A Method Specialisation and Virtualised Execution Environment for Java

A.M. Cheadle, A.J. Field, and J. Nyström-Persson
Imperial College London
{amc4, ajf}@doc.ic.ac.uk

Abstract

We present a virtualisation and method specialisation framework for Java that facilitates efficient, dynamic modification of the behaviour of object accesses at run time. The technique works by virtualising *all* method calls and field accesses associated with selected classes so that all corresponding object accesses are via the invocation of a virtual method. Different access behaviours are then supported by allowing arbitrary specialisations of those methods to be defined. The virtualisation overheads are partially recovered by allowing the JVM's optimisation subsystem to perform guarded inlining of specialised methods. We describe an implementation based on the Jikes RVM and show how the framework can be used to implement an 'implicit' read barrier that supports incremental garbage collection. The performance overhead of full virtualisation, and the performance of the implicit read barrier compared with an existing conventional, explicit barrier, are evaluated using SPEC JVM98 and DaCapo benchmarks. The net virtualisation costs are shown to be remarkably low and the implicit barrier is shown to outperform the explicit barrier substantially in most cases. Other potential applications, including object proxying, caching, and relocation, and instrumentation are also discussed.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Frameworks; D.3.4 [*Processors*]: Memory management (garbage collection)

General Terms Algorithms, Design, Experimentation, Measurement, Performance, Languages.

Keywords Language implementation, Memory management, Incremental garbage collection, Read barrier, Method virtualization.

1. Introduction

The power and expression of object-oriented languages such as Java are realised predominantly by promoting and maximising object extensibility and code reuse and are facilitated by abstract language features such as class local methods (enabling encapsulation and overridden re-implementation), inheritance and message dispatch. Message dispatch is the general mechanism by which a message sent to a receiver object results in the execution of a specific code sequence. Within object-oriented languages, message dispatch, termed *dynamic dispatch* is the invocation of a virtual method, most usually via a method table.

In addition to facilitating the overloading and overriding of methods, dynamic dispatch can also be exploited for the provision of language features and runtime services, most usually by employing *proxy objects*. Examples include persistent object systems and in particular, Orthogonally Persistent Java [14], distributed virtual machine implementations [17, 18], incremental garbage collector support [6, 7], object caching and checkpointing, and fast hot swapping of class-local features, such as debug code and message logging.

In this paper we develop and evaluate a generic framework for supporting a range of applications such as the above, with the objective of balancing efficiency with ease of use. The idea is to exploit the VM's dynamic dispatch mechanism to virtualise *all* method calls and field accesses associated with selected classes so that *all* corresponding object accesses are made via the invocation of a virtual method. In the VM the result is that all object accesses are then routed through a method table.

With this done, we can now change dynamically the way objects are accessed by 'hijacking' references to the method table, routing them instead to *specialised* variants of the access methods, according to need. The idea is that the various specialisations will encode the different behaviours required.

Virtualisation thus ensures that all specified object accesses are handled by a single common mechanism (dynamic dispatch), and specialisation in principle makes it easy to ensure that the correct behavioural variant is invoked, depending on the object's state.

Once the full virtualisation cost has been paid, the additional overheads are very low. Changing the behaviour of an object involves simply modifying its type information block (TIB) pointer ("TIB flipping") or virtual method table (VMT) pointer ("VMT flipping"). Once this has been done all subsequent accesses will be routed to the required specialisation as a side effect of the standard virtual method calling mechanism.

The initial virtualisation overheads can be substantial, however. In order to buy back some of these overheads we therefore allow the RVM's dynamic optimiser to apply run-time optimisations such as method inlining. This requires some care as only one of the several possible specialisations will end up being inlined. Explicit tests therefore need to be added to the inlined code to ensure that the correct specialisation is invoked.

Each of the above applications, and indeed many others, can be formulated in these terms. Indeed, similar mechanisms have already been used to support them, for example auxiliary methods, *façade* objects, etc. as discussed in Section 2.1. Our main objective is to provide an efficient common framework which is widely applicable.

The framework we propose provides support for virtualisation, the introduction of specialised methods and an API for switching

VEE'08, March 5–7, 2008, Seattle, Washington, USA.

between them at run time. These features are provided by a *Class Transformation Toolkit* (CTTk), for rewriting a program's byte code, and a modified VM for supporting specialisation. The rest of this paper describes the design, application and evaluation of this framework. We make the following contributions:

- We describe a virtualisation and specialisation framework within the context of the Jikes RVM [16] (Section 3).
- We discuss how the same concepts can be supported in other virtual machines, for example other JVMs (that are not themselves written in Java), Microsoft's Common Language Runtime, and Intel's Open Runtime Platform (Section 3.5).
- We demonstrate how method inlining can be applied by a dynamic optimiser whilst correctly invoking the required method specialisation in all cases (Section 3.3).
- We describe an application of the framework, namely the development of an 'implicit' *read barrier* on *all* object accesses for supporting incremental copying garbage collection (Section 4).
- Using SPEC JVM98 and DaCapo benchmarks we evaluate in Section 5: i) the overheads that the method specialisation framework places on the infrastructure of the virtual machine; ii) the cost of our read barrier implementation, compared with that of a previously published 'explicit' read barrier implementation for the Jikes RVM.

We also discuss how our framework might be applied to other applications, including those listed above (Section 6.1).

2. Background and Related Work

2.1 Applications Exploiting Dynamic Dispatch

In our earlier work [6, 7] we demonstrated how to exploit the virtualisation that is inherent to a *pure* dynamic dispatch environment, to efficiently implement run-time services that most usually incur significant overhead when implemented in software. Our particular focus was on the efficient design and implementation of incremental and generational garbage collectors for the Haskell programming language and within the context of the Spineless Tagless G-machine.

Existing persistent object systems, and Orthogonally Persistent Java (OPJ), implementation techniques have explored various source and byte code transformations and compiler and runtime system modifications. For example, Marquez et al. [14] implement *portable* OPJ by subclassing Java's standard class loader to insert read and write barriers into the generated code to ensure correct access to/from the persistent store. They propose the use of *façade* objects that proxy the real objects by faulting and swizzling them into memory before dispatching to the target method invocation.

Further exploitation of dynamic dispatch is in distributed JVM implementations which share many of the problems associated with persistence. A distributed JVM must present a *single system image* (SSI) to the Java applications, hiding the distributed infrastructure upon which they run. Our approach provides an obvious way to change efficiently the way an object is accessed depending on its location and state within such an image. Several approaches have already been explored, for example, cluster-aware models adopted by both IBM's *cJVM* [17] and ANU's *dJVM* [18].

