SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **00**(S1), 1–5 (1999) *Prepared using* speauth.cls *[Version: 1999/06/11 v1.1a]*

Fast Portable Orthogonally Persistent Java™

Alonso Marquez, John N Zigman and Stephen M Blackburn
Department of Computer Science, Australian National University,
Canberra ACT 0200, Australia
(email:{Alonso.Marquez,John.Zigman,Steve.Blackburn}@cs.anu.edu.au)

SUMMARY

A powerful feature of the Java™ programming language is its user-definable class loading policy, which when combined with the namespace independence between class loaders, allows portable implementation of semi-dynamic program transformations. Such transformations can be used for a range of purposes, including optimization and semantic extension.

In this paper we present a framework for semantic extensions in Java. This framework consists of a number of simple but powerful transformations that, among other things, allow us to semantically extend Java to provide orthogonal persistence.

The use of semi-dynamic program transformations lends our orthogonally persistent Java a number of important qualities, including simplicity, portability and a clean model of persistence. Significantly, our implementations are efficient and can outperform in some cases PJama™, a well-known orthogonally persistent Java, which is based on a modified virtual machine.

In addition to describing the application of these transformations to orthogonally persistent Java, we foreshadow their use in a number of other contexts, including dynamic instance versioning and instrumentation.

Introduction

In this paper we describe a framework for *portably* and *transparently* extending standard Java semantics. While such extensions have a number of interesting applications including parametric polymorphism and instance versioning, they are described here in the context of realizing a fast, portable, orthogonally persistent Java.

A critical issue in the design of orthogonally persistent Java (OPJ) is the choice of means by which the Java language semantics are extended to include persistence. While Moss and Hosking [21] outline a number of choices, the approach of modifying the underlying virtual machine has been dominant in the literature until now [2, 17, 10]. In this paper we show that OPJ can be realized without modification to the Java virtual machine or compiler, and furthermore that while this approach is relatively simple, it can outperform an OPJ based on a modified Java virtual machine.

After briefly introducing the concepts of semantic extensions and orthogonal persistence, we go on in section to describe a framework for semantic extensions in Java. This framework forms the practical foundation for our approach to orthogonal persistence for Java, in section . The paper concludes with a brief discussion of other applications for the semantic extensions and some outstanding problems.

Semantic Extensions to Java

A system that extends the semantics of Java must maintain fundamental properties of the language and runtime system including: *separate compilation*, *dynamic class linking*, *modularity*, and *portability*. Original language semantics must be maintained in order to preserve the properties of separate compilation and dynamic class linking. New properties, such as fields and methods, may be added to a class only if the existing API contracts are respected. Furthermore, any semantic extensions must compliment existing semantic change mechanisms, i.e. inheritance and overloading.

To adequately explore semantic extensions for Java, it is important to understand some of the fundamental limitations of the Java language and runtime system. These limitations include the lack of multiple inheritance, the frozen semantics of the system classes, and the difficulties of making semantic changes that extend beyond the encapsulation boundary of a single class. The last of these limitations is common to most object oriented programming languages, and is illustrated by the following examples:

*Correspondence to: Department of Computer Science, Australian National University, Canberra ACT 0200 Australia

Contract/grant sponsor: This work was carried within the Cooperative Research Centre for Advanced Computational Systems established under the Australian Government's Cooperative Research Centres Program. (no number).

CCC 0038–0644/99/S10001–05$17.50

- A model could state that the security level of a ‘manager’ object must be at least as high as that of any ‘subordinate’ object’s security level—a policy enforced by raising the manager’s security level whenever necessary. To implement such a model a significant amount of low-level code must be written. Code to enforce this invariant must be added to multiple classes in an ad hoc, piecemeal manner.
- The addition of a side effect or semantic extension over a field in a class could involve an error prone process of changing any simple references to that field. Enforcing the JavaBeans^{TM} encapsulation convention [23] in the development process will not help if the semantic change is dependent on the place where the field is referenced, as such changes will still be ad hoc and piecemeal.

In both cases, we see that implementing even relatively simple semantics change can be complex in a language like Java, which has a strong notion of encapsulation but provides no high-level support for change.

Previous Work on Semantic Extensions to Java

Recently, papers applying program transformation techniques to Java have added to the large body of literature on program transformations. Semantic extensions, such as runtime type dependent method invocation [7] and type polymorphism [24], augment the Java language thus requiring a non-standard compiler. Two type parameterization [1, 6] apply byte-code transformation at class loading time, these transformations are applied by using a specialized class loader. In both cases, special symbols within the class name are used to denote parameterized classes (’`C<p>`’ and ‘`C$p$`’). Parameterized class declarations and references are trapped during the transformation process and the appropriate classes are dynamically constructed. This is achieved entirely through a custom class loader in [6], and by the use of both a custom class loader and compiler in [1] (enabling the use of custom class file formats). Some toolkits and frameworks for other forms of byte-code transformations include JOIE [9] and BCA [15].

Portability and Dynamic Composition of Semantic Extensions

The ‘write once, run anywhere’ philosophy has been central to the success of Java. The approach taken to semantic extensions is in keeping with this philosophy of *portability*. While this could be viewed as a limitation (restricting the programmer to the mechanisms offered by standard Java) it enables portable implementations to exploit the best Java technology available (as it can run on any virtual machine and use any compiler).

An important objective is that the semantic extensions can be *dynamically* composed. Many semantic extensions such as program instrumentation and profiling are volatile by nature. These semantic extensions should be applicable to other more permanent semantic extensions, such as orthogonal persistence or object instance versioning. Consequently, the specific set of semantic extensions to be applied may only be known at runtime, emphasizing the need for dynamic composition of semantic extensions. In fact, it could even be of interest to dynamically extend the semantics of classes already loaded (e.g. the dynamic instrumentation of a program already running). However, the JVM specification forbids modifications of already loaded classes, so the only possible solution in this case is the use of a modified JVM with advanced support for introspection.

There are a number of ways in which standard Java semantics can be transparently extended, including:

- Modifying the virtual machine to directly implement the semantic extensions either through the existing byte-code set [2, 10], or via additional byte-codes [17].
- Modifying the virtual machine to implement extended reflection capabilities through which semantic extensions can be implemented [17].

- Preprocessing source code [7].
- Modifying the compiler [1, 24].
- Preprocessing byte-codes (statically) [13].
- Transforming byte-codes at class load time [6, 1].

The first two approaches clearly violate the goal of portability as they depends on a modified virtual machine. The next three approaches produce portable byte-codes, but require each producer of semantically extended code to have access to a modified compiler or preprocessor. Moreover, the compilation approach precludes the dynamic composition of semantic extensions. Only the last method is compatible with our goals of dynamic composition and portability. Consequently, we have adopted the last approach to semantic extensions as the basis for our semantic extension framework and our OPJ implementation (a semi-dynamic approach).

Orthogonal Persistence

The semantic extensions proposed in this paper and the methods for applying them can be used for many different purposes. One of the principle applications of these semantics extensions is orthogonal persistence.

Orthogonally persistent systems are distinguished from other persistent systems such as object databases by an orthogonality between data use and data persistence. This orthogonality comes as the product of the application of the following principles of persistence [3]:

Persistence Independence The form of a program is independent of the longevity of the data which it manipulates.

Data Type Orthogonality All data types should be allowed the full range of persistence, irrespective of their type.

Persistence Identification The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system.

These principles impart a transparency of persistence from the perspective of the programming language which obviates the need for programmers to maintain mappings between persistent and transient data. The same code will thus operate over persistent and transient data without distinction.

While the value of orthogonal persistence as a technology for managing complex persistent data has long been acknowledged, difficulties in efficiently implementing orthogonally persistent systems seem to have retarded its uptake in the commercial setting. Of the various challenges associated with implementing orthogonal persistence, one of the most important is that of transparently and efficiently introducing persistence semantics into the programming language runtime system. It is this challenge that the techniques described in this paper address.

Previous Work on Orthogonal Persistence for Java

A comprehensive overview of the orthogonal persistence research field, its goals and its challenges, can be found in Atkinson and Morrison's major review paper [3]. The biennial *Persistent Object Systems* workshops are the primary forum for presentation of work relating to orthogonal persistence, while the series of *International Workshops on Persistence and Java* (PJAVA) are particularly focused on Java.

There have been a number of efforts to extend Java with orthogonal persistence [2, 25, 17, 10]. Most prominent among these are PJama [2] and GemStone/J^{TM} [10]. Both of these try and provide the users with an orthogonally persistent Java environment. In both cases this is achieved by replacing the standard JVM with one that extends standard byte-code semantics to include persistence. In both cases, the virtual

machine, although enhanced, remains Java compliant, allowing non persistent Java programs to execute normally. In their taxonomy of approaches to implementing OPJ, Moss and Hosking [21] indicate the possibility of extending the byte-code set to include explicit read and write barriers[†]. This approach is foreshadowed in the OPJ design described by Kutlu and Moss [17], and is implicit to the byte-code optimization techniques described by [13]. Kutlu and Moss [17] also describe a byte-code modification technique which depends on the implementation of an extended reflection mechanism for Java. Tjasink and Berman [25] describe a very light weight approach to adding persistence to their custom virtual machine, which is targeted at extremely small applications. Preprocessing approaches such as JSPIN [14] and POET [22] replace the standard `javac` compiler. Each user class that directly extends `java.lang.Object` is modified to extend a `PersistentObject` class. Additionally, the user classes are modified to incorporate read and write barriers.

