

# Idioms in Ovm

C. Flack      T. Hosking      J. Vitek  
S<sup>3</sup> lab, Department of Computer Sciences, Purdue University  
{flack,hosking,jv}@cs.purdue.edu  
CSD-TR-03-017

## ABSTRACT

The need to express important non-Java<sup>1</sup> behaviors confronts the Ovm virtual machine framework no less than any other VM-in-Java project such as JikesRVM[1].<sup>2</sup> Any such project needs mechanisms for the purpose, but different choices in the VM design affect the shape those mechanisms can take. In exploring how a component that makes non-trivial use of such mechanisms—the JMTk memory management toolkit—can be made to interoperate with Ovm and JikesRVM, some Ovm mechanisms have been both contributed to and borrowed from JikesRVM, and some remain distinct. We describe mechanisms we find useful in Ovm because of its design differences from JikesRVM.

We have developed idioms that use familiar Java syntax at the source level, and familiar, recognizable design patterns, for problems such as VM configurability that involve *ad hoc* techniques in prior work, and we transform the idioms to code without the indirection and inefficiency that would otherwise weigh against the familiar patterns.

## 1. INTRODUCTION

There are mechanisms in Ovm with clear similarities to their counterparts in JikesRVM. The similarities attest to our ongoing work with the developers of JikesRVM and the JMTk memory management toolkit, toward using the toolkit with both platforms. Of more interest for this paper, that collaboration also highlights differences. Exploring the differences can help delineate aspects of problems that are common to any such project, differing VM design choices that facilitate one approach more than another, and optional design goals

<sup>1</sup>Java™ is a trademark of Sun Microsystems, Inc.

<sup>2</sup>JikesRVM refers to the Jikes™ Research Virtual Machine—trademark held by IBM Corporation—a distinct project from Ovm. References to JikesRVM in this paper are to illustrate where these two projects with similar objectives take different approaches.

that motivate a mechanism in one VM that another may omit. We present some mechanisms we have found useful in Ovm, and consider the choices in Ovm design that have motivated our mechanisms or helped in their implementation. Some of the design choices in question relate to the Ovm intermediate representation, a thorough introduction to which may be found in Palacz *et al.* [8] but will not be essential for the material in this paper.

The aims of the Ovm framework are to provide an open source testbed for experimenting with language implementation techniques. A specific goal of the project is to support multiple virtual machine configurations. Thus it should be possible to use the framework to construct virtual machines for problem domains as different as real-time embedded systems and cluster computing.

This paper is organized as follows. The remainder of the introduction will introduce classes of idiom recognition we find useful. Section 2 will describe idioms we have implemented in Ovm, including idioms for VM configurability and object models that we believe are improvements over prior work. Section 3 concludes.

### 1.1 Idiom Recognition

Granting that a VM will need to execute operations at run time that Java semantics cannot express (except by the native interface, not suited to the heavily-used, small operations in question) entails that the execution engines, whether interpreters or compilers, accept a richer language as input than JVM bytecode. The problem of compiling source programs into that form can be approached in two ways:

- A Devise a richer extended-Java source language and a new compiler to produce the richer compiled language from it.
- B Use syntactically standard Java for the source language and compile with existing tools, but adopt certain idioms that can be recognized in the resulting bytecode and transformed into other instruction sequences in the extended language.

We choose the second approach and, compatibly with Hovey *et al.*[5], we use the term *idiom recognition* to describe it.

### 1.1.1 Classifying idiom recognition

To place the techniques we use in OVM within the space of related approaches, we propose these dimensions for comparing idiom recognition designs.

#### 1.1.1.1 Unguarded v. guarded

Pottenger and Eigenmann[9] developed a parallelizing FORTRAN compiler that would attempt to recognize certain uses of variables anywhere in the source code, and automatically apply transformations anywhere the recognition dictated.