2.2 Dynamic Dispatch in Java

In an object-oriented language, message dispatch, termed *dynamic dispatch* is the invocation of a *virtual* method. A virtual method is defined locally within its *base* class and can be subsequently *overridden* in its subclasses. At a particular call site of a virtual method one of a number of possible implementations may be invoked depending on the dynamic type of the receiver. Such call sites are said to be *polymorphic*. For overridden methods, the method signatures are all identical. However, virtual methods may also be *overloaded* — multiple methods with the same name but differing argument signatures may be provided by both base classes and subclasses. The process of method resolution for these *parametrically polymorphic* call sites depends on the semantics of the particular programming language. Not only is the method choice dependent upon the receiver's dynamic type but also the number of method arguments and, potentially, the dynamic types of one or more of these arguments.

For *multi-method* dynamic dispatching languages such as Cecil, method resolution is purely dynamic and occurs at run-time based on the dynamic run-time types of the receiver and method arguments. In Java, however, method targets are partially resolved at compile-time using the declared types and the number of arguments to the method as well as the declared type of the receiver. This compile-time pre-selection may result in either a single method target, in which case the call site is *monomorphic*, or a set of potential method targets, in which case the *most specific matching* method must be sought for use as a *template*. Some polymorphic call sites may also be classed as *almost monomorphic*, which means that there are multiple method targets but that the majority of calls resolve to a single target.

2.2.1 Inlining

For those call sites that are verifiably monomorphic the target method body can be *inlined*, not only eliminating the overhead of virtual method invocation, but also facilitating wider application of classical code optimisations that, in the absence of inlining, would be restricted intra-procedurally. A method target can still be inlined at an almost monomorphic call site. However, a *guard* must be added which verifies that the dynamic type of the receiver is as expected, falling back to the standard virtual method invocation mechanism if it is not.

Inlining [2, 10], method *devirtualisation* [13] and techniques for the efficient implementation of message dispatch [11] have been extensively researched.

2.3 The Jikes Research Virtual Machine

In the Jikes RVM byte codes are compiled to native machine code prior to execution. In our specialisation framework (Section 3) we make use of the RVM optimising compiler that uses traditional static compiler optimisations as well as some which are specific to a Java execution environment. A static size-based heuristic is applied to method call sites to determine whether static and final methods should be inlined. For the call sites of non-final virtual methods, the object on which the virtual call is invoked, its *receiver*, is predicted to be the declared type of the object and the method is inlined. The call site is then *guarded* with a run-time test that verifies the prediction is correct and if it is not, falls back to virtual method invocation.

In addition to static compiler optimisations there is also an adaptive optimisation system (AOS) that selectively optimises both the user code and the JVM, guided by profiling data gathered by statistical sampling at run time.

The AOS maintains an approximation for the dynamic call graph of the running program by statistically sampling method calls and running methods. 'Hot' call graph edges are identified and passed to the optimising compiler and, in many cases, already optimised methods undergo recompilation in order to inline them.

2.3.1 Object Allocation and Object Layout

Scalar objects of the Jikes RVM have a two word object header, while array objects have three. The first word of the object header is a reference to the object's *Type Information Block* (TIB). For

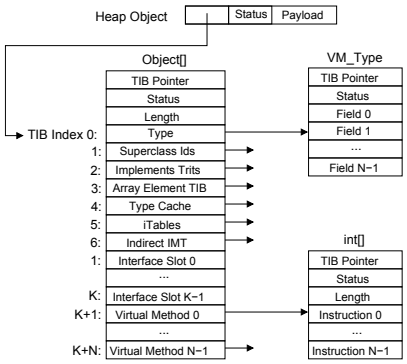

Figure 1. The Jikes RVM Type Information Block.

each class in the RVM there is a unique TIB. The TIB is an array of references to Java `Objects` which describe the object's type and provide the infrastructure that support Java's dynamic operations on objects, such as virtual method dispatch, interface method invocation, and dynamic type checking. The second field of the object header is the `status` word, a bit field, with bits reserved for object locking, the object hash value, and garbage collector support — mark bits, etc. For array objects, the third field stores the length of the array. The remainder of the object contains the payload, i.e. the object's fields.

Figure 1 shows the layout and contents of the TIB. Notice that because the TIB structure is a valid Java array object, it too has a 'TIB pointer' field, that references the TIB for the `Object` array class. The 'Type' field refers to a `VM_Type` (and ultimately `VM_Class`) object, which is an internal RVM object that stores metadata about the object's class, such as its superclass, the interfaces it implements and method and field offsets. This object is constructed when the class is loaded and subsequently accessed during method compilation by the lazy compilation stub, and by some of the more expensive and less frequently invoked run-time mechanisms of the RVM.

The majority of the remaining TIB elements are references to integer arrays, one for each of the class's methods, including those methods it inherits and interfaces it implements. These arrays contain the native code for the compiled method bodies. The TIB therefore implements the class's *Virtual Method Table* (VMT) and *Interface Method Table* (IMT). As a result, method dispatch is achieved via one level of indirection as opposed to two, where the method table would be accessible only via the TIB's `VM_Class` object. Other JVM's, for example Sun's ExactVM / ResearchVM, have previously implemented the object header as a reference to a class object that then references the method tables.

The fields 'K+1' to 'K+N' form the inline VMT and it is indexed by a unique *method identifier* associated with each method. Each virtual method inherited from its superclass resides at the same location in the VMT as in the superclass's VMT. The effect is to clone a local copy of the the superclass VMT and append the class's additional virtual method table entries to the end.

Static fields and all references to static method bodies are stored in the *Jikes RVM Table of Contents* (JTOC), a single array that parallels the Java Language Specification's Constant Pool in housing all literals, numeric constants, and references to string constants. However, unlike the constant pool, all of the RVM's global data structures, or references to them, are stored in the JTOC, including references to *all* TIBs in the RVM, to allow fast common-case dynamic type checking. The JTOC is stored in an 'immortal' part of the heap that is not subject to garbage collection.

3. Our Framework

Our approach is based on the ability to invoke different versions (*specialisations*) of a method, depending on context. In order to achieve this we have developed a *Class Transform Toolkit* (CTTk) for transforming classes as they are loaded by the RVM. We use CTTk to beanify field accesses (generate getter / setter methods) where necessary and to enable user-defined specialisations to be defined and integrated into a common TIB structure in a modified version of the RVM.

