

- The baseline compiler has no intermediate representation (IR) whatsoever and performs no inlining or any other optimisations, so its code runs very slowly.
- The baseline compiler never recompiles a method. Once a method is compiled it stays in place. The optimising compiler and the adaptive system, on the other hand, are based on the idea of recompiling “hot paths”, methods where the greatest portion of CPU time is spent. This means that any compiled methods could potentially be invalidated and replaced.

- **MMTk**
  The MMTk, or *Memory Management Toolkit*, contains memory management routines and several different garbage collectors. Although this project is done with garbage collection in mind as a potential application, we will not have reason to delve into this part of the RVM any further.

- **Adaptive Optimising System (AOS)**
  The Jikes RVM does not have a static optimisation strategy. Instead it adapts its optimisation techniques to the application being executed. It periodically samples itself at run time to find out which methods the application spends the most time in, since these are the ones where applying further optimisation would yield the greatest returns. It then recompiles these methods with the optimising compiler and substituted the newly compiled code for the old baseline compiled code. In addition, the optimising compiler changes its behaviour depending on call graph weights which are updated by the sampling described above, so repeated invocation of the optimising compiler on the same method may result in more accurate optimisations. As an example of this, the optimising compiler’s inlining strategy is directly affected by call graph weights. We shall examine this in greater detail in section 2.1.3.

A more detailed breakdown of the Jikes RVM structure is available in [4].

### 2.1.2 The type information block (TIB)

The type information block, TIB, of an object contains information about the object’s type and also its virtual method table. It is similar in function to the info pointers mentioned in the discussion of Haskell. Since this project is concerned primarily with efficient support for specialisation of methods, we will be interested in keeping multiple versions of TIBs and substituting them for each other at runtime. Each TIB is an Object array. Each object has a two word header, where the first word is a pointer to the TIB and the second is a status word, where locks and other support systems reside. The TIB is used for method invocation and also for dynamic type checking. Sometimes a simple comparison of TIB pointers is sufficient to determine the type of an object.

Figure 2.1 shows the TIB layout in the vanilla Jikes RVM. A description of the function of each field follows. As we will see, there are three kinds of types: classes, arrays and primitives. Not all types use all available fields in the TIB; classes use the most, and primitives use the least.

- The *type* field points to an instance of VM\_Type, which is the VM’s internal representation of a type. Subclasses of VM\_Type include VM\_Class, VM\_Array and VM\_Primitive. We shall discuss these further in section 2.2.3.
2.1. THE JIKES RVM

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Type</td>
</tr>
<tr>
<td>1</td>
<td>Superclass ids</td>
</tr>
<tr>
<td>2</td>
<td>Implements trits</td>
</tr>
<tr>
<td>3</td>
<td>Array element TIB</td>
</tr>
<tr>
<td>4</td>
<td>iTABLES</td>
</tr>
<tr>
<td></td>
<td>Indirect IMT</td>
</tr>
<tr>
<td></td>
<td>Interface slot i</td>
</tr>
<tr>
<td></td>
<td>Virtual method i</td>
</tr>
</tbody>
</table>

Figure 2.1: The layout of the TIB prior to modifications.

- The *superclass ids* field is an array of type `short` which uniquely identifies all superclasses of this class. Unused for primitives.

- The *implements trits* contains an array of type `int` where each integer is a bitmap. These bits indicate which interfaces are implemented by this type. Unused for primitives.

- Unsurprisingly, the *array element TIB* field points to the TIB of the array elements, if this is an array. Otherwise unused.

- *iTABLES* and *indirect IMT* are two mutually exclusive ways of invoking interface methods; which one is to be used is decided by a compile time parameter. Only one of these two fields can be present. Unused for primitives.

In addition to these fields, there is also a fixed number of interface method slots and an arbitrarily large number of virtual method slots. Invocation of any virtual method goes through these, and it is because of this part of the TIB that we are interested in specialising it.

Let us look at an example. Consider the class *UltraList* which extends `java.util.LinkedList`. This means that its superclasses are `java.util.LinkedList`, `java.util.AbstractSequentialList`, `java.util.AbstractList`, `java.util.AbstractCollection` and `java.lang.Object`. Its implemented interfaces are `java.lang.Cloneable`, `java.util.Collection`, `java.util.List` and `java.io.Serializable`. *Type* will of course point to a VM `Class` object that is unique to *UltraList*. *Superclass ids* will be an array of length 5, containing unique identifiers for all the superclasses. *Implements trits* will be an array of bitmaps that indicates all the interfaces. One of the two interface method invocation mechanisms will also be present.

Each virtual method slot will contain a pointer to entry code for a method in *UltraList*. This includes all methods inherited from superclasses. The index in this table of a method is the same in all classes in a virtual hierarchy, to simplify

---

1 Assuming J2SE 1.4.2.
invocation. Note that this table does not include static or private methods, nor
does it include constructors or class initialisers.

2.1.3 Inlining in the optimising compiler

Inlining is an extremely valuable optimisation, since it allows code motion between
methods, makes more information available for data flow analysis, and so on. In
the Jikes RVM it is done in the optimising compiler only. We will outline some
problems associated with it since it will be a very important performance factor.
We will first introduce some terminology:

- A caller is the method whose body contains a call instruction
- A callee is the target method of a call instruction
- A receiver is an object being called
- A guard is a condition that must be true for an inlining to be valid
- A call site is the context of a call instruction in a caller

When method specialisations exist, and can be substituted for each other arbi-
trarily (for instance when triggered by garbage collection), we can no longer inline
freely and preserve correctness. This is a fundamental problem which this project
attempts to address. Consider two methods, $M_a$ and $M_b$, and suppose that $M_b$
has a specialisation $M_b'$. If $M_b$ is inlined into $M_a$, and the TIB is flipped so that $M_b'$
replaces $M_b$, the version of $M_b$ that is inlined into $M_a$ has been invalidated. We
would expect a call to $M_a$ to somehow result in a call to $M_b'$. A means of testing if
an inlined method is up to date becomes necessary.

Fortunately a framework that supports speculative inlining is already present in
the RVM. This is a necessity because of polymorphism. Commonly, the VM will
try to inline a method call, while at the call site the actual type of the receiver
of the call need not be known. In such cases several candidates for inlining will
exist. The VM is forced to use the call graph weights mentioned above, obtained
by periodic sampling, to determine which is the most likely callee. This will then
be inlined and augmented with inlining guards. Each time the inlined method is
reached, the guard will evaluate some condition to test the validity of the inlined
code. This condition is usually some form of type check on the receiver of the call.
If the condition fails, the method is resolved and called normally. The guards al-
ready implemented in the RVM are $ig\_class\_test$, $ig\_method\_test$ and $ig\_patch\_point$.
They function as follows. $ig\_class\_test$ simply compares the TIB of the receiver
against the TIB encoded in the guard, testing the type of the receiver. This is rel-
atively inexpensive. By contrast, $ig\_method\_test$ examines the candidate’s method
pointer against the method pointer encoded in the guard. This is more expensive
since it involves looking at subfields of the TIB, but works in more cases. Finally,$ig\_patch\_point$ emits assembly code that can be overwritten at runtime to update
the guard condition without recompiling the method. These guards are selected by
the inlining oracle, which contains logic for making inlining decisions and protecting
them correctly. The default inlining heuristic has been retained in a modified form
in this project. We present a simplified pseudo code view of the default heuristic
in Figure 2.2. Some details have been left out in order to make the main decision
process clearer.

In addition to the pseudocode presented, other aspects are considered, such as
Inlining pragmas which have the obvious effects - see section 3.3.

We will now give an explanation of some concepts used in Figure 2.2. A trivial
Guardless inline is an inlining action where there is only one possible target (if the
procedure shouldInline(...)

