Proceedings of the Second International Workshop on Persistence and Java

Mick Jordan and Malcolm Atkinson

SMLI TR-97-63 December 1997

Abstract:

These proceedings record the Second International Workshop on Persistence and Java, that was held in Half Moon Bay in the San Francisco Bay Area, in August 1997. The focus of the workshop series is the relationship between the Java platform and long-term storage, such as databases and orthogonal persistence. If future application programmers building large and long-lived systems are to be well supported, it is essential that the lessons of existing research into language and persistence combinations are utilized, and that the research community develops further results needed for the Java platform.

The initial idea for the workshop series came from Malcolm Atkinson who leads the Persistence and Distribution Research Group at Glasgow University, and who is a Visiting Professor at SunLabs. The workshop series is one of the fruits of the collaboration between the Forest group at SunLabs, led by Mick Jordan, and the Glasgow group.

Sun microsystems

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
mick.jordan@eng.sun.com
malcolm.atkinson@eng.sun.com

Contents

Preface

This report contains the proceedings of the Second International Workshop on Persistence and Java. The workshop was held on August 13-15th, 1997 at the Half Moon Bay Lodge in the San Francisco Bay Area and attended by approximately 45 people, comprising paper authors and invited participants. The workshop also hosted a parallel meeting of the Object Database Management Group to facilitate exchange of ideas and informal communication.

A year is a long time in the evolution of the Java platform and this was borne out by the submitted papers and the makeup of the participants. Along with the major academic research groups, practically all of the object database vendors were represented.

In contrast to the first workshop, which was mostly focused on debating the appropriate form of persistence mechanisms for Java, with little in the way of actual implementations, many of the papers in this workshop were concerned with discussing actual implementations. Several papers contained experimental measurements of the systems, although these tended to be for small standard benchmarks, notably OO1. We look forward in the future to more comprehensive and comparative measurements, that will enable the costs and benefits of the different approaches to persistence to be analyzed.

The workshop schedule contained a demonstration segment. Several groups, both academic and commercial, gave demonstrations of their systems in action.

Although we must admit to some bias, the workshop offered some evidence that the notion of orthogonal persistence is becoming accepted as the appropriate choice for object persistence in Java. Indeed, one invited participant mistakenly concluded that the workshop was specifically for the orthogonal persistence community! Evidently we need to increase participation from the relational and object-relational communities.

As last year, we polled the participants on whether we should hold a third workshop in 1998, particularly since the Eighth Persistent Object Systems workshop (POS8) will occur, and many of the participant overlap. The conclusion was that we should jointly hold the two workshops at the same location, with a shortened PJW3 immediately following POS8.

We would like to thank all the participants and the program committee for their contributions to a successful workshop. We owe a special vote of thanks for Linda Browning of SunLabs who handled all the administrative details and local arrangements. Finally, we would like to thank Bert Sutherland, SunLabs director, for his continued support of the workshop series.

Mick Jordan and Malcolm Atkinson

Java Universal Binding: Storing Java Objects in Relational and Object-Oriented Databases

Florian Xhumari[†][⋆] **Cassio Souza dos Santos**[⋆] **Marcin Skubiszewski**[†][⋆]

August 29, 1997

[†]INRIA, Rocquencourt
78153 Le Chesnay, France
FirstName.LastName@inria.fr

[⋆]O_2 Technology
7, rue du Parc de Clagny
78000 Versailles, France
http://www.o2tech.fr

Abstract

We introduce JUB (*Java Universal Binding*), a software tool that stores Java objects in relational and object-oriented databases. JUB supports the object-oriented DBMS O_2, the relational DBMS Oracle and Sybase, and all the relational databases which can be accessed via JDBC.

In the context of O_2, Java objects stored in the database are first-class database objects: they can be accessed by all the clients of O_2 (O_2 supports application programs written in C, C++, Smalltalk, and O_2C).

We describe JUB from the application programmer's point of view. We discuss the architecture of JUB, and the way in which Java classes and objects are translated into O_2 types and objects. We describe the current status and the performance of the product.

1 Introduction

Many different approaches have been proposed to allow Java programmers and applications to take profit from database technology. These approaches can be divided into two main groups: those proposing an extension to the Java language or virtual machine, and those taking the language as it is and defining a persistence service layer on top of the Java language and of an existing database system.

The first group aims at defining a persistent version of Java that requires a nonstandard compiler and/or virtual machine. For example, the systems described in [2, 1, 6] belong to this group.

The second group can be further divided, according to the level of integration between the database system and Java, into database drivers (JDBC) and language bindings (ODMG).

JDBC drivers [7] provide access to existing databases through an API that reflects the underlying database model. Applications must map the native structures of the database (in the relational model, rows and columns) into corresponding Java objects and attributes. Many JDBC drivers are currently available or under development. A ODBC/JDBC bridge is also available.

Language bindings provide transparency to persistence in that the mapping between Java objects and the underlying database structures is performed automatically by the runtime system. Applications manipulate persistent objects as if they were ordinary (transient) Java objects. Persistence is requested either explicitly or indirectly, through the attachment to a persistent root.

The ODMG Java binding is defined in version 2.0 of the standard. Most object database vendors have announced Java bindings to their systems.

JRB (*Java Relational Binding*) [9] is a Java binding to relational databases. It was initially defined on top of Oracle and Sybase databases, and was later extended to run on top of a JDBC driver. JRB provides an API that allows applications to manage database entities (bases, transactions, queries) and to store and retrieve Java objects into/from the underlying database.

The work described in this paper is built on top of JRB. Here, we do not describe the way in which Java objects are stored in a relational database (this issue has been treated in [9], where a complete description of JRB is given). Instead, we describe a general architecture featuring a common runtime running on top of different DBMS. This architecture is called the Java Universal Binding (JUB). A common API running on top of JUB allows applications to access relational and O_2 databases undistinguishably and in a transparent way.

O_2 Java is a specific component of the JUB architecture providing access to O_2 DBMS [3]. It is analogous to JRB, but this time the target database system is the O_2 DBMS.

JUB uses the same algorithm as JRB to store Java objects in a relational database. No other aspects of JUB are inherited from JRB. Most notably, the structure of the product is novel: a well-defined interface has been introduced between the higher-level part of the product, which is independent on the DBMS being used, and the lower-level part, which is DBMS-specific.

The paper is organized as follows. Section 2 describes JUB as seen from the user's point of view. Section 3 describes the structure of the product. Section 4 describes the way in which we translate Java classes and objects into the corresponding O_2 types and objects. Section 5 describes the current status of the project. Performance is discussed in Section 6.

2 Using JUB

This section introduces the key properties of JUB, as seen from the application programmer's point of view.

JUB is a Java database binding, *i.e.* a tool that makes it possible to store Java objects in a database. With JUB, the application programmer is aware of the fact that there is a database involved and that some objects are persistent and are stored in the database, whereas others are not. He/she invokes database synchronization primitives like `transaction()`, `commit()` or `abort()`, and knows that these invocations affect persistent objects. The user may explicitly take database locks (although this is usually unnecessary, because JUB automatically takes the appropriate locks whenever an object is read or written).

Our approach is different from the one used in Persistent Java [2], where every attempt is made to give to the system the appearance of an ordinary (non-persistent) Java runtime system.

2.1 Persistence-capable types

When JUB is used, the application programmer has access to two kinds of Java objects: ordinary or *transient* objects, which behave as if JUB did not exist, and *persistent* objects, which are managed by JUB, and are stored in a database. In order to be persistent, an object must belong to a *persistence-capable* type.

A class or an interface is persistence-capable iff it satisfies two conditions. First, it must implement or extend the interface `PersistentObject` from the package associated with JUB (namely, package `jub.api`). This interface contains all the methods necessary to manage persistence-related properties of objects. Second, the class must have been *imported* by JUB. The import operation creates in the underlying database the data structures necessary for storing objects of a given class: in a relational database, appropriate relations are created [9]; in O_2, a type is created that corresponds to the Java class (see Section 4).

An array type is persistence-capable iff an array of this type can be referenced to by an attribute of a persistence-capable class and the type of the elements of the array is persistence-capable. The import operation detects these conditions and creates the necessary structures in the database.

2.2 Making objects persistent

Two models have been proposed for choosing which objects should be made persistent: explicit persistence and persistence by attachment.

With explicit persistence, the application programmer explicitly requests objects to be added to or deleted from the database, *i.e.* to be made persistent or to be made transient again.

With persistence by attachment, a fixed set of objects is made explicitly persistent. These objects are stored in the database and are directly accessible to the application programmer. They are called *roots of persistence*. For all other objects, the following rule is applied recursively: if a persistent object *a* contains a pointer to an object *b*, then *b* is also persistent. This rule can be equivalently expressed in a non-recursive form: an object is persistent if it is *reachable*, *i.e.* if and application program can reach it by starting from a root of persistence, and by following pointers from one object to another.

With persistence by attachment, objects that are persistent, but are no longer reachable, are deleted from the database by a garbage collector.

JUB implements both kinds of persistence. In both cases, the static members of persistence-capable classes can be accessed directly (*i.e.* without the need to first obtain a pointer to the object) by application programs. Static members are made persistent at import time, except if the user requests otherwise.

The user can request JUB to maintain *class extents*—collections containing all the objects of a given class. When class extents exist, they are directly accessible to application programs.

With persistence by attachment, the static members and the class extents are roots of persistence.

2.3 Using a persistent object

The usage of persistent objects involves three specific issues. First, the user program must notify JUB of reads and writes performed on persistent objects. Second, under certain circumstances, persistent Java objects are *detached* from the database: changes made to detached objects will not be written into the database. And finally, the application program may label certain fields as *transient*. Such fields will not be stored in the database, although they exist in the in-memory version of the object.

2.3.1 Access notifications

Before accessing a persistent object, a Java program must inform JUB of its intention. This is done by invoking the method `access` for the object in question. This invocation causes the object to be actually loaded in the address space of the program. Until then, it may be the case that the real object is only stored in the database, and what appears to be the object in the address space of the Java program is in reality a *shadow object*: a placeholder object that has the appropriate type, but whose contents are meaningless.

The method `access()` is defined for all persistence-capable types.[1] `access` has no effect when invoked for a non-shadow object, either persistent or transient. Invoking `access` many times in a row for the same object is equivalent to invoking it once. Therefore, there is no harm in calling `access` too many times, or in calling it for a transient object. This simplifies the use of `access`: application code can invoke the method before every access to an object that belongs to a persistence-capable class, without knowing

[1]For array types, it is impossible to define methods. Therefore, if `t` is an array, `t.access()` is not defined; instead, we use the static method `access` in class `jub.Database`, like this: `Database.access(t)`. The same remark holds for the method `markModify`, mentioned below.

whether the call is redundant or whether the object is actually persistent. Application code can therefore by instrumented so as to call `access` in a systematic way, before every access to a persistence-capable object. The instrumentation relieves the programmer from the necessity to add calls to `access` by hand. It is done automatically, by a postprocessor of Java bytecode.

2.3.2 Write notifications

When a persistent object is modified by a Java program, this fact must be notified to JUB, so that when the current transaction commits, JUB will propagate the modification to the database. The notification takes the form of a call to the method `markModify`.

`markModify` is only intended to be used with persistent objects. `markModify` is similar to `access` in that it can harmlessly be invoked outside of its intended scope of use (namely, for transient objects) or be invoked many times, instead of just once. It is therefore possible to instrument application code so that every modification of a persistence-capable object is followed by a call to `markModify`. The instrumentation is done automatically, by the same postprocessor which adds calls to `access`.

2.3.3 Invalidating persistent objects

JUB can *invalidate* a persistent object, *i.e.* put the object in a state in which it can no longer be used. All objects are invalidated when the current transaction commits. This is necessary, because at commit time all locks are released, and therefore the state of the persistent objects, as represented in the address space of the Java program, is no longer guaranteed to correctly represent the real contents of the objects.

Additionally, the application program can request the invalidation of individual objects. This is useful (and sometimes necessary) in order to release the resources used by objects that have been accessed at some point, but are unlikely to be accessed again.

The application program must not use invalidated objects. It is recommended that the program destroys all the references to such objects, so that the Java garbage collector can destroy them.

2.3.4 Transient fields

Fields in persistence-capable classes can be labeled as `transient`. Such fields are never stored in the database. Therefore, when a persistent object is brought from the database to the address space of a Java program, transient fields need to be properly initialized. This can be accomplished by the method `activate()`: if this method is defined for the object, it will be invoked every time the object is brought to the address space. The rôle of this method is limited to initialization of transient fields of the object. In particular, it is not allowed to modify any persistent attribute or otherwise manipulate a persistent object. The system enforces this rule.

2.4 Storing Java bytecode in the database

It is possible to store Java bytecode in the database. The bytecode of the classes stored in this way can be loaded into the execution environment by means of the class `DatabaseClassLoader`. This class extends the class `ClassLoader` and provides the mechanism to load the bytecode, pass it to the Java runtime making it possible for the program to manipulate objects belonging to the loaded class. The user may explicitly use the class `DatabaseClassLoader` to load a particular class. But the most interesting use of stored bytecode is when the database contains objects of a class which is not available to the program from the `CLASSPATH`. If the program accesses such an object, the JUB loads the implementation of the class from the database, allowing the object to be properly used.

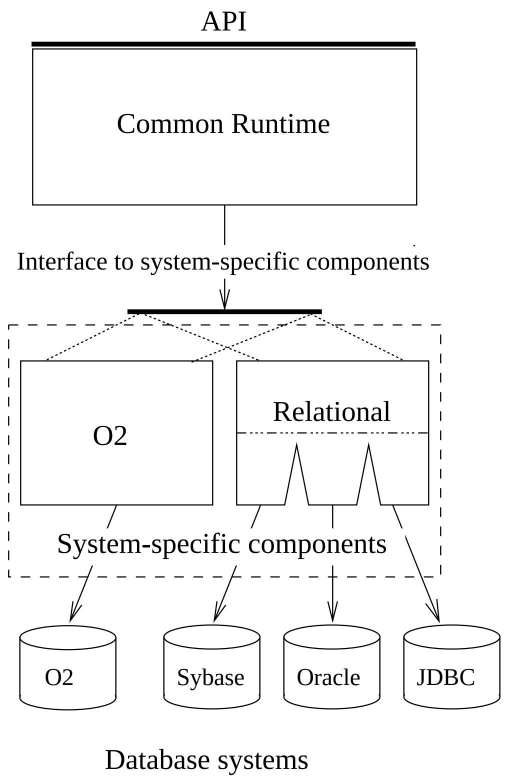

API
Common Runtime
Interface to system-specific components
O2
Relational
System-specific components
O2
Sybase
Oracle
JDBC
Database systems

Figure 1: The runtime system of JUB.

Let's give an example of the utility of this feature. A class `C` which implements the interface `Runnable` is stored in the database and an object belonging to the class is created. The `Runnable` interface defines a method `run()` and the implementation of this method in `C` performs some action. An external program needs to know only the `Runnable` interface; it accesses the stored object and calls its `run()` method and the proper action is performed.

3 The architecture of the runtime system

The runtime system of JUB is divided into a *common runtime* and *system-specific components* (Figure 1). The common runtime is accessed by application programs through an API (*application programmer's interface*). There are two system-specific components: one for relational databases, and one for O_2. The component specific to relational databases contains three sub-components, which access, respectively, Oracle, Sybase, and JDBC. Access to Oracle and to Sybase is done via their respective native SQL interfaces (incidentally, one can also access Oracle and Sybase through a JDBC interface, but this is less efficient).

Both system-specific components have the same interface, through which they are used by the common runtime. They allow the common runtime to connect to a database, to manage transactions (perform operations like `commit()` or `abort()`), and to create, read and modify persistent objects in the database. All these operations are performed without the common runtime knowing whether the database is object-oriented or relational: the common runtime always views the database as a repository of objects.

The system is designed so that more system-specific components, accessible through the same interface, can be added, and can be used by the common runtime.

The common runtime is based on two data structures: the *object table* and the *metadata classes*. Let us describe these structures briefly.

3.1 The object table

The object table is a transient data structure, internal to the common runtime. It contains information about the persistent objects that exist in the local address space. For every such object, it memorizes the object's address in the address space, its type, its state, and its *cache identifier*.

The cache identifier is used to identify the object when communicating with a system-specific component; from the common runtime's point of view, it plays the rôle of the object's address in the database. The cache identifiers of objects are computed by the system-specific component, in a way that allows for easy mapping between the cache identifier and the information necessary to find the object in the database.

The state of the object is either *shadow*, *accessed*, *modified*, or *new*. *Shadow* is the state of shadow objects, *i.e.* of objects that are present in the database and are referenced from within the local address space, but have not been properly copied into this space (see Section 2.3.1). *Accessed* is the state of objects that are present both in the database and in the local address space. *Modified* is the state of objects that are present both in the database and in the local address space, and for which `markModify()` has been called. *Modified* objects are written to the database at commit time. *New* is the state of the objects which have been made persistent by the local application program, during the current transaction, and have not yet been added to the database. *New* objects are added to the database at commit time.

3.2 Metadata classes

The metadata classes contain information about persistence-capable classes. There is one metadata class per persistence-capable class. Information stored about each persistence-capable class includes its name, the name of the parent class (if any), the names and types of the attributes of this class and those of the static attributes of the class.

Metadata classes are generated by the import tool. If the bytecode of a persistence-capable class is stored in the database, then the bytecode of the metadata class is stored there, too.

To use a metadata class, the common runtime creates an object belonging to this class (the object is transient), then accesses the object through the `ClassMetaData` interface. The methods of this interface return information about the persistence-capable class corresponding to the object in question. All metadata classes implement the interface `ClassMetaData`.

The common runtime contains a *metadata manager*—a module capable of finding (or, if necessary, for creating) metadata objects corresponding to any given class name. When the manager needs to create a metadata object whose class is not present in the execution environment, the class bytecode is loaded from the database, as described in Section 2.4.

4 The mapping of Java classes into O_2 types

The mapping from Java classes into O_2 data structures benefits from the similitudes between between the Java type system and the O_2 data model. Incompatibilities also exist, and we discuss the ways they are handled.

Table 1 shows in detail the mapping between Java types and O_2 types.

Classes, interfaces and objects Java classes and interfaces are translated by the import tool into O_2 classes (an O_2 class corresponding with a Java interface contains no attributes). When a class or an interface is imported, all the interfaces and classes from which it inherits (*i.e.* which it extends or implements) are also imported. The inheritance structure is entirely preserved by the translation process. This is possible due to the multiple inheritance mechanism of O_2, whose semantics englobes that of Java.

When a subclass defines an attribute with the same name as an attribute in a superclass, O_2 considers that the attribute is overloaded, whereas Java considers it completely distinct from the superclass' attribute. To overcome this problem, in case of a conflict we create the attribute with a unique name and then rename it to the original name. This causes O_2 to consider the two attributes as different attributes. The runtime system keeps track of renamings so as to load the corresponding Java attributes accordingly.

Scalar types `short`s and `int`s are translated into `integer`s. `float`s and `double`s are translated into `real`s. If there is an overflow in the process of this conversions, an exception is thrown.

Java 64-bit long values are mapped into pairs of 32-bit integer O_2 integer attributes. The runtime system performs the appropriate data conversion when long values are loaded from the database into long variables.

String, Integer, Float,... Objects belonging to `String` and to classes of simple types (like `Integer`, `Float`, etc) do not retain their identity in the database. In this respect, our system changes the semantics of Java language, but we considered that having these objects retain their identity would be too expensive in terms of performance. Furthermore, the values encapsulated in these objects are immutable, which limits the non-transparency to the only case of reference comparison.

Unicode characters in `String`s are converted and stored in the database as ASCII. It is possible to specify in the configuration file that they are to be stored as Unicode. Unicode strings are stored always as UTF strings; the conversion is performed by Java JNI (Java Native Interface) runtime.

Arrays A Java arrays is mapped into an O_2 class encapsulating a list. When a new array is created in the database, the whole corresponding list is filled with nulls. When a list is read in as an array, the size of the list determines the size of the array to create.

Arrays in Java follow the same hierarchy structure as the types of their elements. This structure is closely followed by the corresponding O_2 classes which implement arrays. Such classes are of type list, and the model of O_2 allows inheritance between lists to follow the inheritance between their elements. From this point of view, the O_2 model matches exactly the Java model.

Arrays of Java `long` values are implemented using lists of 32-bit integers, considering two consecutive elements as a single 64-bit integer.

Static variables Static variables are mapped into O_2 names, thus becoming persistent roots of the database.

5 Current Status of the Implementation

An early version of JUB exists today and has been tested. This version has some restrictions. Only explicit persistence is implemented. Persistence by attachment is not available today, although the major difficulties connected to it have been solved: a garbage collector for O_2 exists [8], and persistence by attachment is available in all language bindings to O_2 except Java.

The application programmer must instrument Java code with calls to `access` and to `markModify` by hand. The postprocessor of Java bytecode, which will perform this task automatically, is under development.

Java type	**O_2 type**
Builtin types and corresponding classes	
boolean, Boolean	boolean
char, Character (16 bit, UNICODE)	char (8bit ASCII), or integer (16 bit, UNICODE)
byte, Byte (8 bit)	char (8 bit)
short, Short (16 bit)	integer (32 bit)
int, Integer (32 bit)	integer (32 bit)
long, Long (64 bit)	two attributes of type integer, whose names are suffixed by hi and lo
float, Float	real
double, Double	real
String (UNICODE)	string (UTF coding, as defined in Java JNI)
Classes, interfaces, arrays	
interface I {... }	class I (no data)
interface J extends I {... }	class J inherit I (no data)
class C {... }	class C public type tuple(...)
class C extends S implements I {... }	class C inherit S, I public type tuple(...)
JavaType[]	class o2_list_*O2Type* public type list(*O2Type*)
JavaType[], where class *JavaType* extends S implements I ...	class o2_list_*O2Type* inherit o2_list_S, o2_list_I public type list(*O2Type*)
JavaType[][]	class o2_list_o2_list_*O2Type* public type list(o2_list_*O2Type*)
Special cases	
long[], Long[]	like int[], with two successive elements coding the low and high part of a long
byte[], Byte[]	class ByteArray type public bits
char[], Character[]	class ByteArray type public bits
java.lang.Vector	o2_list_Object, of type list(Object)
Static variables	
in *class*, static *JavaType att*	name o2_var_*class* : tuple (*att* : *O2Type* ...)

Table 1: The mapping of Java types to O_2 types.

Collection classes in the style of those defined in the ODMG Java Binding [5] (sets, bags and lists) are being currently implemented and should be soon available in JUB. At short term, JUB API should evolve towards an ODMG compliant interface.

Today, it is possible to launch OQL or SQL queries in the underlying database. We are in the process of investigating the possibility of the execution of Java methods from an OQL query.

6 Performance measurements

We present preliminary performance measurements of our prototype using the O_2 system. We investigate the impact of the differences between JUB and the C++ binding to O_2, as well as the scalability of JUB.

6.1 The benchmark

Our measurements are based on the OO1 benchmark [4]. We used OO1's database schema and two operations: traversal and creation. We did not include the lookup operation and the reverse traversal operation of OO1 benchmark because we felt that performing these operations would not add significant information to our measurements.

We implemented a simple benchmark because our goal is to measure the impact of the language binding and not the performance of the database system itself.

We use a database composed of objects called *parts*. Each part contains an identifier, a date, a part type and two coordinates: x and y. Each part points to three other parts. It also contains a list of parts from which it is referenced—the reverse links. For each reference to a part there is a type and a length. The type fields are strings of ten characters, the date is a 64-bit integer. The identifier, the coordinates and the length fields are 32-bit integers. A global hash table contains all parts, hashed by the part identifier.

Two database sizes were tested: one *small* database containing 20000 objects and one *large* database with 200000 objects. The fields of each object data occupy about 100 bytes and if we include the space overhead of other structures, this gives databases of about 4 Mb and 40 Mb respectively.

The database is constructed in two phases. First, the parts are created with links set to null. The identifiers of the parts are set in increasing order, from 1 to the total number of parts, while the other fields are set to random values. In the second phase, we link each part to three other parts; the reverse links are updated as appropriate. The parts to link are chosen so as to obtain some locality of reference: 90% of them are randomly selected among the 1% of the parts that are "closest", and the remaining 10% are randomly selected from all parts. The parts are said to be close if their identifiers are numerically close. The global hash table is used to find a part, given its identifier.

Our first experiment is a traversal in the graph of parts. Starting from a random part, we visit all parts connected to it in a depth-first order, up to a depth of 7 levels. In total this makes 3280 parts, with possible duplicates. For each part visited we read the values of all attributes. This experiment is executed in read-only transactional mode.

The second experiment is similar to the first one, with the difference that for each part visited, the field x is incremented. The time measured includes the time to commit the changes to the database.

The third experiment inserts new parts into the database. A new part is created and is linked to other parts following the same rules used to construct the database. We create 100 parts with increasing part identifiers and report the time needed to create the parts and to commit the transaction.

We measure execution times of our tests in two different runs: a cold run, when there is no data cached in the database server or in the client, and a warm run, where cached data are present. Practically, we run the tests 15 times and consider the first run as a cold run and the last 7 times as warm runs.

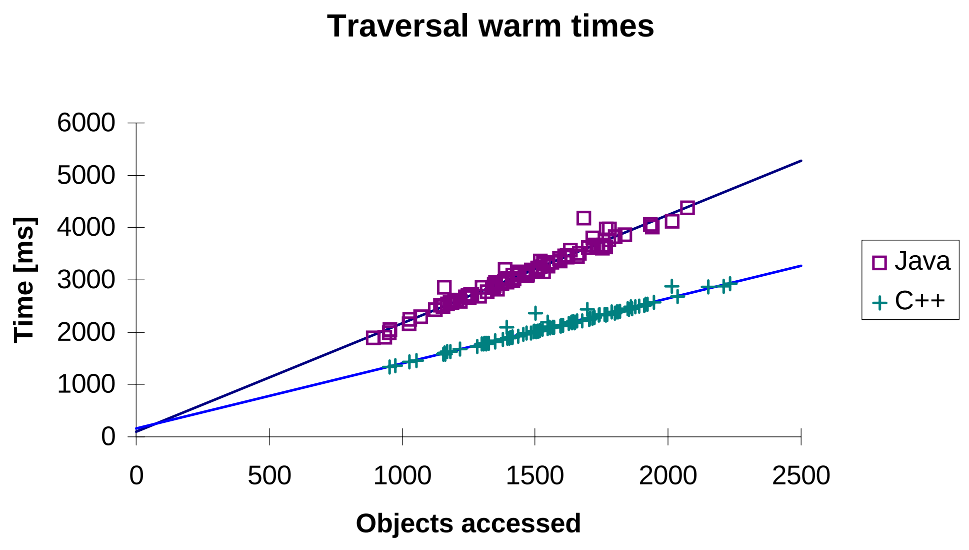


Figure 2: Traversal time depending on the number of distinct objects visited

6.2 System configuration

For our benchmark, we used a Sun UltraSparc 1 workstation with 128 Mb of random access memory, 192 Mb of swap space and two 4 Gb disks connected to two different fast-wide SCSI controllers. One disk contained only the database files and the other contained all the system files and database software. During the measures, the system was used exclusively by our benchmark.

Both the server and the client processes ran in the same machine. The server was configured with 4 Mb of cache and the client had 4Mb of cache too. We used O_2 version 5.0.2.A.4 and the Java Virtual Machine included in Sun's JDK 1.1F.

6.3 The results

Figures 2 and 3 show the results of the traversal and modification experiments in the small database, the warm run. *Objects accessed* is a count of distinct objects visited during the traversal.

We observe significant differences in the number of objects accessed in each run, as well as a difference in the time measured. The time is almost proportional to the number of distinct objects accessed. This suggests that the first access to an object is time-consuming while further accesses are much faster. This is an expected result. In the case of Java, the first visit to an object causes the object fields to be loaded from the database and shadow objects to be created for the three pointed-to objects. A subsequent visit only causes the system to query the object table about the state of the object. In C++ things are much the same: during the first visit the object itself is created and its fields are filled in; subsequent visits just test a bit in C++'s reference variable (`d_Ref`).

Given this behavior, we decided to present in our results the execution time divided by the number of distinct objects accessed. For the insert experiment, we divided the time by the number of objects inserted (100).

Table 2 shows the results we obtained in our experiments on the small database and table 3 shows the results for the large database.

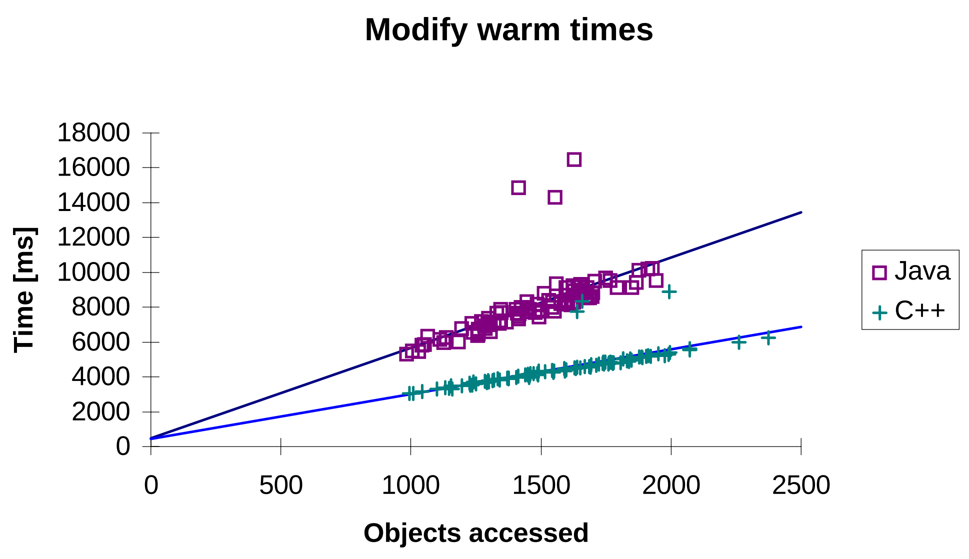


Figure 3: Modify time depending on the number of distinct objects visited

	Time		Percent
	Java	C++	slower
cold traverse	4.95	2.22	123%
warm traverse	2.14	1.35	59%
cold modify	7.13	3.78	88%
warm modify	5.51	2.86	93%
cold insert	23.39	18.05	30%
warm insert	16.81	12.49	35%

Table 2: Small database results

	Time		Percent
	Java	C++	slower
cold traverse	6.57	3.03	117%
warm traverse	3.30	1.41	133%
cold modify	12.09	4.56	165%
warm modify	15.64	3.83	308%
cold insert	46.67	40.79	14%
warm insert	50.50	40.41	25%

Table 3: Large database results

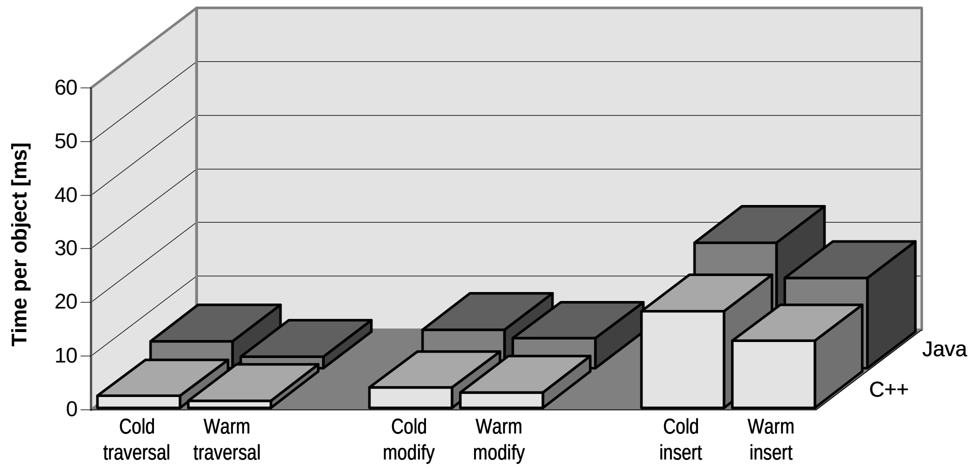


Figure 4: Small database results

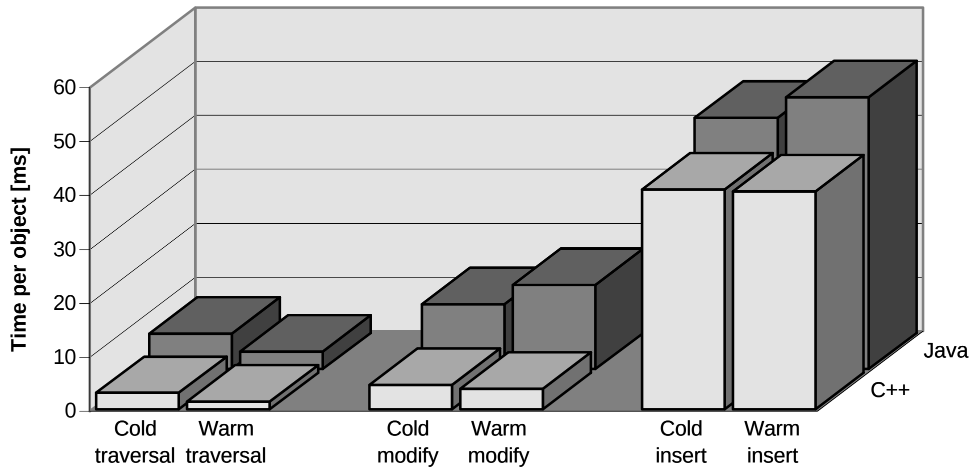


Figure 5: Large database results

The numbers presented in these tables are shown in a graphic form in figures 4 and 5, respectively for the small and for the large database.

