A Generic Approach to Object Migration using Specialised Methods

Will Deacon
`wjd105@doc.ic.ac.uk`

Supervisor: Dr Tony Field

June, 2009

Acknowledgements (`0x06 0x06`)

Firstly, I would like to thank my family for their support and encouragement during my time at university and for putting up with my constant moaning. Secondly, the support offered by my fellow prisoˆHˆHˆHˆHˆHstudents has been immeasurable and for this I'm very grateful. In particular, I would like to thank Robin Bennett and Will Jones for reducing the complexity of the crossword to something tractable and allowing me to get some work done. Finally, I would like to express my gratitude and respect for my supervisor, Dr Tony Field, for giving up so much of his time to listen to me ramble about computers.

Additionally, the following people have played a major part in my academic experience both professionally and personally: Andy Cheadle for motivating the hacker inside me; Jack Griffith for staying aboard the sinking ship that is Microsoft Office, despite his better judgement; and Will Harrower for keeping me company at jazz gigs.

Abstract

This project describes and implements a framework to support the execution of specialised methods within the Jikes RVM. The idea is that by applying suitable bytecode transformations at class loading time, method dispatch can be hijacked to invoke a different method implementation depending upon the state of the receiver. These transformations are specified by the programmer using a simple API and are treated independently from the application on which they operate. We evaluate the overheads incurred by our modified virtual machine and motivate the use of class transformations by means of a read barrier and an implementation of transparent object persistence for Java. Our results show that the performance penalty of our framework is extremely low and our technique for specifying class transformations is both efficient and concise.

Contents

Chapter 1

Introduction

The object-oriented programming paradigm promotes modularity, encapsulation and software reuse. Central to this philosophy is the notion of an object, that provides a means of combining data with the knowledge to manipulate it. The self-contained nature of an object lends itself to the possibility of *object migration* which requires programs to capture the notion of object *state*. By migration we specifically mean the movement of objects around a memory system and possibly across address spaces.

The state of an object is commonly used to describe the value of its fields with respect to a finite state machine, however for the purposes of object migration it signifies the *validity* of the object in question. Consider the following Java code:

```java
private Sphere[] spheres;

/*
 * Return an array containing the spheres
 * that intersect with the given ray.
 */
public Sphere[] intersectsWith(Ray ray) {
  LinkedList<Sphere> intersections = new LinkedList<Sphere>();

  for (Sphere sphere : spheres)
    if (sphere.intersectsWith(ray))
      intersections.add(sphere);

  return intersections.toArray(new Sphere[0]);
}
```

Listing 1.1: A simple ray intersection test.

This straightforward method iterates across an array of `Sphere` objects, invoking a method on each one in turn. Now suppose that the spheres are held on a single node having been lazily loaded from a backing store. As demonstrated in Listing 1.2, the code must be modified to check that each sphere has been faulted into memory before it is dereferenced.

```java
 1 public Sphere[] intersectsWith(Ray ray) {
 2   LinkedList<Sphere> intersections = new LinkedList<Sphere>();
 3
 4   for (Sphere sphere : spheres) {
 5     if (!faultedIn(sphere))     /* Test */
 6       load(sphere);             /* Action */
 7
 8     if (sphere.intersectsWith(ray))
 9       intersections.add(sphere);
10   }
11
12   return intersections.toArray(new Sphere[0]);
13 }
```

Listing 1.2: Lazy loading of spheres from a backing store.

The explicit state **test** at line 5 checks whether or not the object resides on disk or in memory and in the case of the former, **action** is taken to load the sphere from disk prior to its use. This test-action pattern is a common occurrence in programs where the state of an object affects the flow of execution. Once in memory, a sphere will remain in memory for the duration of the program but the dynamic nature of execution may lead to further, redundant state tests. A naïve implementation will simply prefix every object dereference with a test in order to ensure that the object is in memory before it is used. In this example, objects can exist in only two possible states: *faulted* or *unfaulted*.

In general, an object can exist in an arbitrary number of states. For example, a distributed shared memory implementation may label objects as *valid* or *invalid*, allowing valid objects to be cached in the local memory of a node. The cache hierarchy for a given node may be arbitrarily deep and as the object moves through this hierarchy its state will change to reflect its location in the memory system.

The state of an object may be explicit at the source level (as in Listing 1.2) but may also feature at a lower level, for example in a DSM layer or even within the JVM itself. Examples of the latter occur in distributed JVMs (Section 2.1) and distributed shared memory implementations as well as within the read-barrier of an incremental garbage collector (Section 5.2.1). Performance is a critical aspect in the implementation of a JVM and the status checks on object accesses can sometimes prove expensive [1, 2, 3] and may actually be redundant in some situations.

1.1 This Project

The main idea that underpins this project is to exploit Java's existing dynamic dispatch mechanism to implement distinct object behaviours by executing different method variants depending on an object's state.

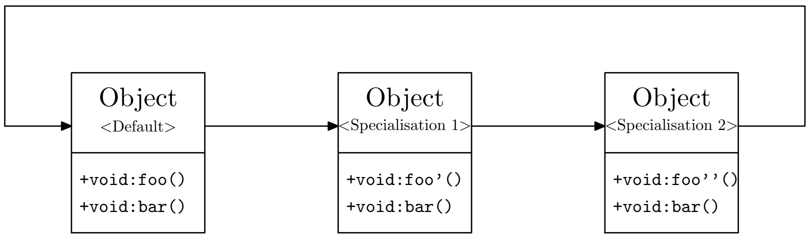


Figure 1.1: Three variations of an object, each with a different implementation of the `foo` method but sharing a common `bar` method.

Figure 1.1 illustrates the idea of specialised objects that initially exist in a 'default' state and can be toggled between a number of specialisations at runtime, each providing alternative method implementations for candidate methods. To achieve this, the VM must be extended to support multiple virtual tables per object and mechanisms have to be introduced to flip between them.

Specifically, we modify the *Type Information Block* (TIB) of an object to contain a pointer to an array of specialised TIBs that the JVM consults during method dispatch. This is similar to the way in which polymorphism is used to implement test-action code when the *type* of an object determines the behaviour of the program (see, for example, Figure 1.2). Implementing a similar system for checking the state of an object requires a class to have multiple implementations or *specialisations* of each method it defines.

```java
interface Animal {...}
class Cat implements Animal {...}
class Dog implements Animal {...}

public class Person {
  public void feed(Animal animal) {
    if (animal instanceof Cat)
      System.out.println("Cats eat tuna");
    else if (animal instanceof Dog)
      System.out.println("Dogs eat cats");
  }
  ...
}
```

(a) Using explicit **instanceof** tests to determine parameter type.

```java
interface Animal {
  public void eat();
}

class Cat implements Animal {
  public void eat() {
    System.out.println("Cats eat tuna");
  }
}

class Dog implements Animal {
  public void eat() {
    System.out.println("Dogs eat cats");
  }
}

public class Person {
  public void feed(Animal animal) {
   /* Invoke eat() on the dynamic type of animal */
    animal.eat();
  }
}
```

(b) Using polymorphism and overriden methods to dispatch to the correct method at runtime.

Figure 1.2: Rather than explicitly test the type of an object, dynamic dispatch provides an efficient, modular procedure to specify different object behaviours.

The key point is that method specialisations remove the need to test the object state explicitly because the specialised code is tailored to it. The current state is thus defined implicitly. This can improve performance significantly in some applications and provides a means to separate the concerns of each state in an *aspect-oriented* manner. Furthermore, specialised methods can be expressed as a *class transform* at the bytecode level (Section 4.4), consolidating any state dependent code and making the application easier to debug and maintain. Crucially, this transform-based approach obviates the need to modify the virtual machine in order to implement state-dependent execution below the application level. The transforms are applied automatically to all code during class loading; this includes both user code and third-party libraries.

The downside to this approach is that all accesses to a specialised object must be dispatched dynamically so as to invoke the correct variant. In Java all methods are implicitly declared virtual, but direct access to object fields can occur without consulting the virtual table. In order to capture all accesses to an object it may be necessary to generate additional `getter` and `setter` methods for its fields and then redirect any field accesses to these methods. We discuss this technique and its impact on performance in Section 4.3.

The long-term objective of this work is to investigate the application of method specialisation to object migration. The encoding of object locality tests into specialised methods paves the way for automatic data movement at the JVM layer and can be used to realise transparent object migration according to an application-specific coherency protocol. For example, in a simple two state system, accessing an object in the *invalid* state will cause it to execute code according to the specified protocol in order to place itself into the *valid* state and change its specialisation accordingly. In this scenario, the programmer's only concern lies in specifying the coherency protocol which, if written correctly, will guarantee the validity of any dereferenced object at runtime.

1.2 Contributions

The rest of this report documents an extended implementation of these ideas in the context of the production build of the Jikes Research Virtual Machine. We make the following contributions:

- We describe the method specialisation framework proposed in [4] (Section 4.1) and implement it within the production build of a recent version of the Jikes Research Virtual Machine. The Jikes RVM is a large and complex piece of code, so an overview of its essential features is presented in Section 3.2.
- We extend the framework (Section 4.2) to support the specialisation of static methods and describe the implementation we have developed.
- We discuss the problems associated with virtualising objects and present solutions for static and instance field access (Section 4.3).
- We give a brief overview of our API for class transformation and method specialisation along with a simple example of its use (Section 4.4).
- Using SPECjvm2008 and DaCapo benchmarks, we evaluate the performance of class transformation and object virtualisation within our framework (Section 5.1). We also offer some insight as to the cost of changing the specialisation of an object at runtime. Our results show that full virtualisation of a class typically adds less than 10% overhead to the converged execution time, but the actual penalty imposed is completely dependent on the nature of the application.
- To exemplify the practicality of our work, we describe and evaluate implementations of persistence for Java applications and a read barrier for an incremental garbage collector (Section 5.2). We show that the cost of our implicit read barrier makes it an attractive alternative to the commonly used explicit barrier for trapping object accesses.

Chapter 2

Related Work

Object migration is prevalent in a number of applications. We discuss and evaluate its implicit presence in the context of *Distributed Java Virtual Machines* and *Object Persistence* systems as well as the explicit use of migration in *Distribution APIs* and *In-memory Database* applications.

2.1 Distributed JVMs

The concept of a distributed Java virtual machine has been around for some time. The idea is to make the JVM run across multiple nodes whilst presenting a unified view, or *Single System Image* to the application. To achieve this, objects must migrate automatically around the system, creating the need for locality checks on object dereference. In this way, the issues surrounding object consistency and remote class loading are hidden from the programmer, who only needs to worry about the application at hand.

Existing implementations of a distributed Java runtime suffer from a number of pitfalls:

Lack of (current) development Active research into distributed JVMs appears to have diminished in recent years. Previous work has been left unmaintained and become subject to ‘bit rot’.

Availability Where the design of a distributed JVM has been described, there is rarely a publicly available implementation of the system. In the few cases where code is available, it is commonly a research prototype offering only a limited subset of the full functionality required by a real Java program.

Performance analysis Often, a paper will describe the algorithms to implement a distributed JVM efficiently, but then fail to disclose any real benchmarks. Where benchmarks are evaluated, they are typically ‘embarrassingly parallel’ and offer no real insight into the overheads of communication and data movement.

No de-facto standard As a result of the above points, no single distributed JVM has entered regular use by a body of programmers. The lack of even a *de-facto* standard leaves the situation as a collection of incomplete, unmaintained and disparate implementations, offering no support or future-proofing to programmers wishing to use them.

With this in mind, we evaluate some proposed solutions to the problem of distributed Java and look specifically at the use of object migration within them.

2.1.1 cJVM

cJVM [5] was developed by IBM around 1999. The authors claim to be the first to implement a cluster-aware JVM, where the cluster is completely hidden from the application. The cluster in mind is *'a collection of homogeneous machines connected by a fast communications medium'* and the application domain is that of *Java Server Applications* (JSAs).

The goal of cJVM is to distribute a standard multi-threaded Java application transparently across a cluster by assigning the application threads to different nodes. In order to provide this illusion, a *distributed heap* is implemented by cJVM. Each node runs a cJVM process capable of creating new objects. When a new object is created, the node responsible for its creation is said to hold the *master* copy of the object. External references (*proxies*) held by other nodes can be used to access the object indirectly. To support the illusion of a single address space, objects passed as arguments to remote procedures are assigned a unique global identifier.

Rather than move data around the cluster, cJVM migrates execution between nodes. That is, when a thread attempts to dereference a proxy object, execution is momentarily migrated to a thread running on the node where the master copy of the object is held. On this node the method is invoked before execution returns to the original thread on the remote node. This process is referred to as *method shipping*. Method shipping requires the concept of *distributed stacks*, the stack of a Java thread being split up over multiple system threads. Each Java thread is also assigned a logical identifier in order to provide a handle on related system threads and any monitors held by the thread.

To compensate for different data access patterns, proxy objects can dynamically change behaviour at run-time. The proxy implementations provided by cJVM are described as:

Simple proxy Default implementation - the method is executed on the master node.

Read-only proxy When data is read-only, the method can be executed locally. Copies of the data are therefore held by this proxy.

Proxy with locally invoked stateless methods Where object fields are not referenced, a method can be invoked locally.

Dynamic switching among proxy implementations is provided by extending the virtual method table of a class into an *array* of method tables, each of which refers to a different proxy implementation. This modification to the object model allows cJVM to handle the problem of object locality without explicit tests in a similar manner to the method specialisation described in Section 4.2.

Load balancing in the cJVM is achieved by intercepting the `new` opcode. If the parameter is a `runnable` object, the opcode is rewritten as a private `remote_new` opcode that can then consult a load balancing system in order to select the optimal node on which to run.

The performance of cJVM is unclear. Benchmarks taken from their website [1] offer little insight into the performance of real-world applications (reproduced in Figure 2.1). They claim ~80% efficiency on a 4 node cluster but this cannot be validated because the source code (or even a binary distribution) has not been made publicly available and the project has been inactive since 2000.

In conclusion, cJVM is a sound proposition for a distributed JVM implementation. The use of proxy objects allows for a fully distributed heap and automatic thread migration hides the cluster architecture from the programmer. The lack of an available implementation and a full suite of benchmarks limits the use of cJVM as a comparison with other distributed JVMs.

2.1.2 dJVM

dJVM [6] is an open-source research project built on top of the *Jikes Research Virtual Machine* described in Section 3.2. Originally, the dJVM project was to explore the themes of performance enhancement, fault-tolerance and memory management, but since 2004 these last two areas have been dropped. The prototype code available is unmaintained, obsolete (supporting Java versions 1.3 and 1.4) and only implemented in the *baseline* compiler of Jikes (as opposed to the *optimising* compiler). The implementation is not robust enough to run any 'real' benchmarks so the performance of the code is largely unknown.

[1] `http://www.haifa.il.ibm.com/projects/systems/cjvm/benchmark.html`

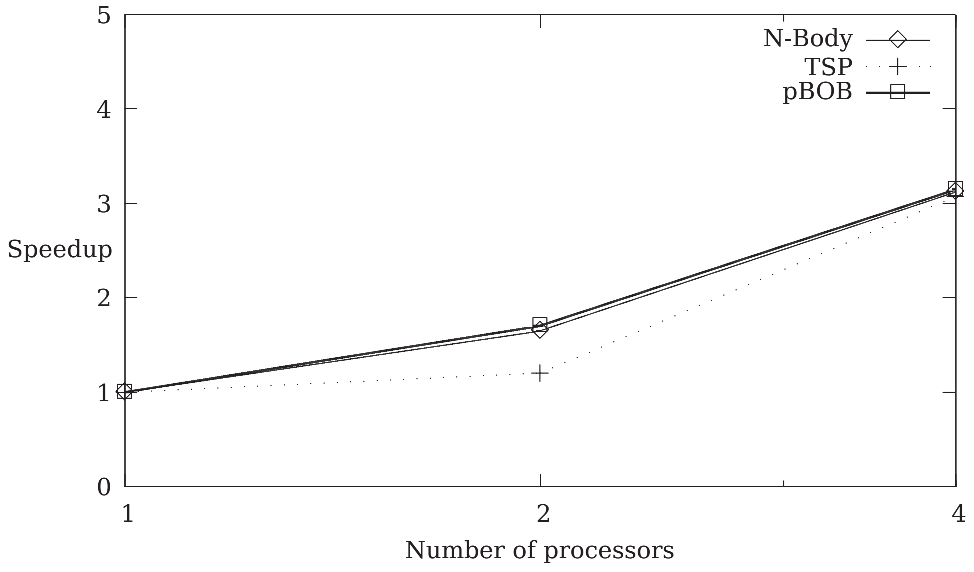


Figure 2.1: cJVM benchmarks for three selected kernels.

The approach taken is similar to that of cJVM. A master/slave architecture is used to coordinate interactions within the system. Upon booting, the master node activates the communication layer in order to connect each slave node to every other node in the cluster, forming a fully connected node graph prior to execution.

Communication within the dJVM occurs via the use of *messages* and consists of a *substrate*, *registry* and *thread pool*. The substrate provides a means of replacing the underlying communications protocol (e.g. TCP/IP) with another, if desired. It also provides a pool of message buffers that can be re-used to mitigate the impact of garbage collection. The registry is used for messages that require a response; the outgoing message registers itself with the registry in order to receive a notification when a response is sent back from the recipient. Finally, the thread pool contains a number of message-processing threads that assemble and decode messages from incoming packets. Messages can be either *synchronous* or *asynchronous* and are of the type *send*, *decode* or *process*.

dJVM makes use of a centralised class loader running on the master node. This creates a bottleneck but is justified by being simple to implement and by exploiting the observation that class loading generally becomes less frequent as a program executes. A class is described in Jikes as a set of *constant* objects, so these are simply copied to the slave node when classes are loaded. More interestingly, method compilation during class instantiation is performed locally on the slave node. Compiled methods are private to the node; sharing these objects is a potential optimisation that could be explored in future work. Globally used classes are instantiated at the master node.

Within the dJVM, objects have either an *Object IDentifier (OID)* in the case that they are held locally, or a *Local logical IDentifier (LID)* to indicate that the object is remote. A LID can be mapped on to a *Universal IDentifier (UID)* that in turn can be used to determine the remote node for the object. An OID is simply the address of the locally held object.

Invoking a method on a remotely held object is therefore performed by:

1. Locating the remote node.

2. Deciding where to execute the method. `static` methods, for example, can be executed locally as static method code is replicated at each node.

3. In the case of remote invocation, the method parameters are transferred to the remote node prior to execution.

Inter-node parameter passing is achieved using a *proxy/stub* model. Methods are compiled along with a `proxy` and a `stub` method. The proxy method creates a message containing the parameters, which it then sends to the remote node. The stub method unpacks the parameters from this message and invokes the relevant method on the (now local) object.

dJVM makes use of both data and thread movement in an attempt to improve performance. However, it is the location of data in the system that decides where execution takes place so, like cJVM, execution migrates to where the data is held. No benchmarks are available as the system is not yet robust enough to execute any substantial pieces of Java code.

It is worth noting dJVM is still being worked on internally at ANU even though the original funding has expired.

2.1.3 Hyperion

Hyperion [2] takes a novel approach to the distribution of Java threads. Rather than directly implement a distributed virtual machine, Java bytecode is generated with a standard Java compiler and then immediately translated into C using the `java2c` tool that forms part of the Hyperion system. The generated C code is linked with the Hyperion runtime library and compiled with a standard C compiler (Figure 2.2). The resulting executable contains an encoding of the original Java program alongside an internal API for distributed thread and data management.

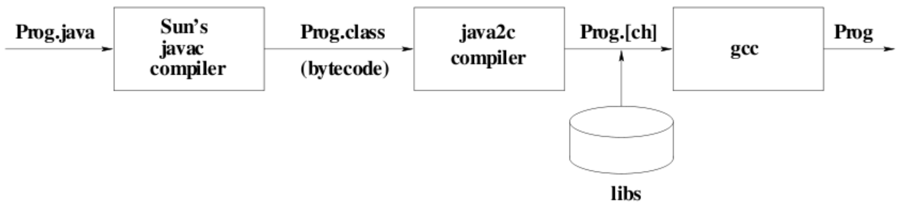


Figure 2.2: The process of compiling a Java program to run using the Hyperion framework (Image taken from [2]).

The runtime system is made up of the following components:

Thread subsystem This is built on top of the *Marcel* [7] thread library from PM2 [2]. The Marcel library offers the ability for RPCs and thread migration between nodes; the latter being used for load balancing in Hyperion.

Communication subsystem Communication in Hyperion is again provided by PM2. RPC invocations between nodes are used to invoke methods and create new threads where appropriate.

Memory subsystem The illusion of a *Distributed Shared Memory* (DSM) is created using the distributed shared memory layer in PM2. The Java Memory Model (JMM) does not require sequential consistency for non-volatile variables, allowing nodes to request data asynchronously from other nodes during execution. The JMM also specifies thread-local storage, or caches, to improve the performance of multi-threaded applications. Hyperion implements these caches, as well as per-node caches that are shared by all the threads on that node. The DSM also provides a number of optimisations; for example objects accessed inside a loop may be cached during the execution of the loop. Hyperion uses a number of memory primitives (`loadIntoCache, updateMainMemory, invalidateCache, get, put`) that are implemented at this layer.