Hosking, Nystrom, Cutts, and Brahnmath [13] have explored techniques for eliminating redundant read and write barriers from Java byte-codes. The byte-code transformation techniques they describe are of relevance to this paper, however currently their approach is static (they use a byte-code preprocessor), and depends on extending the Java byte-code set to make read and write barriers explicit.

A Framework for Semantic Extensions in Java

Java supports the use of user-defined class loaders. A user-defined class loader can modify the content of a class file before calling the `defineClass` method, which finally loads the class into the JVM. The virtual machine will apply all the usual checks to the modified class. By using a user-defined class loader to introduce semantic change, standard compilers and virtual machines may still be used.

The semantic extension of classes in Java is similar to the process of schema evolution, where the semantics of some classes are modified. Using this analogy, our interest lies in an almost 'incremental evolution', where the original semantics form the basis of the extended semantics. Our objective therefore has a similarity to proposals for schema evolution in object oriented databases, where special Data Description Language (DDL) operators are defined that represent high level transformations, such as splitting a class or modifying the class hierarchy. Returning to the Java context and the broader objectives of this paper, there is a need for a family of transformations that allow such incremental extensions to be readily introduced globally and consistently. The development of such transformations is the subject of the remainder of this section.

'Schema Evolution' Transformations

We are interested in semantic extensions that preserve the original semantics of the program. Therefore, a semanticly extended class must respect the API and the semantics of the original class' attributes (i.e. it must maintain the API contract). Asynchronous schema evolution in a distributed environment has motivated the definition of *binary compatibility* in the Java language specification (JLS) [11]. Binary compatibility defines a set of changes that developers are permitted to make while preserving compatibility with existing Java binaries. In addition to the concept of Java binary compatibility we also introduce a similar concept of API compatibility.

Definition 1 (API Compatibility) *A class C' is API compatible with respect to another class C iff:*

- *For each non private field f in C (local or inherited), there exists a field (local or inherited) with the same name and modifiers. Additionally, the field type must be API Compatible as well (references to C replaced*

[†]The terms 'read barrier' and 'write barrier' are used to refer to residency checks and update notifications respectively. Read and write barriers of some form are essential to extending programming language semantics to persistence.

by references to C').

- *For each non private method m in C (local or inherited), there exists a method in C' with the same name and modifiers. Additionally, the return type and signature must be API Compatible.*
- *For each method m in C' (local or inherited) that does not have a corresponding method with the same name and signature in C (a new method), there exists a naming convention that ensures a similar method will not be redefined in any subclass.*
- *The C' superclass must be API compatible with respect to the C superclass. A similar constraint applies to all interfaces implemented by C.*

The previous definition is independent of the class name. Therefore, it is possible to define a transformation that replaces one class for another with a different name, package or class loader, as long as all references to the replaced class are modified. In general, to replace one class for another and maintain binary compatibility we only need to ensure that the external API is consistent. Now we want to be able to replace a class (C) by a new version (C') that represent the semantic extension of the original class.

Definition 2 ('Class Replacement' Transformation) *Given a binary compatible program P, a class C and a new API compatible class C', the 'Class Replacement' transformation replaces every reference to C in P with a reference to C' (except in the classes C and C').*

After applying this transformation, the original class may be discarded from the transformed program if it is no longer accessible. It is trivial to extend the previous definition to replace a set of classes (a subprogram). In general, given a program P, the 'class replacement' transformation of C for C' makes sure that the transformed program will be binary compatible. From the point of view of byte-code modification, the only change needed is the replacement of all references to class C for references to the new class C'. For example, a **`getfield`** [‡] instruction like:

```
getfield C/f1 Ljava/lang/String;
```

will be replaced by

```
getfield C'/f1 Ljava/lang/String;
```

The binary and API compatibility rules are simply syntactic, so they do not take semantic changes into account, allowing two binary compatible classes to have incompatible semantics. The concept of binary compatibility could be seen as a very limited case of class evolution, which does not account for semantic compatibility and more general schema evolution. Binary compatibility and API compatibility only ensure that there is consistency between the use and the definition of fields and methods. We are going to limit the use of the 'class replacement' transformation to semantically sound cases, where the new class is a semantic extension of the old one, in which case the semantic extensions are compatible. Moreover, if the semantic extensions do not modify the semantics of any previously defined methods or fields, we can assume that the transformed program is semantically equivalent to the original program. However, it is not always easy to avoid modifications to method semantics, even when an API compatible class does not modify the original method byte-code (other than through the application of the 'class replacement' transformation). The most obvious cause of such implicit semantic side effects is with respect to introspection operations such as `getClass()`, which may return a class with a different name, package or class loader. Fortunately, the use of these methods is not common and can be automatically detected and intercepted, allowing the property of semantic transparency to be preserved in the semantic extension transformation.

[‡]We have adopted the notation used by JASMIN (Java Assembler Interface) [19] for ASCII descriptions of Java classes.

'JavaBeans Compliance' Transformation

The 'JavaBeans' specification [23] describes the form and style of a JavaBeans compliant object. The form includes a description of the methods for retrieving and setting the properties (fields) of a JavaBeans compliant object (section 8.3 of [23]). Limiting direct access to fields by methods executing over the object owning those fields enforces the JavaBeans specification, greatly enhancing the level of object encapsulation.

The transformation of a class into a 'JavaBeans' style allows the implementation to be manipulated without breaking the API contract. This provides a significant decoupling of the implementation from the form of an object. We use this transformation to simplify the process of semantic extensions.

The transformation must account for the security level of the fields considered. The security level of the field itself is thus given to the accessor methods that replace it. The fields in the transformed class becomes private, thus restricting access to the field via the accessor methods.

Definition 3 (JavaBeans Transform) *Each field that is visible or potentially visible outside the class that contains it is transformed into a JavaBeans like form, i.e. non-private members. For a particular field of the form:*

```
<protection> [static] <type> <field>;
```

where the field is a class or instance field with type type*, field name* field*, and a* protection *level of either* **`package`**, **`protected`** *or* **`public`**, *the transformation of the field results in the following field and accessor method definitions:*

```
private [static] <type> <field>;

<protection> [static] final <type> get$<classname>$<field>() { return <field>; }
<protection> [static] final void set$<classname>$<field>(<type> arg$<field>) {
  <field> = arg$<field>;
}
```

The example in figure 1 defines a simple class '`Example`' in package '`AU.edu.cs`' before transformation, and figure 2 specifies the class resulting from the JavaBeans transformation.

```
package AU.edu.cs;
public class Example {
  public    Example ( int a ) { this.aField = (double)a; }

  protected double  aField;
}
```

Figure 1. Class Example before JavaBeans Transformation

To accommodate this transformation, all code that directly refers to a field within an object is itself transformed into the appropriate method call. The manipulation of the code can easily be done at the *byte-code* level.

A method in one class might access a field in an object instance or class of a different class directly. The code which performs this operation must be transformed to use the appropriate accessor methods. This change is easily made at the Java byte-code level and must be made in all classes that perform this type of access. Only two cases are considered: instance and class field accesses.

Each instance field access (*getfield* or *putfield*) is replaced by a method call (*invokevirtual*) to the get or set accessor method. Each class field access (*getstatic* or *putstatic*) is replaced by a method call (*invokestatic*) to the get or set accessor method. This replacement (along with the correct *constant pool* method descriptor) is stack neutral.

```
package AU.edu.cs;
public class Example {
  public    Example ( int a ) { this.aField = (double)a; }

  protected final double  get$AU$edu$cs$Example$aField ( ) { return aField; }
  protected final void    set$AU$edu$cs$Example$aField ( double arg$aField ) {
    this.aField = arg$aField;
  }

  private   double  aField;
}
```

Figure 2. Class Example after JavaBeans Transformation

'Abstract Class Replacement' Transformation

Simple extension of the semantics of a class may not be sufficient to encapsulate the variation in behaviours desired by the semantic extension. To facilitate the encapsulation of different behaviours for the same type of object an 'abstract class replacement' technique can be used. A particular class can be replaced by an abstract class and one or more subclasses can implement the different desired behaviours.