CTTk builds on Zigman's Bytecode Transformation Tools for Jikes RVM [19] originally developed for effecting distribution within ANU's distributed JVM, dJVM [18]. Zigman's patch hooks the Apache Jakarta Project's Byte Code Engineering Library (BCEL) [20] onto the RVM's class loading mechanism. BCEL enables the analysis, creation, and binary manipulation of Java class files — in particular, APIs are provided to access and modify a class's methods, fields and byte code instructions. By sitting at the end of the class loading chain the BCEL patch is able to transform both user and system loaded classes.

A BCEL transform is a class that implements the `Transform` interface's `process` method:

```
public interface Transform {
  public JavaClass process(JavaClass javaClass);
}
```

The `process` method is passed a `JavaClass` object that is the BCEL representation of a Java class. It returns a potentially new or mutated `JavaClass` object that has been subjected to the BCEL APIs as defined by the user defined implementation of the method.

We modify the signature of the `process` method to take an instance of a `TransformClass`. The `TransformClass` is a wrapper class that encapsulates the `JavaClass` undergoing transformation, and an array of `Method` object arrays, and provides getter and setter methods for these objects.

```
public interface Transform {
  public TransformClass process(TransformClass c) ;
}

public class TransformClass {
  public TransformClass(JavaClass javaClass,
                        Method[][] specMethods) {
    ...
  }
  ...
}
```

The `Method` class is the BCEL representation of a class method. The array of `Method` object arrays allows an arbitrary number of `Method` array specialisations to be presented to the transformation framework; each element provides a specialisation of the `javaClass`'s methods. It is from this object that the array of VMTs is constructed that replace the single VMT in the original RVM TIB structure. Specifically, the first `Method[][]` index identifies the TIB specialisation, while the second indexes the specialised methods for that particular TIB.

During class loading the minimal amount of work necessary for the user program to start executing is performed. At this point the resulting `VM_Class` object is only partially initialised: superclasses are yet to be resolved, methods and fields are neither laid out nor resolved and the TIB is neither created nor instantiated. The class is only fully resolved and the `VM_Class` object fully initialised on first use of the class, either for object access or instantiation. By the time a class is fully resolved for use, the RVM must have built its internal representation for class instances (`VM_Class`), arrays (`VM_Array`), and primitives (`VM_Primitive`). These internal representations

all subclass `VM_Type`, and correspond to run-time types. Similarly, the RVM defines internal representations for method (`VM_MethodReference`) and field (`VM_FieldReference`) references, which subclass `VM_MemberReference` and represent member references in the class files.

The process of class loading is initiated on invocation of the RVM's `VM_ClassLoader.defineClassInternal` method. In the original RVM, only a `VM_TypeReference` and a `VM_Class` instance are created while only the names of fields, methods, and superclasses are read from the class file. Nothing more is done until subsequent invocation of the `resolve` method. However, in our virtualised RVM, in addition to the above, a BCEL `ClassParser` is constructed that *fully* parses the class file, creating a corresponding `JavaClass`.

In the original RVM class loading is completed on invocation of `resolve`, whereupon superclasses are resolved, fields and methods are laid out, and the Type Information Block is created and populated. In our RVM the BCEL patch arranges for the CTTk internal transformations and user-defined transformations to be applied to the `JavaClass` that has been previously created. The `VM_Class` is updated, and its field and method definitions are updated to reflect the corresponding changes to the transformed `JavaClass`. Finally, methods and fields are laid out and the TIB is constructed. If specialised methods exist, then these are similarly processed and their associated TIB specialisations are constructed.

3.1 Specialisation

To implement method specialisation the standard virtual method table is replaced by an array of method tables. Within Jikes RVM this is achieved by providing duplicated Type Information Blocks that differ primarily in the instruction code arrays to which each virtual method slot points. As a result the TIB must be extended with a *TIB array pointer* which references the array of TIBs which carry the required method specialisations.

Figure 2 shows the modified TIB structure, with all additions to Figure 1 shown shaded on the diagram. If there are S specialisations then there are $S + 1$ TIBs which differ only in their method code. The diagram shows the primary TIB and one of the specialisations (Specialisation 1). A heap object is shown whose header field points to the primary TIB.

Each TIB is augmented with two additional fields. The first of these, occurring at TIB index 2, is the *current TIB index*, which identifies which index in the TIB array this TIB is. The second, at TIB index 3, is the *primary TIB pointer* and points to the type's primary (or default) TIB. This is used in operations such as type checking.

Type checking within the RVM is implemented by the comparison of TIB object addresses: two objects are deemed to have the same type if their TIB addresses match. In our modified RVM we have a problem: there may be several specialisations of the same object, i.e. each with the same type. We get round this by using the additional primary TIB pointer field. This points to the TIB array, but could equally well point to the primary TIB, for example. In principle, two objects now have the same type if their primary TIB fields match. However, this requires an extra level of indirection as we have to index the TIBs to obtain the pointers.

Our experiments have shown that these extra references can prove expensive in some applications. We therefore code for the common case: we envisage that in general, the current TIB only rarely corresponds to a specialisation. A type check thus proceeds by comparing TIB addresses, as in the original RVM. If this fails we fall back and reapply using the more expensive primary TIB pointer comparison.

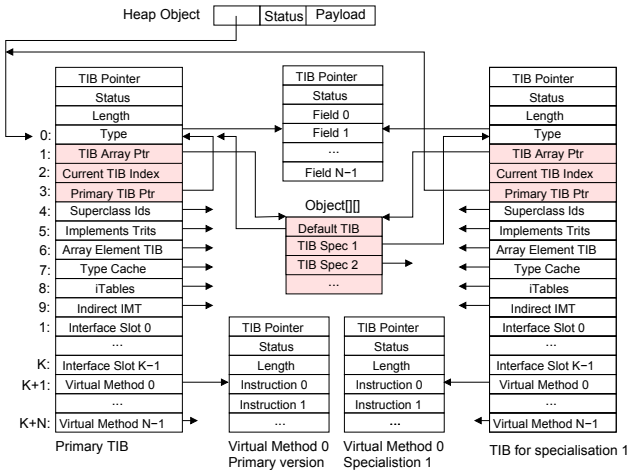

Figure 2. The Type Information Block augmented with TIB specialisations.

It is important to note that, because TIBs are shared by objects of the same type, the addition of these fields does not affect the size of the objects themselves.

3.1.1 TIB Flipping

A particular specialisation is invoked by *flipping* the TIB of an object among the various specialisations, thus *hijacking* accesses that are directed via the method table. In Figure 2, flipping the TIB of the object shown will involve overwriting the header field with a reference to one of the TIB specialisations located in the TIB array. It should be apparent that there is sufficient information in each TIB variant to move arbitrarily among the specialisations.