Object migration in Hyperion occurs when multiple nodes access the same object. Like both cJVM and dJVM, objects are considered to have a master copy located at their home node (the node on which they were created). When another node accesses a remote object, a local copy is created in the requesting node's cache. Writes to this object by the (non-home) node then occur only in the

[2]http://runtime.bordeaux.inria.fr/Runtime/software.html

cache. To synchronise with the home node, `updateMainMemory` is called to write back the object to the home node and ensure that the object is consistent again.

Unfortunately, Hyperion is now a rather dated system. Only a 'small number' of native methods from the Java API 1.1 have been implemented, making Hyperion unsuitable for widespread use. The load balancing system simply uses a round-robin scheme rather than anything more advanced, for instance migrating threads away from overloaded nodes.

The performance of Hyperion is once again unclear. The single benchmark evaluated in [2] is not a widely recognised benchmarking program and the particular implementation is not disclosed. It is therefore difficult to assess whether the Hyperion system is optimally distributing threads and controlling data movement or whether the problem is simply embarrassingly parallel. The results from the benchmark show that the Hyperion system is about 5x slower than native C code when executing a sequential implementation. This overhead is particularly large and is attributed to runtime checks performed by Hyperion. These runtime checks can involve checking array bounds, or more interestingly checking object locality. The DSM underlying Hyperion is object-based rather than page based, each object reference therefore requires an in-line locality check to decide whether to fetch the possibly remote object. Results show that, when executing on a single node (i.e. making all objects local) performance increased by almost 50% due to the absence of these checks. However, the benchmark scales well when executing a parallel implementation achieving between 78% and 90% efficiency across four nodes.

2.1.4 JESSICA2

JESSICA2 [3] is a DJVM implementation from the University of Hong Kong. It is a successor to the JESSICA (Java-Enabled Single-System Image Computing Architecture) DJVM developed by the same team. JESSICA2 differs from the original project in that it uses a JIT to translate Java bytecode into native code prior to execution, as opposed to running the bytecode in an interpreter. The JIT is a custom implementation referred to as *JITEE* and has built-in support for thread migration between nodes at bytecode boundaries.

The single system image of JESSICA2 is realised using a *Global Object Space* (GOS). The GOS is essentially a purpose-built DSM designed with the JMM in mind. This approach offers the advantage that object migration can simply involve moving empty object 'shells' because dereferencing these will cause the GOS to fetch the relevant data automatically.

Both objects and threads can be migrated in JESSICA2. The master/slave architecture is used to distribute multi-threaded Java programs, with the master migrating threads to and between the slave nodes according to a load monitor. Thread migration is achieved using a technique called *stack capturing*. Stack capturing allows threads to migrate even though they are subject to execution by a JIT. This differs from Hyperion (Section 2.1.3) in that JESSICA2 must be able to match bytecode to native machine code, whereas Hyperion runs purely machine code and uses a generic thread migration layer. The migration process may therefore only occur at certain 'coherent' points, i.e. those points where the native code lines up with the end of a bytecode instruction. These points are identified by the JIT and migration code is added into the native instruction stream. When execution reaches a candidate point for thread migration, the added code checks whether or not it should migrate to another node. If so, the machine registers are spilled onto the Java stack, type information about the variables on the stack is encoded and the Java stack frames to migrate are chosen. Utility methods are then called to perform the data transfer and migrate the thread to the remote node.

As with many DJVMs, objects have a home node associated with them. However, JESSICA2 allows the home node of an object to change at runtime depending upon execution heuristics. This allows objects to move towards the nodes that are using them the most and ultimately reduce the amount of network traffic.

The performance of JESSICA2 is largely achieved by using a JIT on each node rather than a bytecode interpreter to execute the program. cJVM (Section 2.1.1) makes use of an interpreter and suffers as a result. Despite using a JIT, the performance of JESSICA2 is still generally worse then the unmodified Kaffe JVM on which it is built. The Java Grande Benchmark suite was used to benchmark the two JVMs and showed that the only two benchmarks in which JESSICA2 outperformed Kaffe by a reasonable margin were *sync* and *SOR*, although this was achieved in part by replacing the locking mechanism of Kaffe with native locking code for JESSICA2. A major part of the performance trouble is accredited to the GOS layer having to check the state of each object on every access.. It is claimed in [3] that this checking code alone contributes to as much as 50% of the usual native code produced by the JIT.

JESSICA2 is available as an open source prototype, implemented using the Kaffe JVM [3].

2.1.5 Conclusion

Object migration exists at the heart of a distributed JVM and contributes substantially to the overheads of such a system. The dynamic proxies implemented by cJVM remove the need for explicit object tests and encode the object state within the proxy. Lack of in-depth benchmarking conceals the costs associated with such a strategy but the efficiency of the whole system is claimed to be high. The proxy idea is taken further by dJVM, which uses it for remote method invocation but requires a way to locate the node holding the receiver object. Object proxies are essentially a more specific notion of method specialisation and could be implemented within a specialisation framework.

Hyperion and JESSICA2 take a more conventional DSM approach to object distribution, but this comes at a cost. The performance impact of this technique is largely attributed to the state checks required on object dereferences and the read barrier imposed by this. Once again, method specialisation could be used to avoid this problem by implicitly encoding the state of an object in specialised variants, potentially improving performance.

2.2 Distribution APIs

An alternative to the transparent approach of the DJVM concept is the use of an explicit API to control the migration of objects. This can be presented to the programmer at various levels of abstraction.

2.2.1 JavaParty

JavaParty [8] is an API designed for easy distribution of multi-threaded programs across cluster hardware. A `remote` keyword is added to the Java language for describing classes whose instances can be accessed by any node in the cluster. Remote objects therefore migrate around the system as and when required by the application. Migration can occur automatically or manually, allowing the programmer to assign priorities to particular nodes and tune migration in order to improve performance. Similarly, remote objects can be declared as `resident` to prevent them from automatically migrating around the system. Initial object placement is determined by a subclass of the `Distributor` class which can be provided by the user if the default strategy is not suitable. Objects may also be decorated with an `@at n` annotation to specify the node on which they are allocated. A basic program using the JavaParty API is shown below.

[3] http://www.kaffe.org

```java
public remote class HelloWorld {
  public void hello() {
    System.out.println("Hello World from machine " +
                       DistributedRuntime.getMachineID() + "!");
  }

  public static void main(String[] args) {
    for (int n = 0; n < 4; ++n) {
      /*
       * Create a remote object on a node determined
       * by the default ObjectDistributor.
       */
      HelloWorld world = new HelloWorld();
      world.hello();
    }
  }
}
```

Listing 2.1: The *Hello World!* program in JavaParty.

The `HelloWorld` class is declared `remote` so when the `main` method instantiates a number of these objects, JavaParty will automatically distribute them onto idle nodes. Invocation of the `hello()` method occurs on these remote objects and prints out the machine ID for the JVM on which they have been placed. An example trace from this program is show below, however the machine IDs are arbitrary in both order and content.

```
Hello World from machine 0!
Hello World from machine 1!
Hello World from machine 2!
Hello World from machine 3!
```

JavaParty provides a simple API for adapting multi-threaded programs to a distributed environment, allowing the programmer to interfere as much or as little as they wish in the distribution strategy. It does, however, assume the existence of a distributed filesystem, a robust network and does not cater for node failure. Exceptions are not thrown, let alone handled, if such failures do occur.

2.2.2 JavaSpaces

JavaSpaces [9] is an API from Sun Microsystems offering a different take on the problem of program distribution. Internally, JavaSpaces uses the RMI and Object Serialisation features of Java to abstract data movement in the form of a *space*. Put simply, a space is a persistent object repository accessible via a network and acts as a store for serialised objects that can be accessed by a number of nodes using the following primitive operations:

`write` Write a new object into a space, making it accessible to all users of the space.

`take` Retrieve and remove an object from the space in which it resides.

`read` Take a local copy of an object from a space.

`notify` Notify an object (i.e. a user of a space) when an object matching a given query is found in a space.

The idea is to co-ordinate processes by capturing the 'flow of objects' between them in a number of different spaces. Objects migrate *through* spaces in order to arrive at one or more destination nodes, giving the space the roles of a distributed scratchpad and a communication channel. Unlike JavaParty, existing code has to be re-architected to benefit from JavaSpaces technology, restricting language portability and tying an application to the specific API used.

2.2.3 Conclusion

Explicit distribution of objects gives the programmer absolute control over the placement of data and the distribution of execution in the system. Moreover, it allows these parameters to be 'fine-tuned' in order to minimise the amount of migration performed at runtime and increase the performance of the application. Such tuning, however, can easily lead to mistakes and debugging these issues is hampered by virtue of using an API.

The presence of API calls and annotations in application code can limit its readability and quickly obscure the algorithm being implemented. Porting such an application to a different distribution layer then often results in an almost total rewrite of the code despite the end result being identical. The tightly coupled nature of the application and the chosen API also manifests when executing the code on a single node because, despite the absence of object migration, the support libraries must still be present in order for the code to operate correctly. These problems can be solved by implementing implicit object migration at a layer below the application (i.e. in the supporting runtime or as direct language features).

2.3 Object Persistence

Persistent applications are designed to preserve their internal state across program executions by periodically writing their working set to a non-volatile backing store. An object can therefore exist in two possible states: either on disk and awaiting loading (*unfaulted*) or in memory as a potential candidate for persistence (*faulted*). We examine three solutions to managing object persistence in Java and summarise the different approaches taken to object migration.

2.3.1 Orthogonally Persistent Java

OPJ [10] is an implementation of transparent object persistence designed by the same team that worked on dJVM (Section 2.1.2). The primary objectives of OPJ are:

Complete transparency To prevent the programmer from having to declare explicitly persistent objects, OPJ uses `static` variables as persistent roots of a program.

Concurrent issue of transactions ACID transactions are chained together, as described in the *chain-and-spawn* transaction system of [11]. The termination of a transaction atomically allows the next transaction to proceed.

Portability To avoid restricting the portability of Java, OPJ implements a separate class loader (`PersistentClassLoader`) for instantiating persistent objects. The backing store is also represented via a storage interface to hide its implementation details from the runtime.

In order to use persistent objects correctly, read and write barriers are inserted into the Java bytecode to ensure that objects are faulted in from and written to the persistent store when necessary. When objects exist only in the persistent store, they are considered to be in the *unfaulted* state. A major design decision when implementing a persistence mechanism is the method of representing an unfaulted object. The type system of Java imposes further constraints on the structure of unfaulted objects. In [10], two methods of representing unfaulted objects are discussed. These use the notions of *shells* and *façades*.

A shell is simply a default, or 'empty' object whose contents is populated when the object is dereferenced. This involves adding a read barrier to all occurrences of the `getfield` bytecode in order to check the status of the object shell and replace it with its persisted contents if required. A disadvantage of this approach is the high memory usage incurred by the empty shell objects and the overhead of the state checks.

The façade approach uses less memory whilst also removing some of the read barriers associated with the shell mechanism. An object façade appears to the programmer as a normal copy of the object it represents and upon initial dereference, transparently replaces itself with the real object. Like the method specialisation framework of [4], this requires all classes to be fully virtualised in order to trap accesses to them. In order to resolve all external references when faulting in an object from the persistent store, a façade maintains back references to all referring objects. This approach means that the read barrier cost is paid only once (i.e. on the first object access) and can be seen as a special case of object specialisation where a façade is the default specialisation for an object.

OPJ does not support persistence of `transient` data and therefore does not support persistence of threads.

2.3.2 Java Data Objects

JDO [12] is another implementation of object persistence but, unlike OPJ, persistence is explicitly managed by the programmer using an XML schema to describe data in a manner similar to a relational database. JDO alone is merely a specification [4], with a number of different implementations available. It is designed to take on the role of a database in an application whilst remaining independent of the underlying store. Since JDO is designed for use in domains where the opportunity for persistence is made explicit, data transparency is only realised due to the environment and the programmer is still in full control of the persistent store that can be queried using the *JDOQL* query language. A forerunner to JDO is the Java Persistence API available as part of Enterprise JavaBeans.

2.3.3 Enterprise JavaBeans (The Java Persistence API)

The *Enterprise JavaBeans* architecture [13] defines the *Java Persistence API* [14] for persisting *entities* to disk. Like JDO, the system is heavily database oriented, with support for explicit data relationships and a query API. The API operates on Plain Old Java Objects (POJOs), separating the data from the control code. Annotations such as `@Entity`, `@Id`, `@Column` and `@Table` are used to specify object behaviour and purpose. The glue holding this system together is the `EntityManager` class. This allows the programmer to persist classes annotated with the `@Entity` annotation and treat the persistent store as though it were a normal database.

2.3.4 Conclusion

Despite the efforts of JDO and Enterprise JavaBeans, object persistence is still a long way from taking over the role of a database in server applications. Persistence is only a worthwhile asset if it can be implemented transparently, as is the case for OPJ. Transparent persistence of data can be used as a convenient way to back up the working set of a running program in the case of system failure, adding redundancy to an application without the need for explicit checkpoints. The object façades used in OPJ are analogous to specialised objects and could easily be implemented as such.

2.4 In-memory Database Systems

There is a fine line and certainly some overlap between object persistence and in-memory database management. The Java Persistence API (Section 2.3.3), for example, is heavily database oriented. We evaluate a simple database system for loading persisted Java collections into memory and allowing them to be queried and retrieved at high speed.

2.4.1 Space4J

Space4J [15] is an implementation of object persistence in Java designed to replace the use of a database in smaller applications. The API centres around the concept of a persistent, serialisable `Space`. A space can contain Java collections (commonly `java.util.Map`) that can be iterated over

[4]The original specification is online at `http://www.jcp.org/en/jsr/detail?id=12`

using methods on the enclosing space. Operations such as insertion and deletion into maps can be performed using the `Command` class.

Explicit persistence is achieved by invoking the `snapshot` method on a space, causing the space to write its contents to disk. Implicitly, the data on disk is updated asynchronously whenever a command is executed that modifies the contents of a space in some way. If the application crashes during this operation, a log file is used to update the database during the next execution.

```java
public class Directory {
  private Space4J space4j;
  private Space space;

  private static final String MAP_NAME = "directoryEntries";
  private static final String SEQ_NAME = "directorySeq";

  public Directory(Space4J space4j) {
    this.space4j = space4j;
    space4j.start();
    space = space4j.getSpace();
  }

  private int getNextId() {
    return space4j.exec(new IncrementSeqCmd(SEQ_NAME));
  }

  // The Entry class must implement Serializable
  public void addEntry(Entry entry) {
    space4j.exec(new PutCmd(MAP_NAME, getNextId(), entry));
  }

  // Assume name is a primary key
  public Entry findEntry(String name) {
    Entry notFound = null;
    Iterator<Object> iter = space.getIterator(MAP_NAME);

    while (iter.hasNext()) {
      Entry cur = (Entry)iter.next();
      if (cur.name.equals(name))
        return cur;
    }

    return notFound;
  }
}
```

Listing 2.2: An example of the Space4J database system being used to create a directory in which entries can reside (adapted from the `phonebook` example of [15]. Exception code has been omitted for clarity).

A basic example of Space4J is shown in Listing 2.2 and, whilst fairly clunky, the interaction with the object store is very similar to the use of a standard Java collection, such as a `Map`. Space4J also allows a distributed mode of operation in which a master node can be contacted by slave nodes that take a replica of any spaces when they are referenced.

Space4J succeeds in hiding the implementation details of an in-memory database from the programmer but it does not take any measures to abstract the location of the user's data. The explicit use of `Command` objects and snapshots to modify the store requires the programmer to keep the location of their data in mind when using the system. Additionally, explicit state checks in Space4J are avoided internally because upon construction of a `Space4J` object, the *entire* persisted store is read into memory. Partial loading of the store from disk in memory constrained systems would introduce the need for tests upon every object dereference by the `Space` object representing the data. Although these tests would be hidden from the programmer, any performance implications brought with them would almost certainly be noticed.

2.5 Conclusion

A common domain for the use of object migration is that of *service applications* which provide to the programmer a means of managing the movement of objects in a system either explicitly or implicitly. Within these applications, the need to identify the state of an arbitrary object in the system proves to be expensive in both performance terms (as demonstrated by the DJVMs evaluated in Section 2.1) and in terms of programmer effort (shown by the explicit nature of the migration APIs in Section 2.2).

To combat these concerns a common strategy is to change the behaviour of an object depending upon the state which it is in, for example the proxy implementations of cJVM (Section 2.1.1) and the object façades of OPJ (Section 2.3.1) remove the need for explicit tests but provide only an application-specific solution to the problem. Specialised methods provide a general means to affect the behaviour of an object depending upon its state without modifying the application code or extending the semantics of the programming language in which it is implemented.

Chapter 3

Java and the Jikes RVM

The implementation of our specialised methods scheme makes use of three existing bodies of software, the relevant parts of which are explained in this Chapter.

3.1 The Java Runtime

Introduced in 1995 by Sun Microsystems [16], the *Java* programming language was hailed as an *architecturally neutral*, *portable* and *high performance* [17] programming language and was quickly adopted by the programming community. Today, Java is a popular general-purpose language, used in a wide variety of applications [18]. Many systems, including mobile devices, contain implementations of the *Java Virtual Machine* (JVM) on which Java *bytecode* is interpreted. The use of a virtual machine facilitates portability and allows for a higher level of abstraction than native machine languages, such as C++. For example, the Java virtual machine provides a number of different garbage collectors to relieve the programmer from the burden of explicit memory management [19]. Pointers and therefore pointer arithmetic are not made available to the programmer in Java, helping to hide the underlying hardware on which the virtual machine is executing.

Execution of Java bytecode by the virtual machine is performed by interpreting the instructions of a method in the context of a private stack frame. The stack frame for a method consists of the following elements:

Local variable array Method parameters are stored in the local variable array. In the case of a virtual method, the received object is placed at index 0, with the arguments held in subsequent indices. Most bytecode instructions do not affect this data structure, with the notable exception of the `astore n` instruction which writes a value to index *n* of the local variable array.

Operand stack The operand stack forms a scratchpad for the intermediate computational results generated during method execution. The majority of bytecode instructions use the operand stack from which to read parameters and store their results.

Constant pool reference The Java constant pool holds references to all of the classes, interfaces and other constant runtime entities in the system. To avoid compiled classes having to rely on the layout of other structures in the JVM, the Java compiler generates symbolic references where other classes or constants are referenced in the bytecode. These symbolic references are used as indices into the constant pool at runtime in order to resolve them to the relevant data structure.

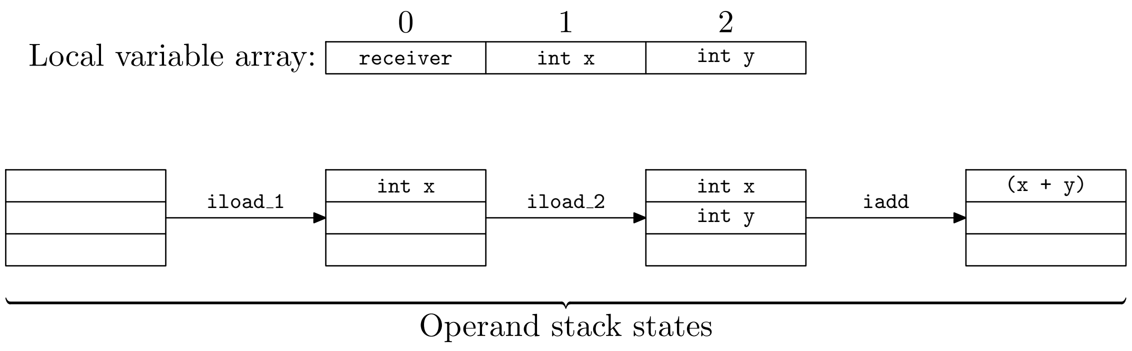
Basic operation of the bytecode interpreter environment is shown in Figure 3.1 and further information about the execution of each bytecode instruction is available in [20].

```java
public int add(int x, int y) {
  return x + y;
}
```

(a) The definition of a simple virtual method in Java which adds its two parameter and returns the result. Note that the receiver object is not used.

```
public int add(int, int) {
  Code:
/* push local_vars[1] (int x) onto the operand stack */
    0: iload_1
/* push local_vars[2] (int y) onto the operand stack */
    1: iload_2
/* Pop the top two entries (x and y) from the operand stack, add them and push the result back. */
    2: iadd
/* Return from the method with the value on top of the operand stack (x + y) */
    3: ireturn
}
```

(b) The bytecode generated by the Java source code from 3.1a. The two method parameters are loaded onto the operand stack from the local variable array before being added together.



(c) A visual representation of the operand stack before and after each bytecode in 3.1b

Figure 3.1: The operation of a bytecode interpreter executing a simple Java method that adds two numbers together.

3.2 The Jikes Research Virtual Machine

Our framework for method specialisation is built on top of the *Jikes Research Virtual Machine* (RVM) [21] [1]. The RVM is an open source implementation of the Java virtual machine and grew out of the *Jalapeño* project before eventually being open-sourced by IBM. It is now targeted as a platform for research projects to build upon. We discuss the design of the RVM, paying particular attention to the aspects of Jikes that are affected by the introduction of specialised methods.

3.2.1 Structure of the RVM

Jikes differs from a standard JVM in two major respects. Firstly, it is itself written in Java and therefore requires some bootstrap code in order to execute without invoking a second virtual machine at runtime. This design decision brings the abstraction afforded by the Java programming language into the JVM itself but complicates initial class loading and garbage collection. Secondly, Jikes does not include a bytecode interpreter, instead opting to compile all bytecode into native machine code prior to execution. To achieve efficient compilation of code at runtime, Jikes includes three separate compilers, each of which can be configured extensively offline.