Hovemeyer *et al.* attempt to recognize sequences of fetch, compare, and store operations that can be implemented with an atomic instruction such as compare-and-swap. To avoid doing the transformation where it is not wanted, though, they introduce a way for the programmer to explicitly mark regions of code where the idiom recognition should be attempted. We propose the term *guarded* for idiom recognition within regions the programmer marks.

#### 1.1.1.2 Heuristic v. tag

Unless the source language provides an explicit way to delimit a region where idiom recognition is wanted, guarded recognition seems to involve regress: what language feature should be recognized to mark a guarded region? Hovemeyer *et al.* use a `synchronized` block on a reference of a pre-arranged reserved type—idiom recognition to guard idiom recognition.

On the bright side, the new idiom has a property the first did not. The heuristic for matching fetch-compare-store sequences could match sequences the programmer would not want transformed, but the test for whether the `synchronized` expression has the reserved type is unequivocal. We will use *tag* to describe an idiom with that property.

A language may have only a few constructs that can be pressed into service as tag idioms; they may look contrived, and they may be simply inadequate to express the range of operations needed. In contrast, heuristic idioms can be based on any appropriate patterns in program text—at best, uncluttered and mnemonic ones—and made as expressive as needed. The risk is of surprising the programmer, either because an idiom was matched incorrectly or because the programmer forgot it was a meaning-carrying idiom. Heuristic idioms guarded by tags help by directing both the recognizer’s and the programmer’s attention to the interesting code regions.

Another risk with heuristic idioms is they may fail to match where intended, either because their matching code is insufficiently general or, more insidiously, because a source compiler has optimized or reordered code that in source form looks like a perfect match. Guarded idioms are again helpful because a guard region implies that the programmer intends an idiom to match within. If nothing is matched, a warning can be issued, instead of silently failing to realize the programmer’s intent.

#### 1.1.1.3 Explicit tag syntax

A proposal, JSR-175, in the Java Community Process would introduce explicit syntax for metadata that can be associated with chosen regions or entities in a Java program. It is likely to offer a form of tag more explicit and less contrived than the tag idioms that can be devised now and, if not expressive enough to convey every meaning we might need, it could still replace tag idioms for the purpose of guarding other idioms.

Important uses of idiom recognition in Ovm will be described in Section 2.

## 1.2 Other background

This section very briefly describes some aspects of Ovm that are not the focus of the paper but will be referred to in later sections.

### 1.2.1 Address and word types

An abstraction to manipulate and dereference arbitrary memory addresses was introduced in JikesRVM and adapted for Ovm. As both projects aim to share a common memory management toolkit, uniformity of this interface has been the subject of much inter-project diplomacy, and so our `VM_Address` differs little from its JikesRVM counterpart. We describe briefly how it is used.

`VM_Address` is the type of a variable or method that holds or returns an address into memory, which may or may not be an object reference. It is declared with instance methods for arithmetic, comparison, dereferencing<sup>3</sup>, and an unsafe “narrowing” cast to type `Object`, which must be used only when the programmer is certain the `VM_Address` is the address of an object. A static method serves as a cast from `Object` to `VM_Address`, which always succeeds. The effect of instance methods is achieved with tag idioms on the methods<sup>4</sup> to replace call sites with inlined instructions that involve no dynamic dispatch. A method that “casts” a `VM_Address` unsafely to another type is trivially implemented by transforming its call sites to nothing.

It is important in using this type that no `VM_Address` into collected space should be live across a point where the thread could be suspended or interrupted by the collector. Methods that manipulate such addresses may require special techniques (*e.g.* an idiom for suppressing yield points) to ensure safety.

`VM_Address` is somewhat painstakingly implemented to behave intuitively, in analogy to its runtime semantics, when used in code that runs at boot image build time when it is an ordinary Java class, and its dereference methods access the memory image being built.