In order to implement TIB flipping, we augment the RVM's object model APIs for getting and setting TIBs:

```
Object[] getTIB(Object o)
void setTIB(Object o, Object[] tib)
```

with three additional (public static) functions:

```
Object[] getTIB(Object o, int tibSpecNum)
void hijackTIB(Object o, int tibSpecNum)
void restoreTIB(Object o)
```

The additional `getTIB` method retrieves the TIB at index `tibSpecNum` from the TIB array. The `hijackTIB` method updates the TIB of the given object/object reference to the TIB retrieved from the TIB array at the index specified by `tibSpecNum`. Finally, the `restoreTIB` method restores the TIB of the given object to its primary TIB.

3.2 Virtualisation of Field and Method Accesses

In order to provide a full virtualisation framework that exploits dynamic dispatch we must *naïvely* ensure that all operations on a Java object are performed using the virtual machine's *invokevirtual* instruction, i.e. a virtual method call — this enforces 'JavaBean' compliance where a class is compliant if *all* field (including `private`) access and assignment operations are performed via getter and setter methods. To achieve this, we build a *BeanifierTransform* transform using CTTk. The beanifier is invoked during class loading and uses the BCEL APIs to generate getter and setter methods for each field in a class. The Beanifier is able to detect existing getter and setter methods and avoids generating duplicates. Additional control is provided over the beanification of fields based on their access level modifiers.

3.3 Recovering Performance via the Adaptive Optimisation Subsystem

Full virtualisation enables the exploitation of dynamic dispatch. However it imposes an additional overhead on some object accesses. In order to buy back this performance we allow the AOS and the optimising compiler to perform a range of optimisations, most notably method inlining.

The optimising compiler of the AOS must often selectively recompile methods, discarding compiled methods and replacing them with new, more aggressively optimised ones. As a result, the method table entries within the TIB must be patched to point at the newly compiled code for the recently optimised replacee method body. In the presence of specialised TIBs, we need to ensure that the correct TIB specialisation is patched during method recompilation.

We introduce a new type of guard, `ig_tib_test` that determines if the inlined method is a method belonging to the current TIB index of the caller. If it is, the inlined code is executed, if not the appropriate method must be invoked using `invokevirtual` on the virtual method slot of the current TIB. The `ig_tib_test` adds at most three native code instructions, (`mov`, `cmp`, and `jne`), and in many cases only two (the `mov` instruction is not emitted when the TIB pointer is already in a register). Each compiled method has a bitmap that identifies the TIBs to which it belongs (in general, they may belong to more than one). The guard thus reduces to a simple bitwise AND operation. We work to preserve the original inlining behaviour of the standard inlining *oracle*. The most significant difference is that whenever an inlining decision has been made, we test the callee to determine if it has any specialisations. If it does, we emit the `ig_tib_test` guard to the final inlining decision. If the call site is polymorphic then an additional class/method guard will be emitted as standard. Inlining may thus introduce 0, 1 or 2 guards in total.

3.4 Thread-safe Execution and Concurrent Operation

The ability to flip the TIB at any point within both user code and the RVM's runtime system raises some interesting issues relating to thread-safe operation within our environment. Code emitted by the compilers for the `invokevirtual` instruction *always* reloads the current TIB pointer before invoking a method. As a result there are no memory ordering issues resulting from the use of a cached, out-of-date, TIB pointer. However, the TIB pointer could change at any point during the execution of a method which means that a subsequent method call may invoke any of the available specialisations. As a result, when writing specialisations, care must be taken to reason about the effect and interaction of all specialisations of a method on the current object state for all potential callers. Similarly, we delegate the responsibility of object locking and thread synchronisation to the method specialisation developer. This provides maximum flexibility, arguably, at the expense of additional complexity. In practice, we expect specialisations to be constructed based on reasoning about global invariants on when TIB replacement may occur and when particular specialisations may execute; indeed, we do exactly this when constructing our garbage collector method specialisations (Section 4.1). Note that where atomic execution is necessary for correctness, the RVM's `UNINTERRUPTIBLE_PRAGMA` can be used to ensure the execution of a method body cannot be preempted by another thread.

3.5 VM 'Neutral' Implementation

Although our implementation uses the Jikes RVM, much of the design is VM 'neutral' and can be applied to VM implementations where the runtime systems are written in native languages such as C or C++. The use of BCEL and the CTTk interface would remain unchanged. CTTk must however be modified to drive the CTTk and BCEL APIs *solely* by overriding the `findClass()` method of one of the standard class loader extension mechanisms (via a bootstrap, system or context classloader). Having read the class file and effected the transformations (including the beanification transformation for full virtualisation) the BCEL `JavaClass.getBytes()` method can hand off the class as a byte array to the standard `defineClass()` implementation of the classloader. Because method specialisations would now appear as standard methods in the mutated class, CTTk must generate the specialisations using a unique naming scheme, such as: `__spec_<specNum>_<methodName>` where `specNum` is the VMT (previously the TIB) index number and `methodName` is the original method name. The VM must now be modified to:

- Implement an array of VMTs not just a single VMT.
- Recognise class method definitions of the form: `__spec_<specNum>_<methodName>` as specialisations and populate the appropriate VMT for the associated specialisation index.
- Expose `hijackTIB` and `restoreTIB` APIs as `hijackVMT` and `restoreVMT` as publicly accessible APIs via JNI VM extensions.
- Emit an `ig_vmt_test` corresponding to the `ig_tib_test` at polymorphic call sites that determines if the inlined method that is about to be executed is the appropriate method for the currently active VMT specialisation.

One of the difficulties of working with a Java-based VM is that some of the run-time structures are attributable to the VM itself, rather than the executing program. The RVM developers dedicated considerable effort to removing such overheads from the original RVM. We have not attempted any such optimisations in our modified RVM. As a result, we would expect a performance hit, even from the use of the BCEL patch alone, as we end up adding intermediate VM data structures that are themselves subject to garbage collection. We revisit this in Section 5.

4. A Dynamic Dispatching Read Barrier

We demonstrate our framework by showing how it can be used to implement a low-cost *conditional read barrier*. Read barriers are used in incremental garbage collectors, which work by by interleaving program execution with small amounts of garbage collection. The objective is to avoid long pauses in program execution associated with "stop the world" schemes such as [9].