Baseline The baseline compiler is designed to generate code where correctness is favoured over efficiency. This is useful for debugging other aspects of the RVM, but the generated code is too slow for use in a production environment. The baseline compiler is used to load other compilers dynamically and so is always present in the boot image.

JNI The JNI compiler cannot be explicitly selected by the user as it is designed specially for compiling Java methods marked with the `native` keyword. Native methods require the virtual machine to execute code written for the underlying platform on which the RVM is running. To facilitate this, the JNI compiler generates code to transfer from the virtual machine's internal calling convention to the ABI of the host.

Optimising The optimising compiler can be used to realise a substantial performance increase in the generated code over the baseline compiler and is the preferred Jikes backend for production environments. However, the quality of the output from this compiler doesn't come for free. The optimising compiler is significantly more complicated than the other Jikes compilers and its compilation rate is around 5% of the rate of the baseline compiler. This is due to the transformation of bytecode through three intermediate representations before machine code is eventually emitted. The optimising compiler is discussed further in Section 4.2.4.

To achieve maximum performance, Jikes forks an *Adaptive Optimisation System* (AOS) thread to balance compilation costs against the potential application speedup available from the increased optimisation of generated code. The AOS performs live profiling and runtime estimation [22] in order to recompile 'hot' methods selectively and asynchronously with more aggressive optimisations, thus reducing the time spent compiling seldom used code and focussing the efforts of the optimising compiler on the code that requires it most.

Once a method has been compiled, it is stored as a protected `CodeArray` field in a `CompiledMethod` object that is in turn stored in a static array, allowing other components in the virtual machine to access the compiled code. The life of a compiled method is depicted as a state chart in Figure 3.2.

Initially, a compiled method is in the `uncompiled` state and cannot be executed until it has been *installed* into the relevant data structures following compilation. The optimising compiler in Jikes may apply speculative optimisations during method compilation that can be invalidated by future class loading. If this invalidation occurs, the method transitions to the `obsolete` state and is garbage collected when it no longer exists on the invocation stack of an executing thread. Another possible reason for invalidating a method (that is, moving it into the `obsolete` state) is because the AOS has recompiled it with a higher level of optimisation. Due to the different dispatch procedures

[1] Subversion revision 15227 - 07/12/08.

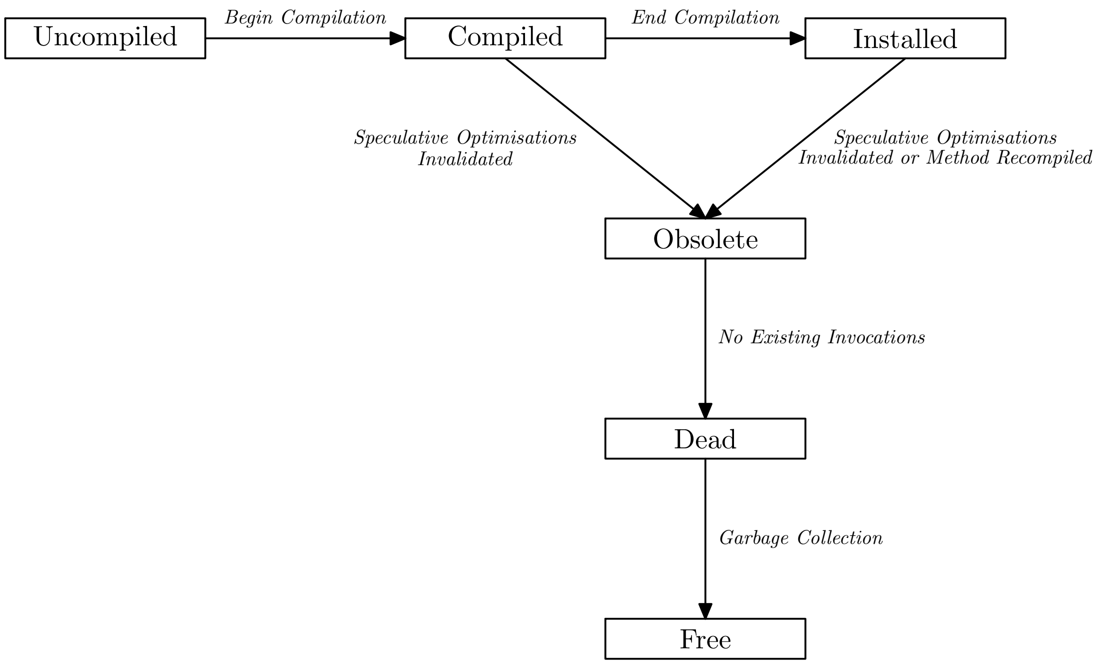


Figure 3.2: The possible states in which a `CompiledMethod` object can exist and the transitions between them.

demanded by `static` and `virtual` methods in Java, compiled method code is stored in one of two data structures.

The *Jikes Table of Contents* (JTOC) is an array of `int` that contains pointers to global, internal data structures as well as static data and pointers to static methods. Jikes accesses the JTOC via a pointer into the middle of the array such that there are 2^{17} entries, or *slots* beneath the pointer and a further $2^{17} - 1$ above it. The slots are addressed using offsets relative to the JTOC pointer and are arranged so that literal values are placed at negative offsets and reference values at positive offsets to allow the garbage collector to distinguish easily between the two.

The JTOC exists at a fixed address so dispatching a static method involves only an indirect *branch-and-link* instruction [2] while accessing a static field directly requires a single indirect move. Figure 3.3d shows the configuration of the JTOC after initialising the `JTOCExample` class listed in Figure 3.3a. The generated assembler code in Figure 3.3c demonstrates the baseline compiler generating static method calls at lines 6 and 14 and static field accesses at lines 3 and 11. The inability of the baseline compiler to perform register allocation for operands means that a value returned from the `getstatic` bytecode is pushed onto the stack rather than moved into a temporary.

The dynamic nature of virtual method dispatch demands a different data structure to hold compiled instance methods that can be accessed on a per-object basis. Objects in Jikes are laid out in memory as an object *header* followed by a *length* field and then the declared fields of the object. In the case that the object is not an array, the first declared field of the object is stored in the slot [3] reserved for the length. In Java there is no such thing as a value object and consequently all objects are accessed via pointers. A pointer to an object in Jikes points to the slot *immediately following* the length field. This has the effect of placing the first declared field of a non-array object at a negative offset from the object pointer. An object header consists of two slots. The first holds a *status* bitmask that is used for locking the object and also for marking it during garbage collection. The other half of the header contains a pointer to the *Type Information Block* (TIB) for the class of the object which is the key to dynamic method dispatch. A TIB contains a wealth of informa-

[2]This corresponds to a `call` instruction on the x86 architecture.

[3]We use the term ‘slot’ instead of ‘word’ to avoid confusion with the native word size of the architecture, which is 16 bits on x86.

```java
public class JTOCExample {
  static int i = 10;
  static String s = "hello";

  static int inc(int i) { return i++; }

  static void print(String s) {
    System.out.println(s);
  }

  public static void main(String[] args) {
    inc(i);
    print(s);
  }
}
```

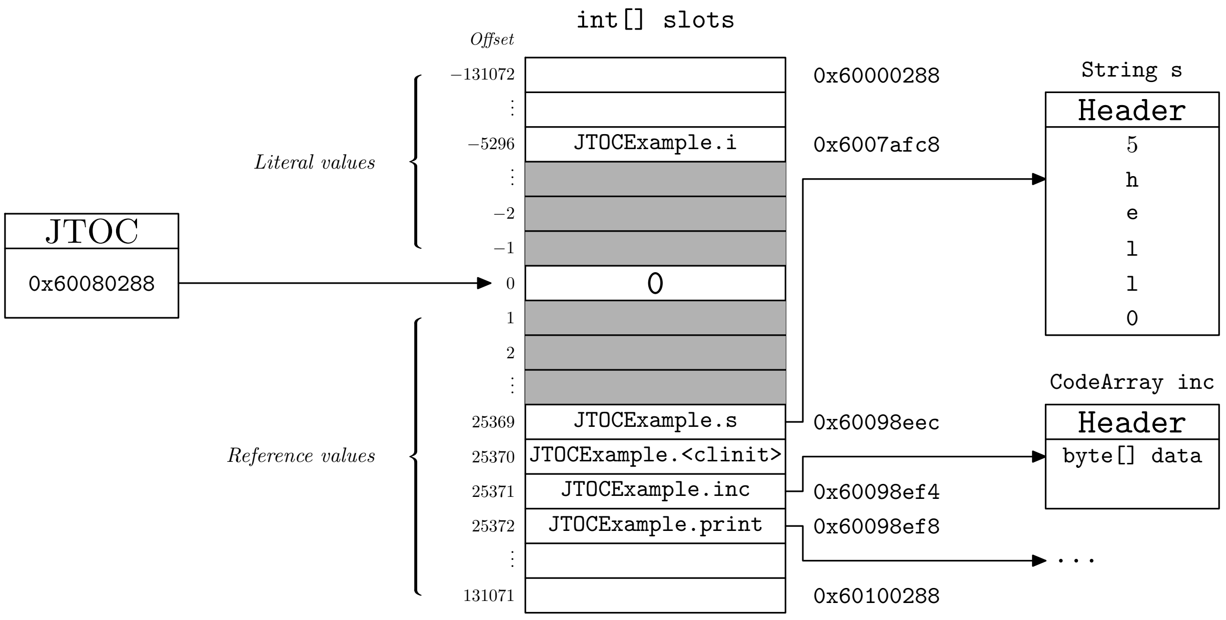
(a) Java source code showing static field access and static method invocation.

```
0:  getstatic     #4; //Field i:I
3:  invokestatic  #5; //Method inc:(I)I
6:  pop
7:  getstatic     #6; //Field s:Ljava/lang/String;
10: invokestatic  #7; //Method print:(Ljava/lang/String;)V
13: return
```

(b) Bytecode corresponding to the main method in 3.3a.

```
;<prologue>
;[0] getstatic < SystemAppCL, LJTOCExample;, i, I >
PUSH        [6007AFC8]
;[3] invokestatic < SystemAppCL, LJTOCExample;, inc, (I)I >
MOV                 EAX           0[ESP]
CALL        [60098EF4]
PUSH                EAX
;[6] pop
POP                 ECX
;[7] getstatic < SystemAppCL, LJTOCExample;, s, Ljava/lang/String; >
PUSH        [60098EEC]
;[10] invokestatic < SystemAppCL, LJTOCExample;, print, (Ljava/lang/String;)V >
MOV                 EAX           0[ESP]
CALL        [60098EF8]
;[13] return
;<epilogue>
```

(c) x86 assembler code generated by the Jikes baseline compiler for the bytecode in 3.3b (prologue and epilogue omitted).



(d) JTOC layout for the code in 3.3c. Shaded slots have been reserved by other classes.

Figure 3.3: An illustration of the mapping between Java source code and the contents of the JTOC on a 32-bit machine. Note that `CodeArray` is an architecture-specific type and becomes an array of primitives at runtime so that it can be branched to directly.

tion about the class that it represents including its position in the class hierarchy, a pointer to the `RVMType` describing the class and information regarding the interface methods that it implements. Most importantly, the TIB holds pointers to all of the virtual methods defined in and inherited by the class it portrays. Figure 3.4 shows a basic Java virtual method invocation and its corresponding bytecode and x86 assembler.

```java
public class TIBExample {
  int i;

  int add(int j) { return i + j; }

  TIBExample(int i) { this.i = i; }

  public static void main(String[] args) {
    TIBExample tibExample = new TIBExample(10);
    tibExample.add(tibExample.i);
  }
}
```

(a) Java source code showing instance field access and virtual method invocation.

```
0:   new             #3;  //class TIBExample
3:   dup
4:   bipush   10
6:   invokespecial #4;  //Method "<init>":(I)V
9:   astore_1
10:  aload_1
11:  aload_1
12:  getfield        #1;  //Field i:I
15:  invokevirtual #5;  //Method add:(I)I
18:  pop
19:  return
```

(b) Bytecode corresponding to the `main` method in 3.4a.

```
 1 ;<prologue and init. tibExample lies on top of stack>
 2 ;[10] aload_1
 3 PUSH                [ESP]
 4 ;[11] aload_1
 5 PUSH               4[ESP]
 6 ;[12] getfield < SystemAppCL, LTIBExample;, i, I >
 7 POP                 ECX
 8 MOV                 EAX          -4[ECX]
 9 PUSH                EAX
10 ;[15] invokevirtual < SystemAppCL, LTIBExample;, add, (I)I >
11 MOV                 EDX           4[ESP]
12 MOV                 ECX         -12[EDX]
13 CALL              72[ECX]
14 PUSH                EAX
15 ;[18] pop
16 POP                 ECX
17 ;[19] return
18 ;<epilogue>
```

(c) x86 assembler code generated by the Jikes baseline compiler for the bytecode in 3.4b (prologue and epilogue omitted).

Figure 3.4: Java source code, bytecode and x86 assembler demonstrating access to instance fields and the invocation of virtual methods in Jikes.

To understand the role of the TIB in virtual method dispatch, we concentrate on the assembler code in Figure 3.4c. Once initialised by the class constructor, a pointer to the new `tibExample` object lies on top of the procedure stack (i.e. at [esp]) at line 1. Lines 3 and 5 push a further two copies of the object pointer onto the stack to take on the role of receiver in the upcoming `getfield` and `invokevirtual` operations. The `getfield` bytecode translates into 3 assembler instructions (lines 7-9) that pop the receiver into `ecx` to use as the base for an indirect load (at an offset of -4, where the length field would reside for an array object) of the target field into `eax`. The resulting value is pushed back onto the stack and forms the parameter for the `invokevirtual` bytecode. All virtual methods are invoked by looking them up in the *virtual method table* of the TIB for the receiver object as illustrated in Figure 3.5. The TIB for the `tibExample` object is loaded into `ecx` at lines 11 and 12 before being used to call the `add` method at offset 72 (slot 18). The reason for the offset being so large is because the VMT is prefixed with methods inherited by the object's superclass - in this case the 11 methods defined in the `Object` class. We can therefore deduce that the TIB contains 2 specialised GC method pointers, that are unrelated to the notion of specialised methods in our framework. This can be confirmed by looking at the implementation of the garbage collector in use; the generational mark-sweep collector used for this example does indeed reserve two slots for these methods.

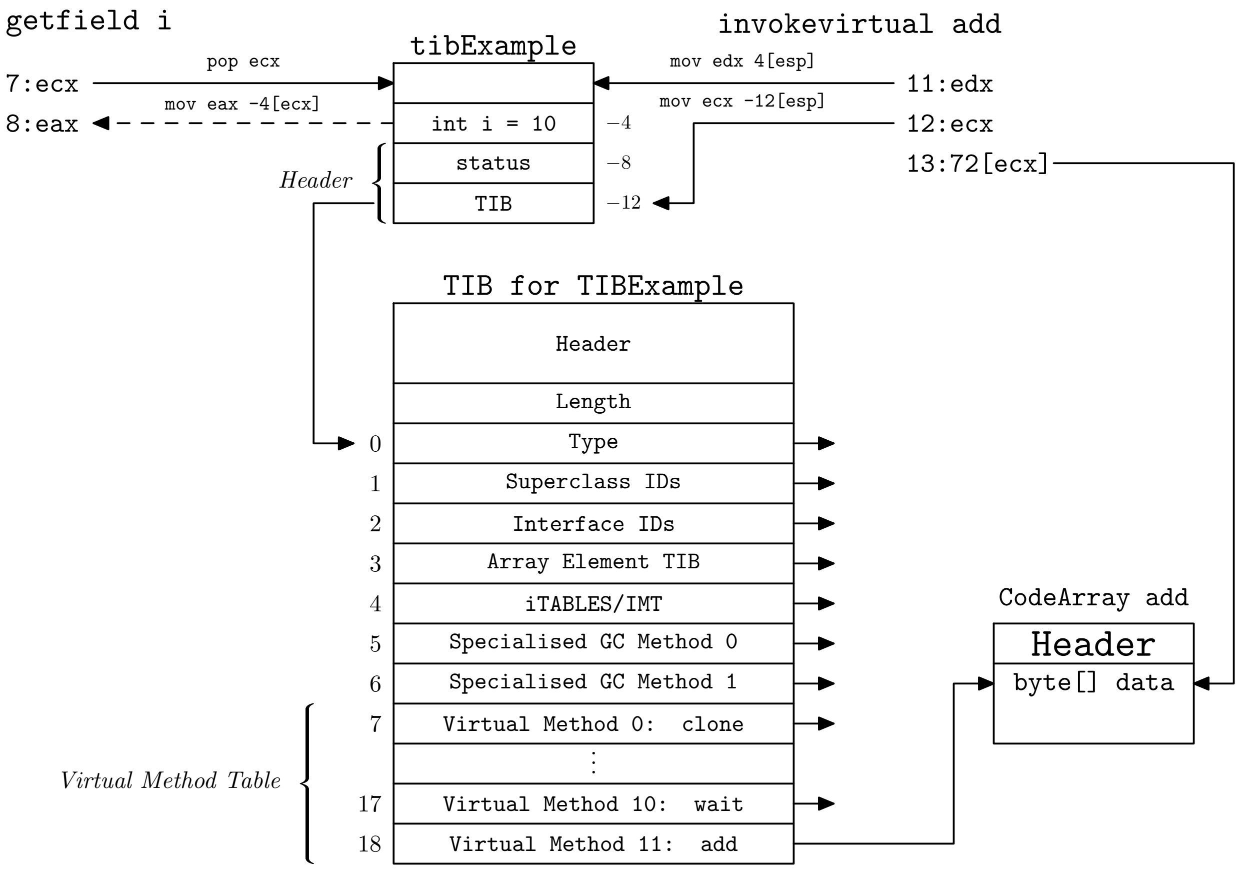


Figure 3.5: The layout of the `tibExample` object from Figure 3.4a on a 32-bit machine. Register operations from Figure 3.4c for `getfield` and `invokevirtual` are illustrated as *line:reg* using solid arrows to represent pointers and dashed arrows to represent data movement.

3.2.2 The Booting Process

Jikes is a *metacircular* virtual machine: it is written in Java and uses Java objects to represent the application that it is executing. This design approach presents a bootstrap problem because Jikes is written in Java and therefore must run on top of another JVM, which would reduce the performance and practicality of the RVM. To remove the need for another JVM at runtime, Jikes uses a piece of *bootstrap* code written in C in order to load Jikes from a *bootimage* created at compile-time [23].

During the build stage of Jikes, its compiled class files are loaded into the system JVM and the 'piggy-backed' RVM is booted into a state where it is capable of performing class loading by itself. At this point, the reflection feature of the Java programming language is used to traverse the heap of the RVM and write to disk the necessary live objects of the bootstrapped RVM. The objects written to disk form a bootimage comprising the *primordial set* of classes necessary for the RVM to boot and initialise the system class loader.

Once built, the RVM is booted by executing the `bootImageRunner` - a comparatively small C program - that registers signal handlers and other low level operations before loading the bootimage into memory and jumping to the *bootrecord* at the beginning of the image. The first Java code to execute is `VM.boot()` that initialises the system, including memory manager, compiler and class loader using the bootrecord callbacks registered by the imagerunner and the primordial classes in the heap.

3.2.3 Class Loading

To avoid polluting the application namespace, Jikes features an internal `BootstrapClassLoader` as well as an `RVMClassLoader` that takes on the role of system class loader. The `BootstrapClassLoader` is used to load classes that form part of the RVM itself and makes calls to the system class loader to avoid duplication of code. During class loading, an `RVMClass` object is created by the static method `RVMClass.readClass` [4] to represent the new class. Class loading is not an atomic operation and advances a class through six states before it can be used.

1. **Vacant:** `readClass` is in the process of parsing the class data including the header and bytecode instructions.
2. **Loaded:** `readClass` has invoked the private constructor for a new `RVMClass` object. Field attributes have been loaded along with any superclasses and interfaces implemented by the class.
3. **Resolved:** Resolution occurs lazily and resolves any superclasses and interfaces implemented by the class as a side-effect. The TIB for the class is allocated and methods are assigned slots in the TIB or the JTOC.
4. **Instantiated:** The TIB and JTOC contain pointers to the methods for the class. These may point to `CodeArray` objects or a `LazyCompilationTrampoline` that compiles the method on-demand. The Superclass is also instantiated.
5. **Initialising:** The class initialiser is executing, assigning values to static fields and executing any static initialisers in the class. The Superclass is initialised.
6. **Initialised:** The class initialiser has completed execution.

3.3 ASM: A Bytecode Engineering Library

To allow for the general specification of specialised object behaviours we require the ability to describe class transformations at the bytecode level. Furthermore, the need to virtualise all object accesses (as described in Section 1.1) is more easily tackled by modifying the bytecode of a class than by performing static analysis at the source level. *ASM* [24] is a bytecode engineering library featuring two APIs for transforming and generating methods and classes that we use in our framework. We discuss only the *visitor* API, that operates in around 75% of the time and uses less memory than the alternative *tree* API.

3.3.1 The Visitor API

As suggested by the name, the visitor API to ASM makes use of the visitor pattern for traversing the contents of a Java class. This brings about the limitation that classes can only be manipulated in the order in which they are visited but helps the library to maintain a small memory footprint. Transformation of a Java class is performed using three primary classes.