`VM_Address`, unlike any Java primitive type, does not have a size fixed by specification; it is the natural width of a pointer on the underlying platform. This is one reason it has its own arithmetic methods, as neither casting to `int`

<sup>3</sup>JikesRVM places the dereference methods in another class

<sup>4</sup>JikesRVM recognizes these methods with special-case code in each compiler.

nor to `long` would be sure to be appropriate. Another type, `VM_Word`, is similar but for platform-width quantities that are not addresses; it has no dereference operations, and a larger complement of arithmetic and logical ones. Casts between `VM_Address` and `VM_Word` are free in either direction.

## 2. IDIOMS IN Ovm

We present important Ovm idioms as of two types. A few primitive idioms were implemented early and by adding dedicated special code into the Ovm class loading and transformation machinery. These primitive idioms are all tags. Since then, access to most new Ovm functionality has been through new idioms built on existing ones, with the implementing code localized to modules being added. These compound idioms are best seen as guarded idioms with a primitive idiom serving as the guard tag.

### 2.1 Primitive Ovm idioms

#### 2.1.1 Marker interfaces

Marker interfaces are a common idiom to tag an entire class or interface with some special behavior. Ovm, for example, has an `Ephemeral` interface to tag classes that are used *during* boot-image building but need to be excluded from the image, and a converse idiom to tag a class live even if naïve reachability declares it dead.<sup>5</sup>

#### 2.1.2 Pragmas

The pragma exception mechanism in Ovm is the same one we contributed to JikesRVM, but has developed differently in the two platforms. `PragmaException`, a subclass of `RuntimeException`, is the parent of various concrete pragmas that tag methods by being mentioned in their throws clauses.<sup>6</sup> Reflection over throws clauses records the presence of pragmas for special treatment by interpreter or compiler. JikesRVM has developed several subclasses of `PragmaException`, handled directly by one or more of the JikesRVM compilers as flags for inlining, optimization, or interruptibility. In that system, each pragma implies special code in a compiler that tests it and acts appropriately.

Three design choices have made Ovm's pragma an especially useful primitive idiom.

1. Ovm's pragma mechanism allows the class hierarchy to delimit families of related pragmas, detected by testing for their common ancestor. New pragmas can be added within a family without new compiler or interpreter modifications. This is, in essence, a roundabout way of getting tags with parameters; when JSR-175 tagging is available, it may allow doing that explicitly, and be simpler than our mechanism.
2. Ovm transforms incoming bytecode to an IR that is the common input of all execution modes (interpret/compile). A pragma whose effect can be stated as a transformation on this IR can be added without touching an interpreter or compiler and will have well-defined semantics independent of the execution mode.

<sup>5</sup>A class accessed reflectively can require that treatment.

<sup>6</sup>Reserved exception types in throws clauses also appear as tag idioms in, *e.g.*, the RTSJ[3].

3. Ovm's transformations to this IR, which include the effects of pragmas as well as replacement of some high-level Java operations with lower-level sequences, are performed iteratively until a fixed point is reached. This allows the effect of a pragma to be expressed in high-level IR constructs that may even include other idioms as long as cycles are avoided.

These three choices in Ovm have effectively decoupled certain pragma families from interpreter and compiler internals, so that new and useful pragmas can be added as small, self-contained modules, defined near the code they tag.

#### 2.1.2.1 Pragma facilities compared

The notion of pragma hierarchies is a simple extension to the pragma mechanism and could as easily be made in JikesRVM. Its chief benefit, pragmas defining new behavior that can be added without compiler modifications, would be more limited in JikesRVM where there is no single IR with which all execution modes begin, so new behavior must be defined in compiler specifics. In JikesRVM a pragma's semantics may vary with the compiler used to compile a method (which may change with adaptive recompilation), and proposals for new behavior must consider whether every JikesRVM compiler will be able to recognize the idiom.

#### 2.1.3 A useful IR-transforming pragma

`PragmaTransformCallsiteIR`, as the name implies, tags any method to bring about, in the simplest case, the following effect: As the input Java bytecode is converted to Ovm's IR, call sites of the tagged method are replaced by an instruction sequence bound to the pragma, leaving the original method body dead.