1: if callee is small and has a precise target then
2:   return decision('Do a trivial guardless inlining')
3: end if
4: if callee is 'huge' then
5:   return decision('Do not inline')
6: end if
7: Find all possible targets by traversing class hierarchy
8: for each possible target $t$ do
9:   Evaluate cost of inlining $t$
10:  if inlining recursively then
11:    return decision('Do not inline')
12: end if
13:  if $t$ is a forbidden speculation and $t$ needs a guard then
14:    return decision('Do not inline')
15: end if
16:  if $t$ was previously inlined then
17:    Decide yes
18:  end if
19:  if not decided yes then
20:    Decide based on call site weight and weight of $t$
21: end if
22: end for
23: Create inlining decision $d$
24: for each possible target $t$ do
25:  if decided yes for $t$ then
26:    Decide the proper guard $g$ if any
27:    Add ($t$, $g$, 'yes') to $d$
28:  end if
29: end for
30: return $d$

Figure 2.2: The default inlining oracle. This can be found in OPT_DefaultInlineOracle.java.
method is final or static, for instance), so that no guards are required, which is smaller than a certain predefined threshold. Such method is always inlined into its caller. Likewise, a huge method is of course a method whose body is larger than another threshold. Such a method is never inlined into. A forbidden speculation is a method which cannot be inlined speculatively into its caller, because the caller cannot be recompiled if the callee gets invalidated at some point. This is currently defined as inlining actions where the caller is in the boot image and the callee is not a part of the RVM.\footnote{Methods in the boot image can’t be recompiled, but if the callee is a part of the RVM it will never be overridden. The Jikes RVM developers admit that this definition is overly conservative but have not been able to find a better one.}

The adaptive optimisation system periodically samples the RVM to determine which methods occupy the most CPU time. To each method a weight is attached. The weight of the call site is considered to calculate a maximum allowable inlining cost; if a call site accounts for a large proportion of CPU time, more expensive inlining actions are allowed. The weight of the callee is also considered as a fraction of the total weight of all the candidates for inlining. An algorithm is then applied to determine whether or not the inlining action will be profitable. Precise details are available in OPT\_DefaultInlineOracle.java.

The inlining oracle is thus trying to find the most optimal decisions in a very large decision space, a nontrivial task. Eager inlining could mean that a long running program will finish sooner but a small program may finish much later. Inlining also means that more methods potentially have to be recompiled: if an inlined callee is overridden, as a result of a new class being loaded perhaps, we have no choice but to recompile all of the caller. The oracle is a delicate balance between several numerical parameters:

- The maximum size for trivial inlining
- The “huge” method size which prevents further inlining
- The maximum inline depth, after which maximum allowed inlining costs go down
- The maximum callee size to inline adaptively
- The minimum call site fraction to consider for inlining
- Several others that affect the cost calculation algorithm

We will of course want to retain as much of the potential performance gains from inlining as possible in our implementation.

### 2.2 The BCEL Byte Code Engineering Library

The Jakarta BCEL is a library developed by the Apache Software Foundation. “The Byte Code Engineering Library is intended to give users a convenient possibility to analyze, create, and manipulate (binary) Java class files” \cite{BCEL}. This project will use it to transform byte code loaded into the VM. In particular, we will be using a transformation framework developed by Zigman and Sankaranarayana as part of the distributed virtual machine dJVM \cite{dJVM}. This framework allows arbitrary transformations to modify classes as they are loaded through the BCEL. We will now look at a particular transformation that is employed in the dJVM and also in a framework for orthogonal persistence. This transformation will prove crucial to this project.
2.2.1 JavaBean compliance

A class is JavaBean compliant if it has no non-private fields. This virtualises all access to instances of that class. Accessing non-private fields is the only way of accessing an object in the JVM without going through the virtual method table in the TIB, so removing this capability ensures that all object access goes through the TIB. In practice, this means ensuring that field accesses are replaced by calls to methods that get or set the field, “getters” and “setters”. This is easily achieved using the BCEL transformation framework mentioned above.

This technique is used in the dJVM to allow local proxy objects to represent objects that reside on remote machines in a cluster. When an object is local, the field access methods access the field directly, and when it is remote, these methods instead route the request to the machine that the object resides on.

Another interesting use of this technique is by Marquez, Zigman and Blackburn to provide orthogonal persistency [7]. In orthogonal persistency, objects are transparently preserved in long term storage, regardless of their type. The authors achieve this by using proxy objects in a similar fashion to the dJVM. The first time an object is accessed, it is faulted in from storage, and the facade that does the faulting in is then overwritten by the actual object. This approach has a complication: when an object is faulted in, references to it, which previously pointed to the facade, must be overwritten with references to the faulted in object. This makes it necessary for the facade to maintain lists of everything that has a reference to it. Maintaining this list introduces an overhead. We will never need to replace references in this way in this project, so this overhead is not a concern.

The JavaBean compliance transformation is essential to this project because we wish to trap each attempt to access an object so that we can scavenge it if necessary, or take some other action. After the transformation there will be no way to access objects except by going through the TIB’s virtual method table. This allows us to substitute specialised self-scavenging methods for the normal ones when, and only when, an object needs to be scavenged. If it has already been scavenged the TIB points to the normal methods and no cost is incurred.

2.2.2 The TIB specialisation patch

Andrew Cheadle has made an experimental TIB specialisation patch which does for the baseline compiler some of what this project aims to do for the entire VM: it specialises methods and creates TIB specialisations which can be substituted for each other on a per object basis through a method call. This is done through two BCEL transformations. However, the baseline compiler does not recompile methods, and it does no inlining or optimisation whatsoever. In order to get decent performance with Jikes RVM, it is necessary to run an adaptive configuration (see 2.1.1) where both the baseline and the optimising compiler are used. Another limitation of this patch is that it cannot specialise native methods, which is a serious flaw, since native methods can access fields and methods of Java objects just as easily as non-native methods.

2.2.3 Class loading

The BCEL patch modifies the class loading procedure in the RVM so that transformations are applied to any classes as they are loaded. We give a brief account of the basic components of the class loading system here, followed by a comparison of class loading before and after the BCEL patch. Class loading revolves around the function VMClassLoader.defineClassInternal(), which takes a raw data input stream corresponding to compiled byte code and returns an object of type VMClass.
As part of the class loading process, instances of the classes `VM_Class`, `VM_Array` and `VM_Primitive` are created to represent classes, arrays and primitive types inside the VM. These are all subclasses of the class `VM_Type`, and correspond to runtime types. There's also a parallel hierarchy consisting of the classes `VM_MethodReference` and `VM_FieldReference`, which are subclasses of `VM_MemberReference`. These correspond to member references in the class files.

In order to allow the VM to start executing a program reasonably quickly, class loading is fundamentally a lazy process. When a class is loaded, minimal information is gathered, and the resulting `VM_Type` object is left in a defunct state: superclasses are not yet resolved, methods and fields are not laid out or resolved and the TIB is not created or set up. Not until the first time code tries access or create an instance of a particular class is the class is resolved and made fully functional.

We will now describe each step. The class loading processes are depicted in Figure 2.3.

1. `VM_ClassLoader.defineClassInternal()` is called.

2. In the vanilla VM, a `VM_TypeReference` and a `VM_Type` are created. Fields, Methods, superclasses etc. are not yet examined, but their names are read from the class file. The `VM_Type` object is returned, and nothing more is done until its resolve() method is called.

   In the patched VM, significantly more work is done. A ClassParser, which is a component of BCEL, is created. It fully parses and deconstructs the class file, creating a corresponding JavaClass. JavaClass objects are representations of classes inside BCEL, as opposed to `VM_Type` objects which are representations in the VM. A new `VM_Type` is then created, and a reference to the JavaClass is stored in it for future use. Again, the `VM_Type` is not populated but returned and nothing more is done until resolve() is called.

3. When resolve() is called the class loading process continues. In the vanilla VM, superclasses are resolved, fields and methods are laid out, and a TIB is created and filled in. This completes class loading.