Although we have not implemented a complete incremental collector we discuss our approach in the context of Baker's incremental algorithm, described in [1]. Each memory allocation event invokes a *scavenger* that copies (*"evacuates"*) a small number of live objects from the current heap region (*from-space*) to a 'spare' region (*to-space*), before switching control back to the program. This continues until all live objects have been evacuated, whereupon the collection cycle terminates and the garbage collector switches off until the to-space fills up again. At that point the two spaces are flipped and a new collection cycle is initiated.[1]

When the collector is on it is crucial that the program sees only objects in to-space, otherwise it may acquire a reference to memory that is sure to be corrupted at some future point, e.g. in the next collection cycle. A conditional read barrier preserves this invariant.

Traditionally, the barrier is an explicit test inserted by the compiler in front of each object reference that checks a) whether the garbage collector is on and, if so, b) whether the object resides in

[1] Note that forwarding pointers, that link from-space objects to their copies in to-space, are also required. We can also build specialisations that perform automatic forwarding, similar to self-scavenging, but this is beyond the scope of the paper. Further details can be found in [8].

Listing 1. The `IncrGCmethodTransform` — The Self-scavenging Specialisation Transform

```
public class IncrGCmethodTransform extends Transform {
  public IncrGCmethodTransform() {}

  public TransformClass transform( TransformClass transformClass ) {
    JavaClass javaClass = transformClass.getJavaClass();

    // Get the class constant pool
    ConstantPoolGen constantPoolGen = new ConstantPoolGen( javaClass.getConstantPool() );

    Method[] methods = javaClass.getMethods();
    int numMethods = methods.length;
    Method[][] specialisedMethods = new Method[1][numMethods];

    for ( int methodIndex = 0; methodIndex < numMethods; methodIndex++ ) {
      String name = methods[methodIndex].getName();
      if (methods[methodIndex].isAbstract() || methods[methodIndex].isStatic()
                                            || methods[methodIndex].isPrivate()
                                            || methods[methodIndex].isProtected()
                                            || name.equals("<init>") || name.equals("<clinit>")) {
        specialisedMethods[0][methodIndex] = methods[methodIndex];
      } else {

        MethodGen methodGen = new MethodGen( methods[methodIndex],
                                             javaClass.getClassName(), constantPoolGen );

        // The method is a GC method, don't let it be interrupted!
        methodGen.addException(UNINTERRUPTIBLE_PRAGMA);

        // Method body -> invoke restoreTIB, scavenge the object, invoke the method again

        InstructionList instructionList = new InstructionList();
        int methodIdx = 0;

        methodIdx = constantPoolGen.addMethodref( "org.jikesrvm.VM_ObjectModel",
                                                  "restoreTIB", "(Ljava/lang/Object;)V" );
        instructionList.append( new ALOAD( 0 ) ); // load this
        instructionList.append( new INVOKESTATIC( methodIdx ) );

        methodIdx = constantPoolGen.addMethodref( "org.jikesrvm.mm.mmtk",
                                                  "scavenge", "(Ljava/lang/Object;)V" );
        instructionList.append( new ALOAD( 0 ) ); // load this
        instructionList.append( new INVOKESTATIC( methodIdx ) );
        instructionList.append( new ALOAD( 0 ) ); // load this

        // Push all arguments required to call a function onto the stack
        Type[] argumentTypes = methodGen.getArgumentTypes();
        for (int argIndex = 0, numArgs = argumentTypes.length, localVarIndex = 1;
             argIndex < numArgs; argIndex++) {
          LocalVariableInstruction load = InstructionFactory.createLoad(argumentTypes[argIndex],
                                                                        localVarIndex);
          instructionList.append(load);
          localVarIndex += argumentTypes[argIndex].getSize();
        }

        // Invoke the virtual method
        methodIdx = constantPoolGen.addMethodref(methodGen);
        instructionList.append( new INVOKEVIRTUAL( methodIdx ) );

        Type returnType = methodGen.getReturnType();
        instructionList.append(InstructionFactory.createReturn(returnType));
        specialisedMethods[0][methodIndex] = finaliseMethod(methodGen, instructionList);
      }
    }

    // Update the constant pool
    javaClass.setConstantPool( constantPoolGen.getFinalConstantPool() );
    javaClass.setMethods(methods);

    transformClass.setSpecialisedMethods( specialisedMethods);

    return transformClass;
  }
}
```

from-space. In this case subsequent code (actually part of the collector, rather than the barrier) evacuates the object before accessing it. Note that the barrier code is executed on each object access regardless of the state of the object or collector.

The idea is to use our virtualisation framework to build an *implicit* read barrier. Rather than statically inserting barrier code, we instead virtualise all object accesses and build a second, specialised, version of each access method that first scavenges the associated object before executing the original method body. The trick is to arrange for the specialised version to be executed only when the object is known to reside in to-space. That way, the act of accessing an object has the side effect of copying it to to-space, *but only when it has yet to be copied*.

Our implicit barrier thus presents a performance tradeoff: full virtualisation potentially adds an overhead on program execution by adding virtual getter/setter methods (although this can in part be bought back via inlining); however, specialisation completely removes the need to perform tests a) and b) above. At the same time, our approach makes it easy to build the barrier in the first place: we get the framework to fully virtualise *all* object accesses; subsequent method specialisation then ensures that no object access can violate the barrier invariant. Section 5 describes a series of experiments designed to explore the performance tradeoff quantitatively. Section 4.1 below, demonstrates the ease with which the read barrier can be built using the framework.

4.1 Implementation

When the collector is off, all objects will have their TIB set to the primary TIB. When the collector is turned on, all objects referenced from the registers and stack of each thread (the *root set*) are evacuated. When an object is evacuated its TIB is set to a *self-scavenging* specialisation. In an incremental collector, this self-scavenging code will evacuate all objects referenced from it, flipping those object TIBs similarly. It will then flip the original object's TIB back to the primary TIB. As a result of this, the invocation of a specialised method occurs *only* in to-space when the collector is on (obviating the need for an explicit 'collector on' test) and at most once per collection cycle per object. Note that this treatment of the stack ensures that the original method cannot be preempted by its self-scavenging variant before it has been evacuated to to-space, which could leak from-space references. Note that it is possible for two threads to be executing the same self-scavenging variant concurrently, but scavenging is an idempotent operation, so correctness is ensured without the use of any additional synchronisation mechanism.

4.1.1 What to Specialise

The Jikes RVM treats the static methods and fields of an object differently to non-static equivalents. Because they are allocated in an immortal (non garbage-collected) area of the heap we do not need to specialise static methods. Recall that a static method can only access the static fields of an object. For the same reason, we do not need to beanify static fields via getter and setter methods.

What about private methods? Because a private method can only be invoked from inside the class in which it is defined we do not need to specialise these either. This is because objects of that class will already have been scavenged by the time any private methods are invoked. The same applies to protected methods.