ClassReader A `ClassReader` object is instantiated with the bytecode of the target class and provides primitives for accessing its raw representation in a low-level fashion. The `accept` method of a `ClassReader` takes a `ClassVisitor` object (as described in Listing 3.1) which subsequently visits the contents of the class at a higher level of abstraction, allowing the user to visit the fields and methods of the class without any regard for their underlying binary data.

[4] A separate `ClassReader` class is used for class reading in the current HEAD revision of Jikes.

ClassWriter The `ClassWriter` class implements the `ClassVisitor` interface and as such can be used as a parameter to the `accept` method of a `ClassReader`, although this would simply visit the class and write it out directly with no intervening modifications. A `ClassWriter` is usually constructed with a `ClassReader` tied to the target class and a set of flags to indicate whether or not ASM should automatically calculate the *stack map frames* and the *stack sizes* for the class. The stack map frames are a feature of Java 6 and relay the type information of the variables residing in the operand and argument stacks during any point in method execution. Calculating these frames automatically is a slow process (it can make the `ClassWriter` 2x slower) and will fail in the presence of `JSR` and `RET` bytecode instructions in classes compiled prior to Java 6. Unlike stack map frames, stack sizes must be computed in order for the `ClassWriter` to generate a valid class. This adds an overhead of only 10% to the writing time and is guaranteed to succeed.

ClassAdapter Like the `ClassWriter`, the `ClassAdapter` class implements `ClassVisitor` and is the key to performing selective class transformations. The constructor for a `ClassAdapter` takes a single `ClassVisitor` parameter to which it delegates all calls made to it by a `ClassReader`. The usefulness of the adapter is realised by creating a subclass which overrides only the methods that the user is interested in modifying, therefore acting as a filter on the contents of the class undergoing transformation. An example transformation using this common idiom is shown in Listing 3.2 by use of an anonymous class.

```java
public interface ClassVisitor {
  void visit(int version, int access, String name,
           String signature, String superName, String[] interfaces);

  void visitSource(String source, String debug);

  void visitOuterClass(String owner, String name, String desc);

  AnnotationVisitor visitAnnotation(String desc, boolean visible);

  void visitAttribute(Attribute attr);

  void visitInnerClass(String name, String outerName, String innerName, int access);

  FieldVisitor visitField(int access, String name, String desc, String signature, Object value);

  MethodVisitor visitMethod(int access, String name, String desc,
                           String signature, String[] exceptions);

  void visitEnd();
}
```

Listing 3.1: The `ClassVisitor` interface for manipulating classes in ASM.

```
package asmtransform;

public class ExampleTransform extends Transform {
  private boolean isTargetMethod(String cname, String mname) {
    return (cname.equals("Greeter") && mname.equals("printGreeting"));
  }

  /* Modify printGreeting to specialise its receiver */
  public int visitMethodStart(String cname, String mname, String mdesc, MethodVisitor mv) {
    if (isTargetMethod(cname, mname))
      specialiseObject(mv);

    return APPEND_ORIGINAL_METHOD;
  }

  /* Create two specialisations of printGreeting */
  public int getSpecialisationMap(String cname, String mname, String mdesc, int access) {
    if (isTargetMethod(cname, mname))
      return addToBitmap(2, addToBitmap(1, 0));
    return 0;
  }

  /* Create the specialised variants */
  public void visitSpecialisation(String cname, String mname, String mdesc,
                                  MethodVisitor specMethodVisitor, int id) {
    specMethodVisitor.visitFieldInsn(Opcodes.GETSTATIC, Type.getInternalName(System.class),
                                     "out", "Ljava/io/PrintStream;");
    switch(id) {
      case 0:
        specMethodVisitor.visitLdcInsn("Hello Galaxy!");
        break;
      case 1:
        specMethodVisitor.visitLdcInsn("Hello Universe!");
        break;
      default:
        specMethodVisitor.visitLdcInsn("Something has gone horribly wrong!");
    }
    specMethodVisitor.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
                                      Type.getInternalName(PrintStream.class),
                                      "println",
                                      "(Ljava/lang/String;)V");
    specialiseObject(specMethodVisitor);
    specMethodVisitor.visitInsn(Opcodes.RETURN);
  }
}
```

Listing 3.2: A transformation that adds a print statement to the beginning of every method in a class. An anonymous class is used to override the `visitMethod` method of the `ClassAdapter` which indirectly calls the implementation in `ClassWriter` before adding the new code.

The verbosity of an ASM pass is largely due to the amount of ‘boilerplate’ code required before any transformation can occur. In Section 4.4 we describe an API we have developed for specifying method specialisations which requires the user to understand only a subset of the ASM framework and eliminates the need for repetitive initialisation code.

Chapter 4

Design and Implementation

Our method specialisation framework can be broken down into three main components.

1. A modified version of the Jikes RVM to support the existence of specialised method variants and an internal API for managing them during execution. This contribution is based on a reimplementation of the framework described in [4] and forms a basis for the rest of our work.

2. Two 'beanifier' transforms in order to hijack field accesses in an arbitrary class hierarchy.

3. An API for describing class transformations and a modular system for applying these transformations without the need to recompile the JVM.

Moreover, we use our framework to implement a transparent object persistence transform which is described in Section 5.2.2.

4.1 Existing Method Specialisation Environment

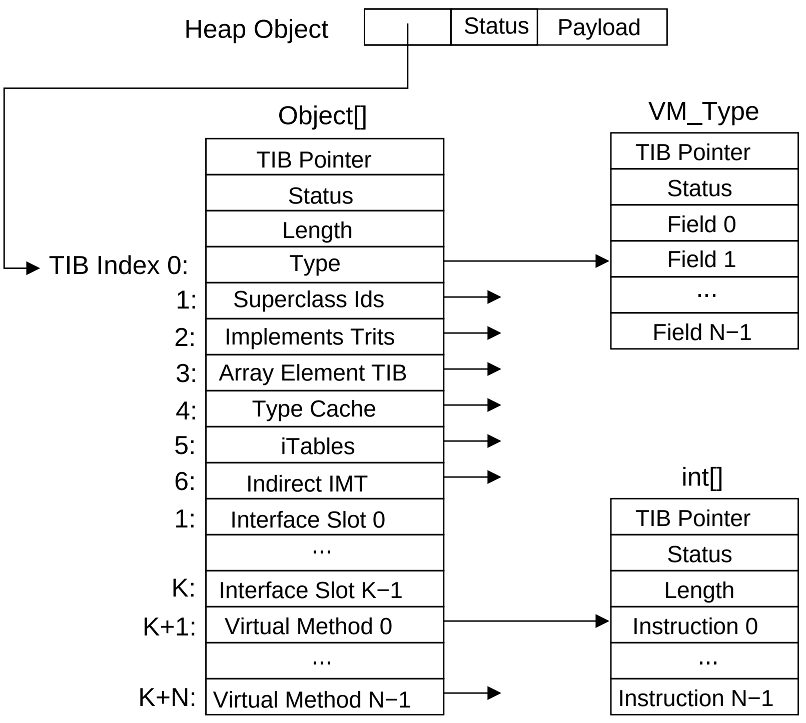
Specialised variants of virtual methods were first proposed and implemented in [4] for the 2.4.3 version of Jikes. The described implementation extends the representation of a class to hold an array of $S + 1$ TIBs for a class with S specialisations. The header of an object remains unchanged and contains a pointer only to the active TIB which in turn contains a pointer to the array of possible TIBs for the class of the object. As shown in Figure 4.1, the layout of a TIB is also extended to include its index into the TIB array and a pointer to the *primary* TIB (the TIB containing unspecialised implementations of the class's methods). The primary TIB pointer is needed to facilitate dynamic type checking of final classes and arrays in the RVM, as illustrated in Figure 4.2.

Switching between TIBs at runtime is performed by a process referred to as *TIB flipping* and is exposed in the RVM object model API as three public static methods:

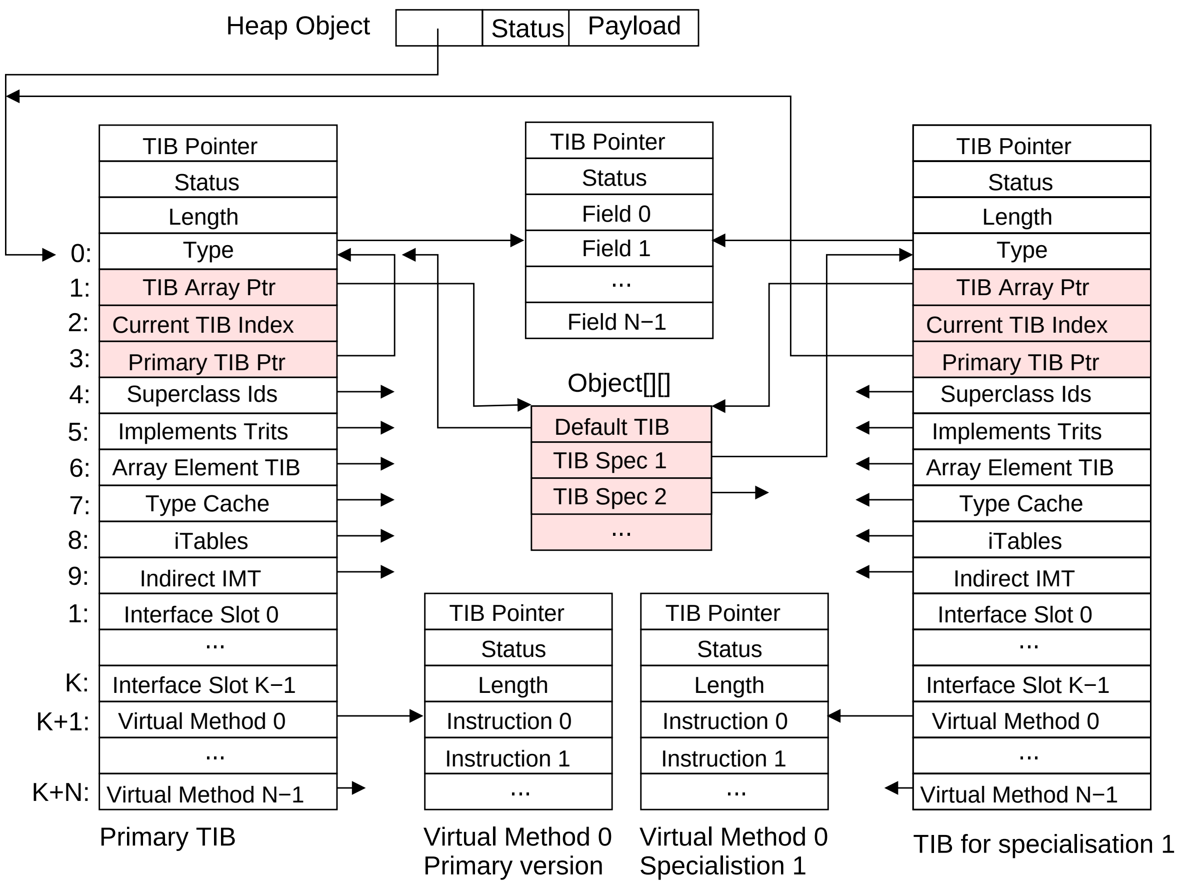
- `Object[] getTIB(Object o, int tibSpecNum)`
  Provides access to the TIB residing at index `tibSpecNum` in the array of TIBs for the class of object `o`.

- `void hijackTIB(Object o, int tibSpecNum)`
  Hijacks the TIB for object `o` by redirecting the TIB pointer in the object header to point to the TIB at index `tibSpecNum` in the array of TIBs for the class of the object.

- `void restoreTIB(Object o)`
  Restores object `o` so that its header points to the primary TIB. This is equivalent to `hijackTIB(o, 0)`.



(a) The normal Jikes 2.4.3 TIB configuration. Each object has a pointer to a single TIB (represented using `Object[]`) accessible via the object's header.



(b) The augmented TIB structure for specialised method dispatch. The object header remains unchanged but the various TIBs are held in an array (`Object[][]`) which can be used to access indirectly all of the object's TIBs from the object itself.

Figure 4.1: The layout of the Type Information Block for Jikes 2.4.3 with and without the framework developed in [4].

```java
class A {}

final class B extends A {}

class DynamicTypeCheck {
  public void check(A a) {
    /* Downcast an A to a B */
    B b = (B)a;
  }
}
```

(a) Java source code involving a downcast which requires a runtime check by the JVM.

```
0:   aload_1
1:   checkcast #2; //class B
4:   astore_2
5:   return
```

(b) The bytecode corresponding to the check method in 4.2a. A `checkcast` instruction has been emitted by the Java compiler.

```
;[1] checkcast < SystemAppCL, LB; >
; Load the parameter (a) into ECX.
000028|          MOV             ECX            [ESP]
; If it's null, we don't bother with the checkcast.
00002B|          TEST            ECX             ECX
00002D|          JZ               10
; It's not null so we load its TIB.
00002F|          MOV             ECX         -12[ECX]
; Since B is final, we perform a pointer comparison with the class TIB accessed via the JTOC.
000032|          CMP             ECX       [60098AC0]
000038|   LIKELY JE                5
; The test failed, we trap to the checkcast exception handler.
00003B|          INT              68
; The test passed, execute remainder of method.
00003D|
```

(c) x86 assembler code generated by the Jikes baseline compiler for the `checkcast` instruction in 4.2b. The `final` decoration of class `B` allows the type check to be performed using a simple pointer comparison.

```
;[1] checkcast < SystemAppCL, LB; >
; Load the parameter (a) into ECX.
000028|          MOV             ECX            [ESP]
; If it's null, we don't bother with the checkcast.
00002B|          TEST            ECX             ECX
00002D|          JZ               13
; It's not null so we load its TIB...
00002F|          MOV             ECX         -12[ECX]
; ...and then its primary TIB from that.
000032|          MOV             ECX          12[ECX]
; Since B is final, we perform a pointer comparison with the primary class TIB accessed via the JTOC.
000035|          CMP             ECX       [60098EF0]
00003B|   LIKELY JE                5
; The test failed, we trap to the checkcast exception handler.
00003E|          INT              68
; The test passed, execute remainder of method.
000040|
```

(d) x86 assembler code generated by the modified Jikes baseline compiler for the `checkcast` instruction in 4.2b. As in 4.2c, a pointer comparison is made but this time between the primary TIB of the object and the class.

Figure 4.2: Dynamic type checking for a `final` class in Jikes. The comparison of TIB pointers presents a problem to the specialisation framework, which may have multiple TIBs for an object.

Users of the framework specify calls to these methods as part of a *BCEL* [25] class transform which serves the purpose of generating and managing specialised methods. Specialisations are described at the bytecode level using the BCEL API and are applied to a class during resolution in the RVM.

Due to the relatively large overheads contributed by BCEL, the framework was ported to Jikes 2.9.2 and BCEL was replaced by ASM but, due to substantial changes made to the RVM in the meantime, only a limited portion of the functionality was preserved. In particular, the optimising compiler and AOS recompilation strategy do not execute correctly in the updated version and the behaviour of the system as a whole can not be considered robust.

4.2 The Updated Framework

Our codebase is a reimplementation of the existing framework using the defective port to Jikes 2.9.2 as a starting point. We seek to achieve the following objectives:

- Port the framework to Jikes 3.0.1, adding support for Java 5 language features, the `x86-64` [1] architecture and general performance improvements.
- Fix problems in the baseline compiler related to dynamic type checking.
- Identify the cause of failed tests relating to reflection and interface method calls and repair the root of the problems.
- Restore the functionality of the AOS thread, which crashed during an early stage of execution.
- Modify the optimising compiler so that it executes correctly and generates meaningful code in the presence of specialised methods.
- Augment the framework to support the specialisation of static as well as virtual methods.

4.2.1 Virtual Methods

We implement virtual method specialisations in much the same way as the previous work. Each `RVMClass` object contains an array of relevant TIBs, each represented using a `TIB` class rather than the previous `Object[]`, which we extend in the same way as in Figure 4.1b. During class parsing, specialised virtual methods (i.e. those methods marked with a `@Specialised` annotation) are separated from other methods in the class and added to a 2D array designed to mirror the final VMT structure which is passed to the `RVMClass` constructor. The TIBs for a class are allocated and placed into the TIB array and JTOC during class resolution, which also installs pointers to the primary TIB and to the array itself. Virtual methods and specialisations are installed into their respective TIBs as part of class instantiation.

To enable the RVM to operate with multiple TIBs for internal classes, the bootimage writing procedure must be modified in order to include TIB arrays in the set of primordial classes. This is achieved by adding the internal name `[Lorg/jikesrvm/objectmodel/TIB;` to the primordials list of the RVM.

The presence of multiple TIBs causes a number of fundamental problems with the RVM. Firstly, invocation of interface methods via the `invokeinterface` bytecode requires a valid *Interface Method Table* (IMT) [2] pointer to be held in each TIB. Additionally, the pointer must be identical for all TIBs of a given class to ensure that the correct method is invoked. This is achieved by creating a single IMT for each class and sharing it between the TIBs by installing multiple pointers to it accordingly. Secondly, care must be taken to ensure that flipping the TIB of a class is an atomic operation otherwise the invocation of a virtual method may dereference an invalid pointer. Atomicity is ensured by decorating the TIB toggling methods with `Uninterruptible` pragmas as shown in Listing 4.1. Finally, some of the optimisations applied by the Jikes compilers are invalidated by the framework

[1] Although only 32-bit code is executed at the current time.

[2] Jikes also supports interface method dispatch via iTables, which follows a similar procedure as far as we are concerned.

and require modifications in order to generate correct code. We discuss the optimising compiler further in Section 4.2.4.

```
@Uninterruptible
@Inline
public static void hijackTIB(Object ref, int tibSpecNum) {
  TIB newTIB = (TIB)((getTIB(ref).getTibArray())[tibSpecNum]);
    if (VM.VerifyAssertions)
      VM._assert(newTIB != null);
  setTIB(ref, newTIB);
}
```

Listing 4.1: The `hijackTIB` method to replace the TIB pointer atomically in the header of an object.

4.2.2 Static Methods

As described in Section 3.2, static methods in the RVM are dispatched via the JTOC. In order to implement specialisations of static methods in a similar manner to that of virtual methods, we first consider their differences:

- Invocation of a virtual method occurs on a per-object basis and specialising an object affects only the method implementation for that specific object instance. Specialisation of static methods, on the other hand, must affect all objects that are instances of the target class.

- Flipping the TIB of an object equates to a basic pointer assignment in the object's header. Static methods reside in the JTOC of the RVM and mimicking the TIB flip approach would require us to maintain a number of JTOCs and swap the JTOC pointer between them when a class specialisation is toggled. This cannot be achieved in Jikes without major disruption to the codebase. The compilers would have to be modified to access the JTOC indirectly via a well known pointer address or register, severely restricting the opportunities for constant folding and direct dispatch (i.e. branch and link) to static methods. Furthermore, flipping the JTOC pointer would halt the entire virtual machine while pointer assignment takes place as the JTOC is shared by all objects (including internal classes) in the system.

- Atomicity of TIB flipping is achieved by ensuring all modifications to the TIB pointer of an object occur in uninterruptible code sections. Whilst JTOC entries can be updated atomically, it is not possible to lock the entire JTOC as this will lead to deadlock inside the RVM.

With these considerations in mind, we describe our approach to static method specialisation.

The class loading operation for specialised static methods is analogous to the procedure described for specialised virtual methods in the previous section. During class resolution, slots are allocated in the JTOC for each unspecialised static method and their specialised variants are allocated the same JTOC offset as their default implementation. Unlike virtual methods, static methods are compiled eagerly and as such we compile all specialised static methods during class instantiation.

In the original RVM, static methods for a class are held in a `staticMethods` field of an `RVMClass` as an array of `RVMMethod`. We add a further dimension to this array and treat the resulting 2D array as an array of static method tables. The table residing at index 0 (`DEFAULT_STATIC_TABLE`) of the array contains all of the unspecialised static method implementations for the class. The remaining tables hold only the methods which differ from their entry in the default table. That is, if a static method has specialised implementations in tables at indices 1 and 3, it will not have an entry in the table at index 2.

To avoid duplicating data in a redundant manner, the specialised static tables (i.e. those with index greater than 0) contain only methods which differ from their entry in the default table. This approach means that the tables are potentially of different lengths and locating the unspecialised implementation of a method can be performed only by a linear search of the default table. Instead, efficient switching between static tables is realised by extending the `RVMMethod` class with an additional two fields: the index into the default static table of the corresponding unspecialised static method and a bitmap representing the tables in which the method has a variant. An example of the layout is given in Figure 4.3.