   In the patched VM, the process is again slightly more involved. First, all transformations that have been defined in the `VM_ClassLoader` are applied to the JavaClass previously created. With the TIB specialisation patch, these include a method specialising patch that may require multiple TIBs and specialised methods to be created. When the transformations have been applied, the `VM_Type` is updated, changing its field and method definitions to correspond to the transformed JavaClass. Methods and fields are then laid out. If specialised methods exist, these are also taken into account. Finally, a TIB, or several TIBs if needed, is created and filled in. This completes class loading.

### 2.3 Garbage collection

Having looked at Jikes RVM and BCEL, we will now turn to garbage collection, from which an important inspiration for this project comes.

Most computer programs cannot know at compile time how much memory they will use at run time. For this reason dynamic memory management, in which programs acquire and release memory while running, is used. Dynamic memory management techniques come in two varieties: explicit memory management and automatic memory management. In the former, the programmer simply requests the allocation or freeing of memory, using system calls like `malloc()` and `free()`. In automatic memory management, one popular technique is garbage collection, in
Figure 2.3: The two class loading processes. To the left, class loading as it is done in the vanilla Jikes RVM. To the right, class loading in Jikes RVM with the BCEL patch and the TIB specialisation patch applied. Without the TIB specialisation patch, the process is nearly identical, except only one TIB is created and specialised methods are not laid out.
which some library, interpreter or language runtime environment automatically finds unused objects and releases the corresponding memory. The programmer must still request memory for new objects but need not worry about releasing memory when they are no longer used. This removes a common cause of performance problems known as memory leaks, in which running programs slowly exhaust all available memory, and simplifies program development. Dangling pointer errors, in which deallocated memory is incorrectly accessed, are also prevented. One downside is that garbage collectors introduce performance overhead.

Since being introduced in the 1950’s, garbage collectors have gained increasing popularity and are now part of the specification of many widely used programming languages, including C# and Java.

2.3.1 Some algorithms

Various common approaches to garbage collection exist. Some key problems that have to be solved are the detection of garbage, whether to move objects and how to move them, whether to partition the objects, based on age or some other characteristic, and how to run the garbage collector concurrently with program execution itself (called the mutator).

Detecting objects to be garbage collected usually involves traversing a graph of object references. This graph begins in a root set and is built by recursively following each reference to another object. Many garbage collectors are generational, that is, they partition objects into generations based on their age. This comes out of the observation that the most recently created objects are highly likely to be garbage collected. This approach is common in many Java virtual machines, for instance Sun’s hotspot VM and Jikes RVM, which this project concerns. Finally, the execution of the garbage collector can have a large impact on performance and responsiveness, particularly in real time systems. At one end of the scale, we find the stop-the-world approach, in which the mutator is suspended to allow the garbage collector to run, and then restarted. The time during which the mutator is suspended is called pause time. Stop-the-world is simple to implement and requires no locking of objects, but has a drastic impact on response time. The other extreme is running the garbage collector concurrently with the mutator, perhaps on another CPU. This makes for short response times but requires complex synchronisation. In between we find incremental garbage collectors which interleave garbage collection with the mutator, reducing the pause time, but increasing the overall time spent in garbage collection compared with stop-the-world collectors.

2.3.2 Barriers

In incremental garbage collector implementations, at any point in time the system will be in a state where some objects have been garbage collected and others await collection. References pointing to the latter are invalid and must not be accessed by the mutator. To ensure this, references are checked for validity before being read or written whenever their state is ambiguous. These checks are usually called the read barrier and the write barrier for reading and writing, respectively.

2.3.3 Baker’s algorithm

In Baker’s algorithm for incremental garbage collection, the heap is partitioned into two spaces: to-space and from-space. New objects are always allocated in to-space, and when the to-space is filled, a flip takes place and the two spaces are swapped. Objects that are in the root set are copied into to-space (evacuation), and objects are then scavenged incrementally. Scavenging an object is done by examining all
2.4. ASSEMBLING THE FRAMEWORK

the references to other objects and evacuating them if necessary. Of course, before a new flip takes place, it is necessary to ensure that all objects have been scavenged. A collector queue keeps track of objects in need of scavenging to ensure that this is eventually done.

Cheadle et al have produced a low cost implementation of Baker’s algorithm for Haskell [2], implemented in the Glasgow Haskell Compiler, GHC. Their approach exploits dynamic dispatch. In virtually every object oriented programming language, polymorphism is provided through function pointers, a mechanism by which method calls are made indirect. Indirect function calls introduce a small overhead but allow functions to be substituted very easily. The authors use this to create self-scavenging code in Haskell. This is achieved by specialising closures, the unit of code and related data in Haskell, so that entry code exists in two varieties. This is illustrated in figure 2.4. One info pointer simply enters the closure, executing code normally. The other scavenges the closure before restoring the closure’s info pointer, which contains the method table, and then executes the code normally. Whenever a flip occurs, every closure in from-space is changed to execute self-scavenging code. Because the info pointer is restored as soon as a closure has scavenged itself, subsequent attempts to enter the closure will not trigger any scavenging.

This technique yields garbage collection at a very low cost since the amount of copying that is potentially done on entering a closure is bounded. Pause times become minimal, making this an attractive implementation of Baker’s algorithm. This project concerns itself with evaluating the possibility of exploiting dynamic dispatch in a similar way in a production level Java VM. As we shall see, this requires overcoming certain challenges that are not present in GHC.

2.4 Assembling the framework

We have looked at the Jikes RVM, how its classloading and inlining works and the BCEL byte code engineering library. We have also looked at the TIB specialisation patch and why it is not in its current form sufficient to meet the project goals. Thus, the key challenges to be solved are

- How can we specialise methods and retain as much performance as possible, particularly through inlining?

- Is it possible to extend the TIB specialisation patch to apply to the optimising compiler? This necessitates proper recompilation of specialised methods, and correct subsequent updates of the TIBs.
• Can we specialise native methods?

To address these problems, we will implement a three component system:

1. Byte code transformations to allow complete virtualisation, protection of native methods and method specialisation. Two of these will be based on transformations from the TIB specialisation patch.

2. Modifications to the Jikes RVM to allow method specialisations to work with the optimising compiler.

3. A new inlining guard and extensions to the default inlining oracle, in order to retain as much of the default inlining heuristic as possible.

The following three chapters will proceed to discuss the implementation of this.
Chapter 3

Byte code transformations

This chapter and the two following ones detail the implementation of this project and how it builds on the concepts that have already been introduced. This project starts out from a CVS snapshot of the Jikes RVM from the 16th of January 2006 to which the abovementioned BCEL patch from the dJVM project has been applied. The TIB specialisation patch has also been applied.

Three byte code transformations are applied to all code loaded into the VM. The packages com.ibm.JikesRVM and org.mmtk are exempt from this for performance reasons. Given that VM objects live in immortal space, where garbage collection never occurs, and that this project is done with garbage collection in mind, the functionality provided by the transformations is not required for these packages. It would be a trivial task however to modify the transformations to apply to all code.

The transformations are:

- **BeanifierTransform** - to enforce the JavaBean contract
- **IncrGCmethodTransform** - to create the method specialisations that would self-scavenge objects
- **NoInlineTransform** - to manage inline pragmas

We will now look at these in detail.

### 3.1 Enforcing the JavaBean contract - Beanifier-Transform

In order to trap each access to a new object it is necessary to ensure that object access is through virtual methods; hence a transformation that ensures JavaBean compliance is applied. This transformation is part of the TIB specialisation patch but extended by this project.

Figure 3.1 shows a pseudocode illustration of the idea behind the main loop of the BeanifierTransform. First, access methods to get and set each field are created. Then, the byte codes of each method are examined. Direct field accesses, which occurs as the `GETFIELD` and `SETFIELD` byte codes, are replaced with calls to the newly created access methods. This feature is part of the unmodified BeanifierTransform.