In summary, the following do not need to be specialised:

- Static methods including constructors and class initialisers
- Private or Protected methods
- Abstract methods (there is no implementation!)
- Methods that have been marked as being pure access methods by the `BeanifierTransform`

The complete source of the `IncrGCmethodTransform` is presented in Listing 1. This code builds an additional self-scavenging specialisation of all methods that require it and installs the necessary specialisations in the modified TIB structure. We briefly outline how it works.

Lines 1 to 11 and 64 - 73 encode class definition, initialisation and finalisation of the transform and the class constant pool. Line 12 declares the array of method specialisations, and in this case there is only a single TIB specialisation. It is worth noting that transforms can be chained in a pipeline. Were this to be the case then Line 12 would read: `Method[][] specialisedMethods = transformClass.getSpecialisedMethods()`.

Lines 14 to 63 is a loop that iterates over each of the original methods defined for the class and provides a specialisation for each of them, if appropriate. Lines 16 to 19 determine whether a specialisation is required. If not then the entry in the specialisation array points to the standard method body. Lines 23 to 24 create a BCEL method generator allowing us to generate byte code instructions for the method body of the specialisation. In line 27 we add a Jikes RVM-specific pragma, implemented as an exception, to the method. Because the self-scavenging method specialisation is essentially an RVM internal method that affects the state of the RVM and the consistency of the heap, it must not be preempted by another thread. The `UNINTERRUPTIBLE_PRAGMA` prevents this.

The remaining lines build the list of instructions for the method body. Lines 34 to 37 add the instructions to invoke our static `restoreTIB` method on its owning object, i.e. `this`. Lines 39 to 42 invoke the RVM's object scavenging garbage collection routine, defined in its memory management toolkit, MMTk. Lines 43 to 57 push all the arguments necessary to reinvoke the method, but in its standard, non-specialised form, onto the stack and reinvokes the method, now on the restored TIB. Lines 59 to 61 add the instructions to effect the method return. The entry in the method specialisation array is then set. For brevity we omit the definition of `finaliseMethod` — it associates the instruction list with the BCEL method generator, generates the method, sets the stack limits and issues BCEL cleanup operations.

Finally, we append to the evacuation routines, `copyScalar` and `copyArray`, within MMTk (`org.mmtk.vm.ObjectModel`) with `VM_ObjectModel.hijackTIB( to, 1 );` to flip the TIB to the incremental GC specialisations on object evacuation. The first parameter, `to` is a reference to the to-space copy of the object, while the second parameter indicates the TIB specialisation to use.

5. Evaluation

We have conducted a series of experiments in order to quantify the overheads of the virtualisation framework and the cost of the implicit read barrier when compared with a conventional, explicit barrier, as previously studied for the Jikes RVM by Blackburn and Hosking [5]. The explicit *conditional* barrier of [5] is particularly efficient as it merges tests a) and b) above by assuming that to- and from-spaces can be distinguished by a single bit in an object's address and by flipping a reserved word between 0 and 1 each time a garbage collection cycle is initiated. The barrier adds six native code instructions to each object access, as documented in [5].

Our framework is integrated with Jikes RVM version 2.4.3 (CVS repository version 16 January 2006). Similarly, Blackburn and Hosking's read barrier patch for version 2.3.3 of the RVM is ported and applied to this 2.4.3 version of the RVM.

We use a subset of the SPEC JVM98 and the DaCapo 'dacapo-2006-10-MR2' benchmarks for evaluation. As we are investigating mutator overhead we chose a subset of the benchmarks that per-

form minimal amounts of garbage collection. The reference execution time for each benchmark is measured by running it on an unmodified RVM (no read barrier) with its generational copying mark-sweep hybrid (GenMS) collector and the default heap size of initially 50Mb growing to a maximum of 100Mb. The overheads of virtualisation, and the cost of the explicit and implicit read barriers, are measured by executing the same benchmark on modified versions of the RVM, again with the 'stop-the-world' collector enabled. Because the amount of garbage collection performed is identical in each run, the difference between the average observed execution time and the average reference time provides the average overhead with the garbage collection time factored out. Note that we are only interested in measuring the cost of the barrier, not the cost of any garbage collection operations (e.g. evacuation) that may follow it.

Measuring the cost of the implicit barrier is not easy because the specialised access methods can only be invoked during incremental garbage collection, specifically when the program tries to access an uncopied object. Although we do not have a complete incremental collector, we can nonetheless compute bounds on the barrier cost, based on two extremes.

At one extreme (lower bound), the specialised methods may not be invoked at all — this will happen if the program only touches objects that have already been copied by the scavenger. The additional cost includes that of full virtualisation and one TIB hijack and one TIB restore for each object copied during garbage collection. Note that the hijack, which replaces the object's original TIB pointer with that of the specialised version, will happen even though the program will never get to invoke the specialised method.

At the other extreme (upper bound), all evacuations may take place as a result of the program touching all currently uncopied objects (i.e. no objects are collector-scavenged) — this will only happen if the program performs no memory allocation at all during the collection cycle (assuming a work-based rather than a time-based collector). The overhead in this case is as above, but with one additional call from the specialised method to the original.[2]

We calculate both bounds by artificially adding code to the scavenger of the original 'stop-the-world' collector. The lower bound execution time is calculated by adding a TIB hijack and restore operation each time the scavenger scavenges a live object, thus yielding the overhead that would be imposed on an incremental scavenger in this extreme case. For the upper bound, we do the same but add an additional method invocation on each scavenge representing the additional method call above. The measurements are estimates of performance bounds for the extreme cases, but we can reasonably expect the measured barrier cost in a 'production' incremental collector to lie somewhere between the two.

5.1 Results

Experiments were conducted on a 2.8GHz Pentium IV with 1GB RAM, a 512K level-2 cache, a 16KB instruction cache and a 16KB, 4-way set-associative level-1 data cache with 32-byte lines. The system runs Mandrake Linux 10.1 with a 2.6.15.1 kernel in single-user mode. All results reported are average overheads taken over five runs.

We use the *pseudo adaptive* driver for the Jikes RVM compiler which applies compiler optimisations deterministically according to an advice file computed ahead of time, so as to avoid variations in those methods that timer-based sampling identifies as 'hot'. We measure both the initial performance and that after optimisation in order to measure extent to which the gross virtualisation overheads can be recovered by the optimiser. When applied, the optimisations are run to convergence, i.e. they are applied until no further improvement in execution time is observed.

[2] Note that this extra call could be eliminated by duplicating the original method's body, but at the expense of substantial code bloat.