The first thing to notice in these results is that Java is slower than C++. This is normal, given that we used a interpreting Java Virtual Machine. We expect that using a Just In Time Java environment will significantly reduce the differences.

An interpreted Java program is generally 5 to 10 times slower than the same program written in C++. Our measurements show execution times slower by 30% to 150% in most cases. This means that a substantial part of the time is spent in the database server and in the client libraries, and the work done in Java is relatively short.

The difference between cold and warm times is due to caching which improves the performances in the warm case. It is interesting to note the behavior of Java in the modify and insert experiments, in the large database case: the warm times are greater than the cold times. We do not have a valid interpretation for this case and we continue to investigate.

When comparing times of the small versus the large database, we notice a general increase of execution times. In the large database there is less locality than in the small database, which causes more pages to be read from disk. The modify time in Java has increased more than the modify time for C++ when going from the small database to the large database. This is a point which merits further investigation.

We are in the process of making performance measurements for comparing the JUB using O_2 versus JUB using a relational database.

7 Conclusion

We presented the main features of the Java Universal Binding, currently under development at O_2 Technology. We briefly discussed some implementation issues and presented performance results.

References

[1] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM Sigmod Record*, December 1996.

[2] M.P. Atkinson, L. Daynès, M.J. Jordan, and S. Spence. Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system. In *Proceedings of the Seventh Workshop on Persistent Object Systems*, May 1996.

[3] F. Bancilhon, C. Delobel, and P. Kannelakis. *Building an Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.

[4] R. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17(1), March 1992.

[5] R.G.G. Cattell, editor. *Object Database Standard : ODMG - 93, Release 1.2*. Morgan Kaufmann, San Francisco, 1996.

[6] A. Garthwaite and S. Nettles. Transactions for Java. *First International Workshop on Persistence and Java - PJ1*, September 1996.

[7] Graham Hamilton and Rick Cattell. *JDBC: A Java SQL API*, June 1996.

[8] Marcin Skubiszewski and Patrick Valduriez. Concurrent garbage collection in O_2. In *International Conference on Very Large Data Bases (to appear)*, 1997.

[9] C. Souza dos Santos and E. Theroude. Persistent Java. *First International Workshop on Persistence and Java - PJ1*, September 1996.

Toward Assessing Approaches to Persistence for Java™

John V. E. Ridgway Craig Thrall Jack C. Wileden

Convergent Computing Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003 USA

{ridgway,cthrall,wileden}@cs.umass.edu
Phone: (413) 545-0289
Fax: (413) 545-1249

Abstract

In a previous paper [9] we described our goals and plans for an approach to seamlessly integrating persistence, interoperability and naming capabilities with Java. Having now completed a prototype implementation of our JSPIN approach, we have begun the process of *assessing* it, and some other alternative approaches, from a variety of perspectives. In particular, we have begun to measure *performance* by adapting a standard benchmark for use with our prototype and some representative alternatives. We have also started to make some qualitative assessments of our approach and some of its competitors based on several *usability* factors, particularly those that were among the goals enunciated in our previous paper. In this paper we outline our JSPIN approach and its implementation, describe the performance benchmark and present initial data resulting from its application, discuss our preliminary observations concerning usability factors, and sketch our plans for further development, assessment and use of JSPIN.

1 Introduction

At the First International Workshop on Persistence and Java we described an approach to seamlessly integrating persistence, interoperability and naming capabilities with Java. Our paper in that workshop [9] identified three main objectives for our work, namely:

- To produce a seamlessly extended version of Java having valuable capabilities beyond those provided in the basic Java language but with minimal barriers to adoption;
- To complement similar extensions to C++ and CLOS, thereby providing a convenient basis for extending our work on polylingual interoperability [13]; and

™ Java is a Trademark of Sun Microsystems, Inc.

- To demonstrate the usefulness and generality of our previously-developed approaches to providing persistence, interoperability and name management.

Having now completed and begun to experiment with a prototype implementation of the approach described in that paper, we are in a position to begin assessing our success at meeting these objectives, as well as some more specific goals derived from them.

To that end, we are presently engaged in assessing not only the current prototype realization of our approach, which we call JSPIN, but also some other alternative approaches, from a variety of perspectives. As a starting point for quantitative measurement of *performance* we have adapted a standard benchmark for object-oriented database systems – the OO1 benchmark [5] – for use with our JSPIN prototype and some representative alternatives. In this paper we describe the adapted benchmark and report initial results obtained from applying it to JSPIN and the selected alternatives. We have also begun to make some more qualitative assessments of JSPIN and a few of its competitors based on several *usability* factors, particularly those implied by the goals and objectives we adopted at the outset of this project. We discuss the preliminary results of these assessments here as well.

It is our hope that the work reported here will contribute to research on persistence for Java in at least three ways:

- By describing, and providing some initial assessment of, one particular approach to extending Java with persistence (and other) capabilities;
- By helping to establish a basis for systematic assessment and comparison of alternative approaches to persistence for Java through presentation of a performance benchmark and some candidate criteria for use in quantitative and qualitative assessment, respectively; and
- By taking a first, albeit quite preliminary, step toward establishing a collection of data useful for assessing and comparing various aspects of various approaches.

Our overall aim is to facilitate ongoing development of approaches to persistence for Java, both our own and others.

The remainder of this paper is organized as follows. In Section 2 we briefly outline the goals and foundations underlying our JSPIN approach. Section 3 describes the JSPIN approach itself, in terms of the APIs provided to JSPIN users and the current implementation of JSPIN. In Section 4 we discuss the OO1 benchmark and how we have adapted it for use in measuring performance of some approaches to providing persistence for Java. Section 5 contains the performance data produced by applying the adapted OO1 benchmark to JSPIN and a few alternative approaches. In Section 6 we discuss some criteria for qualitative assessment of approaches to persistence for Java, and our observations about JSPIN and other approaches based on these criteria. Section 7 summarizes our results and contributions and sketches some future directions for this work.

2 Background

There are many compelling reasons for providing orthogonal persistence capabilities for Java [2], and several efforts are currently aimed at doing so (e.g., [2, 15, 17, 6]). Our own approach to producing a seamless integration of persistence, interoperability and naming with Java, which we now call JSPIN, was outlined in [9]. The foundations for JSPIN are the SPIN framework, the interface to the kernel of the TI/DARPA Open Object-Oriented Database (Open OODB) [22] and Java itself.

The SPIN (Support for Persistence, Interoperability and Naming) framework [8] was developed as a unifying conceptual foundation for integrating extended features in software systems. SPIN has previously been used as a basis for seamlessly integrating persistence, interoperability and naming capabilities in extended versions of the C++ and CLOS APIs of the Open OODB [10, 13]. The SPIN framework itself evolved out of our earlier work on persistence [20, 23, 19], interoperability [24] and name management [11, 12], all of which aimed at minimizing the impact of the extended capability on software developers or pre-existing code. When extended with automated support for polylingual persistence [13], we refer to the framework as PolySPIN.

Our JSPIN approach is motivated by the objectives enumerated in Section 1, which in turn imply several more specific goals. Among those goals are:

Seamless Extension of Java: Our highest priority has been to provide a set of extensions to Java in the most seamless manner possible. Seamlessness implies that our extensions should be compatible with Java and the programming style that it defines, including its type safety and security properties. The specific extensions included in the JSPIN approach are:

- **Persistence:** JSPIN currently provides orthogonal, reachability-based (transitive) persistence for Java. The particular style of the JSPIN persistence capability is similar to that provided for C++ and CLOS by the Open OODB.
- **Enhanced name management:** JSPIN will provide a set of extended name management capabilities, based on the Piccolo model [12] and therefore suitable for use with Conch-style tools [10]. These capabilities will be independent of (that is, orthogonal to) persistence. As a result, this enhanced approach to name management will be uniformly applicable to C++, CLOS, and Java objects.
- **Basis for polylingual interoperability among C++, CLOS, and Java:** The extensions provided by JSPIN transparently incorporate the necessary information into Java objects to support polylingual interoperability among C++, CLOS, and Java [13].

Minimal Barriers to Adoption: Our next highest priority is to make it as easy as possible for Java users to adopt our extensions. In keeping with the philosophy underlying SPIN, we seek to minimize the impact of the extensions on programmers, especially those who might not be (direct) users of the extended capabilities. Hence, this goal is closely related to seamlessness. The specific ways in which we have attempted to minimize barriers to adoption are:

- **No language extensions:** JSPIN does not introduce any modifications in the syntax (including keywords) of Java, nor does it require the use of an additional or separate specification language.
- **No virtual machine modifications:** By making it possible to run JSPIN programs on the standard Java Virtual Machine, we hope to encourage the use of JSPIN from web browsers and in other settings where adoption of a modified virtual machine may be unlikely.

No core class modifications: JSPIN does not make any changes to the core classes, allowing existing code to run unchanged.

No native method calls: The goal is to have no native method calls as part of the JSPIN kernel. As discussed later this turns out to be impossible in the current prototype, but will be possible once we move to JDK 1.1.

Maximal Opportunities for Interoperability: A unique feature of the JSPIN approach is that it will directly facilitate interoperation among C++, CLOS, and Java programs. This is because JSPIN:

- Shares the SPIN (and eventually PolySPIN) conceptual base with the C++ and CLOS APIs of Open OODB 1.0.
- Shares use of the Open OODB kernel with the other Open OODB APIs.

Suitable Basis for Future Research: We intend to use JSPIN as a foundation for various experiments and extensions. To that end, we have endeavored to make the system well modularized, with an open architecture and clean, well-defined interfaces.

In succeeding sections we describe our current prototype realization of JSPIN and present some preliminary assessment of our approach. The qualitative facets of that assessment are closely related to the above-mentioned goals.

3 The JSPIN Approach

In this section we discuss the approach taken in implementing JSPIN. We describe the API, the platform, the implementation strategy, the JSPIN packages, the required compiler changes, and the known limitations.

3.1 API

There is currently one API for JSPIN. It provides basic, Open OODB-style persistence to Java users. The API includes methods added to each class processed by JSPIN, together with several JSPIN-specific classes (in a package named `EDU.umass.cs.ccsl.JSPIN`).

The appearance is that most methods are added to the `Object` class and inherited from it by every other class. In reality this is not exactly the case because of return-type restrictions. Specifically, the `fetch` method of a class is required to return an object of that class and thus must be specific to the class.[1]

The basic API adds the following methods to each class:

`public void persist([String name])` When invoked on any object, this method results in that object, and all objects reachable from it, becoming persistent. The optional `name` parameter can be used to assign a name to the persistent object, by which name it can later be retrieved. If no name is assigned, the object can only be retrieved if it is referenced from some other object.

[1]We could inherit the `fetch` method but then it would have to return `Object` and the programmer would be required to cast the returned `Object` to an object of the desired class. This remains type-safe, but is slightly unpleasant.

`public static` *`class`* `fetch(String name)` When invoked, this method returns the persistent instance of the class corresponding to the name given by the `name` parameter. If there is no such instance in the persistent store, the `UnknownPersistentName` exception is thrown.

These methods are convenience methods. There are related methods in the `PersistentStore` class which provide similar functionality which these methods invoke.

We have chosen not to include an `unpersist` method as being inconsistent with the spirit of Java, which has no explicit operator to free heap objects. We do intend to provide an operator to unbind a name from a persistent object. It is possible that an object unbound in such a way would no longer be referenced and would need to be garbage-collected. We recognize that garbage collection in the persistent store is necessary but feel that it is a part of the implementation of the persistent store and not a part of the implementation of a persistent programming language.

The basic API also defines the `PersistentStore` abstract class (in the `JSPIN` package). All of the methods of this class, abstract or not, may potentially throw a `PersistentStoreException`, which we have chosen to omit from these brief descriptions:

```
public abstract class PersistentStore {
  public void beginTransaction();
  public void commitTransaction();
  public void abortTransaction();

  public void persist(Object obj, String name);
  public void persist(Object obj);
  public Object fetch(String name);
 }
```

3.2 Platform

We chose to implement JSPIN on top of the Java Developers Kit (JDK) version 1.0.2 since it was the latest version to which we had source-code access. We provide interfaces for use of TI/DARPA Open OODB version 1.0 [21] and for Mneme version 5.0.10 [18, 14]. Importantly, JSPIN runs on an unmodified Java Virtual Machine (1.0.2). These decisions have informed some of the implementation choices stated below.

3.3 Implementation Strategy

In this section we discuss the implementation of JSPIN. The interfaces described in this section are not meant for the use of a user of JSPIN. They are for the use of programmers extending JSPIN to work on new persistent stores, and for better understanding of the system.

A major goal of our implementation was to allow the use of our persistence approach in applets running on unmodified hosts. Thus we sought to avoid changes in the Java Virtual Machine, changes to the Java core API, and native methods.[2] Native methods must, of course, be used in interfacing with local persistent stores, but we intend to produce an implementation that communicates with a server using pure Java code. Such an implementation would not require any native method calls in the client.

Our implementation strategy for JSPIN involves exploiting the data abstraction and object-orientation features of Java to seamlessly add the SPIN extensions. As described in the preceding subsection, the extensions are presented to the Java programmer in the form of additional methods for each class and some additional JSPIN-specific classes. We implemented the addition of the methods to each class by modifying the JDK 1.0.2 compiler. Implementation of the added methods themselves, those added to each class and those in the JSPIN-specific classes, is primarily done by calls to the underlying persistent storage manager.

To support orthogonal persistence by reachability requires a small set of basic functionality. The persistence system must be able to identify those objects which are to persist, and must be able to move them between memory and the persistent store. Objects in the store cannot be represented in the same way as objects in memory because addresses do not have the same meaning; consequently the persistence system must do the appropriate address translations.[3] We chose to use a uniform representation of objects in the persistent store. Each object is represented as an array of bytes and an array of *handles*. The byte array holds a serialized form of the primitive fields of the object, while each handle holds a store-unique form of reference to other objects. In addition each object holds a handle to the name of the class of which it is a direct instance.

Our current JSPIN prototype is implemented via changes to the Java compiler and creation of several runtime packages. These are detailed in the subsections that follow.

3.4 The `JSPIN` Package

The `JSPIN` package is the kernel of the JSPIN system. It provides the APIs that programmers using JSPIN see and it maintains all of the mapping and indexing data that are required.

The heart of JSPIN is the `PersistentStore` class. This is a partially abstract class that provides all of the required mapping mechanisms and which contains all of the intelligence on transitive persistence. For each persistent object store which is to host JSPIN a subclass of `PersistentStore` is created.

A central item in the JSPIN kernel is the `PersistenceProxy` interface which realizes the concept of a *persistence proxy*. Every object that is persistent (and possibly some that aren't) has a proxy. This proxy carries persistence data about the related object, and has methods to manipulate the object in the ways required by the kernel. The kernel maintains mappings between objects and their proxies.

The `PersistentObject` class (see below) implements the `PersistenceProxy` interface, so instances of classes that inherit from `PersistentObject` can act as their own proxies. Classes compiled by the JSPIN compiler will generally inherit from `PersistentObject`. Unfortunately, objects of core classes, such as the `Integer` wrapper class, cannot be changed and have not been compiled by our compiler. To deal with this issue we provide, or will provide, special proxy classes for all of the Java core

[2]The goal was to use no native methods. The reality is that we did have to use one native method, as is discussed later.

[3]Aka swizzling.

classes.[4] In addition we provide special proxy classes for arrays of primitive types and arrays of `Object`. Arrays are relatively hard for JSPIN to process because they are instances of classes that have no source.

Classes do not persist in our model, but we do perform some type-checking to ensure that type-safety is not violated. Whenever an object of a given class is made persistent we ensure that some data about the class is also made persistent. When the first object of a given class is fetched we ensure that the saved data about the class is valid. The validity checking is currently skeletal. When a class is first fetched we get its name and attempt to load the class. If necessary any superclasses or interfaces are fetched and checked. Fetching and loading is done on the basis of fully-qualified class names. The signature of the class, i.e. its fields and methods, is not currently checked. If any checks fail a `PersistentStoreException` is thrown. As noted earlier, `fetch` returns objects as instances of the correct class, so no casting of fetched objects is required.

3.5 Compiler Changes

Every class processed by the modified compiler has the following changes made to it:

- If it originally extended `Object` it is logically modified to extend `PersistentObject`. This allows us to insert all of the persistence proxy data into the object itself, and allows us to have a single `persist` method.
- It is decorated with a set of methods to extract and insert bytes and handles. These methods are used for swizzling and unswizzling objects. These methods provide functionality similar to that of the Serialization and Reflection interfaces. These two interfaces were not available when we were implementing JSPIN and neither of them fully provides the functionality that we needed.
- Every non-static method of the object has a residency check inserted as its first statement. This is to support fetch-on-demand. When an object is first fetched only an empty shell of the object is created. It is not until the first reference to the object that the shell is actually filled in, at which time empty shells are created for all of the objects it references.
- Get and Set methods are added for every non-static field in the class. These methods perform residency checks on the object prior to returning or setting the field. Every reference to a field is turned into a method call unless the compiler can verify that the reference is from a method of the same object. We note that this transformation of direct field access to method calls is similar to the definition of some other object-oriented languages, e.g. Dylan.[1]
- A new class is created to act as a creation proxy for the given class. This proxy implements the `CreationProxy` interface, and is necessary when a class is not `public`. In such a case it may not be possible for JSPIN to create an instance of the class. The creation proxy class is created in the same package as the base class. It is a public class that exports a public creator for the base class. JSPIN can then use this creation proxy to create an instance of the base class.

[4]At present we have created proxies for all of the `java.lang` classes, but have not provided proxies for the `java.util` classes.

- All synthetic methods and fields, with the exception of the `fetch` and `persist` methods, have names that include the '$' character. This is to ensure that they do not conflict with user-defined names.

3.6 Persistent Store Interfaces

The kernel has a set of interfaces to underlying persistent stores. Two such interfaces are currently implemented, the `OpenOODBInterface` and the `LocalMnemeInterface`. The kernel contains an abstract class, `PersistentStore`, which the interfaces extend. Another abstract class, `ObjectHandle`, represents a store-specific handle to an object.

Implementations of the `PersistentStore` class provide the means for establishing connection to a specific persistent store. They also provide rudimentary transaction capabilities, mirroring those provided by the Open OODB APIs. An implementation of this class must provide implementations for the following abstract methods:

```
void beginStoreTransaction();
void abortStoreTransaction();
void commitStoreTransaction();

void getStoreHandle(String name, ObjectHandle handle);
void bindStoreName(String name, ObjectHandle handle);

void createStoreObject(ObjectHandle classHandle, int numBytes,
                       int numHandles, ObjectHandle handle);
void writeStoreObject(ObjectHandle handle, byte[] bytes,
                      ObjectHandle[] handles);
void readStoreObject(ObjectHandle handle, byte[] bytes,
                     ObjectHandle[] handles);
ObjectHandle newObjectHandle();
ObjectHandle[] newObjectHandles(int num);
```

They must also provide appropriate constructors. We currently supply two implementations of the `PersistentStore` class: `OpenOODBInterface`, which allows us to use an existing Open OODB database as our persistent store, and `LocalMnemeInterface`, which allows us to create and use local, single-user, Mneme [18, 14] persistent stores. Users satisfied with one of these supplied interfaces need not concern themselves further with implementation of the above abstract methods. This design, however, greatly facilitates porting of JSPIN to any of a number of different underlying persistent stores.

3.6.1 Open OODB Interface

The Open OODB interface uses the C++ interface to the Open OODB. Object handles contain pointers to buffered versions of the underlying C++ object. This interface uses native methods to call out to C and C++ code.

Basing our implementation on (re)use of the Open OODB kernel and our existing (C++-implemented) interoperability code has several advantages. It clearly simplifies the implementation task by leveraging existing, reasonably robust, code. It also lays the groundwork for interoperability by having not just common specifications but largely common implementations of the persistence, naming and interoperability features of JSPIN and the Persistent C++ and Persistent CLOS APIs of the Open OODB.

3.6.2 Mneme Interface

The Mneme[18, 14] interface uses native methods to communicate with Mneme. The current implementation only supports local, i.e. single-user, Mneme. We intend to extend this support to multi-user Mneme.

3.7 Limitations

We are aware of two limitations of our implementation.

- The JSPIN kernel runs on an unmodified Java VM and requires no changes to the core classes. Unfortunately, it does require one native method. There is no way, in JDK 1.0.2, to allocate an array of objects of a class known at runtime but not at compile-time. The `Array` class introduced as part of the Reflection package in JDK 1.1 has this capability, and we have simulated it via a single native method call in our 1.0.2 environment. This limitation will be removed when we port to JDK 1.1.

- Objects that are made persistent, either explicitly or implicitly, are never garbage-collected. Once an object becomes persistent JSPIN needs to be able to find it in memory, and JSPIN maintains a map to do so. This map constitutes a reference, and thus the object is never unreferenced. This limitation is going to be with us until we find some form of weak reference.

4 Adapting the OO1 Benchmark

As a basis for assessing JSPIN and comparing it with some of the other alternative approaches to persistence in Java, we implemented the Object Operations Benchmark (OO1) developed at Sun Microsystems[5]. The OO1 benchmark is an attempt to measure the performance of a database system by repeatedly performing common operations, such as retrieving and inserting records and traversing links through the database. It is a logical decision to use the benchmark to compare persistence implementations, as the tests are a good measure of the performance of any storage system. In addition, comparing code from the benchmark implementations can also bring to light differences in the style and method used to make objects persistent.

Briefly, the benchmark (as we used it to compare different implementations of persistence) performs three tests on a persistent store that holds two different types of objects:

A part object contains an integer that acts as a unique identifier, a pair of integer fields that hold random data, a field that holds a random date, and a string that contains a randomly selected part type.

A connection consists of two integer fields that hold the identifiers of the parts the object connects, a string that holds a randomly selected connection type, and an integer length field that is also filled randomly. There are three connections going from each part to other parts in the store. Ninety percent of the connections in the store go from one part to another randomly selected part within the one percent of parts closest (by part identifier) to the originating part. The remaining ten percent of the connections go to randomly selected parts in the store.

The benchmark tests are run on two different sizes of database. The *small* database contains 20,000 parts and the *large* database contains 200,000 parts. These databases contain, respectively, about 2 megabytes and 20 megabytes of attribute data.

The benchmark tests are:

The lookup test chooses 1,000 parts randomly and fetches them from the persistent store. An empty procedure is called with several parameters filled with data from the part.

The traversal test follows all connections of a part, then the connections of those parts, and so on down the tree for seven hops. Again, an empty procedure is called with data from each visited part. This traversal will touch 3280 parts, possibly with repeats. There is also a reverse traversal test, which is looked upon as less important because the number of parts touched can be widely varying.

The insert test creates 100 parts and three connections from each part to some random part in the store, calling an empty procedure in the process.

Each test is run ten times successively in order to examine the results with different levels of caching. The results of the first run are reported as *cold* results and those of the last run are reported as *warm* results. Between tests the file system buffers are flushed.

The OO1 benchmark is no longer considered the "standard" benchmark for object-oriented benchmarking. We used it for two reasons: because we already had a C version available and because it is simpler than OO7[4]. We believe that the results will nevertheless be of use in early assessment of the performance of persistence mechanisms for Java.

5 Applying the OO1 Benchmark

We applied the OO1 benchmark to several systems: JSPIN, with each of its persistent store interfaces, the Orthogonal Persistent Java implementation, an SQL database server using the JDBC interface, and two versions of a C implementation that used the Mneme persistent object store. JSPIN has been described earlier in this document and OPJ is described elsewhere[2]. We also ran a strictly transient Java version to give a baseline for the Java versions.

Both versions of JSPIN were run with a maximum garbage-collected heap size of 32 megabytes. JSPIN ran on top of version 5.0.10 of Mneme, and version 1.0 of OpenOODB. We used version 0.2.6 of OPJ with default settings except that buffer size was set to 16 megabytes.

For the SQL database we used version 3.20.24a of MySQL, a freeware SQL server, and version 0.92 of the GWE MySQL JDBC driver, also freeware. The GWE driver does not support prepared statements (precompiled SQL statements), and this probably detracted from its performance. The Java interpreter was started up with a 32 megabyte maximum heap size.

The C/Mneme implementations were included as a baseline against which to measure the other implementations. These are written in C and make direct calls to the Mneme persistent object store. The two implementations differ only in that one of them used a single-user version of Mneme while the other used a multi-user version. We expected these implementation to be faster than any of the Java versions and were not disappointed. These versions limited themselves to 5 megabytes of heap space.

The transient Java version was run with a 16MB maximum heap and a 4MB initial heap.

All of the benchmarks were run on a SPARCstation 10 running Solaris 2.5. The machine had 160MB of main memory. All data resided on a single external hard drive, but executables and class files resided on network file systems. The benchmarks were run one at a time with only one user on the machine. We interleaved tests on different systems in order to clear out the system file cache between runs.

We had to use a larger memory size than is specified in the OO1 benchmark to make OO1 work on any of the Java systems.

We were unable to run the *large* database tests as loading times would have been excessive. We hope to run these tests at some future time. Instead we ran tests on a *tiny* database, one with only 2,000 parts, and the *small* database. Table 1 shows the results we obtained on a tiny database, while Table 2 shows the results obtained on a small database.

Table 1: Benchmark Results for Tiny Database

		Transient	JSPIN				C with Mneme	
Measure	Cache	Java	Mneme	O^{3}DB	OPJ	JDBC	Local	Remote
DB Size(kBytes)			488	2400	972	N/A	392	
Load		9.24	44.33	145.63	18.98	100.06	2.28	
Reverse Traverse	cold	.007	.755	.475	.009	67.615	.163	.344
Lookup	cold	.240	8.979	31.119	.285	26.650	.077	.120
	warm	.238	1.146	1.608	.285	26.443	.064	.062
Traversal	cold	.068	3.326	16.257	.088	49.347	.165	.333
	warm	.065	.812	2.877	.084	52.184	.080	.055
Insert	cold	.454	3.894	13.221	.967	5.255	.284	1.565
	warm	.455	4.217	N/A	.900	4.980	.316	2.441
Total	cold	.762	16.199	34.155	1.340	81.252	.526	2.018
	warm	.758	6.175	N/A	1.269	93.607	.460	2.558

All results are in seconds except for the Database size, which is measured in kilobytes.

It should be noted that the reverse traversal benchmark runs are incomparable, as the random nature of the choice of starting part means that each of the above benchmark runs accessed a different number of objects.

In analyzing the results of these benchmark runs we note that the Mneme/C local version is by far the fastest. This was in line with our expectations and reasonable as compared with the systems measured

Table 2: Benchmark Results for Small Database

		Transient	JSPIN				C with Mneme	
Measure	Cache	Java	Mneme	O^3DB	OPJ	JDBC	Local	Remote
DB Size(kBytes)			4296	28240	7160	4755	3208	
Load		135.70	667.82	3981.13	245.71	1086.28	21.541	
Reverse Traverse	cold	2.033	.641	110.681	.134	22.590	.800	2.917
Lookup	cold	.309	20.043	62.397	2.237	26.743	1.398	1.428
	warm	.310	10.428	166.506	.630	26.712	.144	.090
Traversal	cold	.092	25.837	92.140	4.360	51.939	1.073	3.070
	warm	.088	5.974	N/A	.607	49.926	.065	.064
Insert	cold	.498	11.159	26.704	1.440	7.939	.668	7.939
	warm	.515	8.548	N/A	1.073	5.282	.516	6.141
Total	cold	.899	57.039	181.241	8.037	86.621	3.139	12.437
	warm	.913	24.950	N/A	2.310	81.920	.725	6.295

All results are in seconds except for the Database size, which is measured in kilobytes.

in the original OO1 work[5]. The remote results indicate that the multi-user Mneme system needs some improvement, but this is outside of our purview.

OPJ was about three times slower than the Mneme/C combination, and we tentatively attribute that to the difference in the sizes of their stores. (As the purely transient Java results show the interpretation time was not a major factor in the overall time.) The only problem with OPJ is that it did not scale well. We could not load the entire small database as a single transaction, but had to divide it up into one transaction to create the parts and eight separate transactions to create the connections between parts.

The JSPIN results with OpenOODB strongly suggest that there is a bug in our OpenOODB interface that needs to be fixed. The size of the resulting store and the number of crashes in running the benchmarks indicate trouble. We will be looking into this in the near future.

The JSPIN results with Mneme fared somewhat worse than we had expected. We knew that it was unlikely that JSPIN would be as fast as OPJ but it turned out to be a factor of seven times slower. We intend to find out why.

The JDBC results were interesting. The time taken to create the database scaled linearly in the size of the database. The time taken for the other benchmarks was essentially independent of the size of the database. This is not the case for other implementations, and it is our guess that the time taken to parse the SQL requests and generate the queries far outweighed the time to actually execute them. JDBC would not seem to be a good choice for an object-oriented application such as those that OO1 is meant to model.