Java classes commonly already have pure access methods since good coding style dictates that they be used. However, we cannot detect and reuse these, since when transforming a class X we have no way of knowing the names of access methods in another class Y. So we cannot reuse existing access methods. This means that they will amount to pure overhead when our mechanically generated access methods
procedure transform(Class C)

1: for each field f in C do
2:   Create access method to get f
3:   Create access method to set f
4: end for
5: for each method m in C do
6:   if m is an existing access method then
7:     Mark m as always inline
8:   end if
9:   for each byte code b in m do
10:  if b is GETFIELD then
11:     Replace b by a call to the relevant access method
12: end if
13: if b is PUTFIELD then
14:     Replace b by a call to the relevant access method
15: end if
16: end for
17: end for

Figure 3.1: The main loop of the BeanifierTransform

are added inside them. To address this problem, we test every method to see if it is a pure access method. If a previously existing method is found to be such a method, it is marked with a special pragma to ensure that it is always inlined. This ensures that it is not specialised by later transformations, and ensures that it will be factored out completely when it is compiled with the optimising compiler. The ability to detect existing pure access methods is a contribution of this project.

The access methods that we generate have fully formulaic names. For instance, the method getting the value of the field apple of type long in the class com.jungle.Monkey is called gapplecom.jungle.Monkey.J. g indicates getting, as opposed to s for setting, and J is the Java type signature for variables of type long. This naming scheme is necessary to prevent type clashes, as we will see.

The nature of Java’s type system gives rise to some complications here. When a class is accessed through a virtual method, naturally the actual type of an object is used to determine which method is called. However, when fields are accessed, the declared type of the object, in the context of the access, is used to determine which field is accessed. Thus, if we have two classes Car and SubCar, where SubCar is a subclass of Car, and both classes declare a field mileage of type float, two fields exist in SubCar: one that is declared by Car, and one that is declared by SubCar. Which one is accessed is determined by the declared type of a variable. When one class extends another and both declare a given field name, this is termed field hiding. For example, consider this code snippet:

SubCar sc = new SubCar();
sc.mileage = 4.5;
Car c = sc;
c.mileage = 5.7;

Because field access is determined by the declared type, the first access to mileage sets the field declared by sc. The second access sets the field declared by c. If we now apply our transformation and name the generated access methods getMileage and setMileage, as is customary, dynamic type resolution will ensure that the field declared by sc is written to twice. This is unacceptable since it changes program
public class Car {
    float mileage = 7.8;
    native boolean start(short gear);
    public void drive() {
        start(2);
        while (mileage < 500) {
            mileage += 2.1;
        }
    }
}

Figure 3.2: The Car class

behaviour; any changes imposed by the transformations must of course be transparent to the program. Including the full name of the declaring type in the names of the access methods resolves this problem in most cases: the methods called _g_mileage_SubcarF and _g_mileage_CarF will not be in conflict. A similar naming scheme exists in the BeanifierTransform from the TIB specialisation patch.

There is, however, one case that is not covered by this approach. Suppose that Car has a method maximumMileage that sets mileage to 9999.9, and that this method is not overridden by any subclass. If this method is invoked on an object of type Car, it will access the field declared by Car, but if it is invoked on an object of type SubCar, it will access the field declared by SubCar. The transformation described above cannot take this into account. The relevant PUTFIELD instruction will be replaced by a method whose name is of the form _s_mileage_Car since Car is the class that declares the method. The method is then restricted to access only the field declared by Car. Out of the 8 benchmarks in SPECjvm98, one (jack) cannot be run correctly because of this.

The BeanifierTransform is thus fundamental to this project, but not problem free. The main contribution made to it by this project is the ability to detect and factor out existing pure access methods. Some extra control logic has also been added.

For this and other transformations we will illustrate precisely what the transformation does to a small example class, Car, given in figure 3.1.

After transformation, the class will be as in figure 3.1.

3.2 Specialising methods - IncrGCmethodTransform

A second very important transformation is the IncrGCmethodTransform. This transformation creates the method specialisations that will be present in the different TIBs. We will give most methods a specialisation with a very short method body that 1) restores the normal TIB for the object, and 2) invokes the original method. Thus, it would be trivial to add a call to a self-scavenging routine here before invoking the original method, which is what a garbage collector implementation would do. Again, this transformation is part of the TIB specialisation patch but extended by this project.

We want to keep the number of methods we specialise to a minimum, to avoid code bloat and reduce the overhead induced by the transformations. At the same
public class Car {
    private float mileage = 7.8;
    native boolean start(short gear);
    float _g_mileage_Car_F() {
        return mileage;
    }
    void _s_mileage_Car_F(float field1) {
        mileage = field1;
    }
    public void drive() {
        start(2);
        while (_g_mileage_Car_F() < 500) {
            _s_mileage_Car_F(_g_mileage_Car_F() + 2.1);
        }
    }
}

Figure 3.3: The Car class with the Beanifier transform applied. Note that the field has become private - this is an optional feature, though in practice redundant, since any field access in any class will be hijacked and replaced by a method call anyway. We can see clearly how pure virtualisation introduces overheads.

time, we want to make sure that we trap each access to a new object. Thus, we do not specialise the following:

- Static methods including constructors
- Private methods, which do not access new objects
- Abstract methods (nonsensical)
- Methods that have been marked as being pure access methods by the BeanifierTransform

We must also deal with native methods. We cannot introduce a non-native method as a specialisation of a native one, but we have to have specialisations since native methods can access fields etc. just as easily as non-native methods. We solve this by wrapping native methods in non-native ones and then specialising the wrappers. This is somewhat unfortunate since it introduces some overhead, but is absolutely necessary.

The implementation of this is relatively straightforward. The native functions are renamed by adding a prefix, allowing their wrappers to retain the original name. This renaming is then reversed when the VM looks up native symbols in libraries. The main contribution made to the transform by this project is thus the ability to wrap and thus specialise native methods.

We present a pseudocode representation of the main loop of the IncrGCmethodTransform in Figure 3.4.

The specialised method body is always of the same form: it first calls VM_ObjectModel.restoreTIB() and then invokes the corresponding non-specialised method. Compare this with the approach taken in GHC (see Figure 2.4). To illustrate this, the Car class after the IncrGCMethodTransform (but without any other transformation applied) is given in Figure 3.2.
procedure transform(Class C)

1: for each method m in C do
2:  if m is exempt from specialisation then
3:    Proceed to next method
4:  end if
5:  if m is native then
6:    Create a non-native wrapper
7:  end if
8:  Construct the specialised method body
9: end for

Figure 3.4: The main loop of the IncrGCmethodTransform

public class Car {
  float mileage = 7.8;
  native boolean _nat_r_start[0](short gear);
  boolean start[0](short gear) {
    return _nat_r_start(gear);
  }
  public void drive[0]() {
    start(2);
    while (mileage < 500) {
      mileage += 2.1;
    }
  }

  // specialised methods
  boolean start[1](short gear) {
    // scavenging method would have been called here
    VM_ObjectModel.restoreTIB(this);
    return start(gear);
  }
  // will invoke start[0]

  public void drive[1]() {
    // scavenging method would have been called here
    VM_ObjectModel.restoreTIB(this);
    drive(); // will invoke drive[0]
  }
}

Figure 3.5: The Car class after the IncrGCMethodTransform. We have taken liberties with the syntax to indicate method specialisations. The specialised methods ([1]) are always of the same form: they restore the TIB and then invoke the method again, which will result in a call to the corresponding [0]-method since the TIB now is restored. Also note how the native method has been renamed and wrapped in a non-native one, allowing it to be specialised. This is necessary since native and non-native methods are not interchangeable.
3.3 Managing inlining pragmas - NoInlineTransform

The NoInlineTransform serves to edit *pragmas*. It is the most straightforward of the three transforms. Pragmas are special exceptions that serve to inform the VM about special properties of methods. For instance, if a method throws the exception `org.vmmagic.pragma.NoInlinePragma`, the VM will refuse to inline this method. We will be concerned with the following pragmas:

- *NoInlinePragma* - indicates that a particular method should never be inlined
- *InlinePragma* - indicates that a particular method should always be inlined
- *ReallyInlinePragma* - inline pragma introduced by this project
- *UninterruptiblePragma* - indicates that thread switch yieldpoints, etc., should not be emitted inside a method
Chapter 4

General modifications to the VM

4.1 Introducing method specialisations

This section will describe the modifications to Jikes RVM that were necessary to retain full functionality with the optimising compiler and adaptive system even as methods are specialised.