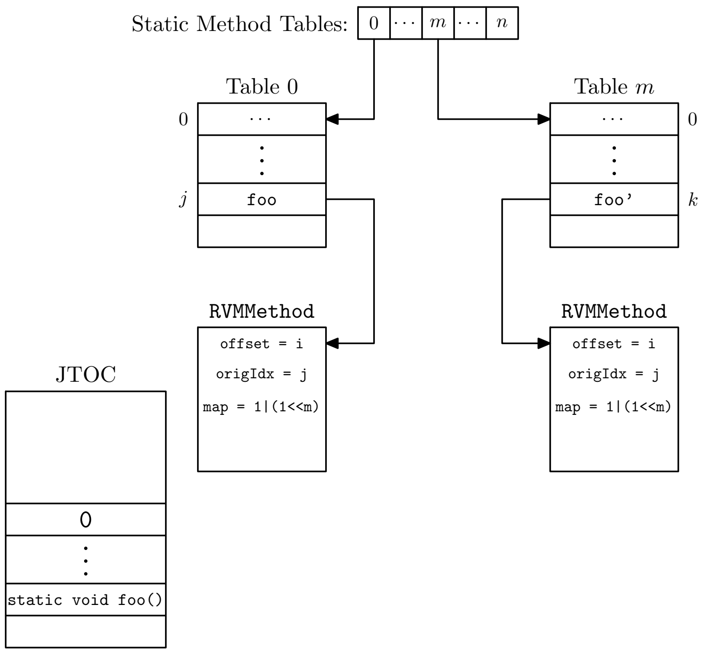


Figure 4.3: The layout of static method tables for a possible class. The default table (table 0) holds all unspecialised methods whilst other tables hold only the methods that differ from their unspecialised variant in that state. Each method is extended to hold the JTOC offset for its unspecialised variant as well as its index into the default table. A bitmap keeps track of the tables in which a method has an implementation.

Toggling between static method tables makes use of the following two internal methods:

`updateMethod(RVMMethod m)` Installs method `m` into the JTOC slot which it has been allocated during class resolution. Specialised methods share the same JTOC offset with their unspecialised variants and so replace the existing entry when they are updated. Calling this method with a parameter that does not belong in the current table causes the call to be silently ignored.

`restoreMethod(RVMMethod m)` Uses the original method index held in `m` to find its unspecialised version in the default table which is then inserted into the JTOC, replacing any specialised implementation that previously resided there. This is analagous to uninstalling a specialised method that was previously installed by a call to `updateMethod`.

We hide these calls behind a `public synchronized void specialiseClass(int table)` method in `RVMClass` which serves as the user entrypoint for managing static tables. Toggling between the static tables for a class is illustrated in Figure 4.4. When moving from an old table to a new table, all of the methods in the new table must be installed into the JTOC (via `updateMethod`) and the methods which are present only in the old table are restored to their unspecialised implementation (via `restoreMethod`). An exception to this rule is when moving to or from the default table, where we can minimise access to the JTOC by making the observation that the largest static table is the default table [3]. Moving *to* the default table is performed by restoring all of the methods in the old table whilst moving *from* the default table updates the methods in the new table but does not need to perform any restorations.

[3]In the worst case, every static method in a class is specialised in all states, leading to an array of tables that are all the same length.

The varying tables lengths and potentially arbitrary ordering of methods within them causes problems when identifying the set of methods to restore during table transition. For each method in the old table, we must perform a linear search of the new table in order to decide whether or not to install its default implementation into the JTOC. To reduce the complexity of this procedure, we traverse the old table once, using the bitmap held in each method to determine whether or not the method is present in the new table. If the bitmap indicates that there is no specialisation in the new table, the method is restored without having to perform any further searching.

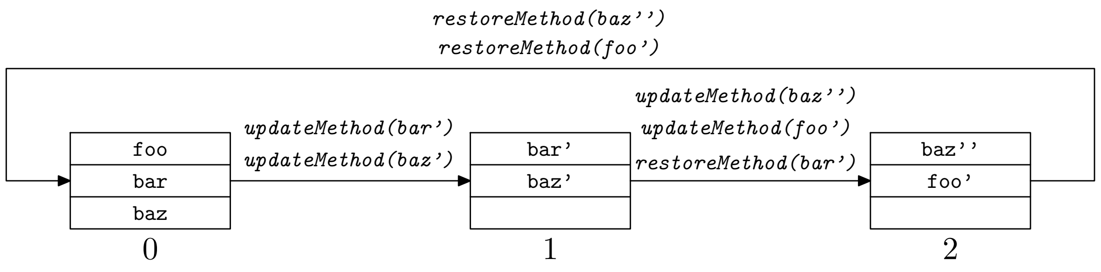


Figure 4.4: An example of the method calls involved in switching between static tables at runtime. Note that it is possible to switch between any two arbitrary tables; we only show one cycle here.

In order to ensure that the active table for a given class is always well-defined in the event of multiple threads changing the specialisation, the entry method is synchronised to maintain consistency. As mentioned previously, the JTOC cannot be locked by a thread as this would result in deadlock within the RVM. A consequence of this is that it is possible to invoke a static method whilst its enclosing class is undergoing a transition between static tables. This scenario can only occur if the programmer specialises a class in one thread whilst executing a static method on the class from another thread. The actual method to be invoked in this situation depends on the interleaving of thread execution but the atomicity guaranteed when replacing the contents of a single JTOC slot ensures that a valid method will always be invoked.

4.2.3 The Baseline Compiler and AOS

The baseline compiler in Jikes maps Java bytecode instructions onto sequences of machine code for the underlying architecture. Instructions are emitted via an `Assembler` object which implements an abstract assembler language for a specific platform. We concern ourselves only with the x86 architecture for the purposes of this report, but PowerPC and x86-64 are also supported internally by Jikes [4].

As described in Section 4.1, dynamic type checking for `final` classes and arrays requires a pointer comparison of the primary TIB for the object with that of the class. We implement this in the baseline compiler by changing the code emitted by `instanceof` and `checkcast` instructions to load the primary TIB for comparison. Specifically, we add a new method to the compiler which loads the primary TIB from an object, as shown below.

```
static void baselineEmitLoadPrimaryTIB(Assembler asm, GPR dest, GPR object) {
  /* Load the TIB for object into the register dest. */
  baselineEmitLoadTIB(asm, dest, object);
  /* Perform an indirect move into dest using TIB_PRIMARY_TIB_INDEX to index the TIB */
  asm.emitMOV_Reg_RegDisp(dest, dest, Offset.fromIntZeroExtend(TIB_PRIMARY_TIB_INDEX << LG_WORDSIZE));
}
```

Listing 4.2: Emitting code to load the primary TIB of an object in the baseline compiler. The current TIB is loaded from the object header and used as a base from which to load the primary TIB.

[4]Although x86 is the main focus of the implementation.

Despite the invasive nature of our framework, no further modifications are required to the baseline compiler. Invoking a virtual method *always* reloads the TIB of the receiver object during dispatch and so any problems associated with caching an out-of-date TIB are avoided.

A further problem caused by the modified TIB structure concerns the AOS recompilation thread in Jikes. When methods are updated with a recompiled version, we must take care to ensure that the correct TIBs are updated with the new code. This is achieved by associating a bitmask with each method to identify the TIBs in which it resides and modifying the `updateVirtualMethod` method in `RVMClass` to update only the relevant TIBs.

4.2.4 The Optimising Compiler

A significant proportion of the Jikes codebase is dedicated to the optimising compiler infrastructure. The compiler is invoked via a `CompilationPlan` for a method, which informs the compiler on how to proceed. Within the plan exists an array of `OptimisationPlanElements`, each of which contains a `perform` method that operates on a supplied intermediate representation (IR) by delegating to a `CompilerPhase` object. It is within subclasses of `CompilerPhase` where the optimising compiler performs operations such as *Global Common Subexpression Elimination* (GCSE), method inlining and tail call elimination in order to generate highly efficient native code.

Rather than generate machine code sequences to implement each bytecode instruction directly as the baseline compiler does, the optimising compiler transforms the bytecode as follows:

1. **Bytecode** is converted into *High-level IR* (HIR) by the `bc2ir` package of the compiler. Any 'magic' instructions (i.e. compiler intrinsics to allow low-level system access which the Java language is unable to provide) that can be represented as machine independent IR are also transformed.

2. **HIR** is initially subjected to transformations of the *Control Flow Graph* (CFG). Splitting of basic blocks in the CFG is performed in order to isolate the common trace from infrequent code paths and loop optimisations such as unrolling and restructuring (e.g. rewriting `while` loops as `until` loops) are also applied to the IR before it is converted into *Static Single-Assignment Form* (SSA) [5]. The SSA code is subjected to further optimisation passes, including constant propagation and load elimination and is then transformed back into HIR which may have further optimisation opportunities as a result of the SSA pass. The IR is then simplified and optimised further before being converted into *Low-level IR* (LIR) by the `hir2lir` package.

3. **LIR** is converted immediately into SSA and manipulated in order to coalesce moves, move loop invariant code out of loops and eliminate common sub-expressions. The SSA code is then returned back to LIR and peephole optimisation of branches occurs as well as overall simplification and further constant propagation. The IR is then transformed into *Machine IR* (MIR) by the `lir2mir` package.

4. **MIR** is a low level IR which makes use of optimisations specific to the native architecture although the main purpose of this representation is to bridge the gap between LIR and machine code. Register allocation is performed on the MIR and a representation of native machine code (MC) is produced by an architecture specific `AssemblerBase` class.

5. **Machine Code** is returned as an architecture-specific subclass of `IR` which contains an `OptCompiledMethod` class that in turn holds relocatable, native code.

We complement the IR (that is, the superclass of the representations mentioned previously) of the compiler with two new operators: `GET_OBJ_PRIMARY_TIB` and `GET_TIB_INDEX_FROM_TIB` which load the primary TIB and the current TIB index from an object respectively. These operators are implemented in both the HIR and LIR passes of the compiler as shown in Listing 4.3.

[5]SSA is a representation of a program where each variable is assigned to exactly once.

```
/*
 * Retrieving the Primary TIB for an object at the HIR level either loads the Class TIB
 * (in the case of a constant operand) or delegates to the LIR.
 */
static Operand getPrimaryTIB(Instruction s, IR ir, Operand obj, Operand guard) {
  if (obj.isObjectConstant()) {
    try {
      RVMType type = obj.getType().resolve();
      return new TIBConstantOperand(type);
    } catch (NoClassDefFoundError e) {
      if (VM.runningVM) throw e;
    }
  }
  RegisterOperand res = ir.regpool.makeTemp(TypeReference.TIB);
  Instruction s2 = GuardedUnary.create(GET_OBJ_PRIMARY_TIB, res, obj, guard);
  s.insertBefore(s2);
  return res.copyD2U();
}

/*
 * Retrieving the Primary TIB for an object at the LIR level loads the current TIB into a
 * register and then performs a load of the Primary TIB using the current TIB as a base.
 */
public void perform(IR ir) {
  Instruction s;
  for (s = ir.firstInstructionInCodeOrder(); s != null; s = s.nextInstructionInCodeOrder()) {
    switch (s.getOpcode()) {
      ...
      case GET_OBJ_PRIMARY_TIB_opcode: {
        Operand objAddress = GuardedUnary.getClearVal(s);

        /* Load current TIB (we create a new HIR instruction) */
        RegisterOperand tib = ir.regpool.makeTemp(TypeReference.TIB);
        tib.setDeclaredType();
        Instruction loadTIB = GuardedUnary.create(GET_OBJ_TIB,
                                                  tib,
                                                  objAddress,
                                                  GuardedUnary.getClearGuard(s));
        s.insertBefore(loadTIB);

        /* Convert the loadTIB instruction into LIR */
        ConvertToLowLevelIR.lowerGET_OBJ_TIB(loadTIB, ir);

        /* Load the primary TIB from the current TIB */
        Load.mutate(s,
                    REF_LOAD,
                    GuardedUnary.getClearResult(s),
                    tib,
                    IRTools.AC(Offset.fromIntZeroExtend(TIB_PRIMARY_TIB_INDEX << LOG_BYTES_IN_ADDRESS)),
                    null,
                    GuardedUnary.getClearGuard(s));
      }
      break;
      ...
    }
  }
}
```

Listing 4.3: Implementation of the `GET_OBJ_PRIMARY_TIB` operator in the HIR and LIR passes of the Jikes optimising compiler.

The dynamic type check implemented for the baseline compiler in Section 4.2.3 is also implemented for the optimising compiler. Dynamic type checking is performed at the HIR level and so we simply replace calls to `getTIB` with calls to our newly implemented `getPrimaryTIB` method for the object being tested.

The *Simplifier* pass of the compiler performs folding of constant operands. If an operand is deemed to be constant (for example, it may be declared `static final` at the source level) then the simplifier will replace it with an `ObjectConstantOperand`. Attempting to retrieve the TIB for such an object will optimise the result into a `TIBConstantOperand` that allows direct access to the TIB at runtime without having to load indirectly via the object's header. Moreover, if the compiler establishes that a receiver object can only be of one possible type at a callsite, then its TIB will be folded into a `TIBConstantOperand` for the inferred type. The folding of TIBs presents us with a problem because the `TIBConstantOperand` that is used may not refer to the correct TIB if the receiver object is specialised. To combat this, we restrict the constant folding of TIBs only to types which we know have no specialised methods. Although this removes some of the benefits of declaring an object `final`, if the programmer chooses to specialise such an object then optimisations that involve folding the TIB are invalidated.

A common optimisation deployed by modern compilers is that of method inlining. Simply put, method inlining replaces a branch instruction to a subroutine with the body of the subroutine itself and in this way avoids the overheads of the `CALL` instruction as well as allowing inter-procedural optimisations to take place more easily. In object-oriented programming, inlining is complicated by the presence of virtual methods. Dispatching to a virtual method transfers execution to a method in a set of candidate methods depending upon the dynamic type of the receiver. If the set of candidate methods contains only one method, the compiler can safely inline that method in the knowledge that the correct code will be executed at runtime. When multiple candidate implementations exist, the compiler can apply heuristics to inline the method which it believes is most likely to be executed and prefix the inlined code with an *inlining guard*.

An inlining guard checks that the inlined code immediately following it is valid for the current receiver object. The performance gained by inlining the target method must outweigh the cost of the guard for this system to pay off. If the guard determines that the code is valid, then execution proceeds normally and the inlined code is executed. The performance hit here is simply the cost of the guard code. If the guard fails however, the inlined method must be invoked in the normal fashion to ensure that the correct implementation is executed. The cost of this operation is the sum of the costs of the guard and the virtual method invocation, demonstrating why it is essential for the compiler to inline the most commonly executed code.

To ensure that specialised methods are inlined correctly, we must emit a new guard to check that the TIB of the receiver matches the TIB in which the inlined method is held. This is achieved by adding a new opcode (`IG_TIB_TEST`) to the HIR of the compiler. The test loads the index of the TIB for the current object and checks it against a bitmap held by the inlined method to identify the TIBs in which it resides (as shown in Figure 4.5). If the index is not present in the map, the guard fails and the method is dispatched via the object TIB, otherwise the inlined method is valid for the current specialisation and is executed. Finally, we modify the inlining oracle to prefix the inlined bodies of potentially specialised virtual methods with a TIB guard. Rather than emit a guard for static methods, we simply avoid inlining them if they have specialisations because the cost of a guard would be greater than the cost of a direct method invocation via the JTOC.

Our modifications to the optimising compiler allow Jikes to produce high performance, correct code in the presence of specialised methods.

```
static void convert(IR ir, OptOptions options) {
  Instruction s;
  for (s = ir.firstInstructionInCodeOrder(); s != null; s = s.nextInstructionInCodeOrder()) {
    switch (s.getOpcode()) {
      ...
      case IG_TIB_TEST_opcode: {
        /* Load the index from the current TIB */
        RegisterOperand tibIndex = getTIBIndex(s, ir, InlineGuard.getClearValue(s),
                                               InlineGuard.getClearGuard(s));
        /*
         * Get hold of the bitmap for the callee.
         * The bitmap is constant at runtime so we create an immediate.
         */
        int TIBMap = InlineGuard.getClearGoal(s).asMethod().getTarget().getTIBMap();
        IntConstantOperand compiledTIBMap = new IntConstantOperand(TIBMap);
        RegisterOperand bitwiseAnd = ir.regpool.makeTemp(TypeReference.Int);

        /* Shift 1 left by the current TIB index */
        Instruction shift = Binary.create(INT_SHL, tibIndex, new IntConstantOperand(1), tibIndex);
        /* Lookup the index in the bitmap */
        Instruction and = Binary.create(INT_AND, bitwiseAnd, compiledTIBMap, tibIndex);

        s.insertBefore(shift);
        s.insertBefore(and);

        /* Is the index in the bitmap? */
        IfCmp.mutate(s, INT_IFCMP, null, bitwiseAnd, tibIndex, ConditionOperand.NOT_EQUAL(),
                     InlineGuard.getClearTarget(s), InlineGuard.getClearBranchProfile(s));
        break;
      }
      ...
    }
  }
}
```

(a) Generating an IG_TIB_TEST inlining guard in the HIR pass of the Jikes optimising compiler.

```
; Load receiver from the JTOC into EAX
MOV          EAX     [6009A324]
; Load its TIB into EAX
MOV          EAX        -12[EAX]
; Load the TIB index array into EAX
MOV          EAX          8[EAX]
; Load the index from the array into EAX
MOV          EAX           [EAX]
; Take a copy into ECX
MOV          ECX           EAX
MOV          EAX             1
; EAX = 1 << TIBIndex
SHL          EAX           ECX
MOV          EDX           EAX
; Bitwise AND of the method TIBMap (1) and (1 << TIBIndex)
AND          EDX             1
; If the Index is in the Map, then the result of the AND will be the same as the index.
CMP          EDX           EAX
```

(b) x86 assembler code generated by the Jikes optimising compiler for the guard described in 4.5a. Notice that the bulk of the instructions making up the guard are used for loading the index from the current TIB. Furthermore, because the TIB can hold only reference values, the index is held in an array of length 1 and therefore requires two loads to access it.

Figure 4.5: The IG_TIB_TEST inlining guard. 8 additional instructions, only 2 of which access memory, are added to the instruction stream. A normal virtual invocation would result in loading the method parameters (including the receiver) into registers before performing an indirect CALL instruction. Similarly, the prologue and epilogue code of the callee may need to save registers onto the stack so as to avoid overwriting data belonging to the caller.

4.3 Class Beanification

Realising transparent object migration using specialised methods requires us to capture all object accesses to ensure that an object cannot be used when it exists in an invalid state. Specifically, we must redirect all object field accesses via method invocations in order to ensure that the receiver is placed into a valid state before its fields made available. This process is referred to as *beanification* because the generation of virtual `getter` and `setter` methods is one of the requirements laid down by the JavaBeans API specification [26].

To avoid the need for modification of application code at the source level, we attempt to beanify user classes automatically and transparently by applying suitable ASM transforms during class loading. The general pattern for beanifying a class is to override the `visitFieldInsn` method of the ASM `methodVisitor` class in order to rewrite `GET/PUTFIELD` and `GET/PUTSTATIC` bytecodes as `INVOKEVIRTUAL` and `INVOKESTATIC` bytecodes respectively. We also override the `visitEnd` method of the ASM `classVisitor` class so that the `getter` and `setter` methods for the fields declared in the target class are generated accordingly. However, the limited view of the application class hierarchy available during class loading introduces some situations where we cannot beanify a class without assistance from the programmer. In these cases we should issue at warning at class loading time indicating the problematic field and class including instructions on how the user can rectify the problem. A simple solution in these cases is for the programmer to beanify the class in question manually, by adding appropriately named `getter` and `setter` methods.

4.3.1 Virtual Methods

Replacing accesses to instance fields with virtual method invocations is achieved by generating methods of the form `_getv_<variable>` and `_setv_<variable>` for each variable within a class. Direct field accesses from other classes are then replaced with a call to the appropriate method which is dispatched based on the dynamic type of the receiver. Due to the differences in the ways in which Java treats method overriding and field hiding, certain class hierarchies cannot be beanified correctly (an example is shown in Listing 4.4).

There are a number of possible solutions to the problem associated with overridden accessor methods. Perhaps the most straightforward approach is to introduce name mangling for the methods so that the name of the enclosing class forms part of the method name, for example the `_getv_x` method in class A of Listing 4.4 would become `_getv_A_x`. The problem with this technique is that in order to generate a call to such a method, the name of the enclosing class must be known at the time at which the call is generated. ASM provides us only with the *dynamic* type of the receiver when visiting a field instruction, which in the case of our example may be `A` or `B` when accessing the definition of `x` in `A`. Furthermore, the transformation takes place in the context of class loading and as such class `A` may not have been loaded by the virtual machine. This makes it impossible to identify the class which declares the field `x` for an object of dynamic type `B` when generating code for class `B` itself. It is possible to invoke the class loader recursively but in the presence of circular class references the RVM will livelock (although we discuss this opportunity further in Section 4.3.2).

A second solution is to dispatch to the accessor methods in a non-virtual manner using the `invokespecial` bytecode. Although this guarantees execution of the correct method, it prevents specialisation of the accessor methods because they are not dispatched via the object's TIB and therefore this approach defies the purpose of generating such methods in the first place. Another consequence of the semantics of `invokespecial` is that `private` methods cannot be specialised since they are dispatched in a non-virtual fashion.

A potentially more expensive solution (in terms of classloading cost) is to apply the ASM transforms *after* class loading has occurred. In this way, all superclasses for a given class will also have been loading into the RVM, providing a complete view of the hierarchy above the class. Alternatively, the Java source code could be preprocessed prior to compilation and beanification could occur in a mechanical fashion, although this is a more invasive approach and requires access to the application source code.

```
 1 class A {
 2   int x;
 3   int _getv_x() {
 4     return x;
 5   }
 6 }
 7
 8 class B extends A {
 9   void print() {
10     System.out.println(x + "," + _getv_x());
11   }
12 }
13
14 class C extends B {
15   int x;
16   int _getv_x() {
17     return x;
18   }
19 }
20
21 public class BeanBreaker {
22   public static void main(String[] args) {
23     C c = new C();
24     /* Write to the x in class A */
25     ((A)c).x = 10;
26     /* Write to the x in class C */
27     c.x = 20;
28     /* Invoke the print method in class B */
29     c.print();
30   }
31 }
```

Listing 4.4: An example of a class hierarchy where beanification of instance fields changes the semantics of the program. The invocation on line 29 prints out the `x` field in class `A` followed by the `x` field in class `C` because the `getter` method in `A` is overridden by the method in class `C`. The output of this code is `10,20`.