Figure 1 shows the code for the traversal benchmark, and figures 2, 3, and 5 show the implementation classes for the different systems.[5] The OpenOODB and Mneme implementations are both extensions of the JSPIN implementation, which is shown in Figure 4. We hope that this will give some insight into the ease of

[5]Complete code for the benchmark is available at `ftp://ccsl.cs.umass.edu/pub/java-oo1`.

use of these systems. The differences lie in the methods of connecting to the persistent store and performing operations.

```
import java.util.Enumeration;

public class TraversalTest {
  private static Implementation impl;

  public static void main(String[] argv) throws Throwable {
    Options opt = new Options(argv);
    RandomWrap rand = new RandomWrap();
    Implementation impl = opt.implementation;
    impl.initiate(opt);

    Database d = impl.fetchDatabase();

    for (int run = 0; run < 10; run++) {
      opt.timer.start();
      int basePartId = rand.nextInt(d.minPartId, d.maxPartId);
      Part basePart = impl.fetchPart(basePartId);
      traversePart(basePart, 7);
      opt.timer.stop();
      System.out.println("Run "+run+" took "+opt.timer.get()+" msecs");
    }
    impl.terminate();
  }
  static void traversePart(Part base, int hops) throws Exception {
    nullProcedure(base.x, base.y, base.partType);
    if (hops <= 0) return;
    for (PartConnection c = base.leftHead; c != null; c = c.leftNext)
      traversePart(c.rightOwner, hops-1);
  }
  public static void nullProcedure(int x, int y, String type) { }
}
```

Figure 1: Code for Traversal Benchmark

6 Qualitative Assessment

In this section we offer some preliminary assessments of several qualitative aspects of the approaches to persistence for Java whose performance we have examined in previous sections. As well as presenting the specific assessments, this section serves to identify and describe a set of qualitative aspects that we consider relevant to the assessment of persistence approaches. These qualitative aspects are largely motivated by the objectives and goals established at the outset of our work on integrating SPIN capabilities with Java. We recognize that both our choice of aspects and our assessments themselves are obviously somewhat

```
import EDU.umass.cs.ccsl.JSPIN.OpenOODBInterface.*;

public class OpenOODBImplementation extends JSPINImplementation
{
  public OpenOODBImplementation() throws Exception { }

  public void create(Options opt) throws Exception {
    initiate(opt); }

  public void initiate(Options opt) throws Exception {
    ps = new OpenOODBStore(opt.name);
    ps.beginTransaction(); }
}
```

Figure 2: Implementation for the JSPIN/OpenOODB benchmark.

subjective. We hope, however, that they are indicative and may inspire others to offer their own candidate qualitative aspects and assessments.

6.1 Seamlessness

For the quality of seamlessness, the key aspects of a persistence approach concern its compatibility with the Java language and programming style. Generally speaking, both JSPIN and OPJ do well in terms of seamlessness, while the JDBC approach does not. More specifically, we consider the following aspects:

Code modifications Our OO1 benchmarking exercise does not involve taking an existing, non-persistent application and adding persistence. Nevertheless, we can still assess the extent to which persistence introduces additional code or modifications to code that would be used to do similar tasks without persistence.

For both JSPIN and OPJ, there is very little additional code and essentially no modifications. Moreover the two approaches are virtually identical in this regard, since both require the addition of calls to `fetch()` and `persist()` or their equivalents. OPJ does have a slight edge in that it makes transaction begin and end implicit, while JSPIN requires that explicit calls be added for transaction control. On the other hand, JSPIN has a slight advantage in that it avoids explicit casts that are required by OPJ.

JDBC, on the other hand, introduces a great deal of additional code and modifications. Most notably, JDBC requires the use of a distinct type system and a completely different interface to persistent objects.

Support tools Another aspect of seamlessness is the extent to which users of support tools such as debuggers or browsers will be made aware of differences between non-persistent and persistent code. OPJ has an advantage over JSPIN here, since the JSPIN approach of invisibly modifying the inheritance hierarchy by introducing the `PersistentObject` class will be apparent through a debugger or

```
import EDU.umass.cs.ccsl.JSPIN.MnemeInterface.LocalMnemeStore;

// Base class for an implementation.
public class LocalMnemeImplementation extends JSPINImplementation
{
  public LocalMnemeImplementation() throws Exception { }

  public void create(Options opt) throws Exception {
    ps = LocalMnemeStore.createMnemeStore(opt.name);
    ps.beginTransaction(); }

  public void initiate(Options opt) throws Exception {
    ps = new LocalMnemeStore(opt.name);
    ps.beginTransaction(); }
}
```

Figure 3: Implementation fragments for the JSPIN/Mneme benchmark.

```
import EDU.umass.cs.ccsl.JSPIN.PersistentStore;

abstract public class JSPINImplementation extends Implementation
{
  PersistentStore ps;

  public void terminate() throws Throwable {
    ps.commitTransaction();
    ps.finalize(); }

  public void makeDatabasePersist(Database d) throws Exception {
    d.persist("Database"); }

  public Database fetchDatabase() throws Exception {
    return Database.fetch("Database"); }

  public void makePartPersist(Part part) throws Exception {
    part.persist(Integer.toString(part.partId)); }

  public Part fetchPart(int id) throws Exception {
    return Part.fetch(Integer.toString(id)); }

  public void stabilize() throws Exception {
      ps.commitTransaction();
      ps.beginTransaction(); }
}
```

Figure 4: Common implementation for both JSPIN benchmarks.

```
import UK.ac.gla.dcs.opj.store.*;

public class OPJImplementation extends Implementation
{
  private PJStore ps;
  private BalancedBinaryTree parts;

  public OPJImplementation() throws Exception { }

  public void create(Options opt) throws Exception {
    ps = new PJStoreImpl();
    parts = new BalancedBinaryTree();
    ps.newPRoot("Parts", parts); }

  public void initiate(Options opt) throws Exception {
    ps = PJStoreImpl.getStore();
    parts = (BalancedBinaryTree)(ps.getPRoot("Parts")); }

  public void terminate() { }

  public void makeDatabasePersist(Database d) throws Exception {
    ps.newPRoot("Database", d); }

  public Database fetchDatabase() throws Exception {
    return (Database)ps.getPRoot("Database"); }

  public void makePartPersist(Part part) throws Exception {
    parts.insert(part.partId, part);}

  public Part fetchPart(int id) throws Exception {
    return (Part)(parts.fetch(id)); }

  public void stabilize() throws Exception {
    ps.stabilizeAll(); }
}
```

Figure 5: Implementation for the OPJ benchmark.

browser. JDBC, of course, fares substantially worse in this aspect, since it introduces non-Java types and a visible boundary between the manipulation of non-persistent and persistent data.

6.2 Barriers to Adoption

Some key aspects related to the ease with which Java users can adopt a persistence approach are considered below. The basic issue here is the impact of the persistence approach on programmers, particularly those who might not be (direct) users of persistence. We believe that OPJ and JSPIN have different strengths and weaknesses in this regard, but that both impose substantially less impact on programmers than does JDBC.

Language extensions Neither OPJ nor JSPIN introduces any modifications to Java syntax nor demands use of a separate specification language. Both add (roughly equivalent) classes that make available their persistence capabilities, but these have no impact on programmers not using persistence. JDBC, however, requires the use of a separate specification language (SQL).

Compiler modifications OPJ and JDBC both work with any Java compiler, while JSPIN depends upon compiler modifications.

Virtual machine modifications Both JSPIN and JDBC run on any standard Java virtual machine, while OPJ depends upon virtual machine modifications.

Operating system dependencies JSPIN can be run on any Unix operating system that supports Java, while OPJ (currently) requires Solaris.

6.3 Interoperability

We consider two kinds of interoperability and assess how the various approaches measure up on the corresponding aspects.

With pre-existing Java classes One aspect of interoperability concerns how easily code incorporating usage of a persistence approach can be integrated with other Java classes that do not (explicitly) make use of the approach. OPJ has an advantage here, since it can be used with any Java class. With OPJ it is even possible to make instances of existing classes persistent, with no modifications or recompilations required. Integration of code that uses the JSPIN approach with other, pre-existing classes does not require modifications or re-compilation so long as no instances of those classes need to become persistent. If instances do need to become persistent, the classes would need to be (re-)compiled with the JSPIN compiler. Integration of code using the JDBC approach with other, pre-existing classes is more problematic. Unlike in the OPJ and JSPIN cases, the pre-existing classes would not necessarily be able to manipulate objects without being aware of their persistence attributes. Moreover, if instances of a pre-existing class needed to become persistent, the class would need to be redefined using SQL, not merely recompiled.

With code and data from other languages OPJ does not provide for sharing of persistent Java objects across language boundaries. JSPIN, on the other hand, is designed to take advantage of the common SPIN framework that it shares with the C++ and CLOS APIs of the OpenOODB and hence will offer transparent polylingual interoperability. JDBC, due to the application-language neutrality of SQL, offers the potential for access from non-Java applications to persistent objects created and stored by a Java program, e.g. using ODBC. Such access, however, will be far from transparent.

7 Conclusion

In this paper, we have presented our JSPIN approach to supporting persistence for Java and outlined its current prototype implementation. We have also described our adaptation of the OO1 performance benchmark for application to JSPIN and some alternative approaches. Finally, we have reported on the preliminary data resulting from application of the adapted benchmark to the various approaches and discussed our preliminary observations concerning usability factors and JSPIN.

It is our hope that the work reported here will contribute to research on persistence for Java in at least three ways:

- By describing, and providing some initial assessment of, one particular approach to extending Java with persistence (and other) capabilities. At this early point in the development of persistence approaches for Java we believe that there is inherent value in exploring a range of alternative approaches;

- By helping to establish a basis for systematic assessment and comparison of alternative approaches to persistence for Java. We see the potential for routine use of our adaptation of the OO1 benchmark as one component in a standard suite of measurements used in assessing and comparing approaches, and our candidate criteria for qualitative assessment serving a similar function;

- By taking a first, albeit quite preliminary, step toward establishing a collection of data useful for assessing and comparing various aspects of various approaches. We find this initial data interesting and suggestive, but we believe that much more data should be gathered and analyzed in order to support more detailed and complete comparisons.

Our overall aim is to facilitate ongoing development of approaches to persistence for Java, both our own and others.

Since we find the results of our initial experiences with JSPIN quite encouraging, we plan to continue development, use and assessment of JSPIN along a number of directions. Some of the immediate directions involve improvements in the prototype JSPIN implementation. We intend to migrate to JDK 1.1 and use the Serialization and Reflection interfaces to reduce our dependence on compiler modification. We do not, however, expect to abandon our compiler changes entirely because without them we have no way of tracking the cleanliness of resident objects. We will soon add the Name Management API that was discussed in [9] to JSPIN, and we will port the Open OODB interface to Open OODB 1.1. We hope also to improve this interface by calling the Open OODB kernel directly rather than pretending that our objects are C++ objects. Other intended enhancements to JSPIN include support for unbinding names from objects, richer and more flexible transaction functionality, and distributed and multi-user versions.

We also plan to continue and expand our assessment and comparison activities. We would like to apply our adaptation of the OO1 benchmark to additional alternative approaches to persistence for Java, and to gradually bring our use of this benchmark closer to the intentions for usage described in the original OO1 report [5]. In addition, we are interested in carrying out other assessments. For example, we hope to experiment with the OO7 benchmark [4] and possibly others for application to Java persistence approaches. The experiments that Jordan carried out with an early version of OPJ [7], although less suited for use with approaches such as JDBC, are also candidates for inclusion in a standard repertoire of assessment benchmarks, so we intend to try applying some or all of them to JSPIN and perhaps other alternative approaches.

Finally, a primary objective of our JSPIN development is to complement similar extensions to C++ and CLOS, thereby providing a convenient basis for extending our work on polylingual interoperability [13, 3]. Support for interoperation between JSPIN and the Persistent C++ and Persistent CLOS APIs of the Open OODB is expected be extremely valuable. It will, for example, facilitate interoperation between Java programs and existing software written in C++ or CLOS. It also represents a simple route to an object querying capability, since Open OODB provides an OQL for C++ which will be directly usable from, and on, Java programs and objects via our polylingual interoperability mechanism.

In sum, we find our JSPIN approach to persistence for Java and its current prototype implementation to be a useful and practical utility, a basis for interesting assessment and comparison activities and a solid foundation for further development and research.

Acknowledgments

This paper is based on work supported in part by the Defense Advanced Research Projects Agency and Texas Instruments, Inc. under Sponsored Research Agreement SRA-2837024. The views and conclusions contained in this document are those of the authors. They should not be interpreted as representing official positions or policies of Texas Instruments or the U.S. Government and no official endorsement should be inferred.

REFERENCES

[1] Apple Computer Inc. *Dylan Reference Manual*. Draft, September 29, 1995.

[2] M. P. Atkinson, L. Daynès, M. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 25(4):68–75, Dec. 1996.

[3] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *Proc. Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 147–155, San Francisco, CA, October 1996. ACM SIGSOFT.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. Technical report, University of Wisconsin-Madison, Jan. 1994.

[5] R. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17(1):1–31, Mar. 1992.

[6] G. Hamilton, R. Cattell, and M. Fisher. *JDBC Database Access with Java: A Tutorial and Annotated Reference*. The Java Series. Addison-Wesley, 1997.

[7] M. Jordan. Early experiences with persistent Java. In PJW1 [16]. To appear.

[8] A. Kaplan. *Name Management: Models, Mechanisms and Applications*. PhD thesis, The University of Massachusetts, Amherst, MA, May 1996.

[9] A. Kaplan, G. A. Myrestrand, J. V. E. Ridgway, and J. C. Wileden. Our SPIN on persistent Java: The JavaSPIN approach. In PJW1 [16]. To appear.

[10] A. Kaplan and J. Wileden. Conch: Experimenting with enhanced name management for persistent object systems. In M. Atkinson, D. Maier, and V. Banzaken, editors, *Sixth International Workshop on Persistent Object Systems*, Workshops in Computing, pages 318–331, Tarascon, Provence, France, Sept. 1994. Springer.

[11] A. Kaplan and J. C. Wileden. Name management and object technology for advanced software. In *International Symposium on Object Technologies for Advanced Software*, number 742 in Lecture Notes in Computer Science, pages 371–392, Kanazawa, Japan, Nov. 1993.

[12] A. Kaplan and J. C. Wileden. Formalization and application of a unifying model for name management. In *The Third Symposium on the Foundations of Software Engineering*, pages 161–172, Washington, D.C., Oct 1995.

[13] A. Kaplan and J. C. Wileden. Toward painless polylingual persistence. In R. Connor and S. Nettles, editors, *Persistent Object Systems, Principles and Practice, Proc. Seventh International Workshop on Persistent Object Systems*, pages 11–22, Cape May, NJ, May 1996. Morgan Kaufmann.

[14] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, Apr. 1990.

[15] Object Design Inc. ObjectStore PSE. http://www.odi.com/products/PSE_Homepage.html.

[16] *Proc. First International Conference on Persistence in Java*, Drymen, Scotland, Sept. 1996. To appear.

[17] POET Software, http://www.poet.com/. *POET Java SDK Documentation*, 1997.

[18] J. V. E. Ridgway. Concurrency control and recovery in the Mneme persistent object store. M.Sc. project report, University of Massachusetts, Amherst, MA, Jan. 1995.

[19] P. L. Tarr, J. C. Wileden, and L. A. Clarke. Extending and limiting PGRAPHITE-style persistence. In *Proc. Fourth International Workshop on Persistent Object Systems*, pages 74–86, August 1990.

[20] P. L. Tarr, J. C. Wileden, and A. L. Wolf. A different tack to providing persistence in a language. In *Proc. Second International Workshop on Database Programming Languages*, pages 41–60, June 1989.

[21] Texas Instruments Inc. *Open OODB 1.0 C++ API Reference Manual*, 1995.

[22] D. L. Wells, J. A. Blakely, and C. W. Thompson. Architecture of an open object-oriented management system. *IEEE Computer*, 25(10):74–82, Oct. 1992.

[23] J. C. Wileden, A. L. Wolf, C. D. Fisher, and P. L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, November 1988.

[24] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr. Specification level interoperability. *Communications of the ACM*, 34(5):73–87, May 1991.

Main-Memory Management to support Orthogonal Persistence for Java™

Laurent Daynès and Malcolm Atkinson*
Computing Science Department,
Glasgow University, Scotland
{laurent,mpa}@dcs.gla.ac.uk

Abstract

The main-memory management of an object cache that provides an orthogonally persistent platform for Java is described. Features of the architecture include: two levels of buffering between the disk and virtual machine, a consistent representation of transient and active objects, an efficient residency checking algorithm in the presence of multiple threads of execution, and well developed object-cache management technology.

We present some of the challenges of providing persistence for the Java virtual machine, which may be typical of any attempt at an industrial-strength orthogonally persistent programming system. Some of these might have been avoided by sacrificing persistence independence, but that is far too valuable to sacrifice.

We report on some detailed investigations of pinning and cache replacement techniques applicable in this context. The information available for eviction-victim selection is inevitably limited by imprecise information about the state of the machine and the cost of collection. The question is raised whether we can do better than random eviction with real rather than synthetic loads given these limits.

1 Introduction

Orthogonally persistent programming languages provide the illusion of a very large space of objects and give seamless access to objects independently of their lifetimes, be they transient or persistent. Supporting this abstraction with limited main-memory resources requires special memory-management techniques to provide executing application programs with in-memory access to persistent objects.

This paper reports on the design and implementation of main-memory management for an orthogonally persistent system for Java™ [17] developed at the University of Glasgow in collaboration with Sun Microsystems Laboratories[1]. It describes the main-memory management used for the first prototype of orthogonal persistence for Java, PJama$_0$, with a focus on object faulting and replacement mechanisms. It also describes how Sun's Java Virtual Machine (JVM) [25], as implemented by the Java Development Kit (JDK), was modified to incorporate this new form of main-memory management[2].

We had to reconsider the design of the object cache and the algorithms for faulting objects in and evicting them from it as our original design did not recycle a sufficient proportion of the space allocated within the object cache. We also found that discovering which objects should be pinned and which could be evicted more complex than we expected. In overcoming these difficulties, we believe we have developed new techniques that are useful for space management during long-running transactions. These depend on

*Visiting professor at Sun Microsystem Laboratories, Mountain View, CA
[1]Sun, Java and JDK are registered trademarks of Sun Microsystems Inc. in the USA and other countries.
[2]The appendix holds a table of definitions of the acronyms and some terms used in this paper.

extracting information about the state and operations of the abstract machine. We present details of the mechanisms developed as we are not aware of them being described elsewhere and we believe they may be useful to other implementors of persistent languages and OODB bindings.

As stated previously [6, 4], our overall strategy was to devise an architecture which made objects that have been faulted in, and their house-keeping data structures, look very similar to the representations used by the unmodified JVM to avoid making extensive changes to the JVM.

Like many persistent object systems, PJama_0 uses a two levels of buffering. A small Page Buffer Pool (PBP) is used as a staging area for the Persistent Object Store (POS) pages. The objects needed by the JVM are copied on-demand from the PBP to an Object Cache (OC), and translated to a format expected by the JVM. All memory areas: garbage collected heap, buffer pool and object cache, are shared by all threads.

The state diagram in figure 1 shows a simplified summary of the object life-cycle in the dual, no-steal, buffer architecture considered in this paper. Transient objects are objects whose life never goes beyond the initial “heap allocated” state. Objects promoted to persistence are destined for a longer life which may involve an arbitrary number of cycles of state transitions depending on how applications operate on them.

In order to be manipulated by applications, a persistent object must go through two transitions (1 and 3), corresponding to the two levels of buffering. The first update to a clean object changes the state of that object such that it cannot be evicted (transition 6). This constraint is the consequence of the no-steal policy. A stabilization brings the object back to an evictable state (transition 7), from which eviction is possible. Eviction of an object changes that object’s state to either “in PBP” (transition 4), or “on disk” (transition 5), depending on whether the page where that object resides is cached in the PBP.

This paper focuses on the support for transitions 1 to 5, but our technology is constrained by the need to implement *all* of the transitions shown consistently and efficiently.

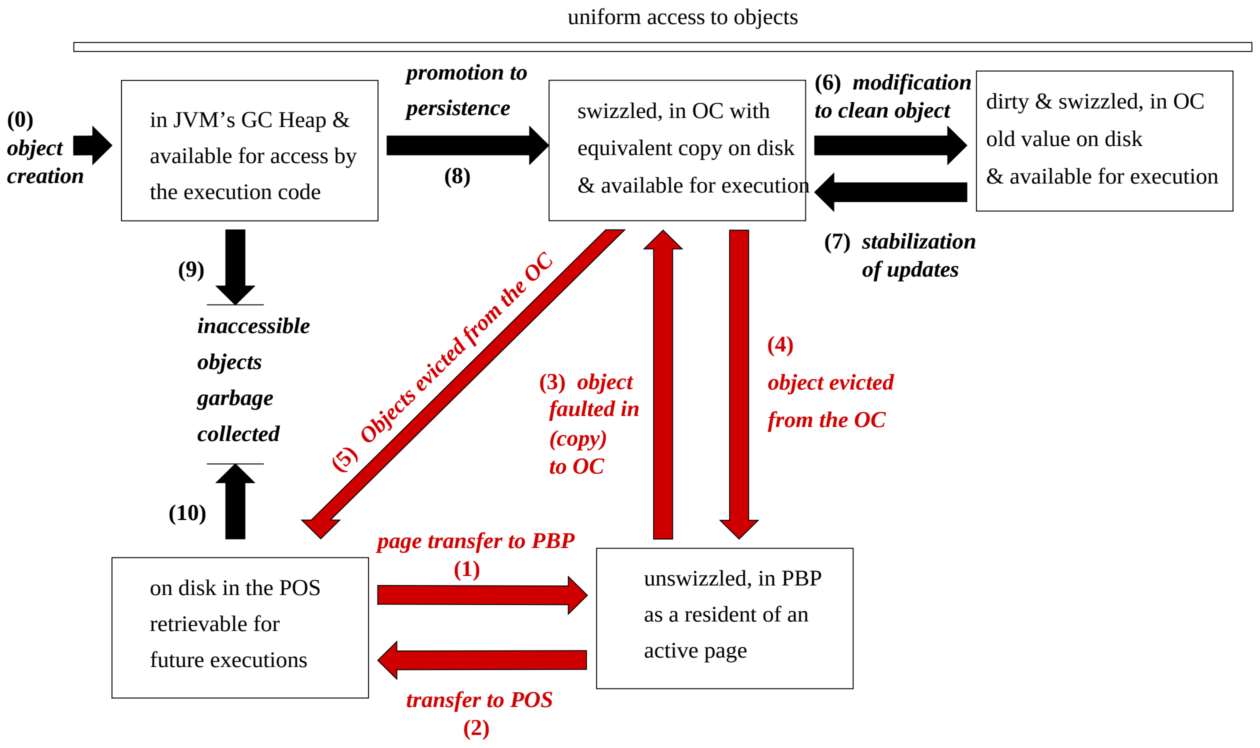

uniform access to objects
(0) object creation
in JVM's GC Heap & available for access by the execution code
promotion to persistence
(8)
swizzled, in OC with equivalent copy on disk & available for execution
(6) modification to clean object
dirty & swizzled, in OC old value on disk & available for execution
(7) stabilization of updates
(9)
inaccessible objects garbage collected
(10)
(5) Objects evicted from the OC
(3) object faulted in (copy) to OC
(4) object evicted from the OC
page transfer to PBP (1)
transfer to POS (2)
on disk in the POS retrievable for future executions
unswizzled, in PBP as a resident of an active page

Figure 1: A simplified persistent object life cycle.

The rest of this paper is organized as follows. The programming model of PJama is briefly described in section 2. Section 3 outlines the modification made to the JVM and the architecture of the resulting virtual machine (OPJVM). It then describes the state transitions 1 to 3. Section 4 discusses more extensively the interface between the JVM and the OC that implements state transition 3. Section 5 contrasts OC management with garbage collection and concludes that garbage collection is not appropriate for implementing state transitions 4 and 5. It then presents in some technical detail several solutions that implement pinning, victim selection and safe OC space reclamation. Section 6 reviews some other OC management research and other implementations of *orthogonal* persistence for Java. Section 7 concludes with a summary and directions of further work.

2 Programming model

The PJama[3] system is intended to provide an alternative platform for the Java language with provision of orthogonal persistence for data, meta data (classes) and code (methods). Persistence is added to the Java language with no perturbation to Java's semantics. All Java classes can be re-used in persistent applications and made persistent as well as their instances without any alteration to either their source or their compiled form. This careful provision of *persistence independence* is important in enabling code, available either as source Java or compiled class files, prepared for other environments, to be re-used in the PJama context *without any transformation.* An extensive discussion of orthogonal persistence is given in [7] and its application to the language Java is described in [4, 6].

Using PJama, the bulk of an application suite consists of Java classes and methods written *exactly* as they would have been written if the application was transient, using only main memory for its data structures. Each program, typically the class containing the `main` method, requires a few lines of code to initiate its binding with the persistent store, and the roots of preserved data structures before calling that standard, main-memory style code. Typically, only a few (< 20) lines of code using the `PJStore` interface enable thousands of lines of unmodified Java using an unrestricted[4] range of classes to operate on a mixture of transient and persistent data. This localised, persistence-aware code typically identifies the roots of persistence, binds these root objects to the application's variables, calls the application, and when it returns, triggers the stabilization[5] of all updates onto the persistent store.

3 An orthogonally persistent Java Virtual Machine

The primary objective of PJama is to demonstrate the feasibility of orthogonal persistence in an industrial-strength programming language such as Java. Our first goal was to develop in six months a virtual machine for Java that supported a prototype, PJama$_0$, with reasonable performance. This OPJVM had to be derived from Sun's JDK. We chose to minimize the changes to the JDK to make it feasible to use successive JDK versions.

To construct our prototype quickly, aspects related to fault-tolerance were consigned to a third-party software. RVM, a *Recoverable Virtual Memory* system from CMU [30], was choosen for its availability. It manages storage as an unstructured address space; no object abstraction is supported. This left us complete freedom to design the structure of the persistent storage. RVM is also very flexible with respect to access to persistent data. RVM applications are left responsible for deciding when and how much data should be loaded into or flushed from virtual memory, as well as where in virtual memory these data should be loaded. These advantages are tempered by one major constraint: RVM imposes a no-steal policy [18], which means that memory-resident regions with uncommitted updates cannot be flushed to persistent storage. This constraint and the decision to stay close to JDK's implementation of the JVM significantly influenced the architecture of OPJVM.

The rest of this section presents that architecture and discusses the rationale behind it. Aspects of JDK1.0.2 that impacted the design decisions are described first. Where differences between JDK1.0.2 and

[3]Originally this was called "PJava", but that name has been taken by Sun for Personal Java a product for PDA devices.

[4]There is at present one restriction, it is not yet possible to make instances of the class `Thread` persist because of the complexities of capturing information from a thread's C stacks.

[5]PJama parlance for atomic propagation of updates onto the POS including *promotion* of objects, classes and methods newly reachable or needed by promoted objects.

JDK1.1 are significant we note how we have adapted to them.

3.1 Crucial features of JDK1.0.2

Two features of JDK1.0.2 make it difficult to reliably locate and adjust pointers, and therefore had a major impact on the design of OPJVM: its memory and thread models.

3.1.1 JDK1.0.2's existing memory model

The right-hand part of figure 2 depicts the memory model of JDK. It uses a garbage collected heap (GC heap) for allocating class instances and arrays. Class objects (i.e., instances of the class `java.lang.Class`), and the related information such as constant pool, field and method blocks, bytecodes (these are byte arrays that each hold a sequence of byte codes generated by a Java compiler from processing one method), and the stacks for each Java thread, are allocated outside of the GC heap.

Objects in the GC heap and class objects are given a *handle* for their (transient) lifetime[6]. Each handle contains the location of the object it references and a pointer to its method table (see 4.1 for details). Objects reference each other via their handles, and the Java stack of each thread references objects via their handles. The locations of references from objects to handles, henceforth simply "references", may be discovered from class objects. In contrast, the locations of references on both native and Java stacks are not recorded. Class objects are often directly referenced using their address in virtual memory, e.g. by the directory of loaded classes, constant pools, method tables, method blocks and field blocks[7].

Handles are *dereferenced* in order to manipulate the contents of an object (see 4.2 for the mechanism). Dereferencing causes the address of objects to be temporarily stored in machine registers or on native thread stacks. Native methods also tend to reduce the number of dereferences by keeping direct pointers obtained from handle on their native stack for the duration of multiple accesses to the objects, often for most of a native method's execution. Because of this, and because of the difficulties of locating all references described above, the JDK's garbage collector uses a conservative approach [34].

The GC heap is organized as two contiguous areas: a pool of handles and a pool of objects. Both objects and handles are 8-byte aligned. Each allocated chunk of the heap is prefixed with a 4-byte header reserved for memory management (the rest of the JVM ignores the existence of this prefix). Headers contain the size of the chunk and a couple of bits. The garbage collector, which operates on both the handle pool and the GC heap, is invoked when the allocator cannot satisfy a request[8] and uses a stop-the-world mark-and-sweep strategy.

Being conservative means that the garbage collector can easily move an object immediately reachable *only* from its handle, but it dares not move any object referenced directly from elsewhere (to attempt to fix up the direct pointers might modify a scalar value that coincidentally looked like a direct pointer). It differentiates between these two cases by whether the object as well as its handle has been marked. The result is a fragmented heap with immovable islands, even after compaction.

3.1.2 Thread model and implementation

JDK uses its own threads package, called *Green Threads*, to implement Java threads. The Green Threads package implements a pre-emptive model with priorities[9]. Each thread is provided with its own native stack, native register values, and Java execution context which includes a Java stack, and other information such as the last exception thrown. A native stack is used for the execution of the JVM and native methods while a Java stack is used for the local variables of methods. All of the threads share the same GC heap.

[6]Versions of the JVM which operate without handles have been developed. At present we are not exploring how to work with such JVMs.

[7]JDK1.1 now uses handles for these references, as a consequence of moving class objects into the GC heap but the constant pool of classes may still contain direct pointers to class objects.

[8]It may also run in the background.

[9]Time-sliced scheduling is allowed as a command line option in JDK 1.1. It is implemented using a hidden daemon thread with a priority higher than any thread in the system.

Threads are either yielded explicitly by the application, or pre-empted by an asynchronous event, such as timeout or I/O. In the latter case, the running thread is immediately pre-empted irrespective of its location in the JVM code. This means that a thread may be suspended in the middle of the execution of a virtual machine instruction or a native method execution. In either case, a thread may be suspended while holding direct pointers to some Java objects[10].

This lack of coordination between direct pointer usage and thread pre-emption has a considerable impact on the design of main-memory management to support persistence.

3.2 Adding a Dual Buffer to the JVM

A dual buffer, shared by all threads was chosen for OPJVM's initial prototype. Figure 2 illustrates how the JDK has been extended to incorporate dual buffering of persistent objects: class instances, arrays, classes and bytecodes. The JVM now interfaces with an object cache (OC) shared by all threads to gain access to persistent objects. Objects in the OC have exactly the same layout as objects in the GC heap which is the format expected by the JVM: they are 8-byte aligned, and prefixed with a 4-byte header. Memory-resident persistent objects are referenced via handles that look like the handles of the GC heap (see 4.1)

The OC itself sits on top of a store abstraction which encapsulates all manipulations of persistent storage. This architecture has met the requirement to minimise changes to the JDK, permitted the rapid implementation we needed and fitted within the constraints imposed by RVM. Interposing an object cache between the JVM and the store layer has simplified the overall system and made it more reliable.

- It reduced interdependencies between the JVM and RVM.
- It simplified the tracking of updates and their propagation to the persistent object store (POS). To have updated objects while they were in the PBP would have required either running the JVM within an RVM transaction, notifying it of every update or starting an RVM transaction at each stabilization and recording the undo images of each object modified. Both solutions were rejected because of their inefficiency and development cost.
- It relieved the PBP manager from having to track direct pointers from the JVM to its buffer, which improves the reliability and performance of the store layer by reducing the number pages pinned and by eliminating the cost of tracking direct pointers (see section 5.2).
- It minimized the time that pages hold updated but uncommitted data and hence the time pages need to be pinned. This in turn reduces the average number of dirty PBP pages which both improves caching and reduces risk of the OC becoming full of pinned pages so that progress is blocked.
- It simplified memory management for objects larger than one page. Such objects are copied in their entirety onto the OC. This does not cope well with very large objects and may incur punitive eviction and copying costs, but it suffices for our initial prototype. More sophisticated techniques would have required too many changes to the JVM.