4.1.1 Augmenting the TIB

In order to accommodate multiple TIBs, we need to add more fields to the TIB. Figure 4.1 shows the augmented TIB, highlighting the added fields. Recall that a TIB is simply an Object array. The added fields have the following functions:

- Field 1 contains a TIB array pointer, which points to an array of all the available TIB specialisations for this object. Calling the TIB switching routine will replace the current TIB with one of these. This field was added by the TIB specialisation patch.

- Field 2 contains the current TIB index, implemented as an integer array of length 1. The TIB index is simply an index into the TIB array above. If inlined methods have specialisations, they are protected by guards that test this field. This field is contributed by this project.

- Field 3 contains a pointer to the default TIB for this type, usually index 0 in the TIB array. The default TIB is important since some type checking is done through simple pointer comparison of the TIBs. Providing a separate field with this pointer speeds up this form of type checking slightly. This field is also contributed by this project.

Note that since TIBs are shared between objects, adding more fields does not bloat objects.

To access the extra fields, two new low level IR operators, \texttt{GET\_OBJ\_PRIMARY\_TIB} and \texttt{GET\_TIB\_INDEX\_FROM\_TIB}, were created. \texttt{GET\_OBJ\_PRIMARY\_TIB} is similar to a byte code that already exists in the vanilla Jikes RVM, \texttt{GET\_OBJ\_TIB}, but more expensive since one extra level of indirection has to be resolved. In order to preserve correctness of type checks and some other operations, some uses of \texttt{GET\_OBJ\_TIB} were changed to use \texttt{GET\_OBJ\_PRIMARY\_TIB}. \texttt{GET\_TIB\_INDEX\_FROM\_TIB} obtains the integer indirectly stored in field 2 of the TIB, telling us which TIB specialisation is the current one. Its use will be clear shortly.
4.1.2 Handling method objects and references

Recall from section 2.2.3 that two parallel hierarchies represent types in the Jikes RVM. VM\_Class, VM\_Field and VM\_Primitive all extend VM\_Type; these are the type objects. VM\_MethodReference and VM\_FieldReference extend VM\_MemberReference. These represent references to class members in class files and thus substitute for VM\_Type objects before class resolution is completed. Sometimes they are also used internally by the compilers. VM\_MemberReferences are normally uniquely identified by their fully qualified name (and descriptor if a method), for example (“println”, “(Ljava/lang/String;)V”). We would like to extend this so that they are uniquely identified by these two and also the TIBs they belong to, in order to make recompilation possible. Strictly speaking we only need this for VM\_MethodReferences, since we don’t specialise VM\_FieldReferences, but we place this functionality in VM\_MemberReference since the logic for creating and managing all VM\_MemberReferences is there. We will now describe the infrastructure that was put in place, and later demonstrate its necessity.

We add a TIB map to each VM\_MethodReference so that we can test easily which TIBs a method reference or its method is in. This can be more than one TIB, since some methods, for example static ones, are usually not specialised, and then duplicating the method makes little sense. The TIB map is simply an integer which has bits set according to which TIBs it is valid for, much like the “implements trits” in the TIB that we saw earlier (section 2.1.2). In this project we are only ever using two different TIBs but it is again possible to extend this to a much greater number; the integer field that is the TIB map could be replaced by an integer array\(^1\).

\(^1\)Unfortunately this could have performance consequence for the inlining guard that we will introduce in the next chapter, but the 32 possible TIBs provided by an integer should surely be
procedure findOrCreate(String name, String descriptor)

1: key = new VM.MemberReference containing (name, descriptor)
2: if dictionary has value for key then
3:    Return value
4: else
5:    value = key
6: end if
7: Add value to dictionary
8: Return value

Figure 4.2: Creating canonicalised VM.MemberReferences in the vanilla RVM.

We track the specialisations by equipping each VM.MemberReference with an array containing references to its specialisations. We then add three instance methods for examining and manipulating the specialisations:

- findSpecialisationForTIB for finding a specialisation for a specific TIB, if any
- addSpecialisation for adding a new specialisation
- isSpecialisationOf for testing if one VM.MemberReference is a specialisation of another.

As we touched on above, member references are uniquely identified by their name and descriptor in the vanilla VM. This is enforced by making the constructors private. When a new member reference needs to be created a static method findOrCreate is called. It is illustrated in Figure 4.2. A hash table keeps track of all VM.MemberReferences in existence at any time. When findOrCreate is called, the hash table is first examined to see if the required reference is there. If it is not, it is added and returned. The hash key of each member reference is the sum of the hash keys of its type, name and descriptor, enforcing the canonicalisation. See Figure 4.2 for an illustration of this.

We extend this idea to include TIB specialisations. A naive way of doing this might be to add the TIB maps to the hash key, but we would then be in trouble when we try to find a member reference for any given TIB. We would be forced to construct all possible TIB maps that allow the TIB we are looking for, and for each one of them try to find a corresponding item in the hash table. This is unacceptable. We have instead opted to store only the member references that are in the primary TIB, which in our case are the first ones created, in the hash table. The other ones are tracked through the specialisation array discussed above. So finding a member reference becomes a two step procedure: first we look up the primary reference in the hash table, and then we find the specialisation we want. See Figure 4.3 for an illustration of the improved findOrCreate.

4.2 Handling method recompilation

Because the Adaptive Optimisation System is based on selective recompilation of methods, the optimising compiler frequently needs to discard compiled methods and replace them with new ones. This involves patching the method table in the TIB to point to the new entry code for the recompiled method. Since we have sufficient for a long time.
procedure findOrCreate(String name, String descriptor, int TIB)

1:   key = new VM_MemberReference containing (name, descriptor, TIB)
2:   if dictionary has value for key then
3:     if value has a specialisation sp in TIB then
4:       Return sp
5:     end if
6:     Add specialisation for TIB to the specialisation list of value
7:   else
8:     value = key
9:   end if
10:  if Did not add method specialisation above then
11:     Add value to dictionary
12:  end if
13:  Return value

Figure 4.3: Creating canonicalised VM_MemberReferences in the enhanced RVM. Note that the hash function remains in its original version, that is, the TIB field does not influence it.

specialised the TIBs we now need to make sure we patch precisely the right TIBs when a method is recompiled. We have access to this information through the TIB maps described in section 4.1.2, since it is easy to obtain the corresponding VM_MethodReference from any VM_Method.

The normal sequence of events when a method is recompiled is the following.

1. The optimising compiler recompiles a method, producing a new instance of VM_CompiledMethod
2. VM_Method.replaceCompiledMethod() is called for the corresponding VM_Method. Objects of type VM_Method live as long as the VM runs, but VM_CompiledMethod objects are continuously invalidated and discarded.
3. VM_Class.updateMethod() is called for the corresponding VM_Class.
4. If the method is virtual, VM_Class.updateVirtualMethod() eventually gets called. Otherwise static method tables are updated.
5. In the vanilla VM, the method slot in the TIB corresponding to this method is overwritten. Interface invocation slots are also updated, if necessary.