5.1.1 Virtualisation Overhead

Table 1 shows the initial and fully optimised (converged) execution times for each benchmark for the original RVM implementation (Original) without the read barrier code. The overheads of the fully virtualised implementation in each case (Virtual) is shown as a percentage overhead on the original RVM execution time (for example a 50% overhead on a 10s run equates to a 15s run). The overheads are like-for-like with respect to optimisation (Init and Conv). Without optimisation, full virtualisation adds between around 12%

Benchmark	Original(s)		Virtual (% o/head)	
	Init	Conv	Init	Conv
_201_compress	6.17	5.02	42.63%	8.37%
_202_jess	3.54	2.83	90.40%	2.83%
_209_db	13.72	13.40	11.88%	2.69%
_213_javac	8.83	5.87	91.50%	21.64%
_222_mpegaudio	6.68	3.72	28.44%	5.65%
_227_mtrt	5.08	2.53	30.31%	7.11%
antlr	5.57	4.46	328.17%	5.38%
bloat	15.02	11.51	165.79%	11.99%
fop	5.33	3.21	711.82%	3.74%
luindex	19.36	15.40	273.46%	21.49%
pmd	15.14	9.78	360.69%	20.96%
Min	3.54	2.83	11.88%	2.69%
Max	19.36	13.4	711.82%	21.49%
Geo. mean	8.23	5.82	102.97%	7.75%

Table 1. Virtualisation overheads

and 700% to the execution time (gross overhead). The overheads come from the use of BCEL (see below), the cost of additional beanification and the cost of maintaining the modified TIB structure. After optimisation, these overheads are reduced substantially: the net overhead is reduced to between around 2.7% and 22% after convergence. The benefits that accrue from using the AOS are thus substantial, although this is perhaps not surprising. Note that use of the AOS targets long-running (e.g. server) applications where the full benefits of feedback-directed profiling and optimisation are obtained. For this reason, we do not concern ourselves with the time taken for the optimisations to converge.

5.1.2 Read Barrier Cost

Table 2 shows percentage overheads, when compared to the original execution times in Table 1, for the lower and upper bound execution times for the implicit read barrier (Implicit) and Blackburn and Hosking's explicit read barrier (Explicit).

The cost of the explicit read barrier varies between around 5% and 44% in the fully optimised case; the results are consistent with the reported measurements in [5] to within a few percent.

The implicit read barrier achieves significantly lower overheads than the explicit barrier, except for `_213_javac`. The lower bound figures all show slightly increased overhead when compared to those for virtualisation alone, as is expected. The difference between the two represents the additional overhead of hijacking and restoring the TIB during incremental garbage collection.

The upper bound measures the effect of an additional method invocation when scavenging objects (the cost of invoking the original method body having restored the object's TIB). Notice that even the upper bounds on overheads are still substantially lower than for the explicit barrier, again with the exception of `_213_javac`.

In order to quantify the effects of the CTTk transformations when generating the implicit read barrier implementations, Table 3 shows counts of: i) The number of getter/setter methods generated for beanification by our framework (Virt^{n}s); ii) The number of such methods that are eventually inlined by the AOS, but guarded by

Benchmark	Implicit (Lower)		Implicit (Upper)		Explicit	
	Init	Conv	Init	Conv	Init	Conv
_201_compress	42.79%	8.37%	42.86%	8.37%	24.47%	43.63%
_202_jess	90.89%	2.95%	91.11%	3.00%	7.62%	7.42%
_209_db	12.10%	2.86%	1.12%	2.93%	12.39%	12.31%
_213_javac	59.52%	23.27%	64.35%	23.99%	5.55%	4.94%
_222_mpegaudio	28.62%	5.66%	28.70%	5.66%	48.20%	34.95%
_227_mtrt	42.79%	8.21%	31.34%	8.70%	2.17%	32.02%
antlr	328.17%	5.62%	328.17%	5.72%	11.67%	8.30%
bloat	165.79%	12.27%	165.79%	12.41%	21.17%	18.11%
fop*	711.83%	4.41%	711.83%	4.41%	32.80%	28.40%
luindex*	273.46%	21.49%	273.46%	21.49%	22.74%	19.03%
pmd*	360.70%	21.0%	360.70%	21.04%	8.11%	7.22%
Min	12.10%	2.86%	1.12%	2.93%	2.17%	4.94%
Max	711.83%	23.27%	711.83%	23.99%	48.20%	43.63%
Geo. mean	102.49%	8.15%	80.87%	8.27%	13.13%	15.56%

Table 2. Read barrier overheads. *Note that Blackburn and Hosking's explicit barrier code causes run-time errors on `bloat`, `fop`, `luindex` and `pmd` benchmarks – as reported in the original paper, the read barrier is not completely 'robust' [5]. The figures reported here are based on measurements taken before the programs crashed.

Benchmark	Virt^{n}s	Devirt^{n}s	Code Bloat(%)
_201_compress	5189	2989	10.1
_202_jess	5977	3684	13.7
_209_db	5078	2564	23.2
_213_javac	9159	2668	11.8
_222_mpegaudio	6172	3673	32.2
_227_mtrt	5499	2635	28.1
antlr	14025	4716	21.2
bloat	9871	9781	12.1
fop	9295	3431	24.2
luindex	6661	2513	18.2
pmd	13898	2928	26.3

Table 3. (De)Virtualisation counts and (static) code bloat

Benchmark	BCEL only (Conv)
_201_compress	7.57%
_202_jess	2.47%
_209_db	1.27%
_213_javac	13.12%
_222_mpegaudio	2.42%
_227_mtrt	3.95%
antlr	4.26%
bloat	8.95%
fop	3.12%
luindex	14.09%
pmd	14.83%
Min	1.27%
Max	14.83%
Geo. mean	5.19%

Table 4. BCEL overheads

`ig_tib_test` guards (Devirt^{n}s); and iii) The code bloat, which is the percentage increase in the original RVM code size attributable to virtualisation (additional beanifier methods and TIB structure in particular) and specialisation (self-scavenging method variants). The average code bloat for the chosen benchmarks is around 20%.

Note that in the case of `bloat` nearly all of the compiler-generated methods of i) end up being inlined. However, this does not necessarily guarantee to maximise the performance improvement obtained from the AOS. Recall that inlining adds a cost, namely that of the additional `ig_tib_test`. At the same time, there may be virtual methods in the original code (i.e. not those added by the virtualisation framework) that will be executed with little or no additional virtualisation cost; these methods may or may not end up being inlined themselves. If a majority of virtual method calls end up being made to these existing virtual methods then inlining of the compiler-generated getter/setter methods may yield proportionally less benefit in performance. This explains why increased devirtualisation does not necessarily yield a corresponding increase in performance after optimisation (Table 2).