The detrimental aspects of this architecture that every object has to be copied, might be ameliorated by an adaptive mechanism that copies objects from the PBP to the OC if their page is about to be evicted and it is expected that they will be used again. This is similar to an adaptive mechanism proposed by Kemper and Kossmann [22] which also uses the term "dual-buffer management" as we do[11], to indicate objects may be either in an object cache or a page pool when in main-memory. They conclude that "lazy object copying combined with an eager relocation strategy is almost always superior and significantly outperforms page-based buffering in most applications". Whether there is an actual saving, depends on several factors:

- the relative size of the page and the most active objects;

[10]Such situations will become even more common in JDK 1.1 when it is used with the time-slice option.

[11]The term was coined originally in [24] with this meaning, though the degree of dynamic adaptability between the space available for object buffering and that for page buffering varies between papers.

- how often the move can be avoided because the page remains resident long enough;
- how much of the object needs to be read and written before it is prepared for use by the JVM[12];
- what proportion of the objects are mutated and therefore need copying back agaian; and
- how good the heuristics are at choosing which objects to copy.

Because of the complexities of the JVM and hence the challenge of making the object cache operate correctly, which we describe in this paper, it has not yet been possible to investigate such adaptive strategies in PJama. However, our cache architecture is designed to make them possible.

3.2.1 The store layer and its use of RVM

The store layer includes the PBP, realizes logging, recovery and disk IO and thereby encapsulates access to the POS. It provides a simple interface for executing the following actions atomically:

- obtaining copies of persistent objects from the POS;
- propagating updates on a set of objects to the POS;
- extending the size of the POS with a set of new objects, and
- creating, opening and closing stores.

It uses a slightly modified version of RVM 3.1 (RVM had to be changed to support extension of segments with mapped regions) which provides primitives to *map* and *unmap* arbitrary regions of persistent *segments* into arbitrary regions of virtual memory of the same size. Regions can be mapped to only one place at a time, they must not overlap, and their size must be a multiple of the page size used by RVM.

Mapping of regions in RVM consists of making ranges of virtual memory addresses coincide with ranges of segment offsets, and effectively fetches the segment region to the specified virtual memory locations. Access to regions of a segment has to be detected first by OPJVM which then invokes the mapping primitive to load the missing region into the PBP. This may be contrasted with the memory-mapping facilities used by operating systems where access detection and loading are automatic.

RVM has been used in a rather unusual way to implement the PBP manager (which supports the object state transitions 1 and 2 described in Figure 1). The buffer manager detects page faults when requests for persistent objects are made. If the page containing a requested object is not in the PBP, a page-fault is raised. Page-faults are processed by allocating a buffer and calling RVM's `map` primitive to load the missing page to that buffer. The object is then copied to the location specified by the request with the buffer being pinned for the duration of that copy. Page-fault detection and resolution are synchronized to prevent multiple threads from processing the same page fault.

Pages are evacuated from their buffer as necessary. A classic two-hand clock algorithm [31] is used for selecting victims for replacement in the buffer pool. The evacuation itself is performed using RVM's `unmap` primitive. Because of RVM's no-steal policy, the victim pages must not contain uncommitted updates.

Stabilizations are implemented using RVM transactions. An RVM transaction is started during a stabilization and used to propagate the updates from the object copies in the OC to the POS. The modified objects are first copied directly to pages loaded in PBP (page-faults may occur to fault in pages on which modified objects reside). These updates are recorded to the log by using the `rvm_set_range` procedure and atomically propagated to the store by committing the RVM transaction. The contents of updated buffers need not be forced to disk before transaction commit, as RVM supports no-force commit.

[12]Normally, all the pointer fields need to be read and written, but if the scalar formats also change, e.g. the store is in big-endian format, but this execution is on a little-endian machine, then a copy does not incraese the number of memory cycles required, nor the disruption of the hardware cahes.

This contrasts with reported use of RVM where entire RVM segments are mapped to main memory [33, 16], thus avoiding issues such as on-demand fetching and automatic eviction of mapped regions at the price of scalability because of the main-memory residency assumption[13].

3.2.2 The Object Cache

The OC implements an object-faulting mechanism to load on-demand (transition 3) any objects required by the JVM. This process is illustrated in figure 2. Upon an object-fault, the OC manager asks the store layer for a copy of the missing object. Once the object has been copied into the OC, inter-object references are translated from a persistent-identifier (PID) format to a virtual-memory address: this is called *pointer swizzling*.

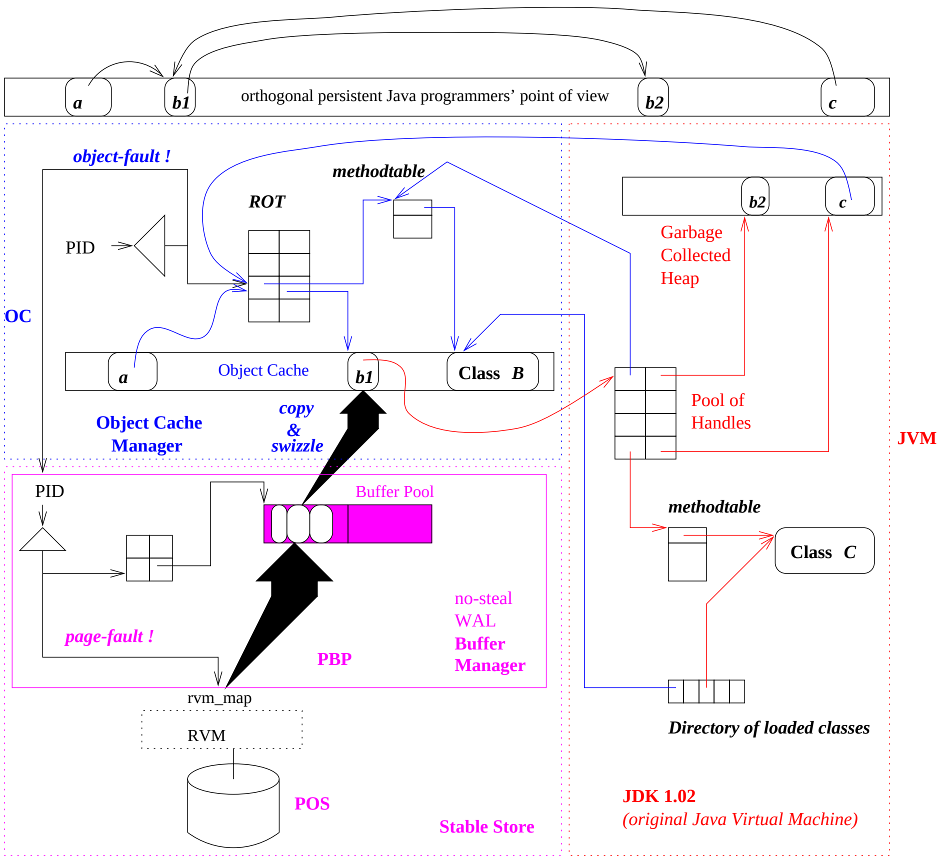


Figure 2: OPJVM's architecture.

The use of handles by JDK to access objects naturally leads to the use of *indirect swizzling* [23, 7] for the OC. Indirect swizzling means that pointer fields in the object being swizzled containing PIDs are overwritten with pointers to handles. An *eager* indirect swizzling strategy is used for class instances, and a *lazy* indirect swizzling strategy is used for arrays of objects. Eager swizzling means that *all* persistent identifiers within an object are overwritten with pointers to handles at object-fault-time. Lazy swizzling means that references are swizzled upon the first use of a reference, which is advantageous for large arrays of objects when only a few references are actually used, but requires a relative expensive read barrier so it does not pay off for small arrays or those that are scanned in their entirity. At present we cannot predict the size and arrage usage so conservatively we use lazy swizzling for all arrays.

[13]In many cases this is actually a mapping to virtual memory, which is often restricted in size per application by the operating system, and which will often perform poorly compared with object caching.

A resident object table (ROT), hashed on PIDs, is used to find handles corresponding to PIDs. The handles in the ROT, called *ROThandles*, correspond to all the objects in the OC, objects referenced from resident objects but representing non-resident objects, and unreclaimed handles referring to evicted objects. When pointer swizzling, an attempt is made to find a handle for the PID in the ROT, and if this fails a new ROThandle is allocated.

The OC also manages several house-keeping data structures including a memory-resident copy of the persistent class directory (PCD). This is a mapping from the names of *all* the classes in the POS to their class descriptors (instances of `ClassClass`). When a class is required by name, a look up determines if it has already been made persistent, in which case it will be loaded from the POS. Consequently, special faulting mechanisms are used for classes and bytecodes. Whenever a class is required by name (Java's dynamic binding) the OPJVM first attempts to find it in the PCD, and uses the size and reference in the returned descriptor to fetch it into the OC. If it is not found in the PCD the normal JVM class loader mechanisms are used. A class object may also be faulted in because a reference to it is dereferenced.

New objects are allocated on the GC heap exactly as in Java. The object allocator is called with a class descriptor which may have been loaded by the JVM or may be persistent. If the class is persistent then the method table reference will be a pointer to the method table in virtual memory constructed by the OC at class-fault time; otherwise it will be a pointer to the method table in memory allocated by the JVM when it loaded the class. A pointer to the method table is stored in the handle returned by the object allocator (see 4.1). Figure 2 illustrates the possible combinations:

- a transient instance of a transient class (object c, instance of C),
- a transient instance of a persistent class (object b2, instance of B),
- a persistent instance of a persistent class (object b1, instance of B).

Note that in the case of class B, the method table is shared by all of B's instances, persistent and transient.

Overall, the original JVM was left intact except for additions in the bytecode interpreter, the native methods of the core classes, and the class loader and resolver (see [25] for details about these components) to support persistence-related functions such as residency checks and update tracking. The garbage collector was modified to retain objects referenced from the OC.

4 Object Faulting

This section considers the implementation of the state transition 3 in Figure 1. The main issues are how to perform: residency checking, object faulting and swizzling.

4.1 Handle and Object formats

These operations depend on four formats for handles which are shown in figure 3. We partition handles into Jhandles and ROThandles, each of which have two possible formats. Jhandles are allocated in the GC heap, whereas ROThandles are allocated in the OC and populate the ROT. Jhandles are created by the bytecode interpreter, upon the execution of a **new** instruction, while ROThandles are created by the OC during the OPJVM bootstrap, by swizzling operations (both eager and lazy), or during object promotion.

The first three of the four handle formats described here are said to be in *indirection* format as they hold the main-memory address of the denoted object.

Standard Jhandles These are exactly the handle format used in the JDK JVM and consist of two pointers, one to a method table (a `struct methodtable`), and one to the corresponding object in the GC heap. The method table has a pointer to its class descriptor (a `ClassClass` structure), and an array of pointers to `methodblocks`. The method table and the class descriptor, as well as all the descriptors reachable from them (e.g., field and method blocks, etc.), are allocated using the JVM's machine dependent allocator `sysMalloc`.

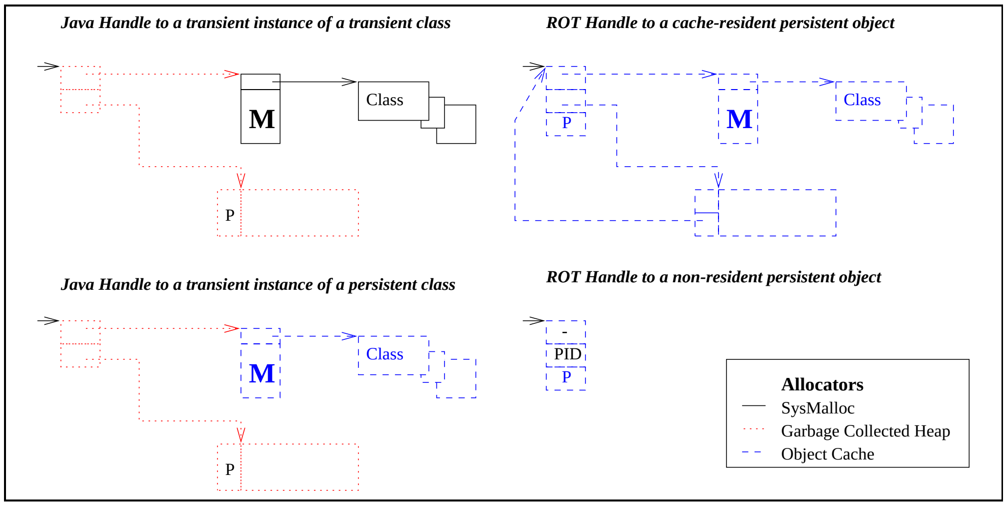

Java Handle to a transient instance of a transient class
ROT Handle to a cache-resident persistent object
Java Handle to a transient instance of a persistent class
ROT Handle to a non-resident persistent object
M
Class
P
PID
Allocators
SysMalloc
Garbage Collected Heap
Object Cache

Figure 3: Handle Formats

Jhandles for instances of a persistent class These are constructed when a **new** instruction is executed for a persistent class. Both the handle and the new instance are allocated on the GC heap as before, but the handle's method table and class are OC-resident.

swizzled ROThandles The object field of this kind of ROThandle contains the address of the object in the OC. The method table and corresponding persistent class are both OC-resident. Thus ROT's indirection handles are strictly equivalent to the JVM's handles, excluding a hidden postfix used to hold ROT's house-keeping information.

fault-blocks Swizzling of PID to non-resident objects results in a fault-block creation. A fault-block is a ROThandle where the object address is its PID so that dereferencing it will cause an object-fault. The method table field will be filled in at that time.

The OC prefixes the cache-resident image of each object with a 4-byte word which holds two pieces of information: the lower three bits are used to keep track of updates, while the upper bits contain an index into the ROT to enable its handle to be found. The objects in the JVM's GC heap are also prefixed with a 4-byte word of house-keeping information. The only components of the JVM that access these prefixes are the GC heap garbage collector for objects on the GC heap and the OC manager for objects in the OC.

4.2 Residency Checks

Every time a handle is dereferenced, a software residency check must be performed to check whether the referenced object is present in main-memory[14]. The problem is twofold: (1) how to perform these checks efficiently without changing the implementation of the Jhandles and (2) how to synchronize object-faulting in the presence of multiple threads of control.

The JVM's class instances and handles are 8-byte aligned. We have arranged that all other objects, e.g. classes and methods are also 8-byte aligned. A memory address therefore has its least significant three bits set to zero. We arrange that all PIDs have at least one of these bits set[15]. A null reference is stored on disk as a `NULLPID` value, which is actually the same value as the JVM's `null` value, that is 0. This avoids any swizzling work for null references.

[14]Optimizations avoiding redundant residency checks are planned for PJama$_1$ [12].

[15]They contain an RVM segment number — allowing up to 7 segments of up to 2^{32} bytes each in an OPJVM POS, as we use 8-byte alignment within segments.

A residency check, testing whether the least significant three bits of the object address field are zero, indicates whether the handle is a *fault-block* or an *indirection*. Thus a dereference consists of the following line of pseudo-C code:

```
p = (h->obj & 7) ? h->obj : object_fault(h)
```

where `h` is a pointer to a handle and `p` is the final location of the object in virtual memory.

Unfortunately, this residency check does not take into account the fact that multiple concurrent threads of control operate over the OC. Both the residency checks and the object-fault processing have to be indivisible to prevent the same object-fault being processed by more than one thread. Taking a mutual exclusion lock on every residency check is illadvised for performance reasons. It is essential to optimize the case where the residency check succeeds, as this includes *all* objects on the GC heap and most other objects if the execution load has reasonable locality.

The double-checking solution described in [14] is directly applicable here: the first check is performed without mutual exclusion on the assumption that the object is already resident. The OC guarantees that objects are removed in an indivisible manner with respect to residency checks (see section 5.5). For the moment, assume that the OC's content is never replaced. If the first check succeeds, then the object is guaranteed to be in memory and there is no need to synchronize a concurrent object-fault.

If the first residency check fails (which is a rare event), an object-fault has potentially been detected. In this case, a mutual exclusion is taken on the handle that raised the object-fault to double check the initial diagnosis safely.

If the second check, protected by the mutual exclusion lock, also diagnoses an object-fault, then the mutual exclusion lock is held until the object fault has been processed. Otherwise, if the second check finds the object resident in memory, the thread T that performed the check has been trapped in a *phantom object-fault* which means that another thread has sneaked in just between the first check and the acquisition of the mutual exclusion lock by T and has performed the object-fault. In this case, T just releases the lock and continues with the dereference. Phantom object-faults are likely to be exceptionally rare. Thus the final multi-threaded residency check looks like this in pseudo-C:

```
register OBJECT *p;
if ( (p = h->obj) & 7 ){        /* initial residency checks */
                                /* got a potential object-fault here */
    mutex_lock(h);              /* Get the object-fault lock for this object. */
    if ( (p = h->obj) & 7)      /* double-checks for phantom object-fault */
         p = object_fault(h)    /* this fault is a true one */
    mutex_unlock(h);            /* release object-fault lock for this object. */
}
p;                              /* memory-resident address */
```

In most cases (i.e., when the object is already in memory), object dereferencing will only consist of the first two lines, which implement exactly the single-threaded residency check.

A similar double-checking strategy is deployed to check residency of bytecodes on method invocation. Residency-check mechanisms for class objects involve a few extra wrinkles because the contents of classes may be directly addressed from an arbitrary number of *resolved* constant pool entries in other classes (see section 3.1.1).

4.3 Faulting

Object-faulting, irrespectively of the kind of the object, is a three-step process. The first step finds the size of the faulting object in order to allocate space in the OC. The second step invokes the store layer to obtain a copy of the object in the space just allocated. The third step prepares the copy of the object for use by the JVM.

All persistent objects are prefixed with a 4-byte word which denotes the type of the object and allows their size to be found. The store layer provides an interface for obtaining the prefix of any object given its PID.

The prefix of an instance of class `Class` contains the PID of its PCD entry object. PCD entry objects contain the PID of the class they refer to along with their size, which includes the size of the `Class` instance as well as the size of all its inner structures such as the constant pool, field and method blocks. This conglomeration we call the *class object*.

The prefixes of all other class instances contain the PID of the PCD entry for their class which also contains their size. This PID can be used to locate a memory-resident copy of the object's class (potentially involving faults for the PCD entry and the class object).

The prefixes of bytecodes and arrays encode their size.

5 Cache Space Management

We have to recycle the content of the main memory in order to support the orthogonal persistence abstraction within the inevitable constraints of limited main-memory resources. For example, a server written in PJama may operate continuously on an indefinitely large population of objects with unpredictably repeated accesses. The PBP, the OC and the GC heap all perform memory allocation and recycling and we therefore need to ensure that they operate synergistically and need to balance resources between them as well as within them. GC heap and OC share similar requirements, namely, low allocation cost, and good locality. In both cases, these may be achieved via linear allocation and re-organization of space to reduce fragmentation.

One primary task of a main-memory manager is to recycle the space taken by objects. An object may be vulnerable to having its space reclaimed either because it is no longer reachable from the executing threads or because there is no expectation that its re-use is imminent and it can be recovered from the POS. Consequently the OC manager must attempt to satisfy the following (conflicting) goals:

- provide space in the OC for every allocation request (these mainly arise from object-faults, promotion and house-keeping);
- retain in the OC as many of the currently resident objects as possible, particularly those that will be used in the near future;
- support rapid allocation, e.g. preserve linear allocation and minimize fragmentation; and
- optimize the locality of objects in the OC, e.g. by reclustering.

So far we have investigated two issues in PJama$_0$: selection of victims for eviction from the OC; and prevention of dangling pointers as a result of object evictions. As explained in section 3.1.2, threads may be suspended while holding direct pointers to objects. Therefore, dangling pointers may be created if such objects are evicted from the OC. The pre-emptive thread model used by Java makes this event likely since low-priority threads may be suspended for long enough that objects that they were using *appear* disused and therefore become vulnerable to eviction.

Two strategies were considered: (1) extend the garbage collector to recover OC space, and (2) develop explicit OC eviction algorithms. There were several difficulties with the first option. The existing JDK garbage collector was unsuitable, as it was insufficiently incremental and (in JDK1.02 at least) incapable of handling classes and methods. More fundamentally, garbage collection algorithms are maladapted to reclaiming cache space, as it may be necessary to reclaim space that is still reachable. Based on these considerations, we chose to explore automatic but explicit OC eviction algorithms.

5.1 Partitioning Object Cache into Specialized Management Regions

The OC is divided into two areas, a static area and dynamic one. The static area holds space permanently allocated to some house-keeping functions of the OC (e.g., the static parts of the ROT), and a *bootstrap* region. The bootstrap region contains the JDK core classes used in the operation of the OPJVM and others activated during the run of OPJVM that initialized the store[16]. It is pinned in the OC and stays memory resident for the entire execution of the OPJVM. The bootstrap region is transferred directly from disk to the OC without stepping through the buffer manager, and is immediatly translated into memory-resident format during start up.

The dynamic area is used not only for caching copies of persistent objects, but also as the storage for dynamically allocated OC house-keeping structures, such as ROThandles. Instead of intermixing very different populations of objects, the dynamic area is divided into homogeneous regions populated with objects of the same kind, i.e., we use "separate cages for incompatible animals". The most prominent kinds of region are designated for: persistent class instances and arrays, for persistent bytecodes, for persistent class objects and for ROThandles.

Each "cage" is provided with its own "keeper", called a *region manager*, each of which is adapted to the characteristics of its own kind of object and can negotiate with the overall OC manager about its share of resources. This division into homogeneous regions with specialized region management offers more opportunities to optimize OC space management than a single global scheme would. It also partitions the OC code into more tractable modules.

For instance, handles and objects have conflicting requirements. Handles need to retain their address for their lifetime because references to them are not easily located (see 3.1.1). Hence, interleaving the two kinds of object would make OC space reclamation more expensive and linear allocation much harder to achieve, because of the fragmentation caused by immovable handles. An object, on the other hand, can be moved or evicted at almost any time, and incrementally, by a single assignment to its handle whereas, reclamation of handles has to be done via pure garbage collection techniques. Similar disparities can be illustrated between the objects in each of the regions.

Another reason for distributing OC management into specialized managers is flexibility and extensibility. In the long term, our aim is to experiment with different policies for each category of object, and to specialize the management for more categories of objects, e.g. large objects, multimedia stream objects, and legacy data. The goal is to allow new specialized region managers to be dynamically added as plug-ins.

Figure 4 describes the architecture of the OC manager, which controls the usage of regions: it allocates regions on demand to region managers, and requests them to give up regions when OC resources are scarce. The OC manager maintains a map of the composition of the OC, which allows it to find quickly, given a virtual memory address, the relevant region-control block.

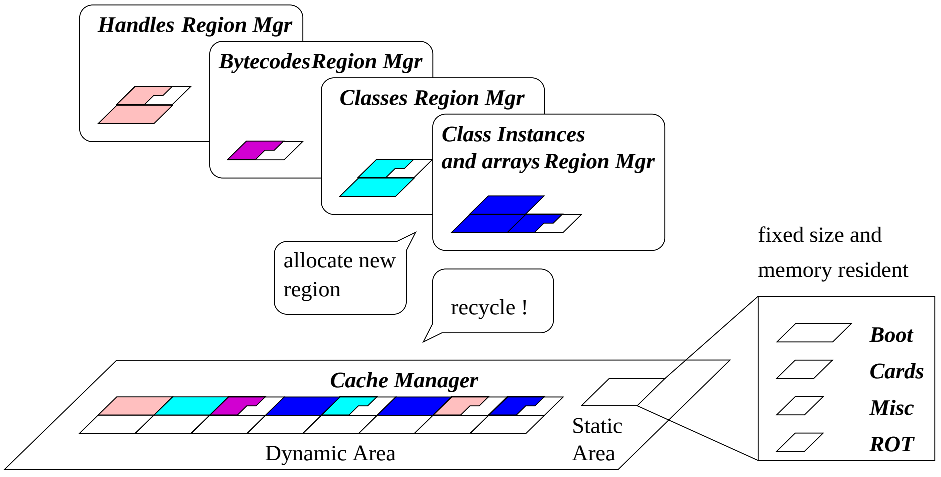


Figure 4: Object cache management.

Each region manager applies its own policy for allocating objects and for reclaiming space within the regions for which it is responsible. The OC manager dispatches requests for allocating space for a given category of

[16]There will be additional classes and other objects based on reachability at the time of the first stabilize.

objects to the region manager specialized in the management of such objects. We next discuss the following issues for implementing OC replacement:

1. efficient identification of objects that are directly addressed by application threads, to prevent their eviction/relocation;
2. selection of appropriate victims for eviction; and
3. mechanisms for reclaiming OC space, independently of any given victim selection policy.

5.2 Low-cost object pinning

A major problem is to detect the presence of direct pointers into the OC to avoid creating dangling pointers when evicting objects. As explained in section 3.1.2, threads may be suspended with some direct pointers to (possibly persistent) objects. Dangling pointers will be created if such objects are flushed from the OC.

A putative solution is to bracket the use of direct pointers with explicit operations to signal the pinning and unpinning of objects. Though explicit pinning may save some residency checks, it is extremely expensive to use because of the per-object storage required for recording the number of pins. Moreover, to be beneficial, a pin/unpin strategy requires compiler support to plant pin and unpin operations[17]. Without compiler support, it is extremely hard to bracket safely more than one JVM instruction with pin/unpin calls.

Another solution is to scan the native stack to detect direct pointers to persistent objects every time the cache manager needs to recover some storage. The tracing would conservatively mark each object in the OC reachable from the native stack of every thread. A similar solution is used by JDK's garbage collector to prevent moving directly addressed objects during compaction. In that case, the cost of scanning the native stacks is unavoidable because of the need to mark handles that are only reachable from native stacks.

Scanning all native stacks is a method of identifying directly addressed objects which does not require explicit action by application threads. Although scanning may be tolerable for garbage collection, it is too expensive for object replacement. This is because garbage collection reclaims as much space as possible at each invocation, reducing the need for further garbage collection. In contrast, cache replacement tries to replace as few objects as possible in order to reduce the number of subsequent cache misses so that cache replacement occurs frequently, thereby making the cumulative cost of scanning prohibitively expensive.

The solution devised for OPJVM consists of recording where on the native stacks there is a variable that may contain a pointer to a cached object's handle. Recording the location of the handle variables rather than the location of the direct pointers themselves is important for performance reasons, because direct pointers are short-lived and much more numerous than the handles from which they are obtained. This is partly because the JVM frequently uses local blocks of code that each declare local variables to store temporary pointers obtained from dereferenced handles. Furthermore, the heavy use of macro-definitions in the JDK JVM makes discovering direct pointer locations intractable. Recording the location of handles is more conservative than recording direct pointers but much more efficient.

For example, the main loop of the JDK bytecode interpreter uses only two handle variables, which are used as "virtual registers" to store the handle of the objects used by the current JVM instructions. The locations of these two variables need only be recorded once before running an instance of the interpreter loop, requiring only two handle-location recording operations per invocation of the interpreter loop. The alternative, if direct pointers themselves were tracked, would have been to bracket every JVM instruction that manipulates the contents of an object with code to register and de-register these direct pointers. The native methods of the core classes as well as the rest of the JVM have been changed correspondingly to record the location of all of their handle variables.

Figure 5 shows how directly accessed objects are identified. Each thread maintains a stack of handle-variable locations[18]. Each stack item is made of an array of handle-variable locations, the address of the

[17]This compiler must operate on classes after they have been loaded, in order not to violate persistence independence.
[18]The stack of locations is actually threaded onto the native stack to simplify memory management.

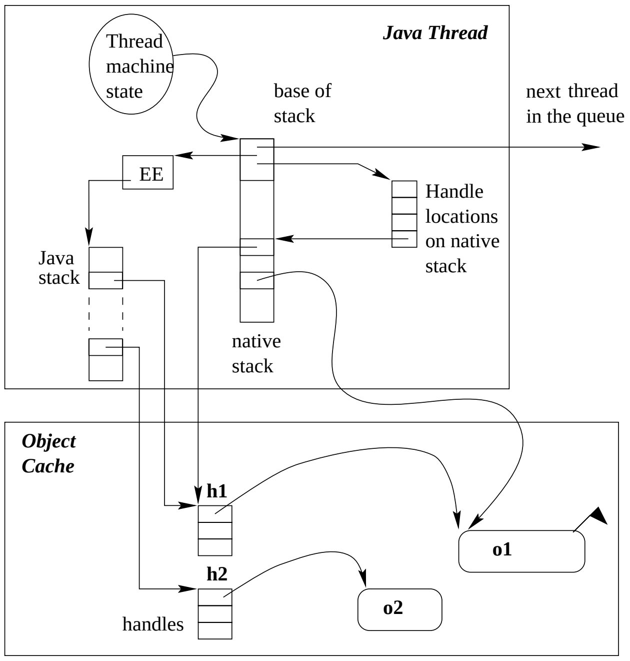


Figure 5: Low-cost object pinning.

native function that recorded these locations, and the address of the JVM program counter, if that native function is the bytecode interpreter.

Every time the OC manager needs to pin objects accessed directly by threads, it scans the queue of threads, and for each thread, it scans its stack of handle locations. For each handle location found, the OC manager checks if the handle refers to an object in the OC and if it does it pins that object. In addition, if the function that recorded the handle locations is an interpreter loop, the OC manager inspects the current JVM instruction to see if it manipulates an object. If it doesn't, then there is no need to pin the object since the content of these "virtual registers" is reloaded systematically at the start of JVM instructions that use them.

Overall, very few objects are ever pinned and most of the regions of the OC are left unpinned. We believe that the cost of pinning is negligible compared with that of explicit pin/unpin or conservative marking strategies.

5.3 Victim selection

The efficiency of cache replacement heavily depends on the correct selection of objects to reduce the number of cache misses. A common heuristic is to replace least recently used (or LRU) objects, on the premise that an object not recently used is unlikely to be used in the near future. LRU as well as other policies based on the notion of recent usage, require the ability to trap every access to the unit of replacement. A classification of the victim selection techniques is shown in figure 6.

Some persistent object-based programming systems benefit from strict encapsulation. If strict encapsulation is enforced, method invocations can be equated to object accesses, and cache replacement information can be obtained at method invocation time (see [13] for instance)[19]. Java does not enforce strict encapsulation. Indeed, it tends to promote direct accesses between instances of classes defined in the same package. As a result, victim selection cannot rely solely on method invocation; an application may traverse a dense graph of objects without invoking a single method. If method invocation were equated to access the traversed objects would appear unused and be (incorrectly) selected for eviction. On the other hand, trapping every object

[19]However, optimising compilers may use in-lining in such a way as to remove this encapsulation at the byte-code level.