In our enhanced VM, we extend the final step to iterate across all possible TIBs, testing, through the TIB map, whether or not the method is in that TIB. If it is, we update the corresponding method slot and the interface invocation slots corresponding to the given TIB index. We also have to use the infrastructure we put in VM_MemberReference to deal with some minor technical subtleties. The interested reader is referred to the source code.

4.3 Handling native methods (JNI)

Recall that as part of the IncrGCMethodTransform we rename native methods and insert non-native wrappers, in order to be able to specialise the native methods. Whenever we call a native method, we reverse this renaming to be able to look up the correct library symbol. For example, a native method writeByte would be renamed _nat_r_writeByte so that a non-native wrapper called writeByte can be
created. This prefix is removed before the library symbol is looked up, making the renaming transparent to the environment.
CHAPTER 4. GENERAL MODIFICATIONS TO THE VM
Chapter 5

Inlining

This chapter will describe the modifications done to the optimising compiler’s inlining subsystem in order to allow it to compile specialised methods.

5.1 A new inlining guard

Inlining specialised methods is in a sense similar to the polymorphism case where multiple candidates for inlining exist and a guard is used to verify the receiver’s type before entering an inlined method. Observing this, we create a new guard that tests the current TIB index of an object. Since each compiled method has a TIB map that determines which TIBs it belongs to, emitting this guard is relatively simple and a bitwise and is enough for the test. We thus emit a guard, called ig_tib_test, that

1. obtains the TIB of the caller
2. obtains the TIB index from the TIB using the new low level operand GET_TIB_INDEX_FROM_TIB
3. tests whether the method that was inlined is valid for this TIB index

When the guard is compiled, the TIB map of the method that is inlined is included as an immediate operand, so this need not be retrieved from anywhere.

5.2 Allowing multiple inlining guards per target

Prior to the modifications introduced by this project, it was never necessary to protect an inlined method by more than one inlining guard. This is no longer the case, as we in many situations need to combine one of the original guards, for type checking, say, with our new guard for TIB index verification. However, the vanilla Jikes RVM only contains support for single guards. Hence, this project extends the VM to support an arbitrary number of inlining guards per target\(^1\). Each inlined call site will be similar to figure 5.1.

5.3 Extending the inlining oracle

We recall the default inlining oracle from section 2.1.3. We would like to preserve as much of its inlining behaviour as possible, so we only change it as much as is

\(^1\)Some of the code assumes at most two guards but this can be extended trivially
Figure 5.1: An inlined callee with multiple guards. If any of the guard conditions fail, the actual type and TIB index of the receiver of the call will be used to resolve the method to be called.

necessary to allow it to work correctly with the rest of the framework. We call the new inlining oracle \textsc{OPT\_TIBHijackingInlineOracle}. Here is a summary of the differences between it and the default policy:

- We only ever inline callees that begin to TIB 0. Other specialisations are assumed to be called only a fraction of the time. (This is true in our case with only one specialisation, but for more complex cases this could be made more clever).

- \texttt{inliningActionCost()}, a method for estimating the cost of an inlining action with guards, contains logic for estimating the cost of an \texttt{ig\_tib\_test} guard.

- The notion of a “complex” guard has been introduced. A complex guard is any guard other than \texttt{ig\_tib\_test}. Where the default inlining oracle tested whether a guard was needed and based decisions on that, in some cases it is now only testing whether complex guards are needed.

- Whenever an inlining decision has been made, we test the callee to see if it has any specialisations. If it does, we add the \texttt{ig\_tib\_test} guard to the final inlining decision. Thus, a complete inlining decision can have 0, 1 or 2 guards, where previously it could only have 0 or 1.

We give a pseudocode view of the extended oracle in figure 5.2. The original oracle can be found, for comparison, in figure 2.2.

Most of the original inlining heuristic has been preserved, except that to each inlining target, we can now assign up to two guards, one of which is the usual guard that the target would have been assigned in the vanilla VM. The other is either \texttt{ig\_tib\_test} or no guard at all. With method specialisations, the number of call sites that will require guards will of course increase dramatically, and many call sites that previously would have been handled by the “trivial” case early in the oracle logic are now passed on to the later stages.
procedure shouldInline(...)  

1: if callee is not in TIB 0/primary then  
2:   return decision(‘‘Do not inline’’)  
3: end if  
4: if callee is small and has a precise target then  
5:   return decision(‘‘Do a trivial guardless inlining’’)  
6: end if  
7: if callee is ‘‘huge’’ then  
8:   return decision(‘‘Do not inline’’)  
9: end if  
10: Find all possible targets by traversing class hierarchy  
11: for each possible target \( t \) do  
12:   Evaluate cost of inlining \( t \)  
13:   if inlining recursively then  
14:     return decision(‘‘Do not inline’’)  
15:   end if  
16:   if \( t \) is a forbidden speculation and \( t \) needs a guard then  
17:     return decision(‘‘Do not inline’’)  
18:   end if  
19:   if \( t \) was previously inlined then  
20:     Decide yes  
21:   end if  
22:   if not decided yes then  
23:     Decide based on call site weight and weight of \( t \)  
24:   end if  
25: end for  
26: Create inlining decision \( d \)  
27: for each possible target \( t \) do  
28:   if decided yes for \( t \) then  
29:     Decide the proper guards \((g_1, g_2)\) if any  
30:     Add \((t, (g_1, g_2) ‘‘yes’’)\) to \( d \)  
31:   end if  
32: end for  
33: return \( d \)  

Figure 5.2: The extended inlining oracle. This can be found in OPT_TIBHijackingInlineOracle2.java.
Chapter 6

Evaluation

6.1 Methodology

This chapter evaluates the changes made to the Jikes RVM, assessing their suitability for inexpensive TIB switching. SPECjvm98, a widely accepted industry standard benchmark [11], has been used for performance evaluation\(^1\). Recall that because of the BeanifierTransform, our VM is not entirely semantically correct - in some cases field hiding breaks (see section 3.1). For this reason we cannot run the SPECjvm98 benchmark _228_jack. All the other benchmarks can be run.

The benchmarks have been run on an Intel Pentium 4 running at 2.60 GHz with 512 KB cache and 1 GB of RAM. The operating system was GNU/Linux 2.6.15 with GNU libc 2.3.4.

Unless otherwise stated, all benchmarks have been run 5 times (or 5 x 2 times in some cases). All benchmark values are measured in seconds and rounded to milliseconds. Where both arithmetic and geometric means are given, the arithmetic mean has been used to calculate percentage overheads.

Arithmetic and geometric averages across the five sequences were calculated to give a measure of the uncertainty. Values have been rounded to milliseconds. Where applicable, the difference between the first and second run of each benchmark can be attributed to transformation and initial compilation costs.

6.2 Benchmarks

6.2.1 Baseline performance

Our baseline performance will be the vanilla Jikes RVM from CVS as of the 16th of January 2006. Other benchmarks will be compared to this as a percentage overhead. Data is given in table 6.2.1. Each benchmark has been run separately twice, and this procedure has been repeated five times.

6.2.2 Performance of the enhanced VM

Having established the baseline performance measure, we will now look at the performance of Jikes RVM with the modifications described in this project. Data is given in table 6.2.