Definition 4 (Abstract API Compatibility) *An abstract class Ac is Abstract API Compatible with respect to a class C that conforms to the JavaBeans encapsulation convention [23] iff:*

- *For each non private, non static, non constructor method m in C (local or inherited), there exists a non-final method in Ac with the same name and access modifier. Moreover, the return type and signature of m must be Abstract API Compatible as well (references to C replaced by references to Ac).*
- *There does not exist a method in Ac without a corresponding method (of the same name and signature) in C.*
- *Ac does not have any non-private fields.*

The 'Class Replacement' transformation is extended with a replacement of a class for an abstract API equivalent class.

Definition 5 ('Abstract Replacement' Transformation) *Given a binary compatible program P, a class C that conforms to the JavaBeans encapsulation convention [23] and a new abstract API Compatible class Ac, the 'Abstract Replacement' transformation replaces every reference to C in P with a reference to Ac (except for references to class constructors and superclasses). The classes C and Ac are not modified.*

It is trivial to extend the previous definition to replace a set of classes for a set of abstract classes. In general, given a program P, the 'Abstract Replacement' transformation of C for Ac ensures that the transformed program will also be binary compatible. This transformation enables the possibility of using more than one implementation class for the same abstract class. In fact, the original class does not need to disappear from the original program. As such, it is possible to simulate some cases of dynamic evolution.

'Proxy Hierarchy' Transformation

The generation of proxy objects for each real object can enable a decoupling of these objects from the application that manipulates them. Thus the 'Proxy Hierarchy' transformation enables the replacement of one class hierarchy by another class hierarchy that reproduces the original class hierarchy semantics.

Definition 6 (Proxy Hierarchy) *Given a binary compatible program P and a class hierarchy H, for each class A belonging to H, a new proxy class A$proxy is generated. If A is the hierarchy root, a new private field*

`proxy$` of class A is generated. Additionally, for each method `m` belonging to A, a new method with the same name, modifiers, signature and return type is implemented in A$proxy. This new method is implemented as a straight call to the original method over the proxy field. All the constructors are extended in a similar way.

A Declarative Semantic Extension Language

Byte-code transformations are error prone. A simple mistake during the transformation process can destroy type safety or the semantics of the program, and may lead to the byte-code modified class being rejected at class loading time. Therefore, a type-safe and declarative way to specify program transformations is essential to the practical application of byte-code transformations. To this end, we have defined a syntactic convention for specifying the most common transformations using an independently defined class that specifies all of the semantic changes.

Our framework allows for both semantic extension of methods and the inclusion of special 'triggers' (similar in concept to database triggers) that are activated on the occurrence of particular events such as the execution of **`getfield`** or **`putfield`** byte-codes. The concept of triggers is illustrated in figure 3, where a special method `pre$getfield$` is defined inside class `java$lang$Object$` which specifies that a 'pre-trigger' be executed prior to each execution of a **`getfield`** instruction for all classes that inherit from `java.lang.Object`. In this example, the trigger instruments a trace message. The `showAccess` method of the class `Tracer` could dynamically determine whether or not the trace message should appear.

```
public class java$lang$Object$ {
    protected final java.lang.Object pre$getfield$() {
      if(Tracer.showAccess(this))
        System.out.println("Going to access object "+this);
      return this;
    }
}
```

Figure 3. A declaration of a parametric semantic extension to achieve trivial instrumentation.

Our approach is to produce a 'black box' framework where the user defines semantic extensions simply by implementing special extension classes. The special extension classes must follow a particular naming convention (like the JavaBeans framework), and use the following syntactic convention: where

```
class EXTENSION_CLASS_NAME {
  // new fields
  // new methods
}
```

Figure 4. Syntax of special extension classes.

`EXTENSION_CLASS_NAME` is the name of the target class, C, terminated with a '$' and with '.'s replaced by '$'s:

```
EXTENSION_CLASS_NAME=C.getName().replace('.','$') + "$"
```

In order to avoid semantic conflicts, semantic extension classes cannot contain abstract methods as all methods must have a body. New methods may be either normal methods (as needed by the semantic extension) or special trigger methods triggered by field accesses, array access, method invocations and parameter loading. The framework supports both 'pre-triggers' (triggered prior to execution of the relevant

byte-code) and 'post-triggers' (triggered immediately after the execution of the relevant byte-code). 'pre-triggers' include: `pre$getfield$`, `pre$putfield$`, `pre$aaload$`, `pre$invokevirtual$`, etc., while 'post-triggers' include: `post$getfield$`, `post$putfield$`, etc. All fields and methods of the class to be manipulated are automatically included in the transformed class (see section).

The semantic extension framework is invoked when a user class is loaded. This action triggers a special semantic extension class loader to search for and load any semantic extension classes that are applicable to the user class being loaded. The semantic extension classes which are applicable to the user class are then used to apply the following transforms over the user class being loaded:

- Add any new fields.
- Apply the 'JavaBeans compliance' transformation that allows all fields to be made private (section).
- Add any new methods. New method names are suffixed with `$+EXTENSION_CLASS_NAME` to ensure that conflicts with existing methods of the transformed class and its subclasses are avoided.
- Add any triggers.

The mechanism for adding new fields and methods depends on the class being transformed. If the class to be transformed is a system class or inherits from a system class, then a proxy (section) must be used, otherwise the class itself is modified. The framework does not propagate triggers into system classes, as the system code cannot be modified. However, the framework user can redefine any system class method and so propagate the semantic extension. Triggers are propagated to all subclasses of the transformed class. In particular, any new field, method or trigger introduced in a system class will be propagated to all the descendents. If a subclass of the extended class is a system class, the subclass will be extended as well. For example, the pre-trigger `pre$getfield$` declared in figure 3 will be propagated to all user classes.

The ByteCodeVisitor Hook

The declarative approach outlined above limits the number of program transformations available. Consequently, a more general hook is provided by the class transformation framework. By extending the default semantics of the `ByteCodeVisitor` class, the user can specify more complex class transformations. This class corresponds to the implementation of a *visitor pattern*[§] which is applied to all the class `MyClassFile` (a Java class that represents a class file at an abstract level). In addition to its application to the `ByteCodeVisitor`, the visitor pattern has been used extensively in the implementation of the framework itself.

```
public class ByteCodeVisitor {
  public init(ConstantPool cp);
  public boolean isByteCodeModifiableMethod(Instruction ins);
  public LinkedListIns replaceByteCode(
                       Instruction nextInstruction,
                       LinkedListIns outputInstructions);
  public Field replaceField(Field f);
  public Method replaceMethod(Method m);
}
```

Figure 5. The `ByteCodeVisitor` hook.

Framework Implementation

[§] A visitor pattern applies a transform object to all the nodes in a structure.

A first prototype of the framework has been implemented. It has been applied to the implementation of a portable OPJ (section) and a portable object versioning framework. We have implemented the framework using the 'PoorMan' library that provides facilities for class file parsing and basic class transformations [6].

We will assume in the rest of the paper that the 'JavaBeans compliant' transformation has been applied over all user classes.

Java Byte-code Modification

We are interested in portable semantic extensions that introduce a minimum of program execution overhead. A family of byte-code transformations can be defined by replacing patterns of byte-code instructions with semantically equivalent instruction sequences. It is essential that any such transformations are stack neutral (preserve stack consistency). For example, the `getfield` byte-code instruction over a non-primitive field could be replaced by any method invocation that returns an object of a compatible type.

Since Java byte-code is tightly coupled with the language itself, we can associate very simple byte-code patterns with particular instructions in the Java language. For example, the invocation of any instance method corresponds to the byte-code instruction `invokevirtual`, an invocation to a static method to `invokestatic`, etc.

Trigger Implementation

In general modifying the byte-code of a method is quite simple. For example, the insertion of a post-trigger for an operation that leaves an object reference on the stack only requires the duplication of that reference and the invocation of the trigger method (with an object reference as its only argument). Similarly for pre-triggers.

The manipulation of the byte-code for a method means that other issues such as relative branch distances and exception ranges may need to be modified to account for the increased number of instructions in the method code. However, these issues are taken care of by the tools used to manipulate the class files, such as the 'PoorMan' library.

Semantic Extensions Over User Classes

It is always possible to extend the semantics of classes loaded using the semantic extension class loader.

Semantic Extensions Over Non-user Classes

The JVM loads system classes, such as `java.lang.Object`, by using the system class loader. This fundamental constraint precludes the byte-code transformation of system classes. Additionally, the array class is generated internally to the JVM and not loaded from a class library.

To introduce orthogonal persistence, ideally, it would be possible to modify `java.lang.Object` (the root object). This can be achieved by generating a complete set of modified system classes in advance in the filesystem. However, this solution is not satisfactory as it cannot be applied to Applets.

If the semantic extension only involve static attributes, we do not need to replace the system class by its semantic extensions. Only the references to extended attributes need to be modified to make reference to the extended class methods. However, if the semantic extension involves instance attributes, we will need to replace the system class hierarchy by a 'Proxy Hierarchy' (section) adding any semantic extensions to the proxies, and replace the system class hierarchy with the proxy hierarchy.