Applications:

- Methods that have one behavior in host JVM for image building, and another at run time. The host JVM is unaware of the pragma idiom and runs the original method body; the substitute code is used at run time. This technique is useful to give a method the *same* behavior in both contexts, when the same *implementation* is not possible. Other code can call the same method whether at build time or run time.
- Methods whose behavior is not Java (*e.g.*, cast reference to int; trivial replacement of callsite with nothing). Can achieve desired non-Java behavior by:
  - operations corresponding to standard Java bytecodes but in an unusual and perhaps nonverifying sequence
  - new IR operations without Java equivalents
  - a combination

A given behavior can be achieved in multiple ways, representing points in a design space balancing size and typability of the Ovm IR, simplicity of pragma definition, and complexity of components that consume the IR.

- Permit source idiom of instance method call on types (e.g. `VM_Word`, `VM_Address`, `Oop`) that might not be object references, by transforming the callsite to inline code with no dynamic dispatch.

`PragmaTransformCallsiteIR` is not limited to inlining a fixed sequence of instructions in place of any call site. A pragma in this family can be bound to a rewrite method that receives control in Ovm's IR editing framework, positioned at the callsite of the tagged method. Thus a method can be given behavior defined by a peephole transformation of the IR at the method's call site, and variable according to context or Ovm configuration selections.

## 2.2 Compound idioms

Except for the unchecked cast and bitfield idioms, Ovm's compound idioms based on pragmas involve simple IR transformations at the single invoke instruction carrying the tag. Defining them requires only subclassing `PragmaTransformCallsiteIR` to create a new tag, and binding to it a `Rewriter` that will recognize the new idiom and transform the callsite IR.

### 2.2.1 VM Configurability

Ovm is a framework for building virtual machines, assembling them from components (e.g. thread implementations, memory management algorithms, schedulers, language support) specialized and selected for a problem domain. The job of selecting components and implementations to include in a VM is one Ovm shares with JikesRVM; the latter uses a preprocessor for conditional compilation combined with shell scripts that select among source files containing implementations of the same class.

Ovm approaches the problem with a combination of AspectJ[6] and an embellishment on the Abstract Factory pattern[4] called `InvisibleStitcher`. As in the classic Abstract Factory, an interface or abstract class describes a component with its factory methods, for which one implementation will be selected in a VM configuration.

We embellish the pattern to address the performance objection to indirection through the Abstract Factory and eliminate editing any Java sources to specify the concrete implementing class. Figure 1 illustrates the mechanism.

```
public static ObjectModel getObjectModel()
    throws InvisibleStitcher.PragmaStitchSingleton {
    return (ObjectModel)InvisibleStitcher
        .singletonFor( "ovm.core.domain.ObjectModel");
}
```

**Figure 1: A stitched Abstract Factory method. It returns a singleton of an abstract implementing class. After code transformation, its call sites have been replaced with constant-loads of the singleton.**

Call sites of the `getObjectModel` method may execute either during boot image building or at Ovm run time, with the same semantics: obtain the singleton instance of the concrete factory, which the `InvisibleStitcher` has instantiated reflectively by looking up the abstract class name in the

user's configuration file. Under the host VM, this method is in fact called, and indirections to the `InvisibleStitcher` to obtain the result. At run time, though, no call site for `getObjectModel` remains. The effect of the pragma is that all call sites have been transformed to LDC, with the correct singleton in the constant pool.<sup>7</sup>

The pragma is a nested class of `InvisibleStitcher` and is completely defined in 32 lines of that source file. It was defined without modification to any Ovm interpreter or compiler. During transformation of a method's IR, any instruction that invokes a method tagged with this pragma will be presented to the pragma's `Rewriter` for rewriting. The method descriptor can be retrieved from the instruction, the return type identifies the abstract factory, and that is sufficient information to look up the configured implementation and replace the `INVOKE` instruction with `LDC`.