We have instrumented the RVM with timers in strategic places to measure the time spent in transformations separately. This is done through calls to

\(^1\)It is not yet possible to run SPECjbb2005 on Jikes RVM since it requires a J2SE 5.0 compliant environment
Figure 6.1: Baseline performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Run</th>
<th>Time (arith)</th>
<th>Time (geom)</th>
</tr>
</thead>
<tbody>
<tr>
<td>201_compress</td>
<td>1</td>
<td>6.722</td>
<td>6.720</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5.671</td>
<td>5.669</td>
</tr>
<tr>
<td>202_jess</td>
<td>1</td>
<td>3.613</td>
<td>3.612</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.704</td>
<td>2.704</td>
</tr>
<tr>
<td>209_db</td>
<td>1</td>
<td>14.840</td>
<td>14.840</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14.623</td>
<td>14.615</td>
</tr>
<tr>
<td>213_javac</td>
<td>1</td>
<td>9.471</td>
<td>9.469</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7.569</td>
<td>7.569</td>
</tr>
<tr>
<td>222_mpegaudio</td>
<td>1</td>
<td>7.012</td>
<td>7.012</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5.347</td>
<td>5.347</td>
</tr>
<tr>
<td>227_mtrt</td>
<td>1</td>
<td>5.273</td>
<td>5.270</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.672</td>
<td>3.671</td>
</tr>
</tbody>
</table>

Figure 6.2: Performance of the enhanced VM

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Run</th>
<th>Time (arith)</th>
<th>Time (geom)</th>
<th>Transforms (arith)</th>
<th>Overhead</th>
<th>Tfm ohd</th>
</tr>
</thead>
<tbody>
<tr>
<td>201_compress</td>
<td>1</td>
<td>11.482</td>
<td>11.479</td>
<td>0.108</td>
<td>70.8%</td>
<td>1.61%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9.257</td>
<td>9.254</td>
<td></td>
<td>63.2%</td>
<td></td>
</tr>
<tr>
<td>202_jess</td>
<td>1</td>
<td>7.037</td>
<td>7.035</td>
<td>1.700</td>
<td>94.8%</td>
<td>22.9%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.615</td>
<td>3.613</td>
<td></td>
<td>33.7%</td>
<td></td>
</tr>
<tr>
<td>209_db</td>
<td>1</td>
<td>17.160</td>
<td>17.160</td>
<td>0.097</td>
<td>15.6%</td>
<td>0.654%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14.845</td>
<td>14.845</td>
<td></td>
<td>1.52%</td>
<td></td>
</tr>
<tr>
<td>213_javac</td>
<td>1</td>
<td>19.206</td>
<td>19.196</td>
<td>4.779</td>
<td>103%</td>
<td>50.5%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11.843</td>
<td>11.828</td>
<td></td>
<td>56.5%</td>
<td></td>
</tr>
<tr>
<td>222_mpegaudio</td>
<td>1</td>
<td>9.547</td>
<td>9.534</td>
<td>0.824</td>
<td>36.2%</td>
<td>11.8%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5.532</td>
<td>5.519</td>
<td></td>
<td>3.46%</td>
<td></td>
</tr>
<tr>
<td>227_mtrt</td>
<td>1</td>
<td>11.596</td>
<td>11.579</td>
<td>0.478</td>
<td>120%</td>
<td>9.07%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7.283</td>
<td>7.237</td>
<td></td>
<td>98.3%</td>
<td></td>
</tr>
</tbody>
</table>

VM.Time.currentTimeMillis(), using one counter for each thread that enters the relevant methods. The time measured will of course also include work done by background threads like garbage collection, but this should be a function of the work done in the foreground threads, so we deem this acceptable. The cost measured here is precisely the cost of applying the IncrGCmethodTransform and the BeanifierTransform as described in this project. This cost can be removed by transforming classes statically outside the RVM. Data is given in table ???. The second percentage indicates the overhead of transformations only, whereas the first percentage is the total overhead. All percentages are proportions of the baseline performance in table 6.2.

We see a substantial overhead: between 1.52% and 120%, and many above 50%. Fortunately the transform cost is generally fairly high; this can be subtracted from the total overhead to give the potential overhead if transformations were to be applied statically.

6.2.3 Cost of wrapping native methods

Table 6.3 shows the overhead of the full enhancements (against the vanilla VM) when native methods are not specialised. We compare this against the overhead when they are, which we get from Table 6.2, to extract the cost of wrapping native
6.2. BENCHMARKS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Run</th>
<th>Time (arith)</th>
<th>Time (geom)</th>
<th>Overhead</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>1</td>
<td>14.769</td>
<td>14.751</td>
<td>120%</td>
<td>-48.9%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11.241</td>
<td>11.236</td>
<td>98.2%</td>
<td>-35.0%</td>
</tr>
<tr>
<td>jess</td>
<td>1</td>
<td>9.150</td>
<td>9.128</td>
<td>153.2%</td>
<td>-58.3%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4.401</td>
<td>4.368</td>
<td>62.7%</td>
<td>-29.1%</td>
</tr>
<tr>
<td>db</td>
<td>1</td>
<td>17.630</td>
<td>17.629</td>
<td>18.8%</td>
<td>-3.2%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16.129</td>
<td>16.124</td>
<td>10.3%</td>
<td>-8.8%</td>
</tr>
<tr>
<td>javac</td>
<td>1</td>
<td>20.723</td>
<td>20.719</td>
<td>119%</td>
<td>-16.0%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13.518</td>
<td>13.501</td>
<td>78.6%</td>
<td>-22.1%</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>1</td>
<td>12.847</td>
<td>12.801</td>
<td>83.2%</td>
<td>-47.1%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9.883</td>
<td>9.833</td>
<td>84.8%</td>
<td>-81.4%</td>
</tr>
<tr>
<td>mtrt</td>
<td>1</td>
<td>9.853</td>
<td>9.848</td>
<td>86.9%</td>
<td>33.1%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6.410</td>
<td>6.306</td>
<td>74.6%</td>
<td>23.8%</td>
</tr>
</tbody>
</table>

Figure 6.3: Cost of wrapping native methods

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Config</th>
<th>Vanilla (arith)</th>
<th>Vanilla (geom)</th>
<th>Enhanced (arith)</th>
<th>Enhanced (geom)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>Std</td>
<td>0.153</td>
<td>0.153</td>
<td>0.277</td>
<td>0.277</td>
<td>81.2%</td>
</tr>
<tr>
<td></td>
<td>Opt</td>
<td>2.107</td>
<td>2.107</td>
<td>2.752</td>
<td>2.752</td>
<td>30.6%</td>
</tr>
<tr>
<td>jess</td>
<td>Std</td>
<td>0.165</td>
<td>0.164</td>
<td>0.455</td>
<td>0.453</td>
<td>176%</td>
</tr>
<tr>
<td></td>
<td>Opt</td>
<td>3.717</td>
<td>3.717</td>
<td>6.929</td>
<td>6.929</td>
<td>86.4%</td>
</tr>
<tr>
<td>db</td>
<td>Std</td>
<td>0.103</td>
<td>0.103</td>
<td>0.297</td>
<td>0.297</td>
<td>188%</td>
</tr>
<tr>
<td></td>
<td>Opt</td>
<td>2.342</td>
<td>2.342</td>
<td>2.873</td>
<td>2.873</td>
<td>22.7%</td>
</tr>
<tr>
<td>javac</td>
<td>Std</td>
<td>0.454</td>
<td>0.452</td>
<td>1.023</td>
<td>1.020</td>
<td>125%</td>
</tr>
<tr>
<td></td>
<td>Opt</td>
<td>7.596</td>
<td>7.596</td>
<td>16.832</td>
<td>16.832</td>
<td>122%</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>Std</td>
<td>0.506</td>
<td>0.506</td>
<td>0.926</td>
<td>0.926</td>
<td>83.0%</td>
</tr>
<tr>
<td></td>
<td>Opt</td>
<td>2.688</td>
<td>2.688</td>
<td>4.468</td>
<td>4.468</td>
<td>66.2%</td>
</tr>
<tr>
<td>mtrt</td>
<td>Std</td>
<td>0.247</td>
<td>0.247</td>
<td>1.621</td>
<td>1.533</td>
<td>557%</td>
</tr>
<tr>
<td></td>
<td>Opt</td>
<td>2.757</td>
<td>2.757</td>
<td>5.685</td>
<td>5.685</td>
<td>106%</td>
</tr>
</tbody>
</table>

Figure 6.4: Compilation cost

methods. Thus, the contents of the “difference” column in table 6.3 has been obtained by subtracting the “overhead” column here from the “overhead” column of table 6.2. As usual, the comparison is based on the arithmetic average.