Our approach to beanification of virtual field accesses is to generate an `INVOKEVIRTUAL` instruction for each access, despite the limitations described above. Further development should issue a warning to the programmer during class loading if an erroneous invocation is generated. The programmer can then provide their own accessor methods for the troublesome field if required.

A further problem we encounter is when an application class extends a system class, for example a `PrettyPrintStream` may extend the `PrintStream` class from the `java.io` package. Continuing with this example, the `PrettyPrintStream` class will likely access the `out` field of its superclass, which will be transformed into an invocation of the `_getv_out` method. The metacircular nature of Jikes means that the `PrintStream` class will have been loaded during creation of the bootimage. We must therefore ensure that the beanifier transforms these system classes during the build process. To prevent virtualising field accesses within the RVM itself [6], we mark internal classes as exempt and, although we generate accessor methods anyway, we do not redirect internal field accesses to invoke them.

Class beanification inevitably has an impact on performance. To minimise the cost of invoking accessor methods, we mark all generated code with the `Inline` pragma in order to force the methods to be inlined into their caller bodies. We investigate the performance penalty of our approach in Section 5.1.1.

[6]Although this may be required, it leads to problems such as the ones already mentioned and can invalidate annotations held by the caller method. For example, if a method marked `@Uninterruptible` calls an accessor method, then the accessor must have the same annotation to ensure that the code is indeed uninterruptible.

4.3.2 Static Methods

Static field access is handled in a similar manner to the access of instance fields. We generate an INVOKESTATIC instruction which dispatches to the correct accessor method. The cost of static method invocation is low as the method resides at a well known address within the JTOC and dispatch is simply an indirect jump.

The difficulty of generating static accessor methods occurs in relation to static final fields. In Java, static final fields are interpreted as constant values and they are treated as such by the virtual machine and by the Java compiler, as shown in Figure 4.6. A further complication with static final fields is that they can exist within an interface, preventing the generation of accessor methods. We take a conservative approach to static final fields by simply avoiding them altogether. If we can deduce that a static field is declared final, we do not generate accessor methods for it and we similarly avoid generating calls to such methods. The latter part of this statement requires us to determine whether or not a field accessed by a method is final or not. Although the Java compiler performs constant folding of static final literals, static final objects are accessed using the normal procedure (i.e. via GET/PUTSTATIC bytecodes).

```java
public class Final {
  public static final int CONSTANT = 42;

  public static void main(String[] args) {
    System.out.println(CONSTANT);
  }
}
```

(a) Java source code utilising a public static final field.

```
1 0:   getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
2 3:   bipush  42
3 5:   invokevirtual #3; //Method java/io/PrintStream.println:(I)V
4 8:   return
```

(b) Bytecode generated by the Sun Java compiler for the source code from 4.6a.

Figure 4.6: Constant folding of a static final field performed by the Java compiler. Line 2 of the bytecode in 4.6b shows the CONSTANT field replaced with its literal value.

In order to determine whether or not a field is final, we end up with the same problem we had in Section 4.3.1 of trying to determine the declaring class of an arbitrary field. We implement a scheme whereby we recursively invoke the class loader for the interfaces and superclasses of the class undergoing transformation in order to perform a breadth-first search of the hierarchy. Using the reflection features provided by Java, we search each class for the field we are testing and, when we eventually locate it, we check to see if it is decorated with the final modifier. To avoid circular class loading within our algorithm, we maintain a cache of explored classes and we also place any fields which we inspect into a static cache that is shared across all transformation instances. Unfortunately, our algorithm can result in failed class loading if Jikes attempts to load a superclass as a consequence of loading one of its subclasses. In the case that we have already forced partial loading of the superclass, this will result in loading the class twice which is a fatal error in the RVM. In such a scenario, we can print an error message to the programmer informing them that they should create their own static accessor methods for the field in question.

To establish the overheads imposed by beanifying a class, we investigate the effects that the process has on a variety of benchmarks in Section 5.1.

4.4 Transform Application

Like the beanification transforms, user transforms are specified using the ASM framework. Typically, a user transform involves two operations:

1. Generation of specialised variants of existing methods. Specialised methods commonly append or prepend instructions to an existing method in a class. The added code may change the state of the object as discussed in Section 1.1 and possibly toggle to another specialisation.

2. Modification of existing methods in a class. In some circumstances, it may be necessary to modify the existing methods in a class so that they toggle the specialisation of the receiver object (or class) during execution. This operation is less common, however, as it is normally specialised methods which toggle the specialisation of an object.

4.4.1 The Transform API

To remove the burden of writing repetitive, boilerplate ASM code and to abstract the means by which specialisations are managed, we provide a simple API to implement the two transformation operations described above. The `LowLevelTransform` class declares an abstract `apply` method which is invoked by the Jikes class loader during class loading, passing an `InputStream` tied to the target class as a parameter. Implementing general transforms (that is, transforms which have no knowledge of the classes and methods which they are to operate on) is best performed at this level and the beanifier transforms discussed previously extend the `LowLevelTransform` class directly to reflect this. Furthermore, the `LowLevelTransform` class provides a number of helper methods to generate code for switching easily between specialisations of objects and classes. Transforms written for a specific application, however, need not implement the `apply` method. Instead, a `Transform` class, which is itself an abstract subclass of `LowLevelTransform`, defines the `apply` method in terms of four simpler methods as shown in Figure 4.7.

We demonstrate the use of our API with an example in Figure 4.8. In the `ExampleTransform` class, we implement the `visitMethodStart` method so that all methods are left unmodified apart from `printGreeting`, to which we prepend code to toggle the TIB for the receiver. We inform the `Transform` class that we wish to implement two specialisations for the `printGreeting` method by returning the value 3 (0b11) from the `getSpecialisationMap` method. The specialisations are implemented in the `visitSpecialisation` method where we switch upon the specialisation ID in order to determine the correct string to load. Although the transform still consists of a reasonable amount of code, its superclasses contribute over five times more.

4.4.2 Runtime Selection of Transforms

In order for the Jikes class loader to instantiate and apply class transforms to application classes, the transforms must be present in the bootimage of the RVM (as described in Section 3.2.2). The tightly-coupled nature of class transforms with the virtual machine requires the entire RVM to be re-compiled if a transform is modified, adding a significant overhead to the turnaround time for the transform developer. To remove this static coupling, we make use of the `BootstrapClassLoader` class within Jikes to load the transform classes dynamically at runtime. Since the bootstrap class loader is not used to load application code, it does not transform classes as it loads them so we avoid the problem of recursively applying transforms to themselves.

Compiling and installing transforms separately from the RVM can be achieved using a new `install-transforms` build target which places the resulting class files alongside the Jikes class files. If Jikes is not built already, the `install-transforms` target compiles it prior to installing the new class files. The classpath for the bootstrap class loader is separate from the application class path and we augment it to include an additional internal directory in which the transforms are installed. Some transforms are required to modify the internal classes of Jikes (the beanifier

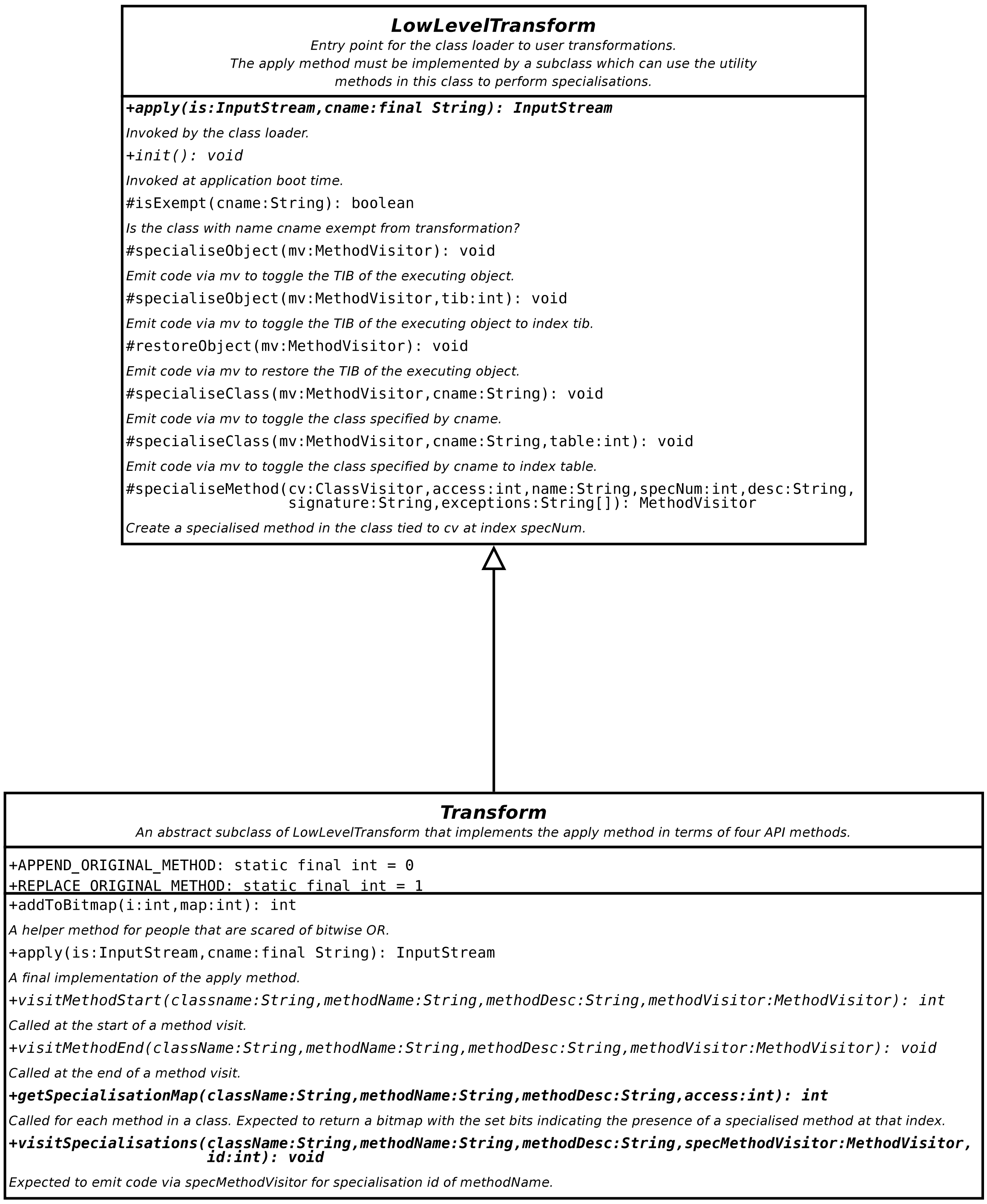

LowLevelTransform
Entry point for the class loader to user transformations.
The apply method must be implemented by a subclass which can use the utility methods in this class to perform specialisations.
+apply(is:InputStream,cname:final String): InputStream
Invoked by the class loader.
+init(): void
Invoked at application boot time.
#isExempt(cname:String): boolean
Is the class with name cname exempt from transformation?
#specialiseObject(mv:MethodVisitor): void
Emit code via mv to toggle the TIB of the executing object.
#specialiseObject(mv:MethodVisitor,tib:int): void
Emit code via mv to toggle the TIB of the executing object to index tib.
#restoreObject(mv:MethodVisitor): void
Emit code via mv to restore the TIB of the executing object.
#specialiseClass(mv:MethodVisitor,cname:String): void
Emit code via mv to toggle the class specified by cname.
#specialiseClass(mv:MethodVisitor,cname:String,table:int): void
Emit code via mv to toggle the class specified by cname to index table.
#specialiseMethod(cv:ClassVisitor,access:int,name:String,specNum:int,desc:String, signature:String,exceptions:String[]): MethodVisitor
Create a specialised method in the class tied to cv at index specNum.
Transform
An abstract subclass of LowLevelTransform that implements the apply method in terms of four API methods.
+APPEND_ORIGINAL_METHOD: static final int = 0
+REPLACE_ORIGINAL_METHOD: static final int = 1
+addToBitmap(i:int,map:int): int
A helper method for people that are scared of bitwise OR.
+apply(is:InputStream,cname:final String): InputStream
A final implementation of the apply method.
+visitMethodStart(classname:String,methodName:String,methodDesc:String,methodVisitor:MethodVisitor): int
Called at the start of a method visit.
+visitMethodEnd(className:String,methodName:String,methodDesc:String,methodVisitor:MethodVisitor): void
Called at the end of a method visit.
+getSpecialisationMap(className:String,methodName:String,methodDesc:String,access:int): int
Called for each method in a class. Expected to return a bitmap with the set bits indicating the presence of a specialised method at that index.
+visitSpecialisations(className:String,methodName:String,methodDesc:String,specMethodVisitor:MethodVisitor, id:int): void
Expected to emit code via specMethodVisitor for specialisation id of methodName.

Figure 4.7: The `LowLevelTransform` and `Transform` classes for class transformation and specialisation.

```java
public class Greeter {
  private void printGreeting() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    Greeter greeter = new Greeter();
    for (int i = 0; i < 4; ++i)
      greeter.printGreeting();
  }
}
```

(a) A simple class to emit a 'Hello World!' message 4 times.

```
Hello World!
Hello Galaxy!
Hello Universe!
Hello World!
```

(b) The output from 4.8a when transformed using 4.8c.

```java
package asmtransform;

public class ExampleTransform extends Transform {
  private boolean isTargetMethod(String cname, String mname) {
    return (cname.equals("Greeter") && mname.equals("printGreeting"));
  }

  /* Modify printGreeting to specialise its receiver */
  public int visitMethodStart(String cname, String mname, String mdesc, MethodVisitor mv) {
    if (isTargetMethod(cname, mname))
      specialiseObject(mv);

    return APPEND_ORIGINAL_METHOD;
  }

  /* Create two specialisations of printGreeting */
  public int getSpecialisationMap(String cname, String mname, String mdesc, int access) {
    if (isTargetMethod(cname, mname))
      return addToBitmap(2, addToBitmap(1, 0));
    return 0;
  }

  /* Create the specialised variants */
  public void visitSpecialisation(String cname, String mname, String mdesc,
                                  MethodVisitor specMethodVisitor, int id) {
    specMethodVisitor.visitFieldInsn(Opcodes.GETSTATIC, Type.getInternalName(System.class),
                                     "out", "Ljava/io/PrintStream;");
    switch(id) {
      case 0:
        specMethodVisitor.visitLdcInsn("Hello Galaxy!");
        break;
      case 1:
        specMethodVisitor.visitLdcInsn("Hello Universe!");
        break;
      default:
        specMethodVisitor.visitLdcInsn("Something has gone horribly wrong!");
    }
    specMethodVisitor.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
                                      Type.getInternalName(PrintStream.class),
                                      "println",
                                      "(Ljava/lang/String;)V");
    specialiseObject(specMethodVisitor);
    specMethodVisitor.visitInsn(Opcodes.RETURN);
  }
}
```

(c) A transform to create 2 specialised versions of the `printGreeting` method from 4.8a, each printing a modified method. All versions of the method toggle the object specialisation to the next one.

Figure 4.8: A transform and its corresponding target class. The output of the class after transformation is show in 4.8b.

transforms, for example) and therefore must remain hard-coded into the RVM in order to transform classes during generation of the boot image.

Runtime transforms are specified to the RVM at boot time as a command line option, `-xforms=<file>`. The contents of the file parameter must be the name of each transform to apply, each on a separate line and prefixed with '`asmtransform.`'. When the virtual machine is booted, the application class loader (`RVMClassLoader`) parses the file and loads the transforms into an array of `LowLevelTransform`. Loading of an application class with this class loader creates a new instance of each transform and the stream representing the bytecode of the class is passed through each transform in turn. The reason we must create a new transform instance for each class is because class loading can occur in parallel as a consequence of the user creating their own subclasses of the system class loader. If a transform relies on internal state to perform its transformation, this may become corrupted in the presence of multiple threads attempting to transform different classes simultaneously. Another solution to this problem is to synchronise class loading for the entire RVM, but the performance repercussions of such a move would mitigate the benefits of parallel class loading altogether.

Chapter 5

Evaluation

The nature of our work demands two separate approaches to its evaluation. Using standard Java benchmark suites, we quantify the performance loss incurred by class beanification and our modified RVM. We attempt to gain an insight into the cost of toggling the TIB of an object and also of switching the statics table for a class. The other half of our evaluation concerns the usability of the software. Specifically, we look at a transform for an incremental garbage collector and show how a concise representation of the problem can be achieved and implemented using our framework.

5.1 Benchmarks

The Jikes RVM ships with an extensive testing harness that can be used to benchmark the virtual machine. We run a selection of benchmarks from the DaCapo [27] [1] and SPECjvm2008 [28] suites in order to investigate the performance impact of our framework on a vanilla build of the RVM. Our experiments are conducted on a 2.7Ghz AMD Phenom X4 9950 quad-core system with 8GB RAM, a 512KB 16-way set associative level-2 cache per core and a 2MB 32-way set associative shared level-3 cache, a 64KB 2-way set associative instruction cache per core and a 64KB 2-way set associative level-1 data cache with 64-byte lines. The operating system is a 64-bit build of Gentoo Linux running a custom built 2.6.29-gentoo-r5 kernel in single user mode.

5.1.1 Virtualisation Costs

Class beanification imposes runtime overheads by introducing method invocations in place of field accesses. The AOS component of Jikes optimises and recompiles methods during execution and will gradually inline the generated accessor methods while the application is running. To cater for this effect, we measure the *initial* (init) and *converged* (conv) times for each benchmark run. The initial time for a given benchmark is the average time taken for the benchmark to complete over a number of separate runs, each run taking place in a fresh invocation of the RVM. For our experiments we take an average over five runs. In this way, we include the 'warmup' time (that is, the time taken for the optimisation and recompilation of methods) in our final results and place an upper bound on the execution time. To obtain a lower bound (the converged time), we run iterations of each benchmark in the same instance of the virtual machine, thus allowing the optimisations applied to the methods in a given iteration to benefit the performance of future iterations. Furthermore, for the converged case we initially compile all methods with the optimising compiler and then discard the time taken to complete the first iteration. Our converged timings were taken over fifty iterations of each benchmark as we are generally interested in long-running applications and services.

We benchmark the following configurations of the RVM:

O - The original build of Jikes (r15227) with no modifications, apart from a bug-fix that is essential for correct benchmark execution.

[1] Version 2006-10-MR2.

M - Our modified version of the RVM with no transforms being applied in the class loader. This configuration helps us to determine our static overheads as a result of creating additional objects on the heap and complicating the class loading process.

MB - Configuration M with the beanifier for instance fields active.

MS - Configuration M with the beanifier for static fields active.

MBS - Configuration M with both beanifier transforms active.

Results obtained from the DaCapo benchmarks are tabulated in Table 5.1. We record the percentage overheads of each configuration in relation to the vanilla RVM (configuration O). It is important to note that when a converged time has a larger percentage overhead than an initial time, this does not mean that the program executed more slowly. For example, with the beanifier active (configuration MB), the overheads sustained by the bloat benchmark were 6.29% and 8.69% for the initial and converged times respectively. This means that the respective initial and converged times were 9.80 and 8.63 seconds. The converged time is still smaller than the initial time, but the increased percentage overhead indicates that the benchmark wasn't as amenable to optimisation as it was in configuration O, perhaps as a consequence of the addition of virtual method invocations by the beanifier.

Benchmark	O (s)		% Overhead							
			M		MB		MS		MBS	
	Init	Conv	Init	Conv	Init	Conv	Init	Conv	Init	Conv
antlr	2.90	1.64	-1.72%	0.00%	32.1%	14.6%	-2.76%	0.61%	35.9%	17.1%
bloat	9.22	7.94	0.542%	0.756%	6.29%	8.69%	-2.17%	-0.252%	9.54%	2.02%
fop	2.11	1.14	-0.948%	0.877%	18.5%	7.02%	4.74%	2.63%	26.5%	7.02%
hsqldb	3.96	2.02	3.54%	1.49%	31.1%	12.9%	1.26%	2.97%	39.1%	10.4%
jython	9.19	5.72	2.18%	1.05%	21.9%	-2.45%	4.35%	-0.524%	21.9%	-2.97%
luindex	10.1	8.50	-0.990%	-0.118%	18.8%	16.7%	-1.98%	2.24%	19.8%	14.9%
lusearch	3.23	1.64	-0.310%	4.88%	13.0%	15.9%	-3.41%	0.610%	12.4%	17.7%
xalan	3.51	2.09	7.12%	5.26%	40.2%	13.9%	11.4%	4.78%	44.4%	13.4%
Min	2.11	1.14	-1.72%	-0.118%	6.29%	-2.45%	-3.41%	-0.524%	9.54%	-2.97%
Max	10.1	8.50	7.12%	5.26%	40.2%	16.7%	11.4%	4.78%	44.4%	17.7%
Arith. mean	5.53	3.84	1.18%	1.77%	22.7%	10.9%	1.43%	1.63%	26.2%	9.95%
Mean (no xalan / lusearch)			0.434%	0.676%	21.4%	9.58%	0.573%	1.28%	25.5%	8.08%

Table 5.1: DaCapo Benchmark results for five configurations of the RVM. Application of both beanification transforms adds, on average, less than 10% overhead to the converged execution time.