#### 2.2.1.1 Repeated application

The technique can be applied at more than one level. The example eliminates indirection in obtaining the concrete factory itself. But the factory interface defines methods that return objects too. Those methods can also carry pragmas. When the abstract factory itself is initialized by the first reference, it can add mappings to `InvisibleStitcher` for the concrete classes to be returned by those methods. Call sites for those methods can then be transformed themselves, to `LDC` of a singleton or direct `NEW` of a concrete class, without touching the factory reference. We achieve source code in classic Abstract Factory style, configuration without source editing, correct execution under a host JVM, and runtime performance of direct `LDC` or `NEW`.

#### 2.2.2 Object models

We follow Bacon *et al.*[2] in using the term *object model* for the collection of data that must be retrievably associated with every object to support the language runtime operations. Abstractly, the object model defines these data and the interface for getting and setting them. A concrete model determines how the association is managed; one or more words of object header are commonly involved.

Configurability places demands on the object model at both levels. Different components configured into the VM (different styles of garbage collector, synchronization manager, etc.) call for different abstract object models, with accessors for the specific data the components need. For a given set of components—fixing the abstract model—concrete models that pack the data with more or less cleverness can be compared.

Pluggability, and support for the many memory management strategies of JMTk, are goals common to Ovm and JikesRVM, so both VMs provide for configurable object models. We describe first our mechanism, then what we suggest are its comparative strengths.

#### 2.2.2.1 Object models in Ovm

<sup>7</sup>After transformation to Ovm IR, arbitrary reference types may exist in the constant pool and be loaded using `LDC`.

Figure 2 shows the `Oop` interface, describing the minimal interface our VM requires to the object model. The name suggests an object-oriented pointer; we use it for an address known to be an object, but for which the VM’s view is wanted instead of the programmer’s.

```
public interface Oop {
    Blueprint getBlueprint() throws PragmaModelOp;
    int getHash() throws PragmaModelOp;
    VM_Address headerSkip() throws PragmaModelOp;
    Oop asAnyOop() throws PragmaEatcast;
}
```

**Figure 2: The `Oop` interface: the minimal VM view of an object.**

Most objects, of course, do not implement `Oop`, but a value with static type `Oop` can be obtained by the unchecked cast idiom, and the “methods” of `Oop` invoked with natural Java syntax to obtain the `Blueprint` (Ovm’s type and dispatch structure), default (identity) hash, or the address of the object’s first field or component. The pragmas effect the rewriting of call sites to non-dispatching inlined code.

The `Rewriter` for a `PragmaModelOp` method obtains the correct code to inline by asking the object model, a concrete class extending the abstract `ObjectModel` and configured via the `InvisibleStitcher`. Despite the configurability through a familiar `Abstract Factory` pattern, what remains in the IR at the call site of a model operation is nothing but the bare instruction sequence to retrieve or store the data.

```
public interface MonitorMapper extends Oop {
    Monitor getMonitor() throws PragmaModelOp;
    void releaseMonitor() throws PragmaModelOp;
}
```

**Figure 3: A synchronization component’s view of an object.**

```
public interface MovingGC extends Oop {
    void markAsForwarded(VM_Address fwdaddr)
        throws PragmaModelOp;
    boolean isForwarded() throws PragmaModelOp;
    VM_Address getForwardAddress() throws PragmaModelOp;
}
```

**Figure 4: A memory management component’s view of an object.**

Figures 3 and 4 show interfaces added to the abstract object model by a synchronization component and a simple copying allocator/collector, respectively. Each interface describes only what the corresponding component needs of the object model. Each component manipulates objects simply by casting them (unchecked) to its own model interface and using the interface methods in familiar Java syntax. Given any `Oop o`, an idiom like `(MovingGC)o.asAnyOop()` is recognized and rewritten away to an unchecked cast.