Note that this cost overlaps with the transformation cost for the first run, and with the compilation cost for both runs.

Note that this cost overlaps with the transformation cost for run 1. We see some erratic behaviour here; in many cases (negative overheads) the RVM that wraps native methods runs faster than the one that does not! I currently have no explanation for this. We will come back to this point when we discuss future work.

6.2.4 Cost of compilation

Each benchmark was run five times to measure pure compilation costs for the vanilla VM and the enhanced VM. Two configurations were measured: Std uses the standard adaptive compilation strategy where each method is first compiled with the baseline compiler, and hot paths are later recompiled with the optimising compiler. Opt uses the optimising compiler for all compilation. We use facilities built into the Jikes RVM for the measurements.

Compilation costs have gone up by between 22.7% and 122% when running with
the optimising compiler without recompilation, and by between 81.2% and 557% when running with the usual adaptive system. But in the latter case, even the worst overheads are only a small fraction of the overall overhead.

6.2.5 Cost of the BCEL infrastructure

We note that incorporating the BCEL framework carries a significant overhead. Most likely this is because of the more expensive class loading through BCEL - see section 2.2.3. This cost can potentially be removed by transforming classes statically outside the RVM.

6.2.6 An explicit read barrier

Steve Blackburn has provided us with an explicit read barrier[1], which we benchmark to get an idea of a typical read barrier cost. Since this is highly experimental software, we have been unable to run all the benchmarks. The ones that did run are shown in table 6.6. Overheads are percentages against the vanilla VM, as usual. This is in a sense a possible justification for our approach, at least with respect to garbage collection; if our performance overhead is below that of the explicit read barrier, an incremental garbage collector built on our framework would buy back the performance loss of full virtualisation.

---

### Figure 6.5: BCEL overhead

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Run</th>
<th>Time (arith)</th>
<th>Time (geom)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>1</td>
<td>7.856</td>
<td>7.854</td>
<td>16.9%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6.557</td>
<td>6.556</td>
<td>15.6%</td>
</tr>
<tr>
<td>_202_jess</td>
<td>1</td>
<td>4.787</td>
<td>4.787</td>
<td>32.5%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.142</td>
<td>3.140</td>
<td>16.2%</td>
</tr>
<tr>
<td>_209_db</td>
<td>1</td>
<td>15.198</td>
<td>15.198</td>
<td>2.41%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>15.474</td>
<td>15.473</td>
<td>5.82%</td>
</tr>
<tr>
<td>_213_javac</td>
<td>1</td>
<td>12.944</td>
<td>12.944</td>
<td>36.7%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9.611</td>
<td>9.603</td>
<td>27.0%</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>1</td>
<td>7.865</td>
<td>7.863</td>
<td>12.2%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5.333</td>
<td>5.333</td>
<td>-0.3%</td>
</tr>
<tr>
<td>_227_mtrt</td>
<td>1</td>
<td>6.317</td>
<td>6.315</td>
<td>19.8%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4.128</td>
<td>4.122</td>
<td>12.4%</td>
</tr>
</tbody>
</table>

### Figure 6.6: Cost of explicit read barrier

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Run</th>
<th>Time (arith)</th>
<th>Time (geom)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>1</td>
<td>9.195</td>
<td>9.194</td>
<td>36.8%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8.050</td>
<td>8.041</td>
<td>42.0%</td>
</tr>
<tr>
<td>_209_db</td>
<td>1</td>
<td>15.789</td>
<td>16.405</td>
<td>6.40%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>15.789</td>
<td>15.789</td>
<td>7.97%</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>1</td>
<td>11.441</td>
<td>11.438</td>
<td>63.2%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8.231</td>
<td>8.199</td>
<td>53.9%</td>
</tr>
</tbody>
</table>
### 6.3 Performance overhead breakdown

![Table](image)

We will now summarise the sources of new overhead that we have been able to identify and their contribution to the overall overhead as seen in table 6.2. Because of the erratic results concerning the overhead of wrapping native methods, we will not include them here. The “other” column has been obtained by subtracting all the component from the corresponding total. This summarisation is based on the first of the two runs, in the cases where benchmarks were run twice.

A large proportion of the overhead has thus not yet been classified. However, because we can potentially do transformations outside the VM, making the BCEL patch and the transformations redundant, we may now put an upper bound on the performance overhead that can be obtained from our approach. For convenience, this is shown in the rightmost column.

The only benchmark where we beat the explicit read barrier is mpegaudio, where the explicit barrier has an overhead of 63.2% and our system would have an overhead of 12.2% if transformations were done statically. However, our framework shows promising performance in all benchmarks except 1 and 6, particularly in 3-5. Since this is an upper bound, given more work there may be scope for further improvement. Particularly the mtrt benchmark has an enormously large chunk of unclassified overhead, which makes up almost all of its bounded overhead.
Chapter 7

Conclusion and future work

This chapter will present the conclusion of this project and suggest some future directions.

How well have the goals of this project been met? We recall from chapter 1 that the project goals were

- give an account of prior work in this field, including some incremental garbage collectors, inlining strategies and extensions to the Jikes RVM.
- investigate the possibility of providing a framework for method specialisation in the Jikes RVM, in which arbitrary actions can optionally be taken before each object is accessed, by exploiting dynamic dispatch. This will be done specifically with a garbage collector in mind.
- minimize the performance impact of the above by developing a suitable inlining protection mechanism and inlining heuristic.
- evaluate the results through the use of suitable benchmarks.

We have given an account of Andrew Cheadle’s work on GHC to produce an incremental garbage collector, the Jikes RVM and the BCEL framework. We have also described some of the problems with inlining in object oriented languages.

We arrived at a three component framework that manages to provide a system for exploiting dynamic dispatch at relatively low cost.

We have made it possible to use the full optimising compiler and inline methods arbitrarily while using this framework.

We have also evaluated the framework thoroughly. We were able to classify and dissect much of the overhead this project introduces, but a large chunk remains unclassified. This is perhaps the most significant weakness of this project. Also we were unable to account for the erratic benchmark results concerning the wrapping of native methods. These shortcomings are mainly the consequence of a lack of time.

Nevertheless, we have been able to present a theoretical upper bound on the performance that can be achieved by hijacking dynamic dispatch using the approach of our framework. This has been shown in table 6.7. For many of the benchmarks the results here are promising, with overheads below 16% for half the benchmarks tested. We have reason to believe that further work invested in exploring the unknown portion of the overhead could prove very fruitful.

In conclusion, the project goals have largely been met. Although there are some unknown sources of performance loss, we have shown that it is indeed possible to construct the system we set out to construct, so that dynamic dispatch can be
exploited through method specialisation while the RVM runs at speeds comparable to normal.

7.1 Future work

One of the most important openings for future work is classifying the remaining overhead so that its sources can be understood. When this is done, it may well be possible to construct a system that yields higher performance than that of the framework described in this report.

It is also possible to handle native methods more gracefully than we have done. One possibility is auto-generation of native methods that can act as specialisations for other native methods. If this could be done, then the extra non-native wrapper layer would be unnecessary.

It appears from the benchmarks that the cost of having the BCEL patch in the RVM is rather large. Work going into optimising this would be valuable.

The enhanced VM is not semantically correct, because it cannot handle field hiding in some cases. I believe that it could be possible to remedy this by improving the BCEL transforms.

It may be possible to combine and hoist inlining guards up one or several levels, so that guards for inlined small callees are lifted up to the caller’s entry point. This could prove valuable since in our system we encounter far more small methods that require guards than the vanilla RVM does.

Also, tweaking of the various numerical parameters that control the inlining oracle could prove beneficial to performance. Again, small guarded callees have become much more frequent, so that the callee distribution is vastly different from what the oracle was designed for. Exploring this design space might be very valuable.
Bibliography