Interfering with Java's Introspection

Applying the 'JavaBeans compliance' transformation (and others) to a class modifies the API of the class. As a result, some of the `java.lang.Class` methods such as **`instanceof`** and `getfield` will need modification. These methods must mask the modification made to the user classes. The system classes must also be modified to return the modified `java.lang.Class` class `java$lang$Class$`. These modifications can be introduced using the pre-triggers and post-triggers described earlier, see figure 6.

```
public class java$lang$Class$ {
  private java.lang.Class realClass;
  private java.lang.Class proxyClass;
  public String getName() {
    return realClass().getName().replace('$','.');
  }
  public Object newInstance() {
    return proxyClass.newInstance();
  }
  ...
}
```

Figure 6. Semantic Extension to java.lang.Class

Framework Overhead

The class transformation overhead for most applications is not significant. The initial overhead of applying the byte-code transformation to a hundred or so classes is only a few seconds.

The runtime overhead is dependent upon the transformations applied. The 'JavaBeans compliance' transformation does not generate any real overhead. This is due to the accessor methods being final and the consequent inlining of these methods by the JVM. The replacement of a class for a subclass of that class adds little to the overhead. However, the overhead of using a system class with a trigger is significant, as a result of the calls through the associated proxy object.

A Portable Orthogonally Persistent Java

Once equipped with a mechanism for transparently inserting read and write barriers into Java byte-code at class load time, it is possible to build a portable OPJ. There are a wide range of issues that must be addressed in making such an implementation. However, the focus of this paper is on the use of program transformations to gain portability, so we give only cursory treatment to the broader implementation issues here. The interested reader is referred to other published papers [5, 4] for a detailed account of other aspects of our approach to orthogonal persistence for Java, including the development of a scalable architectural foundation and the clean and efficient integration of transactional concurrency control into OPJ.

Implementation Issues

Before describing our first and second implementations of OPJ (called ANU-OPJ), we briefly outline a number of important issues that impact on any OPJ implementation and that have influenced our key design decisions.

Barrier Implementation Strategies In section we described a technique for inserting read and write triggers, but did not address the question of how the barriers were implemented. There are many dimensions to this question, including a choice of barrier algorithms, and implementation contexts (Java or native code). Although the choice of barrier algorithms is somewhat interrelated with the issues of object swizzling [¶] strategy, meta-data maintenance, and Java's system classes (which are separately addressed below), some aspects are sufficiently fundamental to be divorced from those issues.

A basic choice between unpacking laziness and eagerness underlies the implementation of read barriers. Barriers may either be inserted before a call to `getfield` ('pre-barrier'), or after a call to `getfield` ('post-barrier'). A pre-barrier checks for the presence of the object containing the field to be accessed, while a post-barrier checks for the presence of an object *referred to* by the field that was accessed by the call to `getfield`. The pre-barrier represents a lazier approach, and requires barriers over both primitive and reference fields, while the post-barrier need only be applied to reference fields and is more eager.

Swizzling Strategy The choice of swizzling strategy is a significant design decision for implementers of persistent systems [20, 16] and has had a major impact on the way we have approached our OPJ implementations. For the purposes of this discussion, four broad options exist: *eager swizzling*, where all references are swizzled as soon as an object is faulted into memory; *lazy swizzling*, where references are swizzled when they are first traversed; *no swizzling*, where references remain in store format and a conversion is made each time such a reference is traversed; and *eager swizzling to handles*, where handles hold references to OIDs and an in-memory pointer to the referenced object is established the first time the handle is traversed.

For each of the first three strategies, the relevant OID must, by definition, be available at swizzle time. This is trivial when eager swizzling is employed—the swizzle takes place while the persistent object is being unpacked from store format. If no swizzling is employed, the OID can be stored in place of a native reference in the object, in which case it will always be available when the reference is traversed. If a lazy strategy is employed, the OID must be somehow associated with the reference field up until the point at which it is swizzled. This can be done by storing it in place of the object reference, by associating extra storage with the object for accommodating OIDs, or by using some structure external to the object such as a lookup table.

These alternatives become severely restricted when Java's strong typing and our decision to maintain portability are taken into account. Strong typing prevents the overloaded use of a reference field as both an OID and a Java object reference, which makes efficient implementation of a lazy swizzling strategy difficult. Similarly, using reference fields to accommodate OIDs for the non-swizzling strategy is problematic. Both can be easily overcome if portability is sacrificed. The use of handles is also problematic with respect to typing as object references must be replaced with references to handles, which is difficult to achieve efficiently and transparently.

Associating Persistent Meta-data with Objects The implementation of read and write barriers depends on the barriers being able to determine and record the state of the referenced object—whether it is in memory and whether it is dirty. An important question is how meta-data describing this state can be associated with each object. Without the constraint of portability, the solution is simple—record the state internally to the virtual machine, either as part of the internal object representation, part of an object handle, or in some other internal structure. However, portability dictates that this information be recorded either in an external structure such as a hashtable, or as additional fields and methods in each persistent object.

[¶]The term 'swizzle' is used to refer to the translation between persistent pointers (OIDs) and in-memory pointers (references).

Limitations of the Default Constructor The faulting and reification‖ process involves the generation of new Java objects. These objects could be created through the default constructor using the `newInstance` method in `java.lang.Class` class. However, in many cases the default constructor is private or not defined in the class. Moreover, introspection methods like `newInstance` are quite expensive**. One solution to this problem is to create a special constructor for each user class for use by the read barrier. This is typical of the sort of problems that motivate our development of mechanisms for parametric changes (like the addition of new methods) over sets of classes (section).

A First Portable Orthogonally Persistent Java

The first portable approach to implementing orthogonally persistent Java (the 'shell' approach) is based on a simple model for the implementation of field access barriers. The mechanisms used to implement this approach are described below.

Swizzling and Faulting Method

The analysis of swizzling strategies outlined in the previous section lead to the adoption of an eager swizzling strategy for the first implementation of ANU-OPJ. An eager swizzling strategy requires the immediate swizzling of OIDs to object references when an object is faulted into memory. This can easily be achieved if all necessary information is available at the time the object is faulted. If a particular object is already in memory, and an OID referring to that object is loaded when the object is faulted into memory, then that OID can be translated into the correct object reference. If an object that is faulted into memory contains a OID referring to an object that is not currently in memory, then that reference must be resolved to the correct object. In this case an actual representation of the object must be created, (that object is called a 'shell').

The creation of a 'shell' object acts as a place holder for the final object. The 'shell' object remains uninitialized until it is necessary to fault in the object. This is an eager swizzling and lazy unpacking mechanism. At the point the object is faulted, all the fields of the objects are instantiated. A shell object consumes as much memory as the instantiated object it represents, and in the case of large objects this may represent a large amount of wasted space.

At the time a shell object is created its type must be known. The shell created must be of the type referenced (and as this may be a subtype of the type of the field containing the reference) the field type itself cannot be used to determine the type of shell to be created. A reference to an object when packed (into the persistent store representation) specifies the OID and the type identifier, this reduces the overhead when the references are swizzled—eliminating a lookup to determine the type of the object being referred to.

To facilitate the construction of a shell representation of an object a special Constructor is generated for each class. Meta-data must be associated with each shell indicating its OID and whether it has been faulted in or made dirty (this data must be initialized when the shell is instantiated). A new constructor taking a type hidden from the user is defined to eliminate any conflict with the default constructor for that type.

The faulting of an object is triggered by reading or writing to a field of the object. The read and write barriers inserted before the `getfield` and `putfield` operations. Every field access, whether it is accessing a field in the current (**`this`**) object or an external object requires a read or write barrier before the field access. The implementation of the read and write barriers are described in the following section.

‖We use the term 'reify' (*'to make real'*) to refer to the translation of an object from an external store form to an internal Java form.

**Our experiments with JDK1.2 suggest that the overhead can be up to one thousand times that of . . .

Barrier, Faulting and Swizzling Implementation

The previous section introduced the general process by which the read and write barriers are achieved and indicated the supporting meta-data for those processes. Here, the read barrier and write barrier transformations are described in more detail.

The read and write barriers are applied to all operations that access a field in an object. In particular, the use of `getfield` and `putfield` must accompany the appropriate read and write barrier respectively.

```
getfield MyClass/field Ljava/lang/int      ; a.field
;
;        translates to
;
dup
invokevirtual MyClass/readBarrier()V       ; a.fault ( )
getfield MyClass/field Ljava/land/int      ; a.field
```

Figure 7. Translating `getfield` to Incorporate Read Barrier

When a field in an object is read, the data contained in that field must be valid. To ensure that valid data is read, a read barrier is inserted prior to the reading of the field. The read barrier takes the following steps to ensure that valid data is read:

1. Fault the object if necessary.
2. Execute `getfield`.

Faulting an object requires all the field values to be instantiated. For primitive fields this is simply a matter of decoding the store values and writing them to the corresponding fields.