Configurability at the level of component selection and the abstract object model comes for free in our design. Pluggable VM components, with their own abstract model interfaces, are independent and unaware of each other, and can be included in a configuration or excluded at will.

When the components included in an Ovm configuration have been selected, fixing the abstract object model, a concrete object model must be provided that implements all of the chosen interfaces. A new concrete object model is made by subclassing `ObjectModel` to implement the necessary interfaces<sup>8</sup> and register the instruction sequences to be inlined for the supported methods. This concrete class can be created by hand. It contains in one place the implementations for the otherwise independent abstract model interfaces, so it is the place to consider the known relationships and redundancies among the data and devise a compact and efficient concrete format. For example, we can exploit the knowledge that the blueprint will not be retrieved from a forwarded object and arrange for a forwarding address to occupy a blueprint’s slot. The bitfield idiom may be used to advantage in a concrete object model.

To write concrete models exploring different ways to pack a given abstract model is straightforward. For example, two existing models differ only in whether an object’s monitor is kept in a header slot or an ancillary hash map. The simple syntax `o.getMonitor()` obtains `o`’s monitor in either case.

### 2.2.2.2 Object models compared

In comparison to JikesRVM’s pluggable object model mechanism,<sup>9</sup> an Ovm abstract model is more abstract, and an Ovm concrete model is more concrete. JikesRVM partitions the object model into a `VM_JavaHeader` (dispatch, hashing, synchronization), a `VM_AllocatorHeader` (allocator and collector data), and a `VM_MiscHeader` (everything else). Different versions of these three are configured by source substitution. The Ovm stitcher and the pragma mechanism, which can tag methods on any interface, make possible Ovm’s straightforward model of independently selectable interfaces.

On the concrete side, the JikesRVM model allows the `VM_JavaHeader` to export as “available” some unused bits, by way of accessor methods that can be used by `VM_AllocatorHeader` and `VM_MiscHeader` to pack some of their own data into those bits. A mechanism is available for the allocator header and misc header to register the number of available bits each will need. The overhead of mutual nested calls to access the bits presumably is obviated by aggressive inlining in the optimizing compiler.

By contrast, the Ovm approach makes no such effort to automate the composition of a concrete model; we look at the interfaces to be supported and write a concrete model that will work—using available Java mechanisms to inherit or reuse functionality of existing models, of course. We believe this simplifies the Ovm model without appreciably sacrificing composability: given the premium on finding clever packing schemes, and the limited value of exporting bits between arbitrary partitions, we are not sure there was appreciable composability there to be sacrificed. An Ovm concrete object model supplies efficient IR instruction sequences

<sup>8</sup>So operations on the object model work at image building time under a host VM.

<sup>9</sup>Touched on in [2], but to see its structure the CVS repository at [www.ibm.com/developerworks/oss/cvs/jikesrvm/rvm/src/vm/objectModel](http://www.ibm.com/developerworks/oss/cvs/jikesrvm/rvm/src/vm/objectModel) is indispensable.

for the object model operations under any interpreted or compiled mode of execution. An optimizing compiler will achieve further speedup, but is not relied on simply to make the operations reasonable.

## 2.3 Harder compound idioms

Two compound idioms in OVM demand slightly more powerful recognition, though still simpler than those of Hove-meyer *et al.*

### 2.3.1 Unchecked cast

An example of the unchecked cast idiom was seen in Section 2.2.2. An idiom of the form `(MovingGC)o.asAnyOop()`, where the method carries an `eat-cast` pragma, represents an unchecked cast. The idiom recognized (in Java bytecode) is an invocation of any cast-eating method, followed by a `CHECKCAST` instruction; both the invocation and the check are removed. The Java compiler enforces that the cast must be compatible with the declared return type of the method, so by declaring `asAnyOop` to return `Oop`, we still impose a modicum of control on the idiom's use. The same pragma serves in `VM_Address` to declare an `asAnyObject` method, an unrestricted unchecked cast.<sup>10</sup>