An immediate observation from our results is that the overheads introduced by the static beanifier are generally negligible. We can attribute this to the small proportion of variables that are declared static in an application and also to the efficient nature of the `INVOKESTATIC` bytecode, which doesn't need to take into account dynamic dispatch. We can also see that our modified RVM affects the performance of some benchmarks even with no transform active (although the average overhead is less than 2%). This is due to the (constant) overheads which we introduce to the class loading procedure in Jikes. If a benchmark contains a large number of classes, these overheads will form a larger proportion of the overall execution time. The initial performance of the xalan transform is unusual, running over 5% slower in our framework with no active transforms. We investigated the performance of this benchmark and found that, outside of the Jikes test framework, the overheads reported were significantly lower (the overheads for configuration M dropped to around 3%). Therefore, we ascribe the anomaly to the Jikes benchmarking framework - this requires further investigation. It is possible that the heap size specified by the framework is simply too small and the garbage collector is adding to the performance overhead, but this remains to be confirmed. Despite the poor initial performance of xalan, it converges well under the optimising compiler.

Another curious result is the overhead associated with the lusearch benchmark. With the exception of the MS configuration, lusearch consistently incurs a larger overhead for the converged run than the initial run. Moreover, these overheads are typically larger than the ones experienced by the other benchmarks. Upon investigation of this problem we found that the garbage collection behaviour of the benchmark (which searches a large body of data using a number of threads) to be erratic. We ran five runs of the benchmark, each run consisting of five iterations, and profiled the collector. The number of collections for each run varied dramatically between 789 and 1540 collections. The standard deviation was calculated as 335 collections, as opposed to the xalan benchmark which had a standard deviation of 3.74!

The average overhead of full program virtualisation (the MBS configuration) is less than 10% for the DaCapo benchmarks. This can be reduced by creating a unique name for each accessor method across the entire class hierarchy and marking each method with the final keyword, thereby allowing the optimising compiler to inline the generated methods without a guard to check the dynamic type of the receiver. Creating unique method names requires class transformation to be delayed until after class loading has occurred, as described in Section 4.3.1. Preliminary experiments show that the average overhead of the fully virtualised DaCapo benchmarks can be reduced to around 5% using this technique.

The short running time of the DaCapo benchmarks exaggerates the impact of constant overheads on the results. The SPECjvm2008 results shown in Table 5.2 were obtained from runs which were allowed to execute for up to half an hour. Note that due to the limitations of the instance field beanifier described in Section 4.3.1, the generated accessor methods for SPECjvm2008 have to use the `INVOKESPECIAL` bytecode instead of `INVOKEVIRTUAL`. The generated (beanified) code therefore contains one fewer indirect move instructions (to load the object TIB for virtual invocation) per field access than for the DaCapo benchmarks.

The overheads of the static beanifier for the SPECjvm2008 benchmarks echo the findings from the DaCapo suite, namely that they incur very little performance penalty. Additionally, the effect on performance of our framework with no transforms active is also negligible, as the constant class loading overheads contribute a tiny amount to the total execution time. Of particular interest is the compress benchmark. The results of compress in configurations M and MS do not look out of line; although the overhead in MS is reported to be less than in M, it is below 0.3% and such differences are considered to be in the noise. The surprise comes when we look at the overheads caused by the presence of the beanifier for instance fields. With the instance beanifier active, compress consistently performs over 20% *faster* than it does for the vanilla build of Jikes. On the face of it, this seems to be nonsensical. How can the redirection of field accesses via virtual methods possibly reduce the execution time? We repeated the compress benchmarks using only the baseline compiler and with the AOS thread disabled. We found that without the presence of the optimising compiler, configuration MB experienced an overhead of around 12% [2]. From this result, we can determine that the speedup experienced from the beanifier is due to the optimising compiler realising an optimisation that it would otherwise not apply. Unfortunately, we have not managed to identify the nature of this optimisation or whether it is a valid operation on the standard compress benchmark.

Even with the omission of the compress benchmark from our results, the overheads experienced by the SPECjvm2008 suite are significantly lower than those for DaCapo, with full object virtualisation adding an average overhead of less than 2% to the converged performance of the application.

[2]This cannot be compared to the other results in the table as the AOS system was disabled.

Benchmark	O (ops/m)		% Overhead							
			M		MB		MS		MBS	
	Init	Conv	Init	Conv	Init	Conv	Init	Conv	Init	Conv
compiler.compiler	188	197	0.532%	-0.508%	3.72%	3.05%	0.00%	0.508%	5.85%	3.05%
compiler.sunflow	67.3	70.4	-0.446%	-0.71%	3.12%	2.56%	-0.446%	0.852%	5.2%	2.13%
compress	71.0	72.0	-1.27%	0.278%	-22.0%	-21.5%	0.423%	0.139%	-21.7%	-21.4%
mpegaudio	34.6	34.9	0.289%	0.00%	-0.289%	-0.573%	0.578%	0.287%	0.289%	-1.15%
Min	34.6	34.9	-1.27%	-0.71%	-22.0%	-21.5%	-0.446%	0.139%	-21.7%	-21.4%
Max	188	197	0.532%	0.278%	3.72%	3.05%	0.578%	0.852%	5.85%	3.05%
Arith. mean	90.2	93.6	-0.224%	-0.235	-3.86%	-4.12%	-0.150%	0.447%	-2.59%	-5.41%
Arith. mean (no compress)			0.125%	-0.406	2.18%	1.68%	0.0440%	0.549%	3.78%	1.34%

Table 5.2: SPECjvm2008 Benchmarks

5.1.2 Specialisation Costs

To get an idea of the costs associated with flipping between specialisations of classes and objects, we carried out a simple experiment. We wrote a Java program that performs `Integer.MAX_VALUE` iterations of a loop, whose body consists of a single call to an empty method (see Listing 5.1). We take timings at the application level (using `System.nanoTime()`) to avoid measuring the cost of class loading and initialisation. We then applied a transform to the program that makes a specialised variant of the empty method (also with an empty body). Finally, we inserted code into both method variants to toggle specialisation of the object or class according to the test. The time taken for an iteration to complete therefore indicates the cost of a single toggling of a specialisation plus the cost of any extra code inserted by the compiler, for example the addition of an inlining guard into the loop body.

```java
public class TimeTIBFlip {
  public static final long ITERATIONS = Integer.MAX_VALUE;
  public void flipMe() {};

  public static void main(String[] args) {
    TimeTIBFlip flipper = new TimeTIBFlip();
    for (long i = 0; i < ITERATIONS; ++i)
      flipper.flipMe();
  }
}

public class TimeStaticsFlip {
  public static final long ITERATIONS = Integer.MAX_VALUE;
  public static void flipMe() {};

  public static void main(String[] args) {
    for (long i = 0; i < ITERATIONS; ++i)
      TimeStaticsFlip.flipMe();
  }
}
```

Listing 5.1: Code to estimate the time taken to flip the TIB of an object and the static method table of a class. We measure the time taken to complete the loop and then calculate the average time to complete each iteration.

From the results in Table 5.3, we can see that flipping of class specialisations is around five times slower than flipping of object specialisations. This is due to the extra work involved in patching up the JTOC for a class when compared to a single pointer assignment in the case of an object. Additionally, class specialisation is synchronized and a lock must be acquired for flipping, although in this example it is always uncontended.

	Object Specialisation (TIB)		Class Specialisation (Table)	
	Base	Specialised	Base	Specialised
	2.42	20.2	2.41	98.9
	2.42	20.0	2.47	99.2
	2.42	20.0	2.49	109
	2.42	20.1	2.48	109
	2.48	20.1	2.41	109
Geo. mean:	2.43	20.1	2.45	105

Table 5.3: Converged per-iteration times (in nanoseconds) recorded from our experiment. The base times are recorded with the empty `flipMe` method whereas the specialised times insert code into `flipMe` to toggle the active specialisation.

5.1.3 Functional Testing

To ensure that our work does not introduce new bugs in the RVM, we make use of the extensive test suite contained within Jikes. The Jikes community requires that the `pre-commit` test run passes before any modifications can be committed to the code repository. Our framework passes all of these tests (which execute for both a baseline build and an optimised build) even with our beanification transforms present [3].

Testing the correctness of method specialisations is performed using a series of toy examples which we have produced. These programs range from simple toggling of specialisations to dynamic type checks of specialised objects and also methods which are designed to follow a particular path through the optimising compiler. As an example of the latter, we place invocation of specialised methods in loops to persuade the compiler to inline the body and emit a TIB guard.

5.2 Applications

We motivate the practicality and the usability of our framework by means of two examples; an implicit read barrier for an incremental garbage collector and a transparent implementation of object persistence for Java applications.

5.2.1 Read Barrier

As described in [4], method specialisation can be used to implement the incremental Baker garbage collector [29]. The Baker collector is an incremental, semi-space, copying collector and divides the heap into two sections; *from-space* and *to-space*. An invariant is established that the *mutator* (that is, the running application) can access only objects residing in to-space and therefore new objects are always allocated there.

The garbage collector is activated when a memory allocation fails due to lack of available memory in to-space. The collector pauses the mutator and flips the two space pointers so that the new to-space now points to the old from-space (which is free of any live objects). Before resuming the mutator, the collector *evacuates* the set of directly accessible live objects (known as the root-set) from from-space to to-space and *scavenges* a fixed number of objects taken from the scavenge queue.

Evacuation of an object places a copy into to-space and installs a *forwarding pointer* in place of the original object. Additionally, the new object is added to the back of the scavenge queue for processing. The forwarding pointer is used to maintain a link between an old object and

[3]The serialization test fails, but only because we introduce new accessor methods and therefore the serialized data does not match the expected output.

its current version residing in to-space. Note that an evacuated object may still contain references to objects located in from-space as its reference fields are not copied out during evacuation.

Scavenging an object is the process of evacuating its immediate references. Objects that require scavenging are placed onto the scavenge queue which is served during each memory allocation requested by the mutator.

The incremental nature of the collector means that it is possible for the mutator to reference objects using a pointer into from-space (for instance, via a field in an unscavenged object). To ensure that the mutator only accesses objects residing in to-space, an *eager read barrier* [30] [4] is used to check the target location for all reference load instructions. If an object is referenced using a pointer into from-space, the read barrier evacuates the target object and returns a pointer to its new location in to-space, thereby preventing the mutator from dereferencing a from-space pointer. The basic read barrier operation is shown in Listing 5.2 and an illustration of the heap layout is shown in Figure 5.1.

```
ObjectReference loadObjectReference(Address a) {

  if (COLLECTOR_ACTIVE && a.inFromSpace()) {
     a.evacuate();
  }

  return a.toObjectReference();
}
```

Listing 5.2: The read barrier described in Baker's algorithm for incremental garbage collection as it might be implemented in the Jikes RVM.

The presence of the read barrier introduces a state test to every object dereference and can prove to be very expensive. For example, the *Blackburn-Hosking* read barrier described in [1] adds between 5-40% in execution time across a range of applications on x86 architectures. We seek to reduce these overheads by making the barrier *implicit* within each object, thereby removing the need to capture all reference loads statically. Specifically, we arrange for objects to exist in one of two specialisations: a default state, whereby dereferencing the object results in normal program behaviour and a *self-scavenging* state, where dereferencing the object causes it to scavenge itself (evacuating its immediate references into to-space and flipping their specialisations), flip its specialisation to the default variant and then proceed with the original dereference. In the context of our framework, a dereference can only occur via a method invocation as we beanify all field accesses. The trick is to place objects into the self-scavenging state only when they exist in to-space and are yet to be scavenged by the collector. We achieve this by augmenting the collector's `copy` routines with calls to specialise objects as they are copied. An example of a transform to implement our implicit read barrier for an incremental collector is shown in Listing 5.3. This experiment implements the idea described in [4] but replaces the BCEL layer with our enhanced ASM-based transformation framework. A pleasing aspect of our solution is the significant reduction in the amount of code required to implement the transform.

To investigate the overheads incurred by our approach, we modify the copy routines for the default *stop-the-world* collector (GenMS) used in Jikes. The absence of an incremental collector in the RVM prevents us from implementing our approach directly, so we instead determine bounds on the mutator overhead of the read barrier using the methodology described in [4]. The lower bound occurs when the read barrier does not perform any scavenging and all of the copying is performed by the collector itself. In this case, we must still flip the specialisation of each object twice; once during evacuation and again during scavenging. The upper bound is achieved in the opposite case. That is, all objects are scavenged by the read barrier as a result of mutator access and the mutator

[4]As opposed to a *lazy* barrier that allows the mutator to reference objects in both from-space and to-space, cleaning up bad references at the end of each cycle.

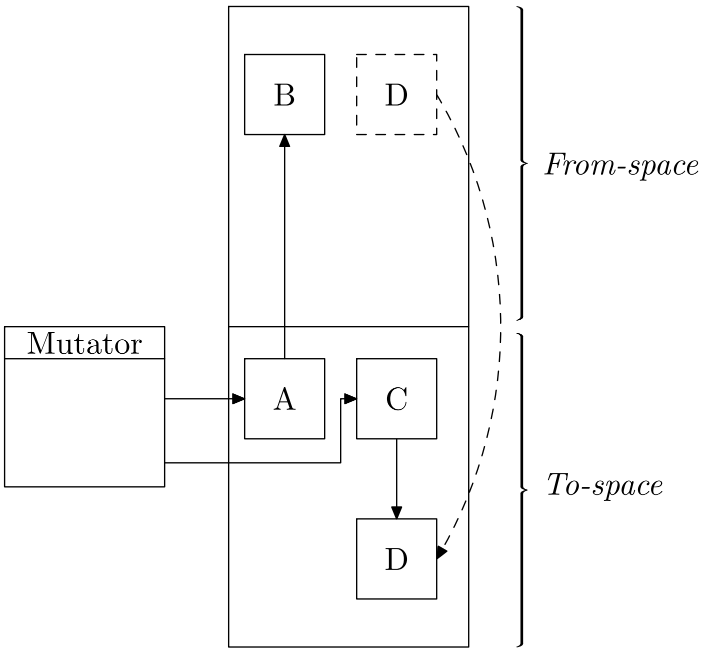


Figure 5.1: A possible configuration of the heap during an active collection of an incremental Baker collector. The mutator sees only objects residing in *to-space* and attempting to dereference a location in *from-space* (for example, B, via A) causes the read-barrier to evacuate the object and install a forwarding pointer in its place. In the diagram, D has been evacuated and the dashed line from *from-space* represents its forwarding pointer.

performs no memory allocation (as memory allocation would give the collector a chance to service the scavenge queue).

We can formulate our bounds in terms of operational cost. The lower bound consists of the virtualisation cost and two specialisation flips. The virtualisation cost for the read barrier is lower than the beanifier costs measured in Section 5.1.1 because we do not need to virtualise field accesses occurring within private and protected methods as the receiver will already have been scavenged. The upper bound is equal to the lower bound plus the cost of a virtual method invocation (namely the invocation of the default variant of the specialised method that caused scavenging to occur).

```java
public class IncrGCTransform extends Transform {
  /* We don't hijack all methods */
  private boolean isExcluded(int access) {
    return ((access & Opcodes.ACC_STATIC) != 0) ||
           ((access & Opcodes.ACC_PRIVATE) != 0) ||
           ((access & Opcodes.ACC_PROTECTED) != 0);
  }

  public int getSpecialisationMap(String cname, String mname, String mdesc, int access) {
    if (isExcluded(access))
      return 0;
    else
      return 1;
  }

  public void visitSpecialisation(String cname, String mname, String mdesc,
                                  MethodVisitor specMethodVisitor, int id) {
    /* Restore the object's TIB */
    restoreObject(specMethodVisitor);

    /* Invoke the scavenger method */
    specMethodVisitor.visitVarInsn(Opcodes.ALOAD, 0);
    specMethodVisitor.visitMethodInsn(Opcodes.INVOKESTATIC,
                                      Type.getInternalName(ObjectModel.class),
                                      "scavenge",
                                      "(Ljava/lang/Object;)V");

    /* Construct the call to the non-specialised version */
    Type[] argTypes = Type.getArgumentTypes(mdesc);
    for (int i = 0; i <= argTypes.length; ++i) {
      specMethodVisitor.visitVarInsn(Opcodes.ALOAD, i);
    }
    specMethodVisitor.visitMethodInsn(Opcodes.INVOKEVIRTUAL, cname, mname, mdesc);

    /* Return the result from the original method */
    specMethodVisitor.visitInsn(Type.getType(mdesc).getOpcode(Opcodes.IRETURN));
  }
}
```

Listing 5.3: A self-scavenging specialisation transform for an incremental, copying garbage collector.

To obtain values for the bounds of the read barrier, we modify our version of the RVM by placing the relevant operations (two TIB flips and a potential call to a virtual method, which we mark with the `@NoInline` pragma) into the copy routines of the GenMS collector. Additionally, we apply two transforms to all classes which are loaded by the class loader, including internal VM classes and those classes loaded during creation of the bootimage. The first transform is a modified version of the beanifier that virtualises all direct field accesses occurring within methods that are not static, private or protected. The second transform generates a specialised variant of each method which performs the same operation as the original code. This is required in order to force the optimising compiler to emit the TIB guard when inlining the accessor methods, as it would need to in the case of the incremental collector. Table 5.4 details our results.

The overheads of our implicit read barrier compare favourably to the results published for the Blackburn-Hosking read barrier in [1]. Moreover, the ease in which the barrier can be correctly implemented using our framework also allows for easy modification of the barrier's behaviour and provides a method for removing the barrier, if need be, with minimal disruption to the RVM.

5.2.2 Persistence

As described in Section 2.3.1, *Orthogonally Persistence Java* (OPJ) implements transparent object persistence by treating static variables as the implicit roots of persistence. To demonstrate our framework, we implement a primitive version of the object *façades* used in OPJ and show how they

Benchmark	Lower Bound	Upper Bound
antlr	15.2%	18.9%
bloat	6.38%	8.13%
fop	9.56%	9.57%
hsqldb	15.1%	15.6%
jython	0.347%	0.347%
luindex	9.54%	19.0%
lusearch	8.14%	9.88%
xalan	9.09%	13.2%
Min	0.347%	0.347%
Max	15.2%	19.0%
Geo. mean	6.56%	8.16%

Table 5.4: Reported overheads over configuration M for the upper and lower bounds on our implicit read barrier implementation. All numbers represent the results of converged runs.

can be used to realise data persistence in Java applications. The approach taken is to capture all static field accesses made within an object and, on the first search access, mark the *class* of the object (as opposed to the object itself) as a persistence candidate. At the end of execution, we walk across the candidates, persisting their static, non-transient fields.

Persistence of objects in Java is made easier due to the *serialisation* API. The `ObjectOutputStream` class in the `java.io` package provides a `writeObject(Object)` method which abstracts the details of object serialization from the programmer and writes out its parameter to disk. Reading a serialised object from disk into memory is the exact counterpart of the write operation, using an `ObjectInputStream` to read in the data. In order for these methods to operate on an object, the object in question must implement the `Serializable` interface. This interface does not require any methods to be implemented and serves as a marker indicating that it is safe to serialise a given class. Serialisation of an object (as occurs in the `writeObject` method) disregards static fields and fields marked as `transient`. Transient fields are simply fields that cannot or should not be serialised, for example a `Thread` object cannot be sensibly serialised as it contains handles on operating system state.

Our persistence implementation utilises the static field beanifier described in Section 4.3.2 to capture all direct accesses to static fields by the application. Once the application has been beanified, we apply a second `PersistenceTransform` transform that performs the following operations:

1. The transform modifies the target class so that it implements the `Serializable` interface if it doesn't already do so.

2. It then prepends all return statements in the `main` method with code to persist any classes identified as persistent.

3. The accessor methods generated by the static field beanifier are specialised.

We specialise each accessor method slightly differently:

Getter: The default variant of a getter method for a static field is invoked when the application first accesses the field. The method body is modified to attempt to load the field (actually, the class of the invoking object, which, due to beanification, must be the class that declares the field) in from disk which, in the absence of a persisted store, fails silently. Somewhat counter-intuitively, we must then mark the invoking class as a candidate for persistence. This is because a later `PUTFIELD` bytecode could modify an instance field of the static object and we would not be aware of it, so we eagerly mark the class now. In the case that it is not written, we simply end up persisting the same value as we faulted in.

Before executing the GETSTATIC bytecode of the getter method, we toggle the specialisation of the invoking class so that we do not attempt to fault it in again on the next access to one of its static fields.

Setter: The default variant of a setter method for a static field is invoked if the application attempts to write to the field before reading it. In this case, we simply mark the class of the invoking object for persistence and toggle its specialisation.