Non-primitive fields (object references) must refer to a valid object. The object referenced must be either an empty shell (yet to be faulted) or a complete object. An in memory mapping is maintained between OID and objects to facilitate the lookup process. In the event a new shell must be created, the type id of the shell is maintained with each reference in the store, reducing the need for a lookup. This process is as follows:

1. Lookup OID to reference mapping.
2. If the object is not resident then create a shell.
3. Set reference to the object in memory.

The example in figure 7 shows the insertion of a read barrier before a `getfield` operation. Each class has code inserted to allow it to fault in the data and set the initial faulted and not dirty status. The read barrier code is encapsulated in a method, and a call to that method is inserted prior to the `getfield` operation.

```
putfield MyClass/field Ljava/lang/int      ; a.field = v
;
;        translates to
;
swap                                       ; swap objref and value
dup
invokevirtual MyClass/writeBarrier()V      ; write barrier
swap
putfield MyClass/field Ljava/land/int      ; a.field = v
```

Figure 8. Translating `putfield` to Incorporate Write Barrier

Similarly to that of the read barrier, the write barrier must ensure that the object is faulted into memory prior to any modification of the object's fields. The write barrier must take the following steps to ensure that the object to be written to is valid, and that the system knows the object has been modified:

1. If necessary fault in the object.
2. Make the object dirty.
3. Execute `putfield`.

The example in figure 8 shows the translation of a simple field write operation (`putfield`) into a write barrier and write operation. This operation is not as straight forward due to the stack manipulation that must take place to ensure that the method invocation is correct and the additions are stack neutral.

A dirty list is maintained to facilitate the update of the objects. As the JVM has its own internal representation for the objects, it is necessary to translate the objects from the persistent store form to the JVM form. The copy out nature of this interaction necessitates the dirty list being used to pack the modified objects into the persistent store form before the transaction is committed.

The write barrier mechanism can also be used to incorporate some form of concurrency control. Prior to writing the data to the object, the write barrier can attempt to obtain a write lock from the underlying store for that object. If the write lock is granted, then the remainder of the write barrier is executed as normal. If a write lock is not granted then an exception is thrown, possibly resulting in an aborted transaction.

System Classes

The byte-code of the system classes cannot be modified. As a result the way in which the system classes are dealt with must be different to that of the user classes. The introduction of the OID and other meta-data fields into the system classes is quite expensive, as it necessitates the introduction of proxy classes, see section .

However, most of the system classes are final and immutable, (such as `java.land.String`). For final and immutable classes it is not necessary to introduce any write barrier, as the objects can not be modified. Furthermore, by eagerly faulting final and immutable system class objects the need for read barriers and the supporting meta-data can be removed. This simplifies the treatment of these objects by eagerly faulting and reifying them during the process of reifying other objects.

In the implementation of ANU-OPJ the Swing library classes were declared transient in order to avoid unnecessary overheads for objects which we did not want to be persistent.

Arrays are a unique mutable class that cannot be extended. In the initial implementation arrays are eagerly faulted in order to simplify their manipulation. In order to avoid the overhead of proxies, a pre-trigger was implemented over array update operations. The write barrier would invoke a native method which would maintain the dirtiness state of the array. This method resulted in a doubling of the execution time for some of the OO7 benchmark queries.

A Second Portable Orthogonally Persistent Java

The ‘shell’ approach of the first implementation has two key drawbacks. First, by allocating space for an object at the point when the first reference to that object is encountered, it eagerly consumes space. Secondly, it requires read barriers to be placed before *all* accesses to object fields—all `getfield` and `putfield` operations (even if they refer to local instance fields). The initial execution of the read barrier faults the object data in, but subsequent executions of the read barriers are redundant.

The primary difference between our first and second OPJ implementations is in the use of *proxy* objects instead of shells. The following sections give a detailed account on the use of proxies in the second ANU-OPJ implementation.

Shells, Handles and Façades

The use of shells in the first implementation was motivated by the need for a portable solution to the problem of swizzling. At the point of reifying an object, OIDs must be swizzled to Java object references. If the referenced object is already in memory this is trivial, but if the referenced object has not yet been faulted in, the swizzle becomes problematic. In our first implementation the problem was resolved by swizzling the OID to a reference to a new (uninitialized) object of the appropriate type (called a shell), which is reified (initialized) only when a read fault occurs over that shell. An OID to object mapping is used to ensure that all subsequent swizzles of a given OID are to the same shell.

If one is prepared to modify the virtual machine, the swizzle problem can be overcome simply and transparently (although with some runtime overhead) by using indirection blocks called 'handles' [2]. When handles are used, the swizzling of OIDs to handles and the association of handles to real objects can be temporally decoupled, allowing the referenced object to be faulted in only when a read fault occurs over the handle.

A portable alternative to the use of shells and handles is the concept of *façade* objects. A façade object is a light-weight representation of the *real* object it masquerades as, the façade standing in for the real object until such time that the real object is required. The façade has two important functions: it must behave like the object that it represents up until the point at which it is first accessed, and once it is accessed, it must transparently replace itself with the real object. For the façade to be able to stand in for the real object, the two must be type equivalent, an issue that is addressed in section below.

The replacement operation comprises three steps (see figure 9).

1. Faulting and reifying the real object.
2. Replacing all references from referring objects to the façade with references to the real object.
3. Calling the corresponding method in the real object and returning its result.

Since all references to the façade are replaced with references to the real object, all subsequent executions will call the real object directly, thus avoiding redundant read barrier executions. Furthermore, by using

```
public class T {
  // ...
  public double calculateIntegral(double x1, double x2) {
    double result;

    // perform calculation

    return result;
  }
}


public class T_F {
  // ...
  public double calculateIntegral(double x1, double x2) {
    T t = getRealObject$();
    changeReferencesTo$(t);
    return t.calculateIntegral(x1, x2);
  }
}
```

Figure 9. The `calculateIntegral` *method for a 'real' object, an instance of class* `T` *(top) and for a 'façade' object, an instance of* `T_F` *(bottom).*

façades, the wastage associated with creation of shells that are never reified is reduced to that of the creation

of façades that are never accessed—which is significant in the case where the average shell size is anything other than small.

The remainder of this section examines aspects of the implementation of façades in more detail.

Field References

Façade objects do not contain any of the data of the objects they represent—they must intercept any attempt to reference the real object and redirect such references in the manner described above. While the previous section describes a mechanism for trapping accesses to the real object through method invocations, this mechanism cannot be directly applied to field accesses.

The problem of direct field accesses can be avoided by hiding the visible fields inside accessor methods or by constraining the façade transformation to classes without public or protected fields.

Ensuring Substitutability of Façades by Using Abstract classes

The use of façades is premised on façades being transparently substitutable for the ‘real’ objects they masquerade as. The conformance to the JavaBeans encapsulation convention [23] reduces the problem of substitutability to that of ensuring that the façade implements the same methods as the real object.

This property allows an interface or abstract class to be generated for each class (or abstract class) which includes all the methods that are implemented by both the façade and the real object. The use of interfaces or abstract classes gives the façade and the real object type equivalence, allowing the same reference types to refer to both façades and real objects. The ’Proxy Hierarchy’ transformation is used to generate a façade hierarchy. The ’Abstract Class Replace’ transformation over the original user class hierarchy and the façade hierarchy is used to generate a Java binary compatible program. We do not need to modify any constructor calls. Only the new trigger methods could generate new façade instances.

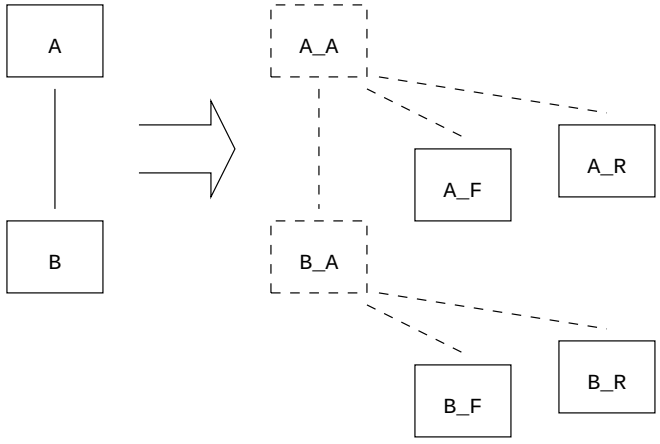

Figure 10. Transformation of a simple hierarchy to implement façades.

Figure 10 represents a class hierarchy consisting of two classes `A` and `B`, and the transformed hierarchy. Class `A` is used to define an abstract class `A_A` and two implementations `A_R` and `A_F`. `A_F` is the façade class for the real implementation `A_R`. Class `B` has a similar translation. The implementation `B_R` of class `B` inherits its functionality and fields from the implementation `A_R` of class `A`.