5.1.3 BCEL Overheads

Table 4 reports the percentage overheads on the original converged execution time (Table 1, column 3) for each application when compiled with Zigman's BCEL patch, but *without* the TIB and method specialisation modifications of CTTk. The results show that use of the BCEL patch alone adds quite significantly to the execution time; this is a result of additional VM data structures that have to be maintained and garbage collected (see Section 3.5). The BCEL overheads account for a substantial proportion of the net overheads observed from the virtualisation and read barrier experiments. The results of Table 4 suggest that, if it were possible to eliminate the BCEL overheads altogether, the virtualisation costs reported above could be reduced to a maximum of around 7%, and the read barrier costs to around a maximum of around 11%. This could be achieved, for example, by working with a native-code VM, rather than the Java-based RVM, or by working to remove the additional overheads in the RVM, as discussed in Section 3.5. There is clearly scope for additional work in this area.

6. Conclusion and Future Work

We have presented a virtualisation and method specialisation framework for Java that facilitates efficient, dynamic modification, of the behaviour of object accesses at run time. The framework can be used to simplify the often complex and error-prone task of efficiently implementing a range of common runtime services. The dynamic, adaptive, optimisation system is exploited to improve performance, partially recovering the overheads of full virtualisation based upon run-time profiling.

Our results suggest that we have reached an excellent compromise between efficiency and ease of use. The implicit read barrier implementation, presented in Section 4, proved relatively easy to build, and the optimisation framework eventually delivered pleasing and extremely promising performance benefits, when compared with an existing conventional barrier implementation. Our experiments also suggest that further, substantial performance improvements could be achieved either by eliminating the overheads that the BCEL library imposes, or by implementing the framework within a native VM implementation.

There are two obvious areas for further development. We are currently implementing a fully operational incremental collector for the Jikes RVM, building on our existing read barrier implementation. A crucial feature of the implementation is the adoption of

the more optimal single-word object model that is facilitated by the use of a "dynamic dispatching forwarding pointer" which uses additional specialisations to direct object accesses in from-space automatically to their copies in to-space [8]. This allows us to reduce the space overhead that is usually incurred when supporting non-destructive object forwarding within incremental copying collectors.

The framework lacks expressiveness and flexibility in the way in which the CTTk transforms byte code and the way specialisations are defined. The addition of a domain specific language (DSL), that provides syntactic sugar for (and therefore hides) many of the BCEL API sequences that are commonly used, will add substantially to expressive power of the framework. In particular, a DSL would eliminate the rather cumbersome way in which individual transformations are currently sequenced within the transformation pipeline.

6.1 Other Applications

The exploitation of dynamic dispatch and proxy objects in OPJ [14] eliminates the persistent read barrier in a similar way to our approach to incremental garbage collection. It would appear to be straightforward to build an OPJ implementation within our framework, building on our current work. Interestingly their measurements suggest that the use of façades results in a slow-down on the order of 20%. It would be interesting to explore the extent to which our approach might reduce these overheads.

The cJVM [17] shares many features in common with our infrastructure except that we support arbitrary and user definable method specialisations and a public API for switching between them. Our framework is eminently applicable to distributed shared memory, and our approach would follow closely that taken by both cJVM and dJVM [18].

An obvious application of our framework is for the rapid deployment and hot swapping of multiple versions of instrumented debug code or enablement of message logging facilities at different trace levels. Previous instrumentation profiling techniques have either relied upon bespoke implementation within the VM, direct modification of application code, or have been aspect oriented [12] in nature. These are seldom hot swappable, therefore requiring (repeated) class loader invocation for enablement and deployment.

Acknowledgments

We would like to thank John Zigman for making available the BCEL patch for Jikes RVM, Steve Blackburn and Anthony Hosking for their read barrier patches to Jikes RVM's MMTk, and Richard Jones for helpful comments on an earlier draft of the paper. We would also like to thank the reviewers for their many detailed comments.

References

[1] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, pages 324–34, San Jose, CA, October 1996. ACM Press.

[3] David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the java object model. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 111–132, London, UK, 2002. Springer-Verlag.

[4] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[5] Stephen M. Blackburn and Tony Hosking. Barriers: Friend or foe? *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[6] Andrew M. Cheadle, Anthony J. Field, Simon Marlow, Simon L. Peyton-Jones, and R.L While. Non-stop Haskell. In *Proceedings of International Conference on Functional Programming*, Montreal, September 2000. ACM Press.

[7] Andrew M. Cheadle, Anthony J. Field, Simon Marlow, Simon L. Peyton-Jones, and Lyndon While. Exploring the barrier to entry — incremental generational garbage collection for Haskell. In *Proceedings of the Fourth International Symposium on Memory Management*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[8] Andrew M. Cheadle, Anthony J. Field, and J. Nyström-Persson. Method Specialisation and Incremental Garbage Collection in Java. Technical Report, Department of Computing, Imperial College, London, May 2007.

[9] C.J. Cheney, *A Non-recursive List Compacting Algorithm*, CACM 13(11), pp. 677-8, 1970.

[10] David Detlefs and Ole Agesen. Inlining of virtual methods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278, London, UK, 1999. Springer-Verlag.

[11] Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 253–282, London, UK, 1995. Springer-Verlag.

[12] David J. Pearce, Matthew Webster, Robert Berry, and Paul H. J. Kelly Profiling with AspectJ. In *Software–Practice & Experience Volume 37, Issue 7*, pages 747–777, 2007, John Wiley & Sons, Inc.

[13] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 294–310, New York, NY, USA, 2000. ACM Press.

[14] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing orthogonally persistent java. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, pages 247–261, London, UK, 2001. Springer-Verlag.

[15] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, 1990.

[16] B. Alpern et al. The Jalapeõ virtual machine. In *IBM System Journal Volume 39 Issue 1*, pages 211–238, January 2000. IBM Corporation.

[17] Yariv Aridor, Michael Factor, Avi Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *Proceedings of the 1999 International Conference on Parallel Processing*, p.4, September 21-24, 1999.

[18] J. N. Zigman and R. Sankaranarayana. Designing a Distributed JVM on a cluster. In *Proceedings of the 17th European Simulation Multiconference*, Nottingham, United Kingdom, 2003.

[19] J. N. Zigman. Bytecode Transformation Tools for Jikes RVM. http://www.wastegate.org/systems.

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

[21] S. M. Blackburn et al. 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*, Portland, OR, USA, October 2006. ACM Press.