The PersistenceTransform is listed in its entirety in Listing 5.4.

```java
public class PersistenceTransform extends LowLevelTransform {
  private static List<Class> persistentClasses;

  /*
   * Register a class for persistence.
   */
  public synchronized static void registerPersistentClass(Class clazz) {
    persistentClasses.add(clazz);
  }

  public static void persistClasses() {
    Iterator<Class> it = persistentClasses.iterator();
    while (it.hasNext())
      persistClass(it.next());
  }

  /*
   * Persist static fields of a class in their entirety.
   */
  public static void persistClass(Class clazz) {
    Field[] fields = null;

    try {
      fields = clazz.getDeclaredFields();
    } catch (SecurityException e) {
      System.out.println("Failed to access class fields for " + clazz + ": " + e);
    }

    for (Field field : fields) {
      if (Modifier.isStatic(field.getModifiers()) &&
          !Modifier.isTransient(field.getModifiers())) {
        persistField(field, clazz.getName());
      }
    }
  }

  /*
   * Restore the static fields of a class from their persisted state.
   */
  public static void restoreClass(Class clazz) {
    Field[] fields = null;

    try {
      fields = clazz.getDeclaredFields();
    } catch (SecurityException e) {
      System.out.println("Failed to access class fields for " + clazz + ": " + e);
    }

    for (Field field : fields) {
      if (Modifier.isStatic(field.getModifiers())) {
        restoreField(field, clazz.getName());
      }
    }
  }

  /* Helper methods for persisting / restoring a field */
  private static void persistField(Field field, String cname) {
    FileOutputStream fos;
    ObjectOutputStream out;
    String name = cname.replace('/', '.') + ":" + field.getName();

    try {
      field.setAccessible(true);
      fos = new FileOutputStream(name);
      out = new ObjectOutputStream(fos);
      out.writeObject(field.get(null));
      out.close();
    } catch (Exception e) {
      System.out.println("Failed to persist field: " +
                         field + " as " + name + "(" + e + ")");
    }
  }

  private static void restoreField(Field field, String cname) {
    FileInputStream fis;
    ObjectInputStream in;
    String name = cname.replace('/', '.') + ":" + field.getName();

    try {
      field.setAccessible(true);
      fis = new FileInputStream(name);
      in = new ObjectInputStream(fis);
      field.set(null, in.readObject());
      in.close();
    } catch (FileNotFoundException e) {
    } catch (Exception e) {
      System.out.println("Failed to restore field: " +
                         field + " from " + name + "(" + e + ")");
    }
  }

  @Override
  public InputStream apply (InputStream is, final String cname) {
    final ClassReader reader;

    if (isExempt(cname))
      return is;

    if (persistentClasses == null)
      persistentClasses = new LinkedList<Class>();

    try {
      reader = new ClassReader(is);
    } catch (IOException e) {
      e.printStackTrace();
      return null;
    }

    final ClassWriter writer = new ClassWriter(reader, ClassWriter.COMPUTE_FRAMES);
    ClassAdapter adapter = new ClassAdapter(writer) {
      String cname;
      @Override
      public void visit(int version, int access, String name, String signature,
                        String superName, String[] interfaces) {
        /* Determine whether or not the target class is Serializable */
        cname = name;
        boolean isSerializable = false;
        for (String iface : interfaces) {
          if (iface.contains("Serializable"))
            isSerializable = true;
        }

        /* If the class is not already Serializable, 'make it so' */
        if (isSerializable)
            super.visit(version, access, name, signature, superName, interfaces);
        else {
          String[] modifiedInterfaces = Arrays.copyOf(interfaces,
                                                       interfaces.length + 1);
          modifiedInterfaces[interfaces.length] = "java/io/Serializable";
          super.visit(version, access, name, signature, superName,
                      modifiedInterfaces);
        }
      }

      @Override public MethodVisitor visitMethod(final int access, final String name,
                                                  final String desc, final String signature,
                                                  final String[] exceptions) {
        /* Leave initialisers alone */
        if (name.equals("<init>") || name.equals("<clinit>"))
          return super.visitMethod(access, name, desc, signature, exceptions);

        MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);

        /*
         * Cheekily prepend all return instructions from main with a call
         * to persistClasses. We could avoid this by modifying the VM internals
         * to perform the call on termination, but we consolidate the code here
         * for now.
         */
        if (name.equals("main")) {
          return new MethodAdapter(mv) {
            @Override public void visitInsn(int opcode) {
              if ((opcode >= Opcodes.IRETURN && opcode <= Opcodes.RETURN)
                  || opcode == Opcodes.ATHROW) {
                mv.visitMethodInsn(Opcodes.INVOKESTATIC,
                                   Type.getInternalName(PersistenceTransform.class),
                                   "persistClasses",
                                   "()V");
              }
              mv.visitInsn(opcode);
            }
          };
          /* Hijack all generated accessor methods */
        } else if (name.startsWith(StaticsBeanifier.SET_PREFIX)) {
          return new MultiMethodAdapter(mv, specialiseMethod(writer, access,
                                                              name, 0, desc,
                                                              signature,
                                                              exceptions)) {
            @Override public void visitCode() {
              mv1.visitCode();
              mv1.visitCode();
              /*
               * We've touched a static object, so mark this class
               * as a candidate for persistence.
               */
              mv1.visitLdcInsn(Type.getObjectType(cname));
              mv1.visitMethodInsn(Opcodes.INVOKESTATIC,
                  Type.getInternalName(PersistenceTransform.class),
                  "registerPersistentClass", "(Ljava/lang/Class;)V");
              /*
               * Flip the specialisation for this class because
               * we've written to a static field before reading it
               * and don't want to overwrite the new value by
               * faulting an old copy in from disk.
               */
              specialiseClass(mv1, cname);
              mv2.visitCode();
            }
          };
        } else if (name.startsWith(StaticsBeanifier.GET_PREFIX)) {
          return new MultiMethodAdapter(mv, specialiseMethod(writer, access,
                                                              name, 0, desc,
                                                              signature,
                                                              exceptions)) {
            @Override public void visitCode() {
              mv1.visitCode();
              /*
               * Attempt to restore from disk.
               * If a persisted store does not exist, this does nothing.
               */
              mv1.visitLdcInsn(Type.getObjectType(cname));
              mv1.visitMethodInsn(Opcodes.INVOKESTATIC,
                  Type.getInternalName(PersistenceTransform.class),
                  "restoreClass", "(Ljava/lang/Class;)V");
              /*
               * We've loaded a static object and may later modify one of its
               * virtual fields, so we eagerly mark this class as a candidate for
               * persistence.
               */
              mv1.visitLdcInsn(Type.getObjectType(cname));
              mv1.visitMethodInsn(Opcodes.INVOKESTATIC,
                  Type.getInternalName(PersistenceTransform.class),
                  "registerPersistentClass", "(Ljava/lang/Class;)V");
              /*
               * Flip the specialisation for this class because
               * it's now faulted in and marked for persistence.
               */
              specialiseClass(mv1, cname);
              mv2.visitCode();
            }
          };
        } else
          return mv;
      }
    };

    reader.accept(adapter, ClassReader.SKIP_FRAMES);
    return new ByteArrayInputStream(writer.toByteArray());
   }
  }
```

Listing 5.4: The persistence transform for a simple implementation of object persistence using static fields as roots.

Although our work does not consider advanced issues, such as persistent transactions and periodic checkpointing, it illustrates the aspect-oriented nature of our framework and shows that it is possible to implement a basic form of object migration in a relatively small amount of code and with no modification to the user application. To show an example of an application using our persistence implementation, we port the currency converter application from the OPJ tutorial [5] to our framework. The original application is shown in Listing 5.5.

[5]http://research.sun.com/forest/opj.tutorial.ccalc_example.html

```
public class Calc {
  private float exchangeRate;

  public static void main(String args[]) {
    Calc self = new Calc();

    try {
      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

      while (true) {
        System.out.print("Input command (h for help): ");
        int command = in.read();
        String arg = in.readLine();

        switch (command) {
          case 'x':
            self.exchangeRate = Float.parseFloat(arg);
            System.out.println("Exchange rate set to " + self.exchangeRate);
            break;
          case 'c': {
            float amount = Float.parseFloat(arg);
            System.out.println("Amount " + amount + " converts to "
                               + amount * self.exchangeRate);
          }
            break;
          case 'h':
            System.out.println("\tx <val>\t : Modify exchange rate.\n\t" +
                               "c <val>\t : Perform a conversion.\n\t" +
                               "h\t : Print this help message.\n\t" +
                               "q\t : Quit.\n");
            break;
          case 'q':
            return;
          default:
            System.out.println("Unknown command " + command);
        }

      }
    } catch (Exception e) {
      System.out.println(e);
      System.exit(-1);
    }
  }
}
```

Listing 5.5: A basic currency converter application without object persistence.

The OPJ implementation requires some slight modifications to the code, as shown in Listing 5.6.

```
 1 public class Calc {
 2   static { OPRuntime.roots.add(Calc.class); }
 3   private static Calc self = new Calc();
 4   private float exchangeRate;
 5
 6   public static void main(String args[]) {
 7     try {
 8       BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
 9
10       while (true) {
11         System.out.print("Input command (h for help): ");
12         int command = in.read();
13         String arg = in.readLine();
14
15         switch (command) {
16           case 'x':
17             self.exchangeRate = Float.parseFloat(arg);
18             System.out.println("Exchange rate set to " + self.exchangeRate);
19             break;
20           case 'c': {
21             float amount = Float.parseFloat(arg);
22             System.out.println("Amount " + amount + " converts to "
23                                + amount * self.exchangeRate);
24           }
25             break;
26           case 'h':
27             System.out.println("\tx <val>\t : Modify exchange rate.\n\t" +
28                                "c <val>\t : Perform a conversion.\n\t" +
29                                "h\t : Print this help message.\n\t" +
30                                "q\t : Quit.\n");
31             break;
32           case 'q':
33             return;
34           default:
35             System.out.println("Unknown command " + command);
36         }
37
38         OPRuntime.checkpoint();
39       }
40     } catch (Exception e) {
41       System.out.println(e);
42       System.exit(-1);
43     }
44   }
45 }
```

Listing 5.6: The application from Listing 5.5 modified for use with the OPJ framework. The `self` field is made static (line 3) in order for persistence to occur and a static initialiser (line 2) adds the `Calc` class to the root set. A checkpoint is taken each time around the while loop at line 38.

All that our framework requires is for any persistent fields to be marked static and non-transient. Objects in Java are considered to be non-transient unless specified otherwise, so the converter application for our persistence implementation simply requires the `self` field to be marked static, as shown in Listing 5.7

```
 1 public class Calc {
 2   private float exchangeRate;
 3   private static Calc self = new Calc();
 4
 5   public static void main(String args[]) {
 6     try {
 7       BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
 8
 9       while (true) {
10         System.out.print("Input command (h for help): ");
11         int command = in.read();
12         String arg = in.readLine();
13
14         switch (command) {
15           case 'x':
16             self.exchangeRate = Float.parseFloat(arg);
17             System.out.println("Exchange rate set to " + self.exchangeRate);
18             break;
19           case 'c': {
20             float amount = Float.parseFloat(arg);
21             System.out.println("Amount " + amount + " converts to "
22                                + amount * self.exchangeRate);
23           }
24             break;
25           case 'h':
26             System.out.println("\tx <val>\t : Modify exchange rate.\n\t" +
27                                "c <val>\t : Perform a conversion.\n\t" +
28                                "h\t : Print this help message.\n\t" +
29                                "q\t : Quit.\n");
30             break;
31           case 'q':
32             return;
33           default:
34             System.out.println("Unknown command " + command);
35         }
36
37       }
38     } catch (Exception e) {
39       System.out.println(e);
40       System.exit(-1);
41     }
42   }
43 }
```

Listing 5.7: The application from Listing 5.7 modified for use with our basic persistence framework. The `self` field is marked static at line 3 and is therefore persisted automatically.

Chapter 6

Conclusions

The evaluation of our work in Chapter 5 identified the overheads induced by our framework and demonstrated the use of our transform API by means of two examples. We now consider the results from our experiments and propose a number of aims for future work on this project.

6.1 Performance

Our modifications to the RVM add only a small overhead to the class loading procedure when not in use. This enables our framework to be permanently active within Jikes, regardless of whether or not the running application is exploiting the availability of specialised methods. The implementation described in [4] reports an average overhead in excess of 5% from the presence of the BCEL library alone. This significant contribution towards the running time of the system is attributed to the presence of additional internal data structures on the heap. As a result of using the ASM framework in preference to BCEL, our work does not suffer from these overheads and the entire framework adds less than 2% to the average runtime.

When operational, our specialisation framework exhibits similar converged performance times to the work in [4]. In particular, the upper bound determined for the implicit read barrier is a little over 8% in both implementations.

6.2 Usability

The purpose of a framework such as ours is to aid the programmer in writing complex applications. Specifically, we separate the application logic (the algorithm) from the state of the object store (the transform) and in this way allow programmers to first write their application for correctness and then implement transforms to introduce specialised behaviours.

By placing the transformation framework above the RVM but beneath the application, we avoid cluttering either tier with the specialised object code. Additionally, this increases the generality of class transforms, which can be applied to any class loaded by the RVM. The use of ASM, and of our specialisation API, allows for quick and concise transform implementations, with complete disregard for the implementation of specialised methods within the virtual machine. In this way, our work acts as an entry point into the class loader of the virtual machine whilst hiding the implementation details of Jikes.

6.3 Future Work

Our primary objective during the development of our implementation was to produce a framework that operated correctly and without introducing substantial overheads to the virtual machine. Now that we have such a framework, an obvious extension is to identify the causes of any overheads introduced by our modifications to the RVM and attempt to reduce the impact this has on application performance. One area which requires further investigation is the impact of our framework on the memory footprint of Jikes, which may have consequences for the amount of garbage collection performed at runtime. Placing internal data structures, whose lifespan is defined by the runtime of the virtual machine, into an *immortal* area of the heap (which we do for TIB arrays) hides them from the collector and therefore removes the potential for increased garbage collection pauses due to changes inside the RVM.

As well as investigating the heap space used by our framework, another potential performance increase may be achieved by modifying the AOS component of Jikes. We believe that specialised objects will spend a significant proportion of their life in a single 'valid' state, toggling to other specialisations only when invalidated and then toggling back to the valid state shortly afterwards. This observation allows us to identify a common-case specialisation which we can use to help us inline the correct specialised variant in the optimising compiler. Currently, our implementation disregards this information and greedily inlines the first method suggested by the AOS. If we extended the AOS to profile the specialised state of each object, then it would be possible to inline specialised methods in a more intelligent fashion.

The simplification pass of the optimising compiler identifies constant TIB slots and folds them accordingly. For example, the superclasses of a class remain constant during execution and so accessing the 'superclass IDs' slot of a TIB can be folded into a single indirect move without having to first load the TIB. This optimisation could be applied to the 'TIB Array' and 'Primary TIB' slots in our augmented scheme as these attributes are constant for all TIBs of a class, regardless of their specialisation.

The limitations of our beanifier transforms (as described in Section 4.3) demand extra attention in order to realise truly transparent beanification of classes. The source of our problem is that transforms are applied during class loading and therefore provide us with only a restricted view of the class hierarchy which we are manipulating. The ASM framework provides an alternative *Tree API* that is suited more to modification of loaded classes and provides a complete view of the application. Future work should implement transform application after initial class loading has occurred and evaluate the tree API of ASM to get an idea of the overheads involved. Additionally, implementation of specialised methods should be performed at the Java source level so that programmers unfamiliar with Java bytecode can still use the framework easily. This is currently possible by annotating Java methods with the `@Specialised` annotation but introduces a tight coupling between the application and the RVM.

The robustness of our framework opens up many opportunities for further work to be built on top of it. The read barrier described in Section 5.2.1 could be implemented as part of an incremental garbage collector for Jikes. This would provide an excellent opportunity to validate the performance bounds which we have measured. Furthermore, a *Distributed Shared Memory* (DSM) layer could be implemented by specialising objects according to their validity. The *Modified Exclusive Shared Invalid* (MESI) protocol for memory coherence [31] places objects into one of four states. These states could be encoded using specialised methods and a transparent implementation of a DSM would be realised. To increase performance, the programmer should be able to annotate data describing whether or not it is potentially shared with other nodes. A high-performance, robust DSM implementation would lay a solid base for the implementation of a *Distributed Java Virtual Machine* (DJVM).

Bibliography

[1] S. M. Blackburn, A. L. Hosking, Barriers: Friend or Foe?, in: ISMM '04: Proceedings of the 4th international symposium on Memory management, ACM, New York, NY, USA, 2004, pp. 143–151.

[2] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, R. Namyst, The Hyperion System: Compiling Multithreaded Java Bytecode for Distributed Execution, Parallel Comput. 27 (10) (2001) 1279–1297.

[3] W. Zhu, C. li Wang, F. C. M. Lau, JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support, in: In IEEE Fourth International Conference on Cluster Computing, 2002.

[4] A. M. Cheadle, A. J. Field, J. Nystrom-Persson, A Method Specialisation and Virtualised Execution Environment for Java, in: VEE '08: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ACM, New York, NY, USA, 2008, pp. 51–60.

[5] IEEE International Conference on Parallel Processing (ICPP-99), cJVM: a Single System Image of a JVM on a Cluster.

[6] John Zigman and Ramesh Sankaranarayana, Designing a Distributed JVM on a Cluster, in: Proceedings of the 17th European Simulation Multiconference, Nottingham, United Kingdom, 2003.
URL http://djvm.anu.edu.au/publications/djvm_design.pdf

[7] S. Thibault, A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines, in: Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2), Cambridge, USA, 2005.
URL http://hal.inria.fr/inria-00000138

[8] JavaParty Trac.
URL http://svn.ipd.uni-karlsruhe.de/trac/javaparty/wiki/JavaParty?redirectedfrom=WikiStart

[9] JavaSpaces Service Specification.
URL http://java.sun.com/products/jini/2.0/doc/specs/html/js-spec.html

[10] A. Marquez, S. Blackburn, G. Mercer, J. N. Zigman, Implementing Orthogonally Persistent Java, in: POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems, Springer-Verlag, London, UK, 2001, pp. 247–261.

[11] S. M. Blackburn, J. N. Zigman, Concurrency - The Fly in The Ointment, Morgan Kaufmann, 1999, pp. 250–258.

[12] Sun Java Data Objects Homepage.
URL http://java.sun.com/jdo/index.jsp

[13] Enterprise JavaBeans Specification.
URL http://java.sun.com/products/ejb/docs.html

[14] The Java Persistence API.
URL http://java.sun.com/javaee/technologies/persistence.jsp

[15] Space4J Homepage.
URL http://www.space4j.org/

[16] Java Technology: The Early Years.
URL http://java.sun.com/features/1998/05/birthday.html

[17] The Java Language Environment.
URL http://java.sun.com/docs/white/langenv/Intro.doc2.html

[18] TIOBE Programming Community Index.
URL http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[19] Tuning Garbage Collection with the 5.0 Java Virtual Machine.
URL http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

[20] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Second Edition, Prentice Hall PTR, 1999.

[21] The Jikes RVM Homepage.
URL http://www.jikesrvm.org

[22] M. Arnold, S. Fink, D. Grove, M. Hind, P. F. Sweeney, Architecture and Policy for Adaptive Optimization in Virtual Machines, Tech. rep. (2004).

[23] I. Rogers, J. Zhao, I. Watson, Boot Image Layout for Jikes RVM, in: Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS), 2008.

[24] E. Bruneton, R. Lenglet, T. Coupaye, ASM: A Code Manipulation Tool to Implement Adaptable Systems, Tech. rep., France Télécom (2002).

[25] The Apache Jakarta Project Byte Code Engineering Library.
URL http://jakarta.apache.org/bcel/

[26] G. Hamilton, The JavaBeans API Specification, Sun Microsystems (1997).

[27] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo Benchmarks: Java Benchmarking Development and Analysis, in: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006, pp. 169–190.

[28] The SPECjvm2008 Java Virtual Machine Benchmark.
URL http://www.spec.org/jvm2008/

[29] H. G. Baker, Jr., List Processing in Real Time on a Serial Computer, Commun. ACM 21 (4) (1978) 280–294.

[30] D. F. Bacon, P. Cheng, V. T. Rajan, A Real-time Garbage Collector with Low Overhead and Consistent Utilization, in: POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 2003, pp. 285–298.

[31] M. S. Papamarcos, J. H. Patel, A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories, in: ISCA '84: Proceedings of the 11th Annual International Symposium on Computer Architecture, ACM, New York, NY, USA, 1984, pp. 348–354.

Appendix A

User Guide

A.1 Building the Modified RVM

1. Download the source code tarball:
   `$ wget http://www.doc.ic.ac.uk/~wjd105/work/wjd105-meng-project.tar.bz2`

2. Extract the contents of the archive:
   `$ tar -xvjf wjd105-meng-project.tar.bz2`

3. Change into the extracted directory:
   `$ cd msf-jikesrvm-r15227/`

4. Build the RVM:
   `$ ant -Dconfig.name=production -Dhost.name=ia32-linux`
   Machines running a 64-bit kernel should use `-Dhost.name=x86_64-linux` instead.

5. Jikes can then be invoked by running:
   `$ ./dist/production_ia32-linux/rvm` (assuming a 32-bit kernel).

A.2 Writing a Transform

Transforms should be written inside the `$ARCHIVE_ROOT/rvm/src/asmtransform/` directory. This directory also contains the `LowLevelTransform` and `Transform` API classes. To install the transforms in this directory without rebuilding the RVM, use the `install-transforms` ant target (which requires the config name and the host name to be set as above).

A.3 Running an Application

Applications are executed by supplying their class files as arguments to the RVM. To specify transforms, place their names (for example `asmtransform.MyTransform`) into a file (for example, `transforms.txt`) and then invoke the RVM with:

```
$ ./dist/production_ia32-linux/rvm -xforms=transforms.txt MyClass
```