Façade Implementation and Operation

The basic operation of façade objects was described in section . Implicit in that description was the façade's capacity to maintain and use sufficient information to be able to fault and reify the real object and replace all references to the façade with references to the real object. By including an `oid` field (which

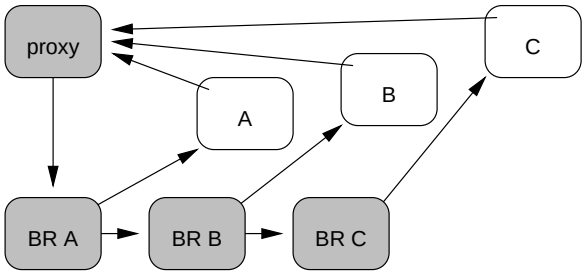

Figure 11. Three 'real' objects referencing a façade, which references each of them via a linked-list of back-references.

contains the OID of the real object) in the façade object, the process of faulting and reification of the real object becomes straightforward. By further including a linked list of back-references to the objects that refer to the façade, the task of replacing all references to the façade with references to the real object is also straightforward. The elements of the linked list will either back-refer to object references (represented by an *object, field* tuple) or array references (*object, index*). An eager approach over static variables avoids the necessity of static references. All classes implement methods that allow outgoing references to be updated by back-referring façades. Figure 11 depicts three 'real' objects (labelled A, B, and C) which refer to the façade object (proxy). The façade has a linked list of back-references (BR A, BR B, and BR C) to each of the referring objects.

After being faulted from the store, an object is reified and each OID is swizzled to an object reference. A 'swizzle map' mapping OIDs to objects is used to determine the target object reference. If, at the time of swizzling the target object is a façade, a back reference to the object being reified is added to the façade's linked list of back references. If no mapping exists for an OID at swizzle time, a new façade object is created in place of the relevant object and the OID and reference to the façade are added to the swizzle map. When a façade takes a read fault and faults and reifies the real object, it also updates the swizzle map to ensure that subsequent swizzles of the OID will be to the real object rather than the façade.

Reference Leakage

As described in the preceding sections, the operation of façades is dependent on each façade being able to transparently replace itself with the 'real' object at the point at which the façade is first accessed. The process of transparent replacement is dependent on the façade maintaining a list of back-references to referring objects so that it can update the references contained in those objects. This approach has the effect of maintaining two key invariants:[††] at no time will references to both façade and real versions of the same object co-exist, and the cost of the read barrier will only be incurred once for each referenced object—all subsequent references will be direct, avoiding the barrier. While the primary consequence of the second invariant is an efficiency gain, the first invariant ensures correctness with respect to reference comparisons. Both invariants depend on *reference leakage* being avoided—references to façades must only be generated in the reification process, thereby ensuring that the façade is aware of all referring objects.

[††]For the moment we will assume that the reference updating process is atomic.

Pointer leakage can occur through the passing of references between methods (‘inter-method’), and through the stack within a given method invocation through (‘intra-method’).

Inter-method leakage Inter-method reference leakage, illustrated in figure 12, can be avoided by byte-code modifying methods to ensure that only references to ‘real’ objects are passed (both as parameters and return values). This can be done by calling the `get$this` method over the reference, which if ‘real’ will

```
getfield MyClass/a LMyClassA;              ;
getfield MyClass/b LMyClassB;              ;
invokevirtual MyClassA/method(LMyClassB)V  ; a.method(b);
                                           ; ...
getfield MyClass/c LMyClassC;              ;
areturn                                    ; return c;
```

Figure 12. Potential for inter-method reference leakage through the passing of a reference as a parameter.

reduce to a no-op, and if ‘façade’ will cause the real object to be faulted in. Figure 13 illustrates a byte-code modified version of the code in figure 12 that avoids inter-method reference leakage. This approach is eager,

```
getfield MyClass/a LMyClassA;               ;
getfield MyClass/b LMyClassB;               ;
invokevirtual MyClassB/get$this()LMyClassB; ;
invokevirtual MyClassA/method(LMyClassB;)V  ; a.method(b.get$this());
                                            ; ...
getfield MyClass/c LMyClassC;               ;
invokevirtual MyClassC/get$this()LMyClassC; ;
areturn                                     ; return c.get$this();
```

Figure 13. Passing a reference as a parameter while avoiding inter-method reference leakage.

as it may lead to the unnecessary faulting of an object in the case where the method being called (or the calling method) does not actually access the object in question.

Intra-method leakage The problem of intra-method reference leakage, illustrated in figure 14, occurs when references are duplicated on the stack. In both cases illustrated in the example, the reference may at first be a façade, the first invocation of the method over the referenced object causing the reference to

```
getfield MyClass/a LMyClassA;     ;
dup                               ;
invokevirtual MyClassA/methodX()V ; a.methodX();
invokevirtual MyClassA/methodY()V ; a.methodY();
                                  ; ...
getfield MyClass/b LMyClassB;     ;
dup                               ;
invokevirtual MyClassB/method()V  ; b.method();
if_acmpeq thenBranch              ; if (b == b)
```

Figure 14. Multiple use of references in a method leading to potential intra-method reference leakage.

be replaced with a real reference. At that point the stack may hold an inconsistent reference to the object, causing subsequent use of that reference to fail (first case), and/or giving an incorrect comparison result (second case). The problem of intra-method reference leakage is solved by byte-code modifying methods to remove any use of **`dup`** that might lead to such leakage.

```
getfield MyClass/a LMyClassA;     ;
invokevirtual MyClassA/methodX()V ; a.methodX()
getfield MyClass/a LMyClassA;     ;
invokevirtual MyClassA/methodY()V ; a.methodY()
                                  ; ...
getfield MyClass/b LMyClassB;     ;
invokevirtual MyClassB/method()V  ;   b.method();
getfield MyClass/b LMyClassB;     ;
if_acmpeq thenBranch              ; if (b == b)
```

Figure 15. Multiple use of references in a method avoiding intra-method reference leakage.

ANU-OPJ Characterization

The previous sections describe in some detail the implementation of both ANU-OPJ-shell and ANU-OPJ-façade. This enables the ANU-OPJ implementations to be characterized according to the taxonomy of approaches to orthogonally persistent Java detailed in Moss and Hosking [21]. Below, each facet of the taxonomy is described along with the choices made in our implementations.

Model Choice 0 *Is persistence by reachability, or by some other means of designating which objects are to persist? For orthogonal persistence it is not sensible to consider alternatives.* The choice of which objects are made persistence is based on the reachability of those objects from the persistent roots. In ANU-OPJ a persistent root is a non-transient class variable (i.e. a static field of a class).

Model Choice 1 *Is (object) code persistent? Is the code maintained in the persistent store?* The classes are made persistent. That is, both the declaration of the fields and the method code are made persistent. A class not held in the persistent store is loaded into memory and is made persistent, so that the class will be retrieved from the persistent store on subsequent invocation of ANU-OPJ.

Model Choice 2 *Is the program execution state persistent? Can an executing program persistent in the store and if so when does it activate?* The program execution state is currently not made persistent in the ANU-OPJ implementation. However, this is not a permanent limitation. Subsequent versions of ANU-OPJ will incorporate the mechanisms outlined by Hohlfeld and Yee [12] which enable the transformation of the method code to allow the execution states of threads to be saved and restored (although some details in relation to system classes must still be addresses).

Related issues such as when do the saved threads become active again, the transaction context and the security settings that the threads are activated in are yet to be dealt with.

Model Choice 3 *What is the transaction model? When and how does the data become persistent? When does the new state become visible to others?* A Chain-and-Spawn transaction model [5] is incorporated into the system to allow transaction based updates to the persistent store. The transaction boundaries are invoked through a transaction class library and all code execution takes place with a transactional context. A chain point is a Commit-Begin point which can maintain read/write semantics from one transaction to the next. These chain points make visible through the store the modification of the data held.

Transparency Choice 1 *Is the source language changed?* The Java source code language remains unchanged.

Transparency Choice 2 *Is the object language changed? If the standard byte-codes are used (with their normal semantics) then immediate portability benefits are obtained (even if the source code is modified).* The object language remains unchanged. However, the object code is transformed at class loading time (into standard Java object code and the original object code is maintained) and all introspection appears to see the original code form. However, some issues still remain with the interaction of the outside world and the JVM(s) that interact with the store. The transformation applied use the standard byte-code semantics and allow our implementation to be portable.

Implementation Choice 1 *Is the Java compiler changed? If the source code is changed then, either a preprocessor of modified compiler must be provided.* The Java compiler remains unchanged.

Implementation Choice 2 *Is the Java interpreter and runtime system modified? Does the JVM remain unchanged? Do the system libraries remain unchanged? What other runtime changes are made?* The JVM and libraries remain unmodified. Persistence is achieved with the use of a special class loader (written in Java) to transform classes on the fly so that persistence can be implemented.