The unchecked cast idiom trivially works in code that executes in the host JVM at image build time, with the same semantics except that the cast is actually checked in that case.<sup>11</sup>

### 2.3.2 Bitfield

Opportunities to define fields of contiguous bits within a word are common in a VM, for example in efficiently packing object model data or encoding operands in a compact IR. Bitfield operations in either context will be frequently executed. Java's shift and logical operations are sufficient to manipulate bitfields, and necessary for fields whose widths or offsets are computed at run time. We up the ante, though, with additional design goals for the common case of predefined, noncomputed bitfields.

- Optimize operations on predefined bitfields. Many architectures have instructions with immediate operands that are better for this purpose than code produced for the general case of a Java mask/shift sequence.
- Improve expression of intent in source code, with dedicated methods to get and set portions of a word according to a named bitfield declaration, instead of a longer expression of nots, ands, shifts, and ors.

<sup>10</sup>The two methods are typically used for slightly different purposes. The object model depends on `asAnyOop` to permit the syntax of `Oop` method invocation on objects that are not of type `Oop`. The chief purpose of `asAnyObject` is to allow the programmer to save the cost of checking a cast from `VM_Address` to its actual type when that type is known.

<sup>11</sup>To support the object model's casts from `VM_Address` to `Oop` subinterfaces under the host JVM, the Java objects that represent `VM_Address` in the hosted case are declared by the concrete object model, obtained via the stitcher, and implement the set of interfaces making up the supported object model.

The two goals complement each other, because by defining a simple source level idiom for bitfield manipulation, we are able to use simple idiom recognition to put appropriate operations into the IR. In the alternative, a highly optimizing compiler might still determine by analysis that a sequence of bit operations involve constant mask and shift operands, and emit the desired code. By introducing a simple idiom to make the programmer's intent explicit, we reduce reliance on the optimizing compiler, and can get good bitfield code out of even a simple compiler that recognizes the corresponding IR instructions.

Development of our idiom was further guided by these criteria:

- Support use of bitfields in code that runs at image build time as well as at run time. The idiom we choose must be meaningful Java that will effect the same bitfield operations when running untransformed under a host JVM. The motivation for this requirement is to facilitate code re-use between build-time and run-time operations; for this we are willing to accept that bitfield operations under the host JVM may be less efficient than their transformed equivalents at run time.
- Isolate code from platform word size. We provide bitfield methods whose parameter and return types are all `VM_Word`, and all uses of these methods are valid independent of the platform word size. We also provide methods to set or retrieve a bitfield as `int` or `long`, and these we define as valid whenever the *bitfield* fits in the desired type—a property that depends only on the bitfield definition and its use in the code, not on the architecture.<sup>12</sup>
- Provide methods to access any bitfield either with shifting semantics (normalizing so the value retrieved from or to be stored in the bitfield has its least significant bit in the unit place) or masking only (for speed when the shift is not needed).
- Keep the syntax to declare a bitfield reasonably simple, subject to the other constraints. Avoid requiring the bitfield name to be repeated, or any other text to be varied except the width and shift values.

With `get/set`, `Word/int/long` variants, and unshifted variants, the total complement of bitfield operations comes to twelve methods. These are instance methods on `VM_Word`, and the last parameter of each method is of type `Bitfield`. Figure 5 illustrates the idiom.