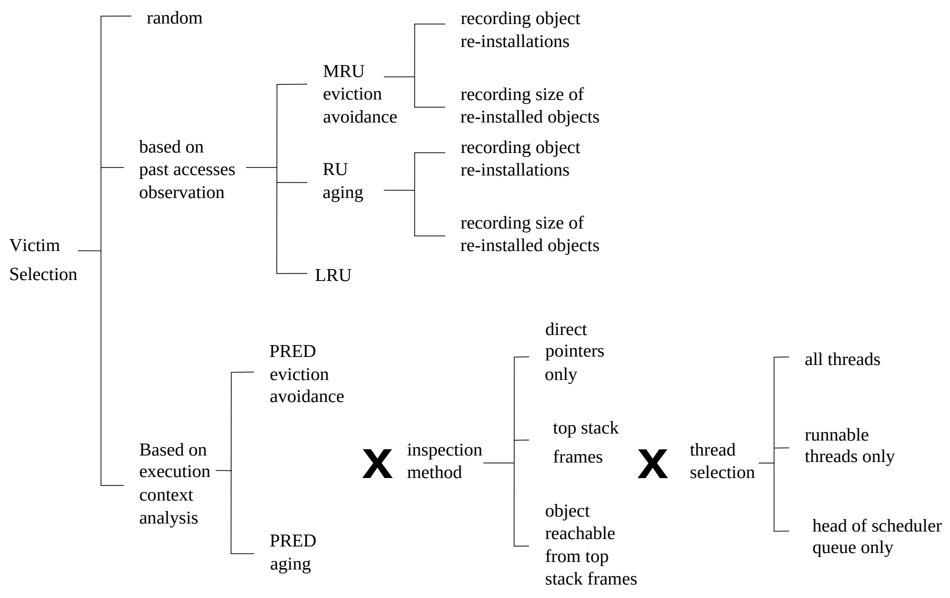

Victim Selection
random
based on past accesses observation
MRU eviction avoidance
recording object re-installations
recording size of re-installed objects
RU aging
recording object re-installations
recording size of re-installed objects
LRU
Based on execution context analysis
PRED eviction avoidance
PRED aging
X
inspection method
direct pointers only
top stack frames
object reachable from top stack frames
X
thread selection
all threads
runnable threads only
head of scheduler queue only

Figure 6: Summary of victim selection methods.

access to optimize victim selection is prohibitive, particularly as studying the past does not yield oracular foresight.

A possible solution is to provide the ability to turn on the recording of accesses to OC-resident objects. The trick is to make the application threads pay for this *only* when such monitoring is necessary. Another solution is to periodically estimate object usage based on execution context analysis.

Another parameter to consider when selecting victims is the granularity of eviction. Evicting individual objects is not efficient because it leads to fragmentation of the OC space. A better solution is to recover a whole region, perhaps at the price of movingm (within main memory) some of the region's objects to avoid OC misses. The victim selection must therefore identify the best candidate region for recycling as well as which objects in that region must be retained. Ideally, the OC manager will look for *quiescent regions*, i.e., regions without any objects to be retained, in order to avoid any copying cost for recycling a region.

We report two methods to maintain information efficiently for victim selection within the constraints just described. The first allows us to record individual accesses to OC-resident objects. The second periodically estimates current and future accesses to OC-resident objects. The former is more precise but more expensive, while the latter may be less accurate but cheaper.

5.3.1 Switchable detection of accesses to OC-resident objects

Detecting accesses to main-memory resident objects is the basis for all object-replacement policies based on past use of objects. Access to OC-resident objects may be easily and efficiently detected using a slightly modified version of the object-faulting mechanism described in section 4. The idea is to turn on false failures of the OC-residency check and to record the accesses to these objects in the resultant *minor*-object-fault trap. There are no extra costs when these traps are switched off, nor after the first access.

To provoke a failure of the residency check for an OC-resident object, the OC manager *hides* that object. Hiding only applies to ROThandles which have an extra field in their prefix holding the PID and consists of exchanging, within such a handle, the object's address in the OC with this PID, which makes a hidden object look like a non-resident object, so that the next residency check will fail. It also eases the resurrection of hidden objects upon a minor-object-fault, since the actual location of a hidden object in the OC is still contained in its handle. Thus handling a minor-object-fault just requires acquisition of the latch of this handle, and re-exchange of two fields in the handle, hence the qualification “minor”. The various formats of a ROThandle, as well as the transitions from one format to another, are illustrated in figure 7.

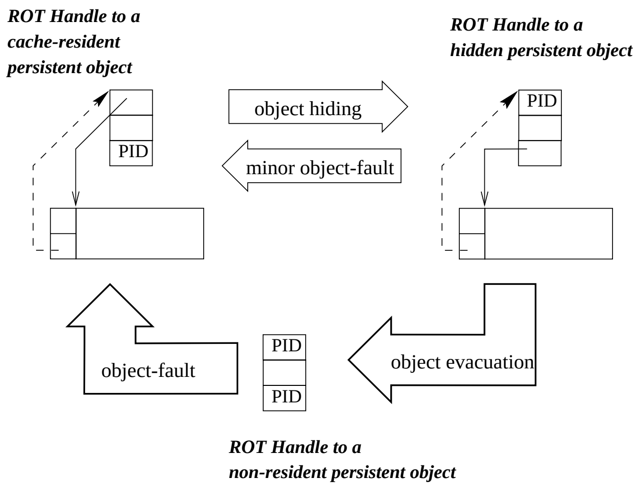

ROT Handle to a cache-resident persistent object
object hiding
minor object-fault
ROT Handle to a hidden persistent object
PID
PID
object-fault
PID
PID
object evacuation
ROT Handle to a non-resident persistent object

Figure 7: Formats for handle to persistent objects, and transition between the various format.

Object hiding and minor-object-faults can be used to identify the set of objects used within a given time frame, and ultimately to obtain a good estimation of the LRU objects. The idea is to *hide* all OC-resident objects and to record, within minor-object-fault traps, information concerning the faulted object. The recording method depends on storage and processing trade-offs for a given victim selection policy. The delay between hiding all of the objects and the moment when information about resurrected objects is processed is called the *usage window*. It may be expressed as a time interval or as some volume measure of objects resurrected.

One way to estimate LRU objects is to wait until only a small fraction of objects are still hidden. The recording method in this case consists of keeping track of the volume of objects resurrected until the desired fraction of objects remain hidden. These remaining objects define a good approximation of LRU objects. A time-out is needed to deal with the case when the set of completely disused objects is larger than the target fraction.

Another possibility is to identify *most recently used objects (MRU)* to inhibit their eviction. These are detected by using a short usage window. Too frequent application of object hiding incurs unacceptable minor-object-fault processing costs.

5.3.2 Periodic estimation of object usage

Detecting access to individual objects may become too expensive. An alternative is to use a background thread to estimate periodically the *age* of the population of resident objects. The age of a unit of memory, be it an object, a page or a whole region, defines how recently that unit has been used. *Aging* is the activity of updating periodically the age of these units according to whether these units have been accessed during the last aging cycle.

Given that it is known which access units have been accessed at the end of each period, aging can be implemented efficiently using a small counter [31] associated with each unit of memory. This is illustrated in figure 8. The smaller the counter value, the less recently used is its unit. Each time an aging period ends, the counter of every unit is right-shifted by 1 and the higher bit of the counter is set to 1 if the corresponding unit has been accessed during the period.

Treating objects as the unit of aging is too expensive. A more appropriate unit is the region. The main problem is then to detect which regions have been accessed during an aging cycle. This could be done using the technique described in the previous section, but it may be prohibitive to hide all of the objects each cycle. Another solution would be to use virtual memory protection to trap the first access to each region, but it was not investigated because of the complexity of handling signals in the JDK JVM and because on

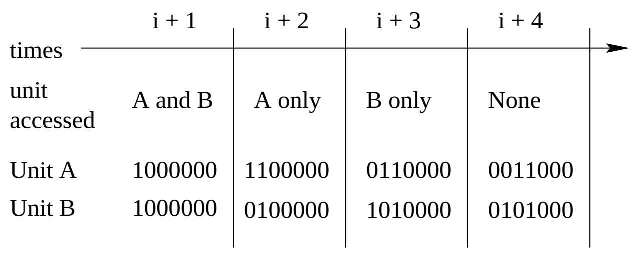


Figure 8: Aging.

many stock operating systems, signal handling and system calls for changing access right to ranges of virtual memory have prooved to be too prohibitive (see [20] for instance) to be used at a small granularity.

5.3.3 Access prediction

The execution context of each thread may be analysed to predict future accesses to objects to assess their eviction vulnerability and hence that of their containing regions. Access prediction may rely on:

- *the direct pointers from native stacks.* Direct pointers are the most precise indicator of which objects have been or will be manipulated by threads. However, it is hard to estimate when their further use will take place. They may never be used again, e.g. the code has finished with a variable but has not yet retracted the stack.
- *the content of the Java stack frames.* Java stack frames may be used to predict the future accesses to the OC by threads but cannot be as precise as direct pointers for two reasons. The location within Java stack frames is not available so conservative methods are required. Handle values may be used only as a left value for an assignment and their referends never accessed. They also suffer from the same unpredictability of re-use as variables do.
- *the position of threads in the scheduler's queue.* OC access predictions for a given thread should provide an estimate of the next time this thread will be executed.

Direct pointers on native stacks or putative handles on Java stacks may suggest either retention or eviction depending on their location. Locations close to the top of a stack suggest retention, while locations near the bottom of stacks *may* suggest eviction. Similarly, objects referenced only from the stacks of a suspended thread well down the process list may be deemed vulnerable to eviction. These are only heuristics, code may be about to retract the stack, a thread may be about to die. Their efficacy can only be established by comparing their performance under real loads, synthetic loads may have a misleading usage patterns.

When comparisons are made among these various mechanisms for selecting eviction victims, it is important to compare performance with random eviction which has no data collection costs.

5.4 OC space reclamation

The unit of OC space reclamation is the region and region managers are required to return whole regions on demand. OC space reclamation is started asynchronously when the free space reaches the low-water mark, and stopped as soon as the amount of free space hits the high-water mark. OC space reclamation is independent of the victim selection policy used. The only requirement is that regions must be sorted with respect to their object populations. Regions are classified as follows:

- *pinned regions*: regions that contain pinned objects, i.e. objects that *appear* to be directly referenced;
- *dirty regions*: regions that contain updated, uncommitted objects which cannot be reclaimed because of the no-steal policy;

- *active regions*: regions that contain “clean” objects that should not be evicted, i.e. the victim-selection algorithm has identified recent or expected activity on objects; and
- *quiescent regions*: regions that contain only objects that are candidates for eviction.

Pinned regions cannot be reclaimed; to do so would create dangling pointers. In extremis, if all regions are pinned and some space must be recycled because the OC is full, then the boundaries of some regions may be re-defined so that one of the redefined regions has no pinned objects. The first step in reclaiming an active region is to move its active objects to another region. To minimize copying costs, active regions with the minimum population of active objects are primary candidates for eviction after quiescent regions.

Selection of victim regions and their reclamation is done in single-thread mode to avoid any resurrection of objects during that process. Objects in non-pinned regions are guaranted not to be operated on at all by any thread. In particular, no objects of non-pinned regions can be involved in residency checks or an object-fault trap. Hence, no particular precaution is required to reclaim or compact non-pinned regions.

Figure 9 illustrates how quiescent and non-quiescent (i.e. dirty or recently used) regions are recycled. Evicting an object from the OC just consists of changing its handle into a fault-block format. Thus, reclaiming a quiescent region just consists of scanning the region linearly. For each object of the region, the handle of that object is changed to a fault-block.

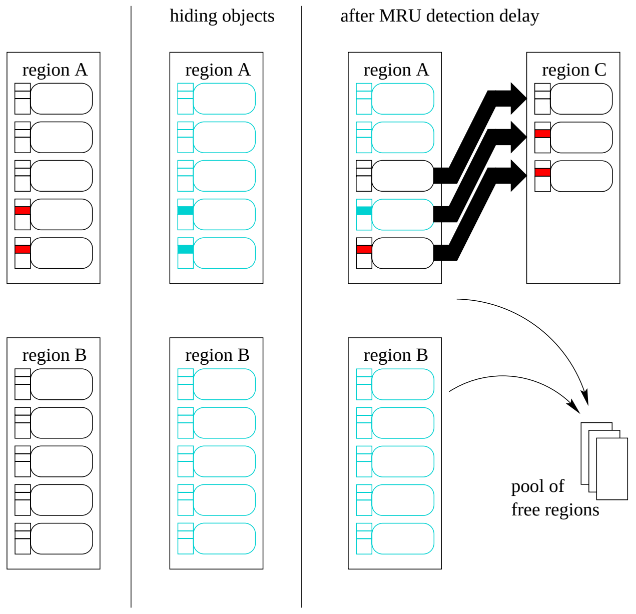


Figure 9: Cache space compaction.

Though reclaiming an active region is more expensive than reclaiming a quiescent one because of the copying costs, it may be useful to improve the locality and avoid expensive OC misses. One can decide to do compaction rather than evict a quiescent region: each region may be tagged with a timestamp of the last time that region has been recycled (the timestamp is essentially a count of the number of OC replacements). Non-quiescent regions with a high timestamp and a small active object population would be chosen first.

When a non-quiescent region is reclaimed, refugee objects are moved to other regions used for compaction. By default, only mutated objects become refugees but the victim selection may identify additional refugees. A possible extension of this scheme is to save families of refugees by including objects that are reachable from the identified refugees if they are also OC-resident.

5.5 Cache replacement strategies

All OC-replacement strategies comply with the following principles:

- OC replacement is performed asynchronously, ahead of OC overflow;
- current allocation regions are exempt from OC replacement;
- the space-reclamation algorithm, described in section 5.1, is used for all replacement of object regions; and
- all policies use the same mechanism to identify pinned and dirty regions (A region is pinned if there are some objects in that region that may be directly used by at least one thread (see 5.2). Similarly a region is dirty if if it holds an updated object.).

5.5.1 Random eviction

Random eviction is the simplest policy and imposes no overhead on applications. It works as follows. When the low-water mark for free space is reached, the OC manager identifies the pinned regions. The OC manager then chooses randomly from just the quiescent regions or, in the hope of attaining better locality, from both the quiescent and dirty regions.

5.5.2 MRU eviction avoidance

MRU eviction avoidance tries to improve over random selection by not evicting objects used within a short interval after the request for OC-space reclamation, on the premise that if they have been used in that interval then they will soon be used again. When the object-region manager is asked to free some space, it initiates the detection of MRU objects by hiding all of the objects in its regions (except the current allocation regions) and waits for a short delay which is long enough for the running application threads to resurrect the working set of objects. When the delay expires, the object-region manager classifies the regions as for the random strategy, except that now, the space used by MRU objects will not be reclaimed. If a region with MRU objects, i.e. with handles in indirection format, is selected for eviction, they become refugees.

6 Related work

Many object-oriented DBMS and persistent programming languages are based on the dual buffer architecture in which an object buffer functions on top of a page buffer. Commercial examples of this architecture are Itasca, Ontos and Versant. Dual buffering is advocated to improve space utilization by filtering out useless objects from the page buffer, thus limiting the dependence of performance on good clustering. This is significant because a large proportion of objects are very much smaller than a page.

However, the difficult issues of discovering all of the direct pointers to buffered objects and realizing efficient object replacement algorithms, such as LRU, at an object granularity have lead many implementors of dual-buffer architecture to cache objects in virtual memory without object replacement, and to leave the responsibility of memory management and swap I/O to the underlying operating systems. That is, all objects are kept in memory until a transaction completes. Example of such design includes Objectivity/DB, Versant, Mneme [26], all versions of Exodus EPVM [32], and Shore [10].

Performance when accessing objects in the dual buffer architecture primarily depends on the object buffer hit ratio. Consequently, given the flush-at-end-of-transaction strategy chosen by many systems, researchers have concentrated on efficient object prefetching to improve object access performance [1]. Exodus EPVM 2.0., ENCORE [19] have adopted an *eager prefetching* strategy where all objects of a buffered page are copied into the object buffer upon the first occurence of an object fault against that page. More sophisticated prefetching policies, based either on semantic knowledge about the structure and operation on complex object structures [11], or on profiling information [28], have also been reported.

Flushing the entire object cache at the end of a transaction, even augmented with good prefetching policies, is not applicable to PJama for two reasons. First, because all threads concurrently access the object cache and their execution contexts may include several direct pointers to OC-resident objects. Such direct pointers must be maintained across stabilizations. This problem will be exacerbated in future versions of PJama which will incorporate the ability to execute an arbitrary number of concurrent, potentially multi-threaded, transactions. Then, objects shared between several transactions cannot be flushed because one of these transaction completes.

Second, PJama intends to support long-lived, uninterruptible servers which will access, over a relatively long period of time, a volume of objects well in excess of the available virtual memory. Even when the volume of accessed objects remains within the capacity of the virtual memory, filling the virtual memory beyond the capacity of physical memory will degrade performance significantly. Page-thrashing is expected to be high compared with an object cache's traffic because the unit of transfer is so much larger than the unit of use (the objects) and there is no way to collect just the objects that are used without the clutter of others on each page. With the ability for active objects to gather into a region as refugees (described above) and with the ability to evict inactive objects, the density of useful data should be maintained, so that object-thrashing is much less probable.

A possible solution to recover some OC space is to extend the existing garbage collector so that it reclaims space in the OC. This solution has been adopted in persistent systems using a local heap architecture. Local heap differs from dual buffer architecture in that new objects are intermingled with cached persistent objects. That is, the local heap is used for allocating space for both new objects and copies of persistent object obtained from the page buffer pool.

PS-algol used a modification of the Lockwood Morris algorithm to evict immutable objects such as strings [9]. Napier88's garbage collector forced a stabilization and eviction of stabilized and then transient objects if it was unable to find sufficient space [8, 2, 27].

The major disadvantage of garbage collection techniques is that they rely only on unreachability to identify recoverable cache space. If a pure garbage collector were used to free OC space, the OC would block permanently when all cached objects were reachable. This is likely to happen in a persistent system as the first cached object is the root of all persistent objects and is usually referenced from an effectively global variable causing all persistent objects to remain live. Hence, if no mutation of the graph of cached objects occurs, the cache will be full of reachable objects, but the application may be attempting a traversal of the whole persistent graph. Extensions to garbage collection techniques may be devised such that reachability is interrupted to circumvent this impasse. Finding appropriate points at which to make these cuts may well be closely related to the problem of victim selection and susceptable to similar techniques. We review the only work we are aware of on combining such heuristics with garbage collection below.

LOOM [21] extend the original garbage collector of the language Smalltalk to recover space in the object cache. LOOM's garbage collector combines reference counting with object contraction: objects are contracted into surrogates called *leaves* when no accesses to these objects have been detected during some period. To detect accesses to an object, LOOM uses a standard clock algorithm: each resident object is given an "untouched" bit. Everytime a field of a resident objects is accessed, the untouched bit is cleared, denoting an access. Whenever some space is needed, the memory is swept and the untouched bit of every object is set. Any object found with an untouched bit still set in an entire pass through memory is a candidate for contraction. Space taken by leaves is recovered using reference counting garbage collection. This scheme makes cache-space compaction very difficult and leads to fragmentation and hence poor allocation performance. Furthermore, the straightforward application of the clock algorithm at the object granularity has a high cost per object access and was rejected in PJama for this reason.

A combination of garbage collection and *object shrinking* very similar to LOOM has been used for Thor's object cache [13]. This combination is dictated by the Thor's use of direct swizzling: eviction of objects requires that they be shrunk into smaller surrogates in order to prevent the creation of a dangling pointer. Pure shrinking would lead to a cache becoming blocked because of the high degree of fragmentation it generates. Garbage collection is therefore needed to relocate surrogates, fix up pointers to relocated surrogates, and compact the cache to reduce fragmentation. The behavior achieved by this shrink and garbage collect strategy is similar to that which we would have obtained by intermixing ROThandles and objects in PJama's

OC. Separating these two categories as we did avoided having to resort to expensive garbage collections for recycling object space.

The selection of objects to shrink is driven by some victim selection policy. Day compares a random victim selection policy to a LRU one and concludes that LRU is superior. However, his implementation of LRU assumed that method invocations can be equated with object accesses and this allowed him to pay the overhead of timestamping objects at method invocation only. This approach is of little use for Java because of the lack of strict encapsulation. Furthermore, his study assumes a single thread of control, whereas the object cache is shared by all threads in PJama.

Our work can be compared with the two other research attempts to make Java orthogonally persistent.

TJava [16] also provides orthogonal persistence but it differs significantly from PJama in that it targets main-memory resident database applications only and hence does not face a cache management problem. TJava also used RVM for recovery purposes, but it also relies on it to implement persistence. Basically, the entire database is mapped in one go into virtual memory at bootstrap time. This has to be contrasted with our use of RVM, where pages are mapped on demand in a buffer, and unmapped when the PBP runs out of space. The decision by the TJava team to change the Java language means that code re-use and persistence independence have effectively been abandoned.

Dearle's group has used its orthogonally persistent operating system, Grasshopper, to provide a platform for Java [15]. Grasshopper does not presume any language support while providing non-intrusive support for persistence, i.e., persistent data may be manipulated without any addition to the original code that manipulated them. This model generalizes PJama's persistence independence principle to arbitrary programming languages. In order to achieve this level of independence, persistence is implemented at the operating system level using standard stock hardware support for virtual memory. Because of this reliance on the virtual-memory hardware, Grasshopper realizes persistence at the page granularity. Accesses and updates to memory resident persistent pages are detected by the virtual memory hardware, and standard operating system techniques can be used to realize page faulting and replacement policies. Though elegant and promising in terms of performance, this approach does not address the crucial need for orthogonal persistence on stock operating systems and therefore does not meet Java's goal of platform independence.

7 Conclusion and future work

We've reported on the integration of a shared, no-steal, dual-buffer technology into a Java Virtual Machine in order to support orthogonal persistence. The advantages of the resulting architecture are:

- separation of the persistent store manager from the details of the Java Virtual machine (JVM);
- provision of a computational space of objects that looks the same to the JVM as the standard space of objects on the GC heap; and
- improved working-set behaviour because the average Java object is much less than a page size or disk transfer unit as it is between 40 and 100 bytes (the mode is even smaller) for typical applications.

The disadvantage of this architecture is the cost of copying between the OC and PBP. The architecture has been implemented and is in use as part of the orthogonally persistent JVM of PJama$_0$. Experience has shown this operating reasonably well for a variety of applications and synthesized loads. For example, one application performed 2000 stabilizations, each adding about 200 objects with an average size of 60 bytes, and performing updates to sumchecks, counts, etc. This was followed in the same execution by three complete traversals of the data (360 MBs) doing various computations, involving accesses to 6 million objects at each pass. This ran on a 16 MBs cache with a 4 MB buffer pool.

A cache observer component has permitted the various behaviours of the cache population to be visualized and studied. One demonstration we have allows: map drawing, interruptable pi calculations, multi-threaded simulations of concurrent bibliographic search, and 007 benchmarks, to be run simultaneously or in succession on the same object cache. The cache observer reveals the redistribution of partitions to different regimes as the balance of this load varies.

We expect to have measurements of some of the alternatives at the workshop and the important results will be included in the final paper.

A major challenge in building this system has been understanding and coping with the complexities of a JVM which was not designed or implemented with persistence in mind. These are presented in the paper for two reasons: as an aid for anyone who has to tread a similar path, and as a record of what we consider to be an almost inevitable part of retrospectively engineering an orthogonally persistent store for an industrial strength language implementation. Some of the solutions to detailed problems that are presented may be specific to the JVM we used but suggest strategies for overcoming similar problems. The generic solutions, such as the double-check residency algorithm and the compromise of tracking handle variables will certainly be useful in other contexts.

A particular interest in the paper are the techniques that may be used for victim selection in the OC and hence of victim pages in the PBP. A variety of techniques, including random selection, approximations to LRU and analysis of the JVM state to estimate future referends have all been presented. In most cases there are a number of parameters and variants to consider. Our future work will include analysis and measurement to better understand this design space which we consider important for continuous operation and long-running transactions.

Our work will include extensions to yield acceptable behaviour during bulk loading (where most objects are not revisited) and during continuous running.

In the longer term we expect to operate with a different persistent object store [29] which will free us from the yoke of the no-steal policy, use different promotion algorithms and have multiple management regimes. We will investigate whether the disadvantage of copying between the PBP and the OC can be ameliorated by a hybrid cache management strategy, such as that sketched earlier. We will then investigate the interaction between more general transaction management schemes and the cache management. We will probably have to deal with significantly changed JVMs by that time, including handle-less JVMs, but perhaps with better information about pointers. We also hope to deploy code-optimization techniques in conjunction with this architecture [12].

Acknowledgements

The research has been conducted at Glasgow University with sponsorship from Sun Microsystems Inc. and from the British Engineering and Physical Sciences Research Council, grant number GR/K87791. Our colleagues Susan Spence and Tony Printezis built complementary parts of the system and contributed to design discussions. Our sponsor Mick Jordan, and our students Robin Scott, Alisdair McLean and Martin Spiers and our colleagues Cathy Waite and Stewart Macneill all produced demanding test loads which drove the work. Mick Jordan, Bernd Mathiske and Kathy Zuckerman all provided useful comments on a draft of the paper.

References

[1] J. Ahn and H. Kim. SEOF: An Adaptable Object Prefetch Policy for Object-Oriented Database Systems. In *Proc. of the 13th Int. Conf. on Data Engineering*, pages 4–13, Birmingham, UK, April 1997.

[2] A.L. Brown and G. Mainetto and F. Matthes and R. Müller and D.J. McNally. An open system architecture for a persistent object store. In *Proc. 25th International Conference on Systems Sciences*, pages 766–776, Hawaii, USA, January 1992.

[3] M.P. Atkinson, V. Benzaken, and D. Maier, editors. *Persistent Object Systems (Proc. of the Sixth Int. Workshop on Persistent Object Systems)*, Workshops in Computing, Tarascon, Provence, France, September 1994. Springer-Verlag in collaboration with the British Computer Society.

[4] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java^{TM}. *SIGMOD RECORD*, 25(4), December 1996.

[5] M.P. Atkinson and M.J. Jordan, editors. *First International Workshop on Persistence and Java*, Sun Microsystems Laboratories M/S 29/01, 2550 Garcia Avenue, Mountain View, CA 94043, April 1997. Sunlabs Technical Report SMLI-TR-96-58.

[6] M.P. Atkinson, M.J. Jordan, L. Daynès, and S. Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system, May 1996. In the pre-proceedings of the 7th International Workshop on Persistent Object System (POS 7).

[7] M.P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[8] A.L. Brown and R. Morrison. A Generic Persistent Object Store. *Software Engineering Journal*, 7(2):161–168, 1992.

[9] R.F. Bruynooghe, J.M. Parker, and J.S. Rowles. Pss: A system for process enactment. In *Proc. 1st International Conference on the Software Process: Manufacturing Complex Systems*, 1991.

[10] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 383–394, 1994.

[11] E.E. Chang and R. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an oject-oriented dbms. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, May 1989.

[12] Q. Cutts and A. Hosking. Optimization. In *(submitted to) second International Workshop on Persistence and Java (PJW2), Half Moon Bay, CA, USA Aug. 1997*, 1997.

[13] M.S. Day. *Client Cache Management in a Distributed Object Database*. PhD thesis, Massachussets Institute of Technology, February 1995.

[14] L. Daynès. *Conception et réalisation de mécanismes flexibles de verrouillage adaptés aux SGBDO client-serveur*. PhD thesis, Université Pierre et Marie Curie (Paris VI – Jussieu), 1995.

[15] A. Dearle, D. Hulse, and A. Farkas. Persistent Operating System Support for Java. In Atkinson and Jordan [5]. Sunlabs Technical Report SMLI-TR-96-58.

[16] A. Garthwaite and S. Nettles. Transactions for Java. In Atkinson and Jordan [5]. Sunlabs Technical Report SMLI-TR-96-58.

[17] J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification*. The Java™ Series. Addison Wesley, December 1996.

[18] J. Gray and A. Reuter. *Transaction Processing : Concept and Techniques*. Morgan-Kaufman, 1993.

[19] M.F. Hornick and S.B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1):70–95, January 1987.

[20] A.L. Hosking and J.E.B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 106–119, 1993.

[21] T. Kaehler and G. Krasner. LOOM - Large Object-Oriented Memory for Smalltalk-80 Systems. In *Smalltalk-80: Bits of History, Words of Advice*, chapter 14, pages 251–270. Addison-Wesley, 1983.

[22] A. Kemper and D. Kossmann. Dual-Buffering Strategies in Object Bases. In *Proc. of the 20th Int. Conf. on Very Large Database*, pages 427–438, Santiago, Chile, September 1994.

[23] A. Kemper and D. Kossmann. Adaptable pointer swizzling stategies in object bases: Design, realization, and quantitative analysis. *VLDB Journal*, 4(3), 1995.

[24] W. Kim, N. Ballou, H. Chou, J.F. Garza, and D. Woelk. Integrating an Object-Oriented Programming System with a Database System. In *OOPSLA '88 Int. Conf.*, pages 142–152, September 1988.

[25] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification.* The Java^{TM} Series. Addison Wesley, September 1996.

[26] J.E.B. Moss. Working With Objects: To Swizzle or Not to Swizzle? *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[27] D.S. Munro, R.C.H. Connor, R. Morrison, S. Scheuerl, and D.W. Stemple. Concurrent shadow paging in the Flask architecture. In Atkinson et al. [3], pages 16–42.

[28] M. Palmer and S.B. Zdonik. Fido: A cache that learns to fetch. In *Proc. of the 17th Int. Conf. on Very Large Database*, Barcelona, September 1991.

[29] T. Printezis, M.P. Atkinson, L. Daynès, S. Spence, and P.J. Bailey. The Architecture of a new Persistent Object Store for PJama: an orthogonally persistent platform for Java. In *(submitted to) second International Workshop on Persistence and Java (PJW2), Half Moon Bay, CA, USA Aug. 1997*, 1997.

[30] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J.J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(2):165–172, May 1994.

[31] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall Int. Editions, 1992.

[32] S. White and D. DeWitt. A Performance Study of Alternative Object Faulting in Pointer Swizzling Strategies. In *Proc. of the 18th Int. Conf. on Very Large Database*, pages 419–431, Vancouver, British Columbia, Canada, 1992.

[33] J. M. Wing, M. Faehndrich, N. Haines, K. Kietze, D. Kindred, J.G. Morisset, and S. M. Nettles. Venari/ML Interfaces and Examples. Technical Report CMU-CS-93-123, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1993.

[34] B. Zorn. The measured cost of conservative garbage collection. *IEEE Software, Practice and Engineering*, 23(7):733–756, July 1993.

The Design of a new Persistent Object Store for PJama

Tony Printezis, Malcolm Atkinson[*], Laurent Daynès,
Susan Spence, and Pete Bailey
{tony,mpa,laurent,susan,pete}@dcs.gla.ac.uk

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
Scotland

June 1997

Abstract

This paper presents the design of a new store layer for PJama. PJama is a platform that provides orthogonal persistence for Java[1]. Based on experience with a prototype, $PJama_0$, a new architecture has been devised to permit incremental store management and to allow a number of object management regimes to co-exist in one store. It uses a plug-in model for composing a Java Virtual Machine (JVM) with the persistent store and a descriptor abstraction to limit the impact of changes in JVMs on store management. Its anticipated advantages over the current scheme include flexibility, adaptability, scalability, and maintainability.

1 Introduction

The PJama project is a collaboration between Malcolm Atkinson's team at the University of Glasgow and Mick Jordan's team at Sun Microsystems Laboratories and is attempting to demonstrate the benefits of an industrial-strength, orthogonally persistent programming language [7]. Opportunistically and for technical reasons we have chosen to build an execution platform and additional class libraries that provide orthogonal persistence for Java [1, 14].

[*] Malcolm Atkinson is currently on leave as a visiting professor at Sun Microsystems Laboratories, Mountain View, CA, USA.

[1] Sun, Java, and PJava are registered trademarks of Sun Microsystems Inc. in the USA and other countries.

The initial design of this platform has been reported [4, 6] and some early experiences with the first prototype, $PJama_0$, have been described [17]. Further experience with building and operating this prototype has suggested a refinement of the store architecture. The pressures for change are given here:

- A succession of ports of our technology between different versions of the JVM [20], an activity that will not diminish, has shown the need for better insulation between the store management code and the JVM.
- At present, parts of our code are highly interrelated which makes maintenance and experimentation difficult. The same problem was discovered in PS-algol [3] and this led to a more modular design for its successor, Napier88 [21].
- A sophisticated model of cache management, with the potential for a variety of complementary management regimes in different regions, is now operational [13]. However, at present the persistent object store (POS) layer only operates one regime and so it is difficult to exploit this potential. The availability of multiple POS management regimes will allow tailored support for special objects, such as those required by multi-media applications.
- The recovery technology of our existing POS prevents us re-cycling cache space that contains updated objects [13]. This limits the amount of data that can be modified within one transaction.