ANU-OPJ Evaluation

The first implementation of OPJ (ANU-OPJ-shell) used the shell approach described in section . The experimental results (next section) show performance problems for the cold[‡‡] execution. This was initially attributed to eager memory consumption for the allocation of shell objects. In order to address this issue, a second implementation using the façade (ANU-OPJ-façade) approach described in section was developed. However, the results of the second implementation were not as good as the first. We believe that a partial explanation is the increased switching between Java and C in the façade implementation. Additionally, some of the transformations used in the façade approach, such as the Abstract Replacement Transformation (section), may reduce the opportunities for code inlining and optimization by the just in time compiler (JIT). Moreover, the memory gain obtained by the use of a façade is less relevant when most of the objects are small (which is the case in the 007 benchmark).

In both these implementations native methods were used for the time consuming tasks associated with read barriers, such as object faulting and reification. However, instrumentation of the system demonstrated that the barriers performed substantially better when the kernel of the barrier was implemented in Java rather than as native methods. This is attributed to the substantial overhead of native method invocation and the capacity for the JIT to optimize frequently executed code. The results of the instrumentation also confirm that the additional overhead introduced by read and write barriers is not relevant in the cold runs.

Benchmarking Environment

The OO7 benchmark [8] is used for evaluating our approaches in a wide range of data manipulations. The results presented compare the performance of ANU-OPJ (ANU-OPJ-shell and ANU-OPJ-facade), PJama [2] (versions 0.5.7.10 and 1.2), PSI-SSM (a SHORE-based implementation using C++), and Java running without persistence. The PJama implementations are unable to take advantage of JIT technology, whereas ANU-OPJ can leverage this technology to produce competitive performance and portability. The ‘small’ OO7 benchmarks are used as we were not able to run the ‘medium’ database over any of the PJama versions or PSI-SSM.

[‡‡]Cold execution times are for the initial run where the data has not been faulted into memory and must be retrieved from the underlying store.

It is possible that the hot times could degrade as the database size is increased. In this case, techniques such as object cache eviction and promotion will become a necessity. None of our implementations support these techniques, but this support could be easily built on top of the normal Java garbage collection (using finalizer methods and weak references), with the support of an object cache that can flush objects and still maintain their locks.

The same Java implementation was used for the ANU-OPJ, PJama and JDK systems with only minor modifications required for each. For the base code to run on either ANU-OPJ-shell or ANU-OPJ-façade it was only necessary to insert a call to `Chain()` at the point where the benchmark required a commit. The non-persistent version (JDK 1.2.2) needed minor changes to allow the 'generation' and 'benchmark' phases of OO7 to occur in a single execution. For PJama it was necessary to add code which opened the store, retrieved the persistent roots, and called `stabilizeAll()` at the point where the benchmark required a commit.

For PSI-SSM, the OO7 benchmark was implemented in C++. The implementation does not use swizzling, but instead explicitly translates persistent references to pointers at each traversal, and makes explicit update notifications. 'Smart pointers' [18] were used to perform the reference translations, affording a degree of syntactic transparency.

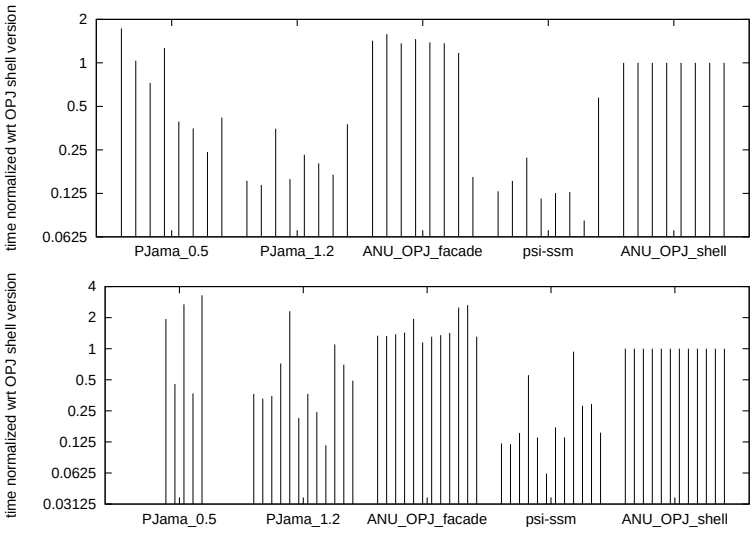

Figure 16. Cold query (top) and update (bottom) times relative to corresponding ANU-OPJ-shell time.

The benchmarks were executed on a single Sun Ultra-170 with 128MB of RAM and separate hard disks for the persistent store and log. Both version 0.5.7.10 (which required JDK 1.1.7) and version 1.2 of PJama were used. JDK 1.2.2 (with the Hot Spot JIT) was used to execute the ANU-OPJ-shell, ANU-OPJ-façade and non-persistent (JDK) versions of the OO7 benchmarks. ANU-OPJ-shell, ANU-OPJ-façade and PSI-SSM used the Shore storage manager.

Performance Results

The complete suite of OO7 benchmarks could not be used. Benchmarks t2a, t2b and t3a did not run in either ANU-OPJ implementation nor PJama. Benchmarks i and d did not run due to a (unidentified) failure in the Shore storage manager. Each impulse reported corresponds to a particular benchmark. The

benchmarks reported are queries q1 to q8 and updates t1, t2c, t3b, t4, t5do, t5undo, t6 to t10 and wu, in this order.

Cold time The cold execution results in figure 16 indicate that ANU-OPJ implementations performs worse than PJama1.2 when cold (2.2 times slower than PJama version 1.2 on average). We attribute these results to the excessive context switching between Java and C using JNI.

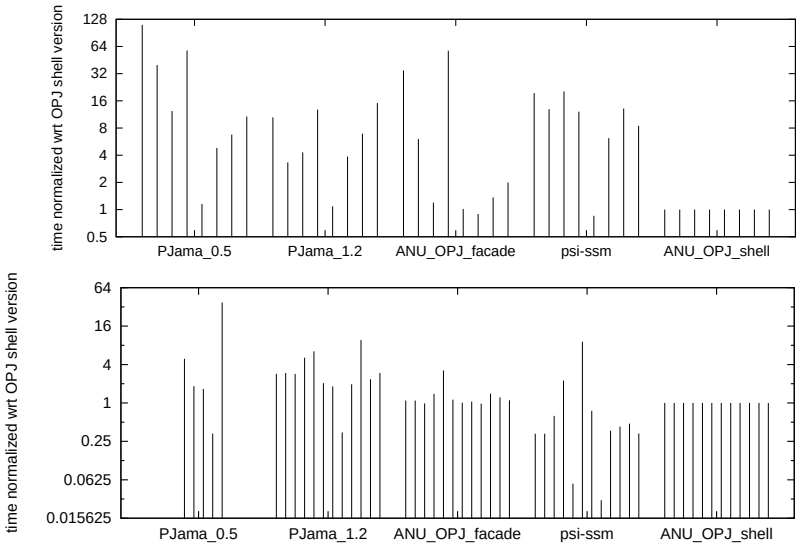

Figure 17. Hot query (top) and update (bottom) times relative to corresponding ANU-OPJ-shell times.

Hot time The hot execution results in figure 17 indicate that ANU-OPJ-shell implementation performs well when hot (5 times better than PJama version 1.2 and 3 times better than the ANU-OPJ-façade implementation on average), outperforming both PJama implementations in almost all the operations. The ANU-OPJ-shell performs better than any other implementation in read only operations (almost 8 times better than PJama 1.2 on average), even outperforming a C++ based implementation over the same store. We attribute the strength of these results to the JDK1.2.2 JIT compiler and to the cost of explicit reference translation in the C++ implementation.

Both ANU-OPJ implementations and the JDK 1.2 benchmark show a variance over 15% in a third of the benchmarks, reaching a maximum of 234% of variance with respect to the average value in bench q3 of JDK-1.2. Each benchmark was repeted ten consecutive times within the same program execution, commiting at the end of each repetition. We attribute this erratic behavior to be caused by the interference of the garbage collector during the benchmarking process. The standard deviation of queries speedup in the shell version with respect to PJama1.2 is 66% of the average speedup.

Read and Write barrier overhead The figure 18 compares the ANU-OPJ-shell and a non-persistent implementation of the OO7 benchmark operations. This graphic demonstrates a reasonable overhead for all query operations (150 percent on average). However, update operations were much less favorable (370

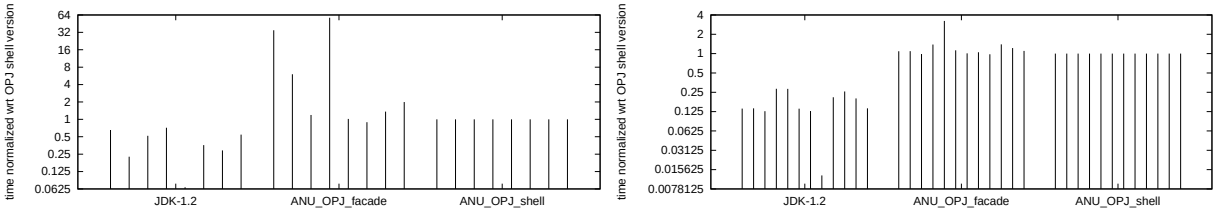

Figure 18. Hot query (left) and update (right) times relative to corresponding ANU-OPJ-shell times.

percent on average), as a consequence of the transaction commit costs. We believe that these results are significant, because they suggest that the runtime efficiency of the Java language environment is unlikely to be a stumbling block, for the use of Java in the context of persistent data management.