The idiom that is recognized and replaced in the IR is the access of a `Bitfield` static field `bf` immediately preceding (because it is the last argument) invocation of one of the bitfield methods of `VM_Word` (`set`, in the figure). This idiom makes the heaviest demands so far on our IR editing framework, because it is the `VM_Word` method that is tagged. The

<sup>12</sup>For this reason, these are our only non-deprecated methods to “cast” between `VM_Word` and primitive integral types.

```

class HashState extends Bitfield {
    static final int WIDTH=2, SHIFT=5;
    static final Bitfield bf = bf(WIDTH,SHIFT);
}
...
w.set(HASHED_AND_MOVED, HashState.bf);

```

**Figure 5: Defining and using a bitfield.** The value `HASHED_AND_MOVED` will be stored in bits 5:6 of the VM word `w`. The bitfield being set, `HashState`, is known by the mention of its `bf` field as the last parameter to the `set` method. The entire idiom is recognized and replaced with IR for bitfield access. The code executes correctly under a host JVM, only less efficiently; the method call really happens.

pragma’s `Rewriter` must look backward in the control flow to find the preceding static field access and identify the bitfield. Our IR editing framework is not especially convenient for looking backward, but this bitfield idiom is much cleaner than the best one we could devise for recognition in a pure linear scan.

An idiom like this one lends itself to introduction in two steps. The first step is to implement the guard pragma with a rewrite method that at first only recognizes the idiom and transforms the IR to a sequence of existing instructions with the right effect. This requires only coding the pragma itself; we touch no interpreter or compiler code, and run and test the system for the intended semantics. In the second step, we introduce the new dedicated IR instructions and update the pragma’s rewrite method to emit them. Only at this point must we update interpreters and compilers, and only to implement the new IR instructions.

### 3. CONCLUSION

From a small set of primitive tag idioms, we have been able to implement several useful compound idioms in Ovm while confining new code to the new modules affected. We have developed idioms that use familiar Java syntax at the source level, and familiar, recognizable design patterns, for problems such as VM configuration that involve *ad hoc* techniques in prior work, and we recognize the idioms to produce code without the indirection and inefficiency that would otherwise weigh against the familiar patterns. As the bitfield example shows, the capabilities of our IR analysis and editing framework affect the universe of idioms we can devise. Our current IR and peephole editing framework has been sufficient to support a selection of useful idioms with comfortable syntax.

There is clearly a tradeoff in adding a common IR to which Java bytecode must be converted before *any* mode of execution. Balancing the benefit of easily added new behavior with well-defined semantics, time is required for the conversion, the IR may be less compact than bytecode or require many allocations, and some IR choices (e.g. register rather than stack based) may not be amenable to direct interpretation or fast nonoptimizing compilation. Something resembling the high-level (HIR) format of JikesRVM’s optimizing compiler would not be ideal as a common IR, as the

time to generate it could impose a fivefold<sup>13</sup> slowdown on a nonoptimizing “baseline” compiler. Ovm’s common IR is a compact, stack-based, executable representation for that reason.

The choice to do transformations on our IR iteratively to a fixed point complicates the tradeoff. A cost of our current representation is the need to resolve relative offsets and variable-size instructions after a transformation pass. There is opportunity to compare different points in the IR design space for the balance of compactness, executability, and cost of manipulation.

### 4. REFERENCES

- [1] Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen Smith. Implementing jalapeño in java. In A. Michael Berman, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324. ACM Press, 1999.
- [2] David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the Java object model. In Magnusson [7], pages 111–132.
- [3] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, 2000. [www.javaseries.com/rtj.pdf](http://www.javaseries.com/rtj.pdf).
- [4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
- [5] David Hovemeyer, William Pugh, and Jaime Spacco. Atomic instructions in java. In Magnusson [7], pages 133–154.
- [6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary: proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, New York, June 2001. Springer-Verlag.
- [7] Boris Magnusson, editor. *ECOOP 2002 — Object-Oriented Programming: 16th European Conference, Málaga, Spain: proceedings*, volume 2374 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, June 2002. Springer-Verlag.
- [8] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. Technical report, Purdue University Department of Computer Sciences, 2003.
- [9] W. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction

<sup>13</sup>Vivek Sarkar, to question in JikesRVM presentation, 6 March 2003

variables. Technical Report 1396, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Research & Development, January 1995.