- The present monolithic POS is not convenient for incremental algorithms, such as garbage collection or archiving. This places an upper limit on the size of the stores over which PJama$_0$ can operate.

For these reasons, and with some data from a year of operation, we set out to design a new architecture for our orthogonally persistent Java virtual machine (OPJVM) as a prelude to implementing the next version of PJama[2], PJama$_1$. This paper reports on the design of one part of that OPJVM, the *PJama Store Layer* (PJSL). This interfaces with the object-caching technology [13] and has descriptors to tell it the representations used by the supported JVM. The infinite variety of types of object that a POS may be asked to preserve are reduced to a small number of *kinds*. The objects are stored in *partitions* to allow incremental POS-management operations and each partition is under the control of a particular *regime*. The operations that the OPJVM uses to execute against a store are specialised by kind and regime.

All the issues presented in this paper are discussed in greater detail in a techinical report [25].

1.1 Design Goals

The primary aim of PJSL is to support the operation of PJama$_1$ when running real workloads. The typical workload makes long-running and complex use of highly structured data, such as that concerned with software construction [18]. Ultimately, we want concurrent access to the store to be organised as long running and flexible transactions. It is hoped that the design of PJSL is sufficiently general that it will service a wide range of applications and will be used to support various language implementations. Its flexibility should allow for a series of store implementation experiments.

The specific and immediate goals are:

- to support complete orthogonality, so that *any* object type can be accommodated, including instances of all classes and arrays whatever their size[3];

[2]PJama was formerly known as PJava, but that epithet has been trademarked by Sun to denote Personal Java.

[3]For engineering reasons the current upper bound for arrays is 2^{27} bytes.

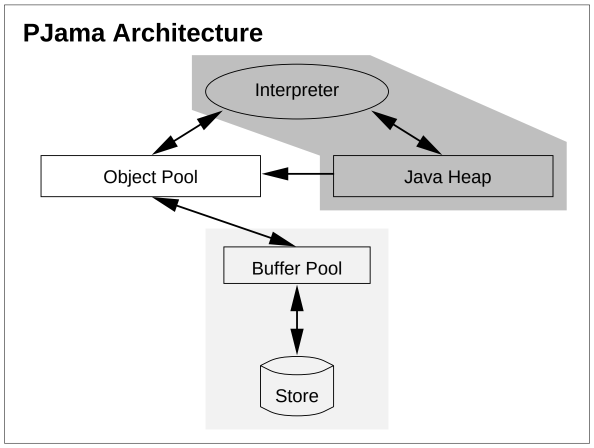


Figure 1: PJama Architecture.

- to accommodate at least 10GB of highly structured data, typically dominated by large numbers of small objects;
- to be capable of continuous operation with incremental algorithms for disk garbage collection, archiving, etc.;
- to be capable of running on a file system or on raw disk, with a minimum amount of operating-system dependent code; and
- to be appropriate for our planned developments, which are flexible and long-running transactions, schema evolution, archiving, and distribution.

The implementation will be biased towards complex computations that make repeated traversals over a sub-graph of the objects that includes a moderate proportion of the total population. However, the system must survive both total traversals and large bulk-loading operations. This must be achieved without requiring guidance from application programmers, otherwise persistence independence [7] will be lost.

1.2 PJama Architecture

This section briefly presents the current architecture of PJama$_0$, which is based on the JVM developed by Sun Microsystems[4]. In Figure 1, the darker of the shaded regions, which comprises the core of the

[4]Currently, the release of PJama$_0$ is based on JDK1.0.2. However, the port to JDK1.1.2 is close to completion.

interpreter and the *heap*, represents the original JVM. The interpreter allocates, modifies, and reads objects in from the heap.

In order for the JVM to support persistence, three new components were added to it. The *store* is kept on disk. It contains the persistent data and it is cached at the page level in the *buffer pool*. All persistent objects, before they can be accessed by the interpreter, are copied from the buffer pool into the *object pool* in a format similar to those in the heap. Because of this, the code used to operate over objects in the heap, can also operate over objects in the object pool with minimal changes.

New objects are still allocated in the heap. However, if they become persistent (by being rendered reachable from other persistent objects, according to the definition of *persistence by reachability* [7]), they are migrated to the object pool and are also copied to the store via the buffer pool. The operation which migrates them is called *promotion*. Finally, any updates to persistent objects are propagated from the object pool to the store, again via the buffer pool.

This paper will concentrate on the components bounded by the lighter of the shaded regions in Figure 1, namely the store and the buffer pool, which will be referred to as the PJama Store Layer or PJSL.

1.3 Paper Overview

Section 2 introduces partitions, kinds, and regimes and shows how the appropriate method of an operation is selected. Section 3 describes the internal layout of partitions, the format of persistent identifiers (PIDs), and the use of descriptors. Section 4 contains our initial views on disk and object space management. Finally, Sections 5 and 6 present, respectively, related work and conclusions.

2 Store Organisation

It has been decided that PJSL will adopt a *Partitioning Scheme* [32]. This means that the store will be split into smaller parts (partitions) so that each of them can be garbage collected independently[5]. This partitioning

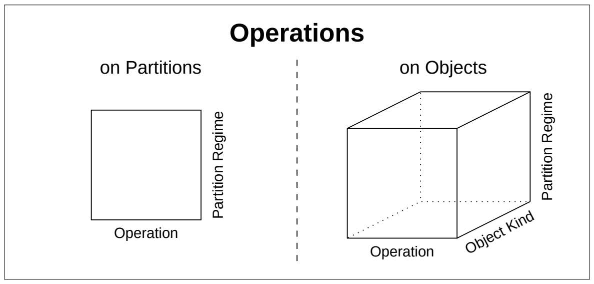


Figure 2: Operation Matrices.

scheme is considered to be the most efficient way to incrementally garbage collect large spaces, such as persistent object stores [11, 12, 23]. This view is also supported by recent experiments conducted by Printezis on garbage collecting small stores [24]. These experiments showed that the time needed to garbage collect stores of sizes between 27MB and 30MB varied from 3 secs to 43 secs, depending on the object kinds included and the degree of connectivity. It is obvious that, if these times were extrapolated to apply to a 10GB store (which is roughly 300 times larger than the sizes mentioned), the garbage collector will require a prohibitively long pause to process the entire store in a single operation.

Managing free-space inside a partition can be achieved in many ways: compaction, free-lists, etc. The combination of the free-space management scheme, along with some additional organisation parameters, will be referred to as the *Partition Regime*. Partitions of the same regime will have the same internal structure and will usually contain similar (in structure, size, behaviour, etc.) objects. One regime can be more appropriate than another for certain kinds of objects, therefore several regimes can co-exist in the same store, applied to different partitions. Based on this, operations on partitions can be organised in a two-dimensional array, indexed by the regime and operation (see Figure 2). This is similar to the single dispatch operation used in object-oriented languages to invoke a method on a given object [16].

Currently, objects in PJama can be divided into four different categories: class objects, instances, arrays, and bytecodes[6], each of which has a different internal struc-

[5]Other algorithms, such as class evolution reformatting, archiving, statistics gathering, etc. will also exploit this partition structure.

[6]These are the byte arrays holding the results of compiling methods to byte-coded instruction sequences.

ture. These categories will be referred to as *Object Kinds* or just *Kinds*. There are several operations defined on objects, some being the same for all kinds (e.g. move) and others requiring a different implementation for each kind (pointer identification, faulting-in, etc.). Further, it may be the case that some of these operations are regime-specific. So, in a similar manner to operations on partitions, operations on objects can be organised in a three-dimensional array, indexed by the object kind, regime, and operation (see Figure 2). Again, from an object-oriented point of view, this is a simple implementation of a double-dispatch operation [16].

2.1 Partition Regimes

Six regimes will be implemented in the first version of PJSL. Notice that here *small* objects are those which are small enough to fit into a single *Transfer Unit* (TU)[7]. In the same way, *large* objects are those which are larger than a single TU. The *initial* six regimes are listed here.

Small Arrays : scalar and object arrays.

Small Instances : instances of classes.

Class Objects & Bytecodes : all instances of class `Class`[8], i.e. all class objects and their bytecodes. Clustering bytecodes with their classes minimizes accesses to other partitions during class faulting.

Large Instances : instances of classes spanning TU boundaries[9].

Large Scalar Arrays : scalar arrays spanning TU boundaries.

Large Object Arrays : arrays of instances or arrays spanning TU boundaries.

Some of the reasons why partitions are organised in this way, which relate to the store organisation on which they are based (see Section 3), are presented below.

[7]A TU is the unit of transfer of data from the disk store to main memory. It is a similar concept to a page, however it is named differently to avoid confusion, since its size might not be the same as the page. In fact, different regimes might use TUs of different sizes.

[8]Strictly `java.lang.Class` but we omit the `java.lang.` where we believe it is easily understood.

[9]Assuming that the minimum TU size is 8KB, a large class instance would have over 1,000 non-static fields, which is extremely unusual. However, automatic generation of Java code (by program translators, user-interface builders, etc.) occasionally results in such classes.

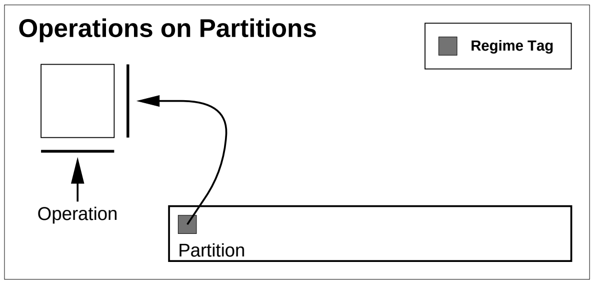


Figure 3: Invoking an Operation on a Partition.

- Separating small objects from large ones avoids many boundary checks upon object-faulting as they are unnecessary for small objects since they are guaranteed not to span multiple TUs.

- When partitions only contain arrays, they do not need to include descriptors and their management structures (see Section 3.6), as arrays have a compressed-type encoding in their header.

- Partitions containing only large scalar arrays[10] can be very large, since they do not need to be scanned to identify intra-partition references during garbage collection. Their reference counts (see Section 3.3) determine whether they are garbage.

It is worth mentioning here that the Mneme object store [22] established a notion similar to regimes. In Mneme, they are referred to as *pools* and can be managed independently, allowing object formats to vary implementing different buffer management. They even provide greater flexibility since it is up to the pool implementor to define their internal structure. This is not the case for the partition regimes of PJSL, which have to conform to the structure described in Section 3.2. This decision was taken as a compromise between flexibility and ease of implementing new regimes.

2.2 Invoking Operations on Partitions

Figure 3 shows how an operation on a partition is invoked. Each partition contains (in its header) a tag which determines its regime. This tag serves as an index into the two-dimensional operations matrix and, along with the operation index, yields the code for the desired operation. Then the code is executed, accepting as argument the partition ID.

[10]Commonly images, sound samples, and numeric data.

2.3 Object Kinds

The minimum set of object kinds required by PJama are as follows.

Class Objects : instances of class `Class` [14]. These require special implementations for the OPJVM bootstrap and for swizzling [13]. Each of them is an image of the `Classjava_lang_Class` C structure and of the other C structures that it points to: `constantpool`, `methodblocks`, `fieldblocks`, etc. [20].

Instances of any class, apart from `Class`.

Bytecodes pointed to from the `methodblocks` of the class objects [20]. These could have been represented as byte arrays, but they need to be handled differently.

Scalar Arrays : arrays of any scalar type.

Object Arrays : arrays of objects (either of instances or other arrays).

Descriptors : a kind defined for internal use by PJSL (see Section 3.6).

Scalar and object arrays are separated since the pointer identification operation on them is fundamentally different (returning either none or all of the array entries, respectively).

It is easy to introduce new object kinds and new operation implementations appropriate for them. This makes it possible to optimise the handling of some objects. Examples are presented below.

Strings : strings in Java (i.e. instances of the class `String`) are made up of two separate objects [14]. Since the space-overhead of an object in PJSL is 16 bytes (see the technical report [25] for more information on this), it might be more space-efficient to transform small strings into single objects when they are written to the PJSL and transform them back into Java memory format when they are faulted-in.

Compressed Objects : large scalar arrays might be compressed when moved onto disk to save transfer time and disk space. Examples are images, sound samples, etc.

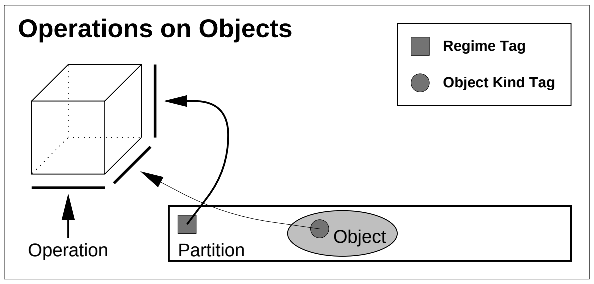


Figure 4: Invoking an Operation on an Object.

Stacks : stack objects which will be used when threads (i.e. instances of class `Thread` [14]) are allowed to be persistent.

Distribution Proxies will be needed to denote references to objects in remote stores [27].

2.4 Invoking Operations on Objects

Operations on objects are invoked in a similar fashion to operations on partitions. The regime tag and operation index are still needed, only this time a kind tag is also required. This is contained in the object's header and will serve as the third index in the three-dimensional operations matrix (see Figure 4). Once the code has been retrieved, it is executed with the partition and object IDs as arguments[11].

2.5 Clustering Considerations

It might seem that grouping objects in different partitions according to their kind, as mentioned above, would cause a high degree of declustering and hence a decrease in the performance of PJSL. However, this is not necessarily the case. Large data structures which typically need to be clustered together (linked lists, trees, etc.) tend to be constructed from only a few distinct types of object, usually instances of a few classes and arrays. Hence, even though the instances and arrays will be written to different partitions, as long as they are clustered close to each other within these partitions, the overall impact on performance will be low. It has also been observed that such data structures are usually larger in persistent systems than

[11]The PID of the object encodes or refers to all of this information (see Section 3.4) so it would suffice as the only argument, though then some decoding would be repeated.

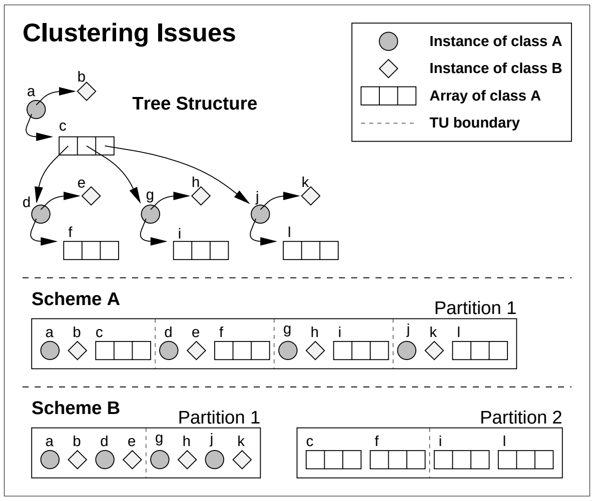


Figure 5: Clustering Issues.

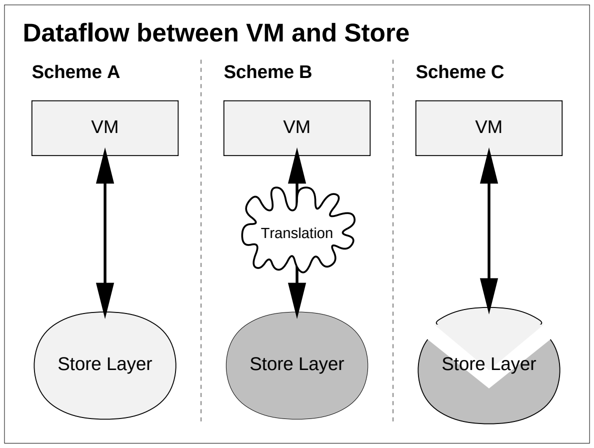


Figure 6: Dataflow between the Virtual Machine and the Store Layer.

in traditional ones [5].

A concrete example is given here. Consider the tree structure seen in Figure 5 and the way it will be copied to the store. According to scheme A, all objects are clustered in the same partition, irrespective of their kind. This keeps them close together and minimises disk accesses when the tree is traversed. However, object management within the partition is harder and less efficient, since it has to deal with objects of different structure, size, and behaviour.

Alternatively, according to scheme B, instances are separated from arrays, when copied to the store. However, objects of both kinds will be clustered close to each other within each partition. Object management within the partition is now more efficient because it only has to deal with objects of the same kind. Initially, when the tree is traversed, TUs from both partitions have to be read, making the startup cost more expensive than in scheme A. However, assuming that the entire tree structure is big enough not to fit in a single TU, this cost will be absorbed as the rest of the tree is traversed and more TUs are accessed.

In the example in Figure 5, when the first node of the tree, containing objects a, b, and c, is accessed, scheme A will touch one TU and scheme B two TUs. However, when the next node, containing objects d, e, and f, is accessed, scheme A will touch a new TU whereas scheme B will touch the same two TUs it previously touched, which are very likely to still be in the cache. Therefore, the initial cost of touching two TUs has already been absorbed. Obviously, this is a very specific example and the performance impact of either scheme is very application dependent. However, there will always be pathological cases for both of them.

As far as class objects are concerned, keeping them close to instances is not very important since they are typically faulted-in once per execution (assuming that they are not evicted from the object cache). Also, there will usually be a large number of instances of a given class and it will be impossible to cluster all of them close to the class object. It is more important to cluster the bytecodes close to their corresponding class object, since they are very likely to be faulted-in shortly after it. PJSL will in fact do this, as explained in Section 2.1.

2.6 Optimising Dataflow

There are several ways to arrange the flow of data between the store layer and the virtual machine. Figure 6 illustrates three of them:

- Scheme A assumes that the store has been written specifically for the given virtual machine, therefore the virtual machine talks to it directly. This offers the highest *potential* performance. However, the store code is not generic and it is very prone to change when the specification of the virtual machine changes.

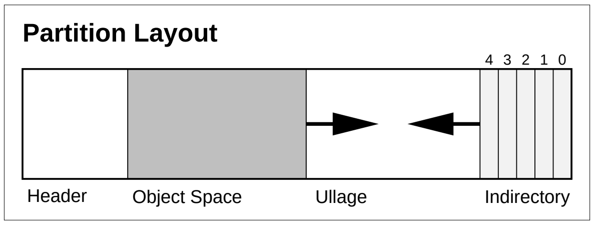


Figure 7: Partition Layout.

- In scheme B, the store layer is general-purpose and totally independent of the virtual machine. However, since it is very likely that the object format it supports is different from the one the virtual machine uses, an extra translation layer is introduced to cope with this. This has a negative impact on the performance of the system. However, the store layer code is totally independent from the layer above it and can be easily re-used with only the translation layer having to be re-written.

- Scheme C is the one which will be adopted in PJSL and has been proposed as a compromise between schemes A and B. The core of the store layer is generic, with only a set of well-specified operations (which define, among other things, the object format) having to be implemented specifically for the given virtual machine or application which uses the store directly. This way no translation layer intervenes to impact performance and the store can be adapted to and optimised for particular situations. However, the use of the store is not trivial, since the persistent programming language implementor has to write the plug-in operations contained in the two operation matrices described in Sections 2.2 and 2.4.

3 Partition Organisation

This section presents a brief discussion on how the partitions are going to be organised in PJSL. It is included here to give a feel for how the store will operate, what facilities will provide to the higher-levels, and what information will require from them. The contents of this section are discussed in greater detail in the technical report [25].

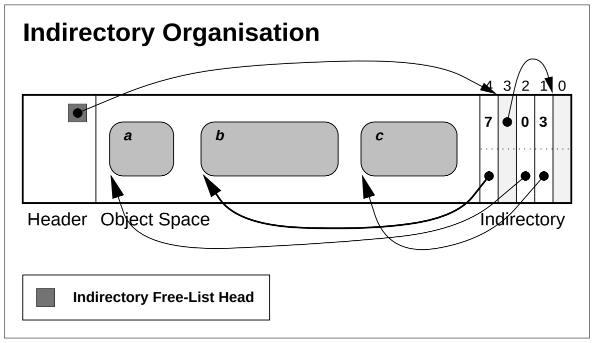


Figure 8: Organisation of the Indirectory.

3.1 Partition Identifiers

When a partition is created, it will be allocated an ID which will stay attached to that partition, until it becomes empty and is reclaimed (if this ever happens). This ID will be independent of the position of the partition within the store. This way, it is possible to easily move, resize, and garbage collect a partition without changing its ID and, therefore, any PIDs in objects in other partitions which point to it (see Section 3.4).

3.2 Partition Layout

Figure 7 illustrates how the three main components of a partition will be laid-out in the store.

Header : where the information describing a partition is stored.

Object Space : where objects are allocated. As its size increases, the object space grows forward in the partition.

Indirectory : where indirection entries, also containing reference counts, are stored (see Section 3.3). As its size increases, the indirectory grows backwards in the partition.

Ullage : free space on disc into which both the object-allocation front and indirectory grow.

3.3 Indirectory

An indirectory entry contains the following fields.

Object Offset : the offset of the corresponding object inside the partition from the start of this partition A 4-byte word is enough for this, as we believe it

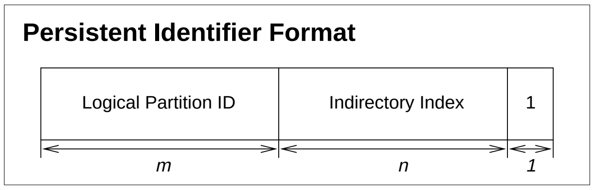


Figure 9: Persistent Identifier Format.

acceptable to limit the maximum partition size to 4GB.

Reference Count : the number of references to the corresponding object from objects in *other* partitions. A 4-byte word is sufficient for this as well, since it is unlikely that there will be more than 4 billion cross-partition references to a single object.

The use of the indirectory is illustrated in Figure 8. When an indirectory entry is allocated for an object, it keeps the same position inside the indirectory during the entire life-time of that object. If the object is moved inside the partition (due to compaction), only the object offset in its indirectory entry is updated.

Indirectory entries which have been freed (when their corresponding objects have been reclaimed) are linked together in a list called the *Indirectory Free-List*. The indirectory will grow only when this list is empty. It can also shrink, if a number of contiguous entries at its end have been freed.

3.4 PID Format

Figure 9 illustrates the format of the *Persistent Identifiers* (PIDs) in PJSL. The least-significant bit of a PID is always 1 to distinguish it from a memory address. In the current JVM, these are all 8-byte aligned, both for objects and handles, hence their least-significant bit is 0[12]. The remaining space is split between the partition ID and the index of the indirectory entry corresponding to the object.

Even though 31 bit addressing might sound inadequate, it must be made clear that in PJSL we address objects rather than data, since the indirectory index is used as part of the PID rather than the position of the object inside the partition. It turns out that 31 bits are enough

[12]Any JVM combined with PJSL will have to (or will be changed to) allocate its objects and handles so they are at least 2-byte aligned.

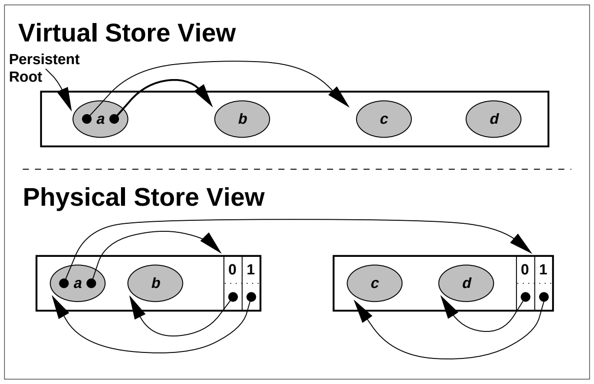


Figure 10: Virtual and Physical Store Views.

to address stores larger than 10GB (our target size), even after making pessimal assumptions about object sizes. A full proof of this is given in the technical report [25].

It is possible for PIDs to be exhausted within a partition, without the partition being full. This happens when there have been allocated 2^n objects in the partition without the object space having reached the indirectory space. In this case, the partition is considered to be full and, during the next garbage collection, an attempt will be made to decrease its overall size by contracting the ullage. Similarly, if the disk garbage collector detects that the ullage is nearly exhausted, but the PID availability isn't, it will attempt an overall expansion to increase the ullage.

3.5 Virtual Store View

Figure 10 shows the virtual view of the store that is presented to the layer above, typically the object-cache manager. The object-cache manager will specify the regime under which an object has to be stored and any subsequent updates to that object and the store layer will handle the rest: object allocation, reference count management, garbage collection, partition re-organisation, etc. These operations might occur synchronously (triggered by events such as updates, allocations, etc.) or (in later versions of PJSL) asynchronously, by daemons running in the background.

Another important point, illustrated in Figure 10, is that both intra-partition and cross-partition references will go via the indirectory. It would be possible to optimise

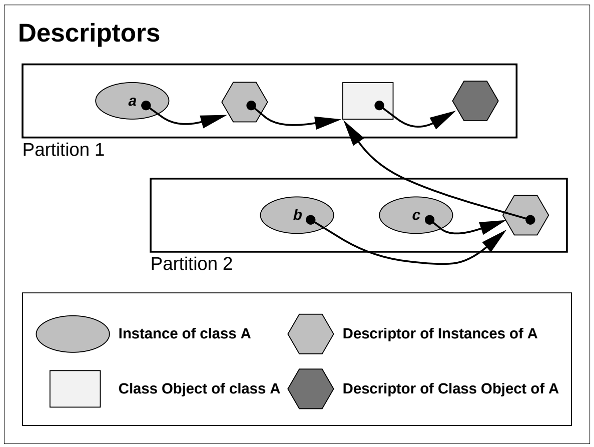


Figure 11: Use of Descriptors.

the intra-partition references to point directly to the object, since i) they do not affect the reference counts and ii) the indirectory will not need to be visited, avoiding a potential disk access. There are two reasons why this will not be done. The most important one is that by pointing directly to the object, it is not easy to deduce its PID since this requires the index of its indirectory entry (see the PID format in Section 3.4) and there is no efficient way to retrieve it from the offset of the object inside the partition. The second reason is a payoff during compaction since, if all references go via the indirectory, only the indirectory entries need to be updated, rather than all the intra-partition references in every object. This can accelerate significantly the compacting phase of the disk garbage collector, especially in highly inter-connected partitions [24].

3.6 Descriptors

It is important to be able to identify efficiently all pointers inside an object to speed-up the pointer swizzling / un-swizzling operations, the scanning phase of garbage collection, etc. Some language designers optimise the object format itself to facilitate this. For example, the pointers in all objects of Napier88 [8, 21] are grouped together at the beginning of the object and can be identified efficiently and uniformly. Unfortunately, this is not possible for PJama, since the object format used by the JVM does not guarantee this. To keep the implementation simple, uniform, and generic, a new scheme needs to be adopted to deal with this complication.

A *Descriptor* is a special object, introduced to abstract over the JVM's layout conventions, which contains information about the structure of all objects with the same internal structure (at least the position of pointers in them). Not all kinds of object need a descriptor, e.g. bytecodes and scalar arrays don't need one (there are no pointers in them) nor do object arrays (all their entries are pointers). However, pointers in instances and class objects intermingle with scalars and it is not trivial to identify them, hence descriptors need to be introduced for both of these object kinds. All instances of the same class can point to the same descriptor, since they have the same internal structure. However, class objects will each need a different descriptor, since their contents, i.e. number and position of their pointers, will vary.

The use of descriptors is illustrated in Figure 11. All instances of class A point to the descriptor of instances of A, which describes where the pointers inside the instances are. This descriptor points to the class object itself. This is necessary, since instances must point to their corresponding class objects and, since they point to the descriptor anyway, it is more space-efficient to make the descriptor point to it rather than introducing a new pointer inside each instance[13]. Finally, the descriptor of the class object of A, which describes where the pointers are inside the class object itself, is included in partition 1 and is pointed to by the class object. Notice that the descriptor of instances of A is replicated inside each partition which contains at least one instance of A. This helps to keep the descriptors close to the instances and to minimise access to other partitions during disk garbage collection.

The introduction of descriptors, apart from contributing towards the efficient and uniform identification of pointers inside objects, also has the following advantages.

- Descriptors can facilitate schema evolution, in the case when the object format does not change. If a class object needs to be replaced, only the pointers in the descriptors need to be updated and not pointers in all instances.
- Descriptors can also optimise the heap garbage

[13]When instances are faulted into main memory, this indirection is eliminated.

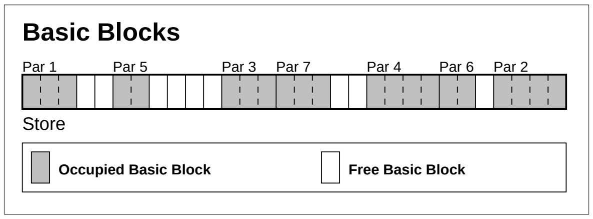


Figure 12: Use of Basic Blocks.

collector of PJama, if the notion of object kinds is retained while the objects are in memory.

- The fact that instances must point to their corresponding class object would normally increase the number of cross-partition references to class objects and hence would also increase the number of changes to their reference counts. However, the introduction of descriptors avoids this, since all instances of a given class would point to the descriptor inside their home partition and only the descriptors, at most one per partition, will point to the class object, via a cross-partition reference.

- Keeping the descriptors close to the corresponding objects improves locality and avoids the disk garbage collector from having to access other partitions.

- On disk at least, descriptors will also include the type of the fields of the corresponding objects so that a store can be used on platforms with different byte-order.

4 Free-Space Management

As mentioned in previous sections, the persistent object store will be divided into partitions whose size will vary and will depend on the kind of objects they contain. Because of this, two levels of free-space management are needed: one for allocating partitions inside the store and one for allocating objects inside a partition. The next two sections present a discussion of the differences in trade-offs, behaviour, and assumptions between the two levels.

4.1 In the Store

The store will be split into fixed-size blocks, called *Basic Blocks*[14] (BBs). Their size will be between 256KB and 1MB and probably equal to the smallest partition size. When a new partition needs to be allocated in the store, a number of contiguous BBs will be reserved for it, which of course implies that a partition size can only be a multiple of the BB size. On the other hand, when a partition needs to be de-allocated, the BBs it occupies will be marked as free in order to be re-used later. The use of BBs is illustrated in Figure 12.

Managing the free BBs and allocating space for partitions might seem a similar concept to managing free-space and dynamically allocating memory for programs [30]. Some of the properties of a good dynamic memory allocator are i) to minimise fragmentation, ii) to adapt quickly to changes in allocation patterns, iii) to minimise wasted space, and iv) to be fast. However, the trade-offs in managing free BBs are very different to managing free-space in memory, as discussed below.

Fragmentation : Since persistent stores are very long-lived (several orders of magnitude greater than a program heap), it is vital that fragmentation is kept as low as possible. Otherwise, it will have a negative impact on the performance and size of the store, as its life-time increases, and might introduce indefinetely accumulating space leaks, which are unacceptable in the context of a long-lived persistent store.

Adaptation to Changes : Again, due to the store being long-lived and different applications being able to run over it at different times (or even concurrently), the BB manager should be able to adapt easily to new allocation patterns.

Wasted Space : Disks these days are large and relatively cheap and, since the first two properties are so important, the space taken up by the store can be a small percentage (up to 10% or 15%) larger than its real size, in order to deal with them more efficiently.

Speed : Even though speed is vital for a dynamic memory allocator (since the programs which use it can exhibit a very high allocation rate), it is not as

[14]A better name for them would be *Minimum Blocks*, but unfortunately this is abbreviated to MBs, same as Megabytes.

important in allocating and freeing BBs. Partition allocation and de-allocation will not be extremely frequent events in PJSL and they will usually be followed by several disk accesses. Therefore, speed can be sacrificed in order to manage space more efficiently[15].

Flexible Partition Size : Sometimes partitions might need to grow or shrink. However, when a new size for one is proposed, the BB allocator can be allowed to change it within some limits. For example, if a 2MB partition needs to grow, it probably does not matter whether it becomes 3MB or 3.5MB (but does matter if it becomes 20MB). This can allow the BB allocator to be more efficient in dealing with fragmentation.

Partition Mobility : Since PIDs do not depend on the position of the partition in the store (see Section 3.4), it is possible to move a partition, in order to make a larger number of contiguous BBs available for a big partition. This clashes with the typical assumptions a memory allocator usually makes (e.g. objects allocated dynamically in languages like C or Pascal are non-migratable). The movement of partitions can only be used as a very last resort, after all other possible solutions have been exhausted.

The decision on the algorithm to be used for the BB management is still being researched. Ideas will be drawn from previous work in free-space management for file systems [15, 28] and dynamic memory management and allocation [30].

4.2 In a Partition

Once a partition has been allocated in the store, the free space inside it will be managed at the object level. The object space will be reclaimed and compacted using garbage collection [29]. Additionally, due to the introduction of partition regimes (see Section 2.1), it is possible for different partitions to implement different free-space management policies, optimised for the kinds of objects they contain.

[15]Of course, this does not mean that the BB manager might require 1 sec or more to allocate a partition. It just means that *some* speed might be sacrificed in order to achieve more efficient BB management.

For example, compaction can be beneficial for small objects because it can deal with the big number of small “holes” which are created as small objects become garbage. Also the fast allocation that it provides can improve the performance of promotion, if a large number of objects are allocated in the same partition. Alternatively, free-lists might apply better to larger objects since it is inadvisable to copy them unnecessarily[16] and, because of their size, fewer large objects can be accommodated inside a partition, which has the potential to keep the free-lists short.

It is also worth pointing out that clustering objects of similar size inside each partition has the potential to reduce fragmentation considerably.

5 Related Work

A large number of persistent stores have been constructed for a variety of systems and purposes. Mentioning all of them would be too lengthy. Therefore this section is selective.

ObjectStore [19] from Object Design Inc. is considered to be the most successful commercial object store. It uses a client-server model and was initially targetted for C++ applications, therefore space re-use relied on explicit deletes rather than garbage collection. Its latest version (5.0) provides an API to store Java objects.

Object Design Inc. have also announced lately a new product called ObjectStore PSE (Persistent Storage Engine), which is a lightweight version of their main product. The main difference is that it is written entirely in 100% Pure Java, thus trading-off performance for portability.

The Texas object store [26] from the University of Texas at Austin is similar to ObjectStore in that it was targetted for C++ and explicit deletes. It implements pointer-swizzling at page-fault time [31] and uses a technique similar to the descriptors (see Section 3.6) in order to do so.

[16]It has been observed that the average lifetime of large objects is usually greater than that of smaller ones [30]. This argument still needs supporting experimental evidence in the context of persistent stores. However, if it does hold and compaction is used, large objects will be forced to be copied unnecessarily, causing an increased number of disk accesses.

The object store implemented for the persistent language Napier88 [9, 10, 21], from the University of St Andrews, Scotland, has a good model of reachability and hence makes disk garbage collection possible. However, the object format which it uses groups all pointers in the beginning of the objects [8, 9, 10]. If the application which uses it does not have a similar object format (which is the case for PJama), expensive translations are necessary when objects are copied to and from the store.

Finally PJSL was influenced by the Mneme object store [22]. As mentioned in Section 2.1, it has a similar partitions and regimes called pools. Each pool can be independently managed and can support different object formats. Also, Mneme was designed with disk garbage collection in mind. It is not known whether a garbage collector has actually been implemented for it.

6 Conclusions and Future Work

The design of a store layer for the support of an orthogonally persistent platform for Java has been described. Important features are:

- the grouping of store-objects into a small number of kinds;
- the partitioning of disk space into partitions;
- local regimes for space and transfer management;
- the introduction of descriptors that abstract over the store formats used by a virtual machine; and
- the use of these features at the store-layer interface. They will be presented in a structured way for use by the adaption code, which must be written when a new (version of a) virtual machine is combined with the store layer.

The first three points contribute to flexibility and will allow experiments with regimes that are thought to be optimal for particular categories of data. The final two are expected to yield benefits when binding to a new virtual machine. They can be considered a satisfactory compromise, trading performance against maintenance costs, between stores that are tailored to a particular JVM and stores that incur large translation costs because they choose a neutral format of their own. The partition structure is also intended to allow incremental store administration algorithms.

Construction of this new store will take place this summer and we plan to report on the extent to which the design matches our expectations at the workshop. The store will be integrated with a JVM and performance measurement and tuning will quickly follow. The next phase will involve three parallel investigations:

- exploration of disk garbage collection strategies;
- evaluation of the utility of specialized partition regimes; and
- validation that the store will support its intended load and planned functionalities:
  - flexible and long transactions;
  - concurrent archiving and disk garbage collection;
  - schema evolution; and
  - a model of distribution [27].

7 Acknowledgements

This work is funded by the British Engineering and Science Research Council, grant number GR/K87791 and by a collaborative research grant from Sun Microsystems Inc. The authors would like to thank Dr Peter Dickman for his input on the free-space management issues, Huw Evans for providing feedback on consecutive versions of the paper, and Prof. Andrew Black, Dr Quintin Cutts, Dr Mick Jordan, and Dr Bernd Mathiske for their constructive comments.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] M. P. Atkinson, editor. *Fully Integrated Data Environments*. Springer-Verlag, 1997. To be published.

[3] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-Algol: an Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[4] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *SIGMOD Record*, December 1996.

[5] M. P. Atkinson and M. J. Jordan. Improved Hash Coding Methods for Java. Technical report, Sun Microsystems Laboratories Inc., MTV29/1, 2550 Garcia Avenue, Mountain View, CA 94043, USA, 1997. In preparation.

[6] 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 *Proceedings of POS'7*, Cape May, New Jersey, USA, May 1996.

[7] M. P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

[8] A. L. Brown. *Persistent Object Stores*. PhD thesis, University of St Andrews, Scotland, October 1989.

[9] A. L. Brown, G. Mainetto, F. Matthes, R. Mueller, and D. J. McNally. An Open System Architecture for a Persistent Object Store. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, pages 766–776, Hawaii, USA, 1992. Also appears as FIDE Technical Report FIDE/91/31.

[10] A. L. Brown and R. Morrison. A Generic Persistent Object Store. *Software Engineering Journal*, pages 161–168, 1992. Also appears as FIDE Technical Report FIDE/92/39.

[11] E. J. Cook, A. W. Klauser, A. L. Wolf, and B. G. Zorn. Semi-Automatic, Self-Adaptive Control of Garbage Collection Rates in Object Databases. In *Proceedings of SIGMOD'96*, pages 377–388, Montreal, Canada, October 1996.

[12] J. E. Cook, A. L. Wolf, and B. G. Zorn. Partition Selection Policies in Object Database Garbage Collection. In *Proceedings of SIGMOD'94*, pages 371–382, Minneapolis, USA, May 1994.

[13] L. Daynès and M. P. Atkinson. Main-Memory Management to support Orthogonal Persistence for Java. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, August 1997. To be published.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[16] D. H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proceedings of OOPSLA'86*, Portland, Oregon, USA, 1986.

[17] M. J. Jordan. Early Experiences with Persistent Java. In *Proceedings of the First International Workshop on Persistence and Java*, Drymen, Scotland, September 1996.

[18] M. J. Jordan and M. L. Van De Vanter. Modular System Building with Java Packages. In *Eighth International Conference on Software Engineering Environments*, pages 155–163, Cottbus, Germany, May 1997.

[19] C. Lamb, G. Landis, J. Orestein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[21] R. Morrison, R. C. H. Connor, Q. I. Cutts, G. N. C. Kirby, D. S. Munro, and M. P. Atkinson. The Napier88 Persistent Programming Language and Environment. In Atkinson [2], chapter 1.5.3. To be published.

[22] J. E. B. Moss. Design of the Mneme Persistent Object Store. Technical report, Department of Computer and Information Science, University of Massachussets, August 1990.

[23] J. E. B. Moss, D. S. Munro, and R. L. Hudson. PMOS: A Complete and Coarse-Grained Incremental Garbage Collector for Persistent Object Stores. In *Proceedings of POS'7*, Cape May, New Jersey, USA, May 1996.

[24] T. Printezis. Analysing a Simple Disk Garbage Collector. In *Proceedings of the First International Workshop on Persistence and Java*, Drymen, Scotland, September 1996.

[25] T. Printezis, M. P. Atkinson, L. Daynès, S. Spence, and P. Bailey. The Design of a Scalable, Flexible, and Extensible Persistent Object Store for

PJama. Technical report, Dept. of Computing Science, University of Glasgow, Scotland, May 1997.

[26] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Proceedings of POS'5*. Springer-Verlag, September 1992.

[27] S. Spence and M. P. Atkinson. A Scalable Model of Distribution Promoting Autonomy of and Cooperation Between PJava Object Stores. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, Hawaii, USA, January 1997.

[28] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall International Editions, 1992.

[29] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[30] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *International Workshop on Memory Management*, Kinross, Scotland, September 1995.

[31] P. R. Wilson and S. V. Kakkad. Pointer-swizzling at page-fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press.

[32] V. F. Yong, J. Naughton, and J. B. Yu. Storage Reclamation and Reorganization in Client-Server Persistent Object Stores. In *Proceedings of the International Conference on Data Engineering*, 1994.

The Transactional Object Cache as a Basis for Persistent Java System Construction

Stephen M. Blackburn and Robin B. Stanton[*]

Department of Computer Science
Australian National University
Canberra ACT 0200 Australia
{Steve.Blackburn,Robin.Stanton}@cs.anu.edu.au

Abstract

The promise of Java as the vehicle for widely used, industrial strength orthogonally persistent systems places a renewed emphasis on implementation technologies for orthogonally persistent systems. The implementation of such systems has been held back by a number of factors, including a breadth of technologies spanning database and programming language research domains, and difficulty in capitalizing on the fruits of the mainstream database research community.

In this paper we present PSI—a practical storage abstraction that separates database and programming language concerns and facilitates the adoption of mainstream transactional storage technology within orthogonally persistent systems. We argue for PSI as the basis for persistent Java system construction with particular reference to how it might be applied to PJama$_0$ [Atkinson et al. 1996].

1 Introduction

In Carey and DeWitt's retrospective on the past ten years in the database community [Carey and DeWitt 1996], two items are singled out for special attention as "casualties" of the past decade. These are *persistent programming languages* and *database toolkits*. The work reported in this paper raises the suggestion that Carey and DeWitt may have jumped horses a little early, as we claim that something akin to the database toolkit approach will play an fundamental part in the realization of industrial strength, widely used orthogonally persistent systems.

The work reported in this paper has developed in response to our desire to build efficient, robust, scalable orthogonally persistent programming environments. It has been shaped by our attempts to come to terms with the breadth of the technologies involved and the relative smallness of the persistence research community. The effect of these factors has been articulated by Atkinson and Morrison in their 1995 review of orthogonally persistent object systems [Atkinson and Morrison 1995], where they say:

> However, [existing systems] do not manage to provide full database facilities—that is, few can actually demonstrate a complete repertoire of incrementality, transactions, recovery, concurrency, distribution, and scalability. (It appears that this is more a consequence of teams being unable to muster the effort to tackle all of these issues together rather than of any fundamental limits.)

The focus on *orthogonally persistent* systems (i.e. systems where the persistence of data is orthogonal to all other properties of the data [Atkinson and Morrison 1995]), stems from a desire to build systems that elegantly unify the divergent database and programming language paradigms. While ODBMSs and object-relational systems bring programming languages and databases *closer* together, they do not seek to *unify* the two paradigms. The challenge of building efficient orthogonally persistent systems thus goes beyond the pragmatic appeal of building systems for today and instead focuses on systems for the future.

It seems clear to us that wide-spread uptake of orthogonally persistent systems will depend on the efficient and robust delivery of database facilities such as those outlined by Atkinson and Morrison.

[*]The authors wish to acknowledge that this work was carried out within the Cooperative Research Center for Advanced Computational Systems established under the Australian Government's Cooperative Research Centers Program. We also wish to acknowledge generous support from Fujitsu Ltd and Object Technology International.

The realization of this objective will depend on either a substantial increase in the size of the persistence community's research effort or a change of implementation approach.

This paper outlines a new implementation approach which can be characterized in terms of the following attributes: separation of concerns and concentration of expertise; maximal capitalization on the work of the database research community; and the portability and amenability to collaboration of implementations.

The remainder of this paper is structured as follows: First we will introduce and motivate an architectural framework for store implementation. Next we will present an interface based on an abstraction of that architecture. Finally we will discuss the practical application of PSI to persistent Java.

2 The Transactional Object Cache

The choice of the transactional object cache as the basis for a new design approach is rooted in the authors' experience with scalable store construction in the design of MC-Texas [Blackburn and Stanton 1996] and MC-DataSafe [Blackburn et al. 1997] and in the observation that the persistence community has not been able to build systems with the level of efficiency, robustness and functionality found in database products.

From the perspective of the future development of scalable persistent systems, perhaps the three most important lessons of the MC-Texas and MC-DataSafe experiments are these [Blackburn 1997]:

- The importance and appropriateness of a transactional model of concurrent computation when working in a distributed persistent space.
- The extent to which concurrency control and recovery fall within the mainstream of database research—research which the most (orthogonally) persistent architectures are not suited to extensively exploiting.
- The impact of temporal and morphological grain on performance.

These lessons can be interpreted as indicating a need for an architecture for scalable (orthogonally) persistent systems which:

1. Embodies a transactional model of concurrency control.
2. Separates database and programming language concerns in a way that facilitates the capitalization on research by the database community, particularly in the areas of concurrency control and recovery.
3. Offers an object-grained interface to clients which minimizes the need for copying of data.

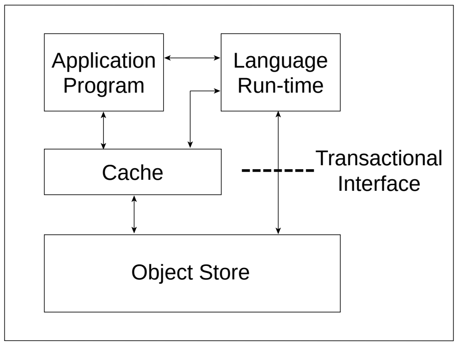


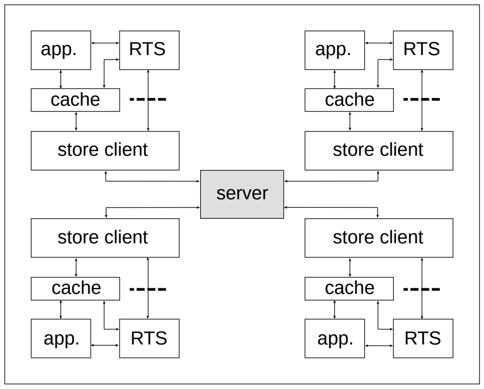
Figure 1: The transactional object cache architecture. The architecture of the object store is transparent to the application.

The transactional object cache architecture (figure 1) is one which satisfies all of these criteria. The basic architecture consists of five key components: an application program; an (optional) language run-time system (RTS); a cache; an object store; and a transactional interface. The basic model is that of the application program operating (via direct memory access) over a cached image of the store. The validity (in transactional terms) of the cached image seen by the application is ensured by appropriate use of the transactional interface.

The first of the above criteria is addressed by virtue of the transactional framework in which all cache consistency actions occur—the architecture is intrinsically transactional. Criterion two is met through the existence of the transactional interface, which separates the store and RTS. The extent to which the second criterion is met will be largely a function of the the interface definition. The dominance of the transactional object cache paradigm in the ODBMS literature [Carey and DeWitt 1986; Carey et al. 1994; Franklin 1996] has lead to mainstream database technology often being targeted at or sympathetic to that approach. Orthogonally persistent programming systems that adopt the transactional object cache architecture therefore stand to profit from the database community's research outcomes in a very direct way. The final criterion is met by virtue of the direct cache access given to the application and the object grained nature of the interface.

2.1 The Transactional Object Cache as a Platform for Distribution

While caching has a natural role in persistent system design as a means of hiding disk latency, it is also important to distributed systems where it is used to hide network latencies. There exists a well established literature on the distributed cache coherency problem. Approaches to this problem fall into two broad camps: transactional cache coherency [Franklin et al. 1997], based on the transactional notion of *isolation*; and distributed shared memory [Adve and Gharachorloo 1995], based on the programming language community's *cooperation-oriented* view of concurrency. Given the transactional nature of most orthogonally persistent systems, the first approach is of most interest here.



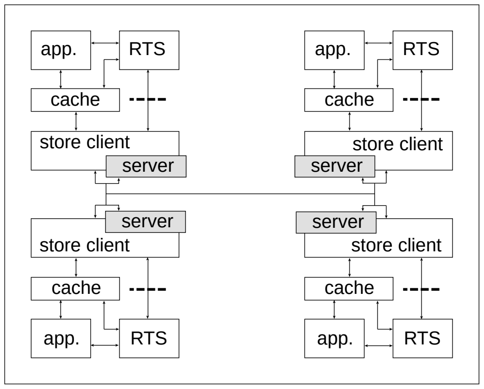


Figure 2: Client-server (left) and client-peer (right) transactional object cache architectures. In both cases the distributed nature of the underlying architecture is transparent to the run-time systems and applications.

Although not explicitly object-based, all of the the wide range of approaches to transactional cache coherency surveyed by Franklin et al. in [Franklin et al. 1997] are directly applicable to the transactional object cache architecture. These approaches span almost all dimensions of the concurrency control design space and include classic optimistic and locking architectures.

A powerful property of the architecture is that its semantics are defined in strict transactional terms and as a consequence embody the distribution-independent concurrency that flows from the isolation-oriented transactional view of concurrency, thus opening the door to transparency of distribution in implementations[1]. Approaches to transparent distribution in this context include client-server (distributed clients, single store), and client-peer [Carey et al. 1994; Blackburn and Stanton 1996; Blackburn 1997] (distributed clients, distributed store) architectures (figure 2).

3 PSI: A Transactional Storage Interface

Having identified the transactional object cache architecture and the value of a common storage interface, we will now briefly describe PSI, which we have developed to fulfill this role.

An important element of the interface is the identification of core and extended functionality. The core interface provides the client with the minimal functionality needed for basic ACID transactional

[1]The isolation guarantee of ACID transactions gives us this property. In the case where isolation is weakened, such as in some advanced transaction models, concurrency and distribution become more opaque.

object caching, whereas the PSI extensions place greater demands on the PSI implementer but give the client features such as advanced storage structures (indexes and collections) and intra-transactional checkpointing and rollback. To keep the description of PSI brief, we will only define the core PSI functionality here. A complete description of PSI appears in [Blackburn 1997].

3.1 A Semantic Framework

In order to effectively articulate the semantics of the interface, we first identify a clear abstraction of the transactional object cache architecture.

Although the key building blocks for a transactional interface may be fairly clear (begin, commit, abort etc.), the goal of flexibility both above and below the interface makes the identification of the precise semantics of these operations with respect to the various areas of store management more difficult. For example, a number of questions are raised by a simple write to the cache. When is that write made stable? When may the buffer associated with that data be freed? When will that change be made visible to other transactions? Formulating answers to these questions is made all the more difficult by a tendency for the various concerns to be blurred in the literature. In order to help meet our objective of flexibility in our interface design, three key concerns of a transactional object cache are separated and identified:

- stability,
- visibility,
- and cache management.

The first two are unambiguously central to a transactional storage interface; the third is included in a pragmatic response to the demands of efficient store construction. By establishing abstract interfaces to each of these concerns, a store interface can be built up and described precisely in terms of its semantics with respect to each of the separate concerns. The remainder of this section will focus on the development of those abstract interfaces. The approach taken will be to treat the three concerns as *orthogonal*—separate models of stability, visibility and cache management will be developed. Having developed the models and abstract interfaces to them, there will be a discussion on how the three concerns come together to form a rich abstraction of the transactional object cache architecture.

3.1.1 Stability

Stability is fundamental to most transaction models. The commit of an ACID transaction requires that all changes made by that transaction be made *durable*. Durability combines stability with irrevocability. By contrast, changes made stable (but not durable) may be subsequently rolled back.

In order to properly describe the stability semantics of a transactional object cache, it is helpful to first develop an abstract model of stability. Stability in this context concerns the maintenance of a stable image of a system state that corresponds in a meaningful way with the state of a dynamic system that is otherwise volatile.

The state of such a system can be represented as a history, h of (state changing) atomic events, e_i:

$$h = e_0.e_1.e_2 \ldots e_n$$

By identifying a durable global stability history, hs_g, and a set of volatile local stability histories, $HS_l = \{\langle 0, hs_0\rangle, \langle 1, hs_1\rangle, \ldots, \langle n, hs_n\rangle\}$, the atomicity and durability of a simple transactional system can be defined by describing the changes of state associated with a given transaction t in hs_t (where $\langle t, hs_t\rangle \in HS_l$). In this model, a transaction is made durable via an operation which appends hs_t to hs_g (i.e. $hs'_g = hs_g.hs_t$, where hs_g and hs'_g denote before and after values of hs_g respectively). With the addition of marker events, m, and stability events, s, the semantics of checkpoint/rollback and intra-transactional stability can be described respectively.

Having set in place a simple abstract model of stability capable of representing a wide range of stability scenarios, the remainder of this section will identify a series of stability primitives in terms of that model.

The global stability history is initially empty and there are no local stability histories ($hs_g^{initial} = empty \wedge HS_l^{initial} = \{\}$). In the following shorthand will be used to refer to simple modifications of local history, the effects on HS_l being implicit. For example, the notation

$$hs'_t = hs_t.e$$

Core	**Logging**	**Extended Trans.**
BeginUpdates	CheckpointUpdates	DelegateUpdates
NotifyUpdate	RollbackUpdates	
AbortUpdates	StabilizeUpdates	
MakeDurable		
EvictVolatile		

Table 1: Stability primitives. Only the semantics of core primitives are described in this paper. See [Blackburn 1997] for a complete description of all primitives.

should be read as shorthand for:

$$HS'_l = \{\langle t_i, hs_{t_i}\rangle \in HS_l | t_i \neq t\} \cup \{\langle t, hs_t.e\rangle | \langle t, hs_t\rangle \in HS_l\}$$

The following primitives, described in terms of the above stability model, are sufficient to describe the stability semantics of a basic flat ACID transaction, t:

BeginUpdates(t) $hs_t = empty \ \wedge \ HS'_l = HS_l \cup \{\langle t, hs_t\rangle\}$

NotifyUpdate(t,o) $hs'_t = hs_t.e_o$, where e_o is an event describing a change of state to some object, o.

AbortUpdates(t) $HS'_l = \{\langle t_i, hs_{t_i}\rangle \in HS_l | t_i \neq t\}$

MakeDurable(t) $hs'_g = hs_g.hs_t \ \wedge \ HS'_l = \{\langle t_i, hs_{t_i}\rangle \in HS_l | t_i \neq t\}$

In addition to these, there needs to be a primitive with global scope that describes the eviction of all volatile data (allowing system crash and program termination to be modeled):

EvictVolatile $\forall \langle t, hs_t\rangle \in HS_l \ \ hs'_t.s.E_c = hs_t$, where $E_c = e_{c_0}.e_{c_1}\ldots e_{c_n} \ \wedge \ s \notin E_c$

A simple ACID transaction would thus consist of BeginUpdates followed by zero or more NotifyUpdates and then one of MakeDurable, EvictVolatile or AbortUpdates.

3.1.2 Visibility

Visibility is another issue of fundamental importance to transaction models. ACID transactions ensure *isolation* by restricting visibility of changes made by uncommitted transactions. Extended transaction models often allow the controlled relaxation of isolation. There are a wide range of approaches to implementing visibility control, the design space for which spans many dimensions [Franklin et al. 1997].

Central to an understanding of visibility is the notion of transactions operating over potentially invalid images of the state of a store. The responsibility of the visibility control mechanism is to ensure that no transaction that saw an invalid image of the store be allowed to commit. As outlined by Franklin et al. [1997], there are two broad implementation alternatives: *avoidance* based schemes, where transactions are prevented from ever being exposed to invalid images of the store; and *detection* based schemes, where exposure to an invalid image of the store is detected and the transaction prevented from committing[2]. In either case, the visibility control mechanism must be able to determine the validity of the image of a store seen by a given transaction. Validity is usually defined in terms of serializability—a transaction is valid only if it can be serialized with respect to all previously validated transactions.

In order to describe visibility semantics concisely, a reference model for visibility will first be described. Note that this model is *orthogonal* to the model for stability presented in the previous section. The integration of stability, visibility and cache management semantics to fully capture the semantics of the transactional object cache is addressed at the end of this chapter.

The visibility semantics of a transactional system can be described in terms a single history, hv, of visibility events, e_i:

$$hv = e_0.e_1.e_2\ldots e_n$$

[2]Franklin et al. [1997] argue for a taxonomy of concurrency control approaches based on a separation into avoidance and detection based schemes. The taxonomy specifically avoids the ambiguity of the related pessimistic/optimistic distinction.

A transaction, t, is then modeled as a sub-history of hv, hv^t, and the store image seen by t is defined by the visibility events composing hv^t. T denotes the set of all transactions in hv, where all transactions are disjoint with respect to hv and T completely covers hv:

$$(e \in hv) \Rightarrow ((\exists t_i \in T | (e \in t_i)) \wedge (\forall t_j \in T, t_i \neq t_j \;\; e \notin t_j)))$$

The notion of irrevocability, which is central to modeling transactions, is introduced by defining $\mathit{irrevocable}(e)$ to denote that e is irrevocably part of hv. More generally, $\mathit{immutable}(t)$ is defined such that hv^t is a fixed sub-history of hv (i.e. membership of hv^t is static) and $\mathit{immutable}(t) \Rightarrow ((e \in hv^t) \Rightarrow \mathit{irrevocable}(e))$. The property of immutabilty can be used to capture the notion of transaction commit—all committed transactions are immutable while uncommitted transactions are mutable (both revocable and appendable).

The visibility events which compose the histories must capture sufficient semantic detail such that the validity of the store image as projected by a given sub-history can be determined. Furthermore, the events must capture the range of visibility scenarios possible in a cached store, most notably: shared access to an image of an object and the possibility of multiple 'versions' of objects existing as a result of replication. These facets of visibility are covered by the definition of read and write begin and end events with respect to versions, v, of objects, o, in particular workspaces, w: $r_{o_{v,w}}$, $\overline{r}_{o_w}$, w_{o_w}, and $\overline{w}_{o_{v,w}}$.

The concept of workspace is used here to refer to a single, potentially shared, image of an object. Interactions and potential conflicts between transactions sharing a single image of an object (for space efficiency reasons, for example) can thus be modeled. Object version numbers, v, monotonically increase and are incremented as part of each $\overline{w}_{o_v}$ event (which corresponds to the new version of o becoming visible in some scope). Read events, $r_{o_{v,w}}$, may be with respect to any existing version, v, of o and any workspace w.

Having constructed such a model of visibility, a number of functions are defined that will enable a user to reason about the validity of an image of the store as seen by a particular transaction t. The first of these is a termination function $T(hv^i)$ which tests termination on all reads and writes within a sub-history hv^i (the notation $a \rightarrow b$ is used to denote a preceding b in hv):

$$T(hv^i) \;\; = \;\; (\forall r_o \in hv^i \; (\exists \overline{r}_o \in hv^i \;\; (r_o \rightarrow \overline{r}_o))) \;\; \wedge \;\; (\forall w_o \in hv^i \; (\exists \overline{w}_o \in hv^i \;\; (w_o \rightarrow \overline{w}_o)))$$

In addition, a workspace isolation function, $W(hv^i, hv^j)$, is defined such that it is true only if no read events composing a given sub-history hv^i overlap with any write events in sub-history hv^j *and* are with respect to a common workspace image of an object:

$$\begin{aligned} W(hv^i, hv^j) \;\; = \;\; & (\forall r_{o_w}, \overline{r}_{o_w} \in hv^i \;\; (\nexists w_{o_w} \in hv^j \;\; (r_{o_w} \rightarrow w_{o_w} \rightarrow \overline{r}_{o_w}))) \;\; \wedge \\ & (\forall w_{o_w}, \overline{w}_{o_w} \in hv^j \;\; (\nexists r_{o_w} \in hv^i \;\; (w_{o_w} \rightarrow r_{o_w} \rightarrow \overline{w}_{o_w}))) \end{aligned}$$

Finally, a serializability function $S(hv^i, hv^j, hv^k)$ is defined such that $S(hv^i, hv^j, hv^k)$ is true only if the store image as seen by hv^i is consistent (serializable) with respect to hv^j, where hv^k denotes a sub-history of all events with which conflicts are ignored:

$$\begin{aligned} S(hv^i, hv^j, hv^k) \;\; = \;\; & \forall r_{o_v} \in hv^i \;\; (((\exists \overline{w}_{o_v} \in hv^j) \; \vee \; (\exists \overline{w}_{o_v} \in (hv^i \cup hv^k))) \;\; \wedge \\ & (\nexists \overline{w}_{o_{v'}} \in hv^j \;\; (\overline{w}_{o_v} \rightarrow \overline{w}_{o_{v'}}))) \end{aligned}$$

The inclusion of hv^k is necessary because given a decision to ignore conflicts between events in hv^i and hv^k, update events in hv^k form part of the valid store image seen by hv^i.

With the visibility model and the three validity functions defined, an abstract interface with respect to visibility in a transactional object cache can now be defined. The model is sufficiently rich to allow the user of the abstract interface to assess the transactional validity of a very wide range of visibility scenarios. The abstract interface will be introduced in terms of core and extended functionality (as with the stability interface) and consequently begins with the particular (i.e. basic ACID) and extends to the general.

In the following description, a number of conventions will be used:

- Appending an event to a sub-history implies appending the event to hv: $(hv^{t'} = hv^t.e) \Rightarrow hv.e$.
- Truncating a sub-history implies removal of events from hv: $(hv^{t'}.e_i = hv^t) \Rightarrow (hv' = hv \setminus e_i)$, where $\setminus$ denotes history difference.
- The operation $hv^i \cup hv^j$ denotes the order-preserving merging (union) of two sub-histories.

Furthermore, by definition any manipulation of a sub-history corresponding to an immutable transaction is not permitted.

Core	Logging	Extended Trans.
BeginVisibility	CheckpointVisibilty	DelegateVisibility
ReadIntention	RollbackVisibility	IgnoreConflict
ReadComplete		
WriteIntention		
WriteComplete		
AbortVisibility		
Terminated		
Finalize		
Expose		

Table 2: Visibility primitives. Only the semantics of core primitives are described in this paper. See [Blackburn 1997] for a complete description of all primitives.

3.2 Visibility and Core Functionality

Using the above model of visibility, the following primitives are sufficient to describe the visibility semantics of a simple flat ACID transaction, *t*:

BeginVisibility(t) $hv^t = empty \ \wedge \ T' = T \cup \{t\}$

ReadIntention(t,o) $hv^{t'} = hv^t.r_o$

ReadComplete(t,o) $hv^{t'} = hv^t.\overline{r_o}$

WriteIntention(t,o) $hv^{t'} = hv^t.w_o$

WriteComplete(t,o) $hv^{t'} = hv^t.\overline{w_{o_v}}$

AbortVisibility(t) $(hv' = hv \setminus hv^t) \ \wedge \ (T' = T \setminus \{t\})$, where the symbol $\setminus$ denotes history difference and set difference respectively (i.e. the events composing sub-history hv^t are removed from hv).

Terminated(t,o) $T(hv^{t_o})$, where hv^{t_o} refers to a sub-history of hv consisting of all events in transaction t relating to object o.

Finalize(t) $(T(hv^t) \wedge S(hv^t, hv^i, hv^{ic_t}) \wedge W(hv^t, hv^w))$, where hv^i is the sub-history of hv consisting of all irrevocable events, hv^{ic_t} is the sub-history of hv consisting of all events with which t is ignoring conflicts, and $hv^w = hv \setminus (hv^t \cup hv^{ic_t})$.

Expose(t) $immutable(t) = true$

3.3 Cache Management

A third dimension of the transactional cache architecture is cache management. A cached store design is motivated by the desire to hide IO latency and introduce replication through caching. While visibility is concerned with the state of the store as it might be seen by a given transaction, cache management is concerned with the *availability* of that image to the transaction.

Cache management can be modeled in terms of each active transaction, t, operating over a logically distinct cache c_t within which are present some set of objects: $c_t = \{o_0, o_1, \ldots, o_n\}$. An object is only available to a transaction if present in that transaction's (logically distinct) cache.

Core
Fix
Unfix

Table 3: Caching primitives.

Only two primitives are necessary for the implementation of a cache management scheme:

Fix(t,o) $c_t' = c_t \cup \{o\}$.

Unfix(t,o) $c'_t = c_t \setminus \{o\}$.

With these the client can notify the store of when it requires availability to a given object. The state of the available objects is a function of the visibility control mechanism.

3.3.1 Generality and Completeness

Each of the three orthogonal abstractions outlined above are general—in the sense that they are premised only by intrinsics of scalable persistent systems, namely *caching*, *atomicity* by way of transactions, and *layered software abstractions*—and complete in so far as they support the wide range of scenarios derivable from a combination of ACID transactions, delegation, isolation relaxation, intra-transactional stability, and checkpoint / rollback.[3] When brought together, the orthogonal abstractions yield a full abstraction of the transactional object cache with the same generality and completeness. The remainder of this section gives a brief overview of the full abstraction.

The relationships between each of the abstractions are not symmetric. Visibility can be thought of as dominant because it is visibility that defines the image of the store seen by each transaction. By contrast, stability and cache management have ancillary roles of defining the stability and availability of the store image as determined by the visibility model.

By and large the integrated semantics of the full abstraction are straight-forward. However, it should be emphasized that the cache is merely a means of accessing the store image as defined by the visibility model. Any access to the cache outside the context of a fix(t,o), unfix(t,o) pair is not meaningful and any access within the context of a fix(t,o), unfix(t,o) pair is only meaningful insofar the visibility model indicates the validity of such an access.

Finally it should be noted that although the abstraction is presented in terms of object-grained semantics, it is applicable to data movement and coherency at any granularity and so may be trivially adapted to account for such.

3.4 Separation of Concerns

Having presented the semantics of a transactional object cache, we now look at how various elements of persistent programming system design impact on the PSI interface design. We start by identifying key design choices for a persistent programming language (PPL) implementer. Having identified these, we determine the extent to which PSI will take a role in that aspect of PPL construction on the basis of its relative proximity to storage and language issues. In the remainder of this section, we identify six key issues and for each argue the case for PSI's role with respect to that issue. We refer to the PPL as the interface's "client" throughout this section.

Persistence Identification PSI presents its clients with a store that supports persistence by reachability from a single root and that guarantees the referential integrity of object identifiers (OIDs) *within the scope of a transaction*. An OID is undefined as soon as it leaves its transactional scope. Support for extended transaction models allows clients to use delegation of transactional scope to avoid re-traversing from the root of persistence at the start of each transaction. This approach does not inhibit the client PPL from presenting applications with a more elaborate space of named entry points, it is left to the client to ensure that all advertised entry points remain reachable. In order to efficiently implement persistence by reachability, PSI introduces the concept of *descriptors*. PSI associates each object with a (hidden) descriptor field. The descriptor field points to an object that encodes the location of references (pointers) within the object.

PSI's approach to object typing contrasts strongly with ODMG's ODL [Cattell and Barry 1997] and SHORE's SDL [Carey et al. 1994], which are object definition languages that attempt to provide a means for storing objects and their types in a language independent manner. PSI is language independent, but does not attempt to address the issue of *poly-lingual access* to objects. The need for PSI to be aware of types is limited to the requirement that it be able to efficiently and correctly garbage collect the store—a need that is adequately met by the identification of references and is-one-of relationships as provided by the descriptor model.

[3]It is hard to *prove* completeness, however the literature *suggests* the completeness of the range of scenarios covered by the abstractions.

Residency Checks and Write Detection Object faulting is usually achieved through some sort of *residency check* at each object access. Similarly, some form of *write detection* is typically used to identify updated objects for writing back to stable store. The spectrum of approaches to residency checking and write detection include explicit checks by an interpreter at the time of word access or update (Napier88 [Munro 1993]) and use of page-grained hardware memory protection (the Texas persistent store [Wilson and Kakkad 1992]). The choice of the most appropriate mechanism involves tradeoffs which, for a particular client, are likely to be heavily context-dependent [Hosking 1995]. For this reason PSI relies on the client making *explicit* read and write requests, leaving the choice of detection mechanism to the client's implementer.

Swizzling The choice of an appropriate swizzling strategy is complex and influenced by application characteristics and PPL implementation approach (interpreted versus compiled PPL, for example). For this reason PSI does not *impose* a particular approach to swizzling on the client PPL, but instead provides hooks that facilitate swizzling—by allowing the client to identify references through descriptors, for example.

Concurrency Control and Cache Management The question as to which level the management of concurrency control should live within a persistent programming system is an important and difficult one. One approach is to give the upper layers levers in the form of transactional primitives within a flexible, extended transaction framework such as that formalized in ACTA [Chrysanthis and Ramamritham 1994]. The PSI interface is based on a rich abstraction of a transactional object cache which gives the client the levers necessary for the execution of a wide range of transaction models while leaving the store designer considerable implementation scope [Blackburn 1997].

Recovery There exist a wide variety of approaches to recovery. Most are compatible with the semantics of durability and failure in the context of basic ACID transactions and some support more sophisticated stability semantics such as intra-transaction stabilization and roll-back. The ACID notion of durability is included in the core PSI interface while intra-transaction stabilization and rollback form the basis for PSI's logging extension.

Advanced Storage Structures Some prospective PSI clients are likely to make use of extended storage structures such as index and collection types [Albano et al. 1995]. While these can be readily constructed on top of an object store (including PSI), the specialized nature of such data types has lead to the publication within the database community of considerable implementation optimizations with respect to concurrency control and storage management. In response to this, the implementation of index and collection types is the basis for one of PSI's extensions.

3.5 The interface

The PSI core interface is now defined in terms of the semantic framework outlined in section 3.1. In addition to the extra functionality of extensions for logging, extended transactions and advanced storage structures, the interface includes a number of housekeeping functions for opening and closing the store (including store recovery), setting the cache size etc. For the sake of brevity, only the core interface is described here (the interested reader is referred to [Blackburn 1997]). The transactional elements of the PSI interface are listed in table 4.

The modules that are not transactional in nature are illustrated in table 5. In addition there is a function, PSI_LIO, which gives asynchronous and list semantics to most of the transactional operations in much the same way as the POSIX lio (list-directed IO) interface [ISO/IEC and IEEE 1990] does for Unix file operations.

PSI_Read A copy of a specified object is forced into the cache. While PSI will not guarantee the object to be fresh, it will ensure that no transaction exposed to a stale object be allowed to commit (see section 3.1.2). The object will either be left in-place—in which case the client is returned a pointer to the object—or copied to a client-specified buffer, depending on the value of a boolean, copy, passed by the client. In terms of the semantics outlined in section 3.1, the read has no impact on stability but implements ReadIntention(t,o) with respect to visibility and if the read is in-place, Fix(t,o) with respect to cache management.

Core	**Logging**	**Extended Trans.**	**Indexing**
PSI_Read	PSI_Checkpoint	PSI_Delegate	PSI_Insert
PSI_Write	PSI_Rollback	PSI_IgnoreConflict	PSI_Fetch
PSI_New	PSI_ThisCheckpoint		PSI_Delete
PSI_NewTransaction	PSI_Stabilize		
PSI_Commit			
PSI_Abort			
PSI_Unfix			
PSI_Fix			

Table 4: The PSI transactional interface.

Extended OID	**Housekeeping**
PSI_GetSA	PSI_Init
PSI_GetOID	PSI_Open
	PSI_Close
	PSI_Recover

Table 5: PSI non-transactional calls.

PSI_Write The client's intention to update an (existing) object is asserted. The reference passed by the client may be to the in-place version of the object or to a private copy. The call implements WriteIntention(t,o). In the case where the object reference is in-place it also asserts Fix(t,o) semantics. ReadIntention(t,o) semantics are not asserted—a client would therefore typically call PSI_Read before PSI_Write (although this is not necessary if the client is not concerned with the object's prior state).

PSI_New Space is allocated space for a new object or a new array object, the call returning a cache pointer and an OID (see [Blackburn 1997] for a description of the PSI storage model). The nearOID parameter allows the caller to nominate an object as a placement hint (predefined constants allow users to nominate other sorts of hints, for example NEAR_ANY). PSI_New takes a descriptorOID argument which allows the user to define *is-one-of* relationships and to identify the structure of the new object. In addition to the allocation of space, PSI_New implements WriteIntention(t,o) and Fix(t,o) semantics.

PSI_NewTransaction The data structures associated with a new transaction are created and a transaction handle is initialized with respect to that transaction. The caller may provide a callback for handling pre-emptive aborts (see section 3.1.2). BeginUpdates(t) and BeginVisibility(t) semantics are asserted.

PSI_Commit The commit process is two phase. The first phase terminates the transaction by first asserting DelegateUpdates and DelegateVisibility with respect to any delegations flagged for commit time, and then asserting Unfix(t,o), ReadComplete(t,o), WriteComplete(t,o) and NotifyUpdate(t,o) with respect to all objects accessed by the transaction. The second phase is conditional on Isolate(t) and Finalize(t) being true. If they are true MakeDurable(t) and then Expose(t) semantics are asserted. Otherwise PSI_Commit returns failure, and the transaction is aborted (AbortUpdates(t) and AbortVisibility(t) are asserted). ACI transactions can be constructed by delegating all updates prior to commit. Although PSI_Commit is described here in terms of a series of steps, its implementation must be atomic.

PSI_Abort All resources associated with the transaction are released. Unfix(t,o) is asserted with respect to all objects accessed by the transaction and then AbortUpdates(t) and AbortVisibility(t) are asserted.

PSI_Unfix The availability of the specified object is removed by asserting Unfix(t,o) (the object remains unavailable until the need for it is re-asserted via PSI_Fix, PSI_Read, or PSI_Write). If the object was being updated in-place (i.e. PSI_Write was asserted with respect to an in-place version of the same object), NotifyUpdate(t,o) semantics are asserted with respect to the object.

PSI_Fix The need for an object to be made available is asserted through Fix(t,o). This call is only valid in the context of read or write intentions for that object already being asserted but not completed (PSI_Unfix must have been called subsequent to the PSI_Read, or PSI_Write in order to make the object unavailable).

4 Applying PSI to Persistent Java Implementations

Having defined the PSI core, we now investigate the application of PSI to persistent Java implementations, using the PJama$_0$ architecture [Atkinson et al. 1996] as a reference point[4].

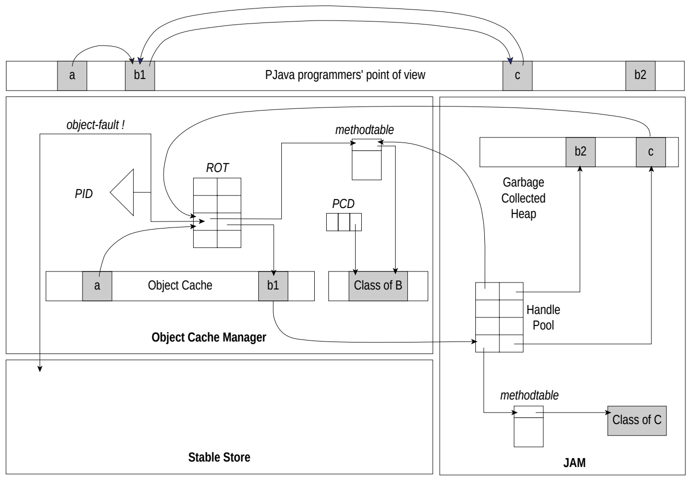


Figure 3: PJama$_0$ architecture [Atkinson et al. 1996], illustrating JAM, Object Cache Manager, and Stable Store modules. The details of the PJama$_0$ Stable Store implementation have been hidden. (Figure adapted from figure 2 in [Atkinson et al. 1996].)

The PJama$_0$ architecture (figure 3) is comprises the three key modules: the Java Abstract Machine (JAM), an Object Cache Manager, and a Stable Store. A key feature of the PJama$_0$ architecture is the minimal extent to which the JAM is disturbed [Atkinson et al. 1996]. The persistence mechanisms in PJama$_0$ are thus concentrated in the the Object Cache Manager and the Stable Store, the Object Cache Manager faulting objects from Stable Store and requesting stabilization of data by the Stable Store when necessary. The Stable Store is implemented on top of RVM [Satyanarayanan et al. 1994], a segment-based transactional storage system.

4.1 Implementation Approaches

By including an object cache manager as a central component, the PJama$_0$ architecture lends itself to a PSI-based implementation. Minimally, PSI could be used in place of the Stable Store. A more tightly coupled approach might see the object cache manager operating over PSI's cache rather than copying objects into its own cache. Given the amenability of the PJama$_0$ architecture to a PSI implementation, the dominant PSI/PJama design issue is therefore likely to be whether a single-level or two-level (as in PJama$_0$), buffering strategy should be employed.

Copying Versus Non-copying o maximize data transfer efficiency, caching stores usually move data in and out of the cache at as coarse a grain as possible. In the case of an object cache, this can result in many objects needlessly being brought into the cache at each object fault. Object clustering can help but it cannot *guarantee* good results because of the stochastic nature of the problem. Another approach is to re-pack objects in a second level cache. This can increase the efficiency of memory use significantly, but incurs the overhead of a memory-to-memory copy for each object fault, which, in the face of memory bandwidth bottlenecks becoming a dominant feature of modern processor architectures, is an increasingly unattractive option. The implementer is thus faced with a time-space tradeoff: either optimize for

[4]We chose PJama$_0$ because to our knowledge it is the only orthogonally persistent Java with a detailed published account of its architecture.

time by avoiding copies, or optimize for space by introducing a packed cache. The approach taken in the design of PJama$_0$ is to optimize for space [Atkinson et al. 1996]. A third alternative is to adopt a hybrid approach [Kemper and Kossmann 1994] which involves adaptively switching between the two polices as making each of the respective tradeoffs becomes more essential.

The results presented in [Kemper and Kossmann 1994] suggest that a hybrid approach is likely to be optimal. Such an approach could either be implemented within PSI or on top of PSI. Experimentation is necessary to determine which of these schemes will perform best.

4.2 Stable Store Alternatives

Having pointed to the suitability of PSI as the basis for persistent Java construction, we will now briefly assess alternatives.

4.2.1 RVM

Initial performance results for PJama$_0$ [Jordan 1996] suggest that RVM performs reasonably well. However, the stated design goals for RVM [Satyanarayanan et al. 1994], which include simplicity and portability of the RVM implementation suggest that a more targeted storage system, such as PSI, is likely to perform better and give the PJama implementer more flexibility.

The simplicity of the RVM design comes at the cost of efficiency, of loading the client with implementation responsibilities, and of reduced functionality such as resilience to media failure [Satyanarayanan et al. 1994]. The client implementer is left to take care of important elements of the transactional storage system such as *distribution*, *nesting*, and *serializability*. RVM's authors state that these areas of functionality were excluded in order to provide clients with flexibility with respect to the implementation choices associated with each [Satyanarayanan et al. 1994]. In terms of the semantic framework of section 3.1, RVM's transactional semantics are very weak. It supports only basic *stability* and *cache management* semantics (no support for logging or delegation) and makes no guarantees about *visibility* semantics other than those that are implicit in stability semantics (commit stabilization followed by access by a later transaction).

By contrast, PSI provides a simple yet rich abstraction of a transactional object cache which gives the *client implementer* flexibility through a collection of powerful levers by way of fully implemented transactional mechanisms. Furthermore, *PSI implementors* are given a great deal of scope to explore different implementation strategies, so to the extent that implementations of these are made available to client implementers, client implementers will have a wide choice of storage implementation approaches available to them.

While care was taken in the RVM design to maximize the portability of the RVM *implementation*, the design philosophy for PSI is closer to that of MPI (a widely used message passing interface [Message Passing Interface Forum 1994]), where the emphasis is not on the interface *implementation* being portable, but rather the interface efficiently *providing portability* to its clients by appropriately abstracting complex, vendor specific message-passing mechanisms. In the case of MPI, such efficiency is usually obtained precisely by exploiting non-portable hardware features on each target platform. It seems likely that stable store implementations will need to do the same in order to deliver performance comparable with that offered by commercial database vendors.[5]

4.3 Other Approaches

A range of alternative implementation approaches exist. The two most obvious being an integrated Stable Store / RTS implementation and the use of another storage layer like RVM or PSI.

Integrated Implementation Approaches The "integrated" implementation approach has been used by many, if not most orthogonally persistent programming systems to date. By "integrated", we mean that the design does not attempt to strongly separate database and programming language technologies. The absence of any such implementations that efficiently offer a full range of database features (as outlined in [Atkinson and Morrison 1995]) gives support to the view that the approach is ultimately inappropriate in the context of a small research community such as the persistent systems community.

[5]For example some operating systems offer non-portable control over memory management that can greatly improve caching performance and reduce TLB misses. Also, in the context of distributed stores, optimal communication mechanisms vary from platform to platform (e.g. TCP / IP versus MPI or native message passing).

Alternative Storage Layers Mneme [Moss 1990] is perhaps the most natural candidate as an alternative storage layer. Like PSI, Mneme is based on the transactional object cache architecture and so is well suited to the PJama architecture. However, Mneme is distinguished from PSI in two key respects: First, Mneme does not directly support extended transaction models, which are a feature of the PJama design. Mneme leaves the implementation of extended transaction models to higher levels of abstraction, only directly supporting simple transactions. Secondly, Mneme is a store implementation rather than an interface definition, so it does not offer the same opportunities for collaboration between store implementers and PJama builders as PSI.

There are few other purpose-designed stand-alone transactional storage layers. Carey and DeWitt's review [Carey and DeWitt 1996] lists a number of key examples of database system toolkit projects, some of which come close to the transactional storage layer approach. Carey et al. illustrate problems with the database toolkit approach by reporting a number of the problems they and other users encountered with EXODUS [Carey and DeWitt 1996; Carey et al. 1994]. These include: users wanting to use EXODUS to build an object *server* and being stuck with a client-server architecture, their server thus becoming a server-on-a-client; control over low level details being hidden from 'serious' implementers (*too high* an abstraction); and application programmers finding it a bit too low level (*too low* an abstraction).

All of the database toolkit projects differ from PSI in a number of ways, perhaps most important of those being that PSI is not a *system implementation* but an *interface specification* based on a rich storage abstraction. A system based on PSI is therefore not limited to a single storage implementation approach or architecture. To the contrary, it may be used as the platform for experimentation with a range of approaches to storage management.

5 Conclusions

We have presented PSI, an interface based on an abstraction of the transactional object cache architecture. We argue that the use of such an interface will play an important role in overcoming the problem of mastering the breadth of technology involved in the construction of orthogonally persistent systems by separating concerns and so allowing a concentration of expertise. PSI should also encourage portability of persistent systems and enhance opportunities for collaboration. Finally, we have shown how PSI might be integrated into a persistent Java implementation, using PJama$_0$ as an example.

Bibliography

ADVE, S. V. AND GHARACHORLOO, K. 1995. Shared memory consistency models: A tutorial. Technical Report WRL Research Report 95/7 (Sept.), Digital Equipment Corporation, Palo Alto, CA, U.S.A.

ALBANO, A., BERGAMINI, R., GHELLI, G., AND ORSINI, R. 1995. An introduction to the database programming language Fibonacci. *The VLDB Journal 4*, 3 (July), 403–444.

ATKINSON, M. P., JORDAN, M. J., DAYNÈS, L., AND SPENCE, S. 1996. Design issues for Persistent Java: A type-safe, object-oriented, orthogonally persistent system. In R. CONNOR AND S. NETTLES Eds., *Seventh International Workshop on Persistent Object Systems* (Cape May, NJ, U.S.A., May 1996), pp. 33–47. Morgan Kaufmann.

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent systems. *The VLDB Journal 4*, 3 (July), 319–402.

BLACKBURN, S. M. 1997. *High Level Storage Abstractions: A foundation for scalable persistent system design*. PhD thesis, Australian National University, Canberra, Australia. Available online at `http://cs.anu.edu.au/~Steve.Blackburn/`.

BLACKBURN, S. M., SCHEUERL, S. J. G., STANTON, R. B., AND JOHNSON, C. W. 1997. Recovery and page coherency for a scalable multicomputer object store. In H. EL-REWINI AND Y. N. PATT Eds., *30th Hawaii International Conference on System Sciences* (Hawaii, U.S.A., Jan. 7–10 1997), pp. 523–532.

BLACKBURN, S. M. AND STANTON, R. B. 1996. Multicomputer object stores: The Multicomputer Texas experiment. In R. CONNOR AND S. NETTLES Eds., *Seventh International Workshop on Persistent Object Systems* (Cape May, NJ, U.S.A., May 29–31 1996), pp. 250–262. Morgan Kaufmann.

CAREY, M. J. AND DEWITT, D. J. 1986. The architecture of the EXODUS extensible DBMS. In *Proceedings of the First International Workshop on Object-Oriented Database Systems* (Pacific Grove, CA, U.S.A., Sept. 1986), pp. 52–65. IEEE.

CAREY, M. J. AND DEWITT, D. J. 1996. Of objects and databases: A decade of turmoil. In T. M. VIJAYARAMAN, A. P. BUCHMANN, C. MOHAN, AND N. L. SARDA Eds., *VLDB'96, Proceedings of the 22th International Conference on Very Large Data Bases* (Mumbai (Bombay), India, Sept. 3–6 1996), pp. 3–14. Morgan Kaufmann.

CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M. L., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O. G., AND WHITE, S. J. 1994. Shoring up persistent applications. In R. T. SNODGRASS AND M. WINSLETT Eds., *Proceedings on the 1994 ACM-SIGMOD Conference on the Management of Data*, Volume 23 of *SIGMOD Record* (Minneapolis, MN, U.S.A., May 24–27 1994), pp. 383–394. ACM.

CAREY, M. J., FRANKLIN, M. J., AND ZAHARIOUDAKIS, M. 1994. Fine-grained sharing in a page server OODBMS. In R. T. SNODGRASS AND M. WINSLETT Eds., *Proceedings of the 1994 ACM-SIGMOD International Conference on the Management of Data*, Volume 23 of *SIGMOD Record* (Minneapolis, MN, U.S.A., May 24–27 1994), pp. 359–370. ACM.

CATTELL, R. G. G. AND BARRY, D. K. Eds. 1997. *The Object Database Standard: ODMG 2.0*. Morgan Kaufman.

CHRYSANTHIS, P. AND RAMAMRITHAM, K. 1994. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems 19*, 3 (Sept.), 450–491.

FRANKLIN, M. J. 1996. *Client Data Caching: A Foundation for High Performance Object Database Systems*, Volume 354 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, U.S.A. This book is an updated and extended version of Franklin's PhD thesis.

FRANKLIN, M. J., CAREY, M. J., AND LIVNY, M. 1997. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems 22*, 3 (Sept.). Accepted for publication: see `http://www.acm.org/tods/`.

HOSKING, A. L. 1995. Benchmarking persistent programming languages: Quantifying the language / database interface. In *OOPSLA'95 Workshop on Object Database Behavior, Benchmarks and Performance* (Austin, TX, U.S.A., Oct. 15–19 1995).

ISO / IEC AND IEEE. 1990. *ISO/IEC 9945-1, IEEE Std 1003.1 Information technology — Portable Operating System Interface (POSIX)*.

JORDAN, M. 1996. Early experiences with Persistent Java. In M. JORDAN AND M. ATKINSON Eds., *First International Workshop on Persistence and Java* (Drymen, Scotland, Sept. 16–18 1996). Available online at DB&LP server: `http://www.informatik.uni-trier.de/~ley/db/`.

KEMPER, A. AND KOSSMANN, D. 1994. Dual-buffering strategies in object bases. In J. B. BOCCA, M. JARKE, AND C. ZANIOLO Eds., *VLDB'94, Proceedings of the 20th International Conference on Very Large Data Bases* (Santiago de Chile, Chile, Sept. 12–15 1994), pp. 427–438. Morgan Kaufmann.

MESSAGE PASSING INTERFACE FORUM. 1994. MPI: A message-passing interface standard. In *International Journal of Supercomputing Applications*, Volume 8 (Nov. 1994). Also available as University of Tennessee Technical Report CS-94-230.

MOSS, J. E. B. 1990. Design of the Mneme persistent object store. *ACM Transactions on Information Systems 8*, 2 (April), 103–139.

MUNRO, D. S. 1993. *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, University of St Andrews, St Andrews, Scotland. Available online at: `http://www-ppg.dcs.st-andrews.ac.uk/Publications/`.

SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. 1994. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems 12*, 1 (Feb.), 33–57. Important corrigendum appears in *ACM Transactions on Computer Systems 12*, 2.

WILSON, P. R. AND KAKKAD, S. V. 1992. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *1992 International Workshop on Object Orientation in Operating Systems* (Dourdan, France, 1992), pp. 364–377. IEEE: IEEE Computer Society Press.

OCB: An Object/Class Browser for Java

Graham N.C. Kirby and Ron Morrison

School of Mathematical and Computational Sciences,
University of St Andrews, North Haugh, St Andrews KY16 9SS, Scotland
{graham, ron}@dcs.st-and.ac.uk

Abstract

This paper describes an interactive browser used for exploring the structure of Java objects and their classes. It is implemented in Java and uses JDK 1.1 core reflection classes to discover details of the objects passed to it. The initial motivation for development arose from the need to browse persistent Java stores; the browser may also be useful as part of a symbolic debugging or visualisation tool.

1 Introduction

The provision of orthogonal persistence for Java, e.g. [ADJ+96, DHF96, GN96, MCK+96], allows the programmer to create potentially large persistent stores of Java objects. There are a number of ways of discovering the contents of these stores, including interactive browsing, writing programs which navigate inter-object references, and the use of a query language. All of these, and others, will probably be required in practice. This paper describes a visualisation tool called OCB (Object/Class Browser) which addresses the first requirement, supporting the interactive display of Java objects and classes, and the navigation of references linking them.

The main design goals for the initial version of OCB were:

- to provide a simple and clean user interface;
- to produce an implementation quickly; and
- to implement the browser using only standard Java for maximum portability[1].

The OCB browser was designed in response to a need identified by the developers of the PJama persistent Java implementation [ADJ+96]. It was quickly recognised, however, that most of its facilities would also be useful in conventional Java systems and other persistent versions. All OCB facilities other than access to persistent roots, and in some cases method invocation, will work with any Java system. Persistent root access for other persistent versions can be added simply on a per-system basis; the details depend on the model of persistence provided.

The remainder of this paper contains a summary of related work in Section 2, descriptions of the user and program interfaces in Sections 3 and 4, an outline of the implementation in Section 5 and a discussion of avenues for further development in Section 6.

2 Related Work

There are several aspects of a persistent object store which it may be useful to visualise, including:

- the states of the objects in the store and the graph of references linking them;
- the states of the threads running in the store; and
- the class hierarchy or type structure associated with the objects in the store.

Other systems support these requirements for Java and other languages to varying degrees. The OCB browser in its current implementation addresses only the first and third requirements, although it is planned to extend it to include thread states in a future version.

[1] No native methods are used, although OCB is not 100% Pure Java™ for reasons explained in Section 6.

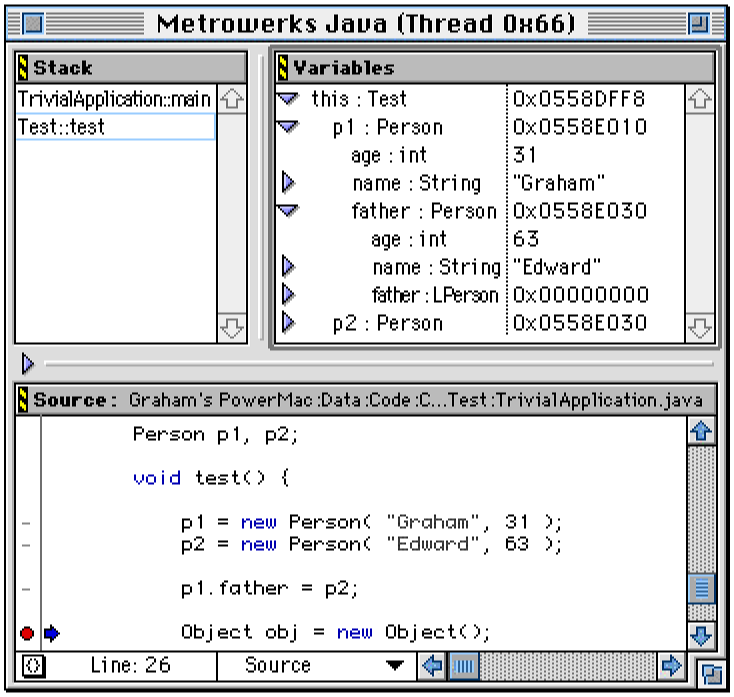
2.1 Commercial Programming Environments

Several commercial products offer integrated programming environments which support the visualisation of the state of an executing Java Virtual Machine. Although these run on non-persistent Java systems, there is considerable overlap with the facilities needed for persistent store visualisation. Such products include Metrowerks CodeWarrior [Met97] and Symantec Visual Café [Sym96] which also support other languages such as C, C++ and Pascal.

CodeWarrior provides a *Hierarchy* window which displays the class tree of the Java program being edited; clicking on a node in the tree brings up the source code for that class. During execution a symbolic debugger may be used to examine local variables accessible by a thread at a particular break-point, and the states of objects reachable via those variables. Similar facilities are available in Visual Café.

The class hierarchy display and symbolic debugger in CodeWarrior are accessed interactively by the programmer via the programming environment user interface. These facilities are only available if the program has been compiled with appropriate flags set. In contrast, OCB can be accessed through an API by any normally-compiled running Java program and made to display objects and classes in scope at the current point of execution. It can also be invoked as a stand-alone Java application to display persistent objects and classes. Since it is written solely in Java, OCB is also relatively portable, whereas CodeWarrior and Visual Café are available in multiple versions tailored to particular platforms, involving greater porting effort.

CodeWarrior[2] does not make it easy to distinguish object graph structures. For example, the *Variables* pane in Figure 1 shows an object of class *Person*. The variable *p1* contains the object at the program break-point indicated by the arrow in the *Source* pane