Other Applications for Semantic Extensions in Java

The previous section describes in detail the application of the semantic extension framework to provide a portable orthogonally persistent Java. The semantic extension framework (which forms the core of this work) can be applied to produce other types of semantic extensions, some of these extensions are outlined in this section.

A read barrier can be used for any number of purposes. In a persistence context it is usually used to assert the presence of a referenced object in memory, faulting the object in from secondary storage if necessary. It may also be used to effect some form of concurrency control (e.g. by acquiring a read lock over the referenced object). In a similar vein, a write barrier can be used both to flag data as having been updated (and therefore requiring propagation to secondary storage) and as a concurrency control mechanism, by acquiring a write lock over a the referenced object.

Read and write barriers can be applied to any context where it is necessary to trap reads and writes to objects. Another example is the extension of Java semantics to support object versions. In this setting, a read barrier could be used to assert the presence of a particular version of an object, while the write barrier could be used to ensure that the update is propagated to the appropriate ‘release’ or ‘configuration’. Other examples include the implementation of a simple heap profiler, where the barriers could be used to log object read and write events.

We are exploring the use of dynamic byte-code modification for the creation of other orthogonal extensions to the language. We have built an orthogonal object versioning (OOV) framework and mechanisms for schema evolution and versioning.

Orthogonal Object Versioning (OOV) Framework

The object versioning framework is based on Orthogonal Object Versioning and is built on top of our ANU-OPJ prototype. The general idea of Orthogonal Versioning is extends the principles of Persistence Independence and Data Type Orthogonality to the problem of deal with multiple versions of the same object.

Code Profiling and Instrumentation

Another important application for the class of transformations described here is in trapping method invocations. Such a technique could be used to transparently instrument byte-codes for call profiling. This

could be applied generally (trapping all method invocations) or selectively (to profile certain classes or methods).

Conclusions

Java allows user-defined class loaders to be defined which perform byte-code to byte-code transformations over class files at the point when they are loaded into the Java runtime system. Such transformations can be used to implement a range of interesting semantic extensions to Java classes in a semi-dynamic manner.

This portable approach to semantic extensions in Java has several key advantages. Any Java to byte-code compiler can be used, and it does not rely on Java source being preprocessed. This contrasts with techniques that extend the Java language syntax, such as those described in [1, 24] et al. Because the transformations are applied at the byte-code level, the framework is not tightly coupled to a particular JVM. This gives the user the freedom to use the most suitable JVM and leverage rapid advances in Java technology immediately. Our portable approach therefore has a significant advantage over approaches to semantic extension which modify the JVM itself and so are tied-down to a particular technology.

Such a framework has a natural application in extending the semantics of Java to orthogonal persistence. This paper reports on two implementations of orthogonally persistent Java using the semantic extension framework. The first of these is characterized by the use of object ‘shells’ and eager swizzling and lazy reification policies with small read barrier overheads. The second uses the concept of ‘façades’ to remove read barriers and has lazy swizzling and slightly eager faulting and reification policies. Both approaches are novel insofar as they represent the first portable implementations of orthogonally persistent Java, and each explores quite different implementation strategies.

The framework and tools outlined provide mechanisms for semi-dynamic semantic extensions which are powerful, platform independent, and through portability can take immediate advantage of improvements in JVM and JIT technologies. Significantly, a platform independent, efficient, orthogonally persistent Java has been presented and implemented using these techniques.

REFERENCES

1. Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *OOPSLA’97, Proceedings on the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 32 of *SIGPLAN Notices*, pages 49–65, Atlanta, GA, U.S.A., October 5–9 1997. ACM.
2. M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. Design issues for Persistent Java: A type-safe, object-oriented, orthogonally persistent system. In Richard Connor and Scott Nettles, editors, *Seventh International Workshop on Persistent Object Systems*, pages 33–47, Cape May, NJ, U.S.A., May 1996. Morgan Kaufmann.
3. Malcolm P. Atkinson and Ronald Morrison. Orthogonally persistent systems. *The VLDB Journal*, 4(3):319–402, July 1995.
4. Stephen M. Blackburn and Robin B. Stanton. The transactional object cache: A foundation for high performance persistent system construction. In Ronald Morrison, Mick Jordan, and Malcolm Atkinson, editors, *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, August 30–September 1, 1998, Tiburon, CA, U.S.A.*, pages 37–50, San Francisco, 1999. Morgan Kaufmann.
5. Stephen M Blackburn and John N Zigman. Concurrency—The fly in the ointment? In Ronald Morrison, Mick Jordan, and Malcolm Atkinson, editors, *Advances in Persistent Object Systems: Third International Workshop on Persistence and Java, September 1–3, 1998, Tiburon, CA, U.S.A.*, San Francisco, 1999. Morgan Kaufmann.
6. Boris Bokowski and Markus Dahm. Poor man’s genericity for Java. In *Proceedings of JIT’98*, Frankfurt am Main Germany, November 12–13 1998. Springer Verlag.
7. John Boyland and Giuseppe Catsagna. Parasitic methods: An implementation of multi-methods for Java. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA ’97), Atlanta, Georgia, October 5-9, 199*, volume 32 of *SIGPLAN Notices*, pages 66–76. ACM Press, October 1997.
8. Michael J Carey, David J. De Witt, and Jeffrey F. Naughton. The OO7 benchmark. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26–28, 1993*, volume 22 of *SIGMOD Record*, pages 12–21. ACM Press, June 1993.

9. Geoff A. Cohen and Jeffrey S. Chase. Automatic program transformation with joie. In Fred Douglis, editor, *USENIX Annual Technical Conference (NO 98), June 17–19, 1998, New Orleans, Louisiana, U.S.A.*, New Orleans, 1998.
10. GemStone Systems. GemStone/J. http://www.gemstone.com/, 1999.
11. James Gosling, Bill Joy, and Guy L. Steel, Jr. *The Java Language Specification*. Addision Wesley, August 1996.
12. Matthew Hohlfeld and Bennet Yee. How to migrate agents. http://www.cs.ucsd.edu/users/bsy/pub/migrate.ps, August 1998.
13. Antony Hosking, Nathaniel Nystrom, Quintin Cutts, and Kumar Brahnmath. Optimizing the read and write barriers for orthogonal persistence. In Ronald Morrison, Mick Jordan, and Malcolm Atkinson, editors, *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, Tiburon, CA, U.S.A., August 30–September 1, 1998*, pages 37–50, San Francisco, 1998. Morgan Kaufmann.
14. Craig Thrall John V. E. Ridgway and Jack C. Wileden. Toward assessing approaches to persistence for java. In Mick Jordan and Malcolm Atkinson, editors, *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, August 13–15, 1998, San Francisco, CA, U.S.A.*, San Francisco, 1998. Morgan Kaufmann.
15. Ralph Keller and Urs Holzle. *Binary Component Adaptation*. 1998.
16. Alfons Kemper and Donald Kossmann. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *VLDB Journal*, 4(3):519–566, 1995.
17. Gökhan Kutlu and J. Eliot. B. Moss. Exploiting reflection to add persistence and query optimization to a statically typed object-oriented language. In Ronald Morrison and Malcolm Atkinson, editors, *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems*, pages 123–135, Tiburon, CA, U.S.A., August 30–September 1 1998. Morgan Kaufmann.
18. Stanley B Lippman. *C++ Primer*. Addision Wesley, September 1991.
19. Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, March 1997.
20. J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, SE-18(8):657–673, August 1992.
21. J Eliot B Moss and Antony L Hosking. Approaches to adding persistence to Java. In Mick Jordan and Malcolm Atkinson, editors, *First International Workshop on Persistence and Java*, Drymen, Scotland, September 16–18 1996. Available online at: `http://www.dcs.gla.ac.uk/~carol/Workshops/PJ1Programme.html`.
22. POET. POET Programmer's Guide, SDK Java Edition. Product documentation, POET Software Corporation, 1998.
23. Sun Microsystems. JavaBeans. API specification, Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043, 24 July 1997.
24. Kresten Krab Thorup. Genericity in Java with virtual types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECCOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9–13, 1997*, number 1241 in Lecture Notes in Computer Science (LNCS), pages 444–471. Springer-Verlag, 1997.
25. Stephen J Tjasink and Sonja Berman. Providing persistence on small machines. In Ronald Morrison and Malcolm Atkinson, editors, *Eighth International Workshop on Persistent Object Systems*, Tiburon, CA, U.S.A., August 30–September 1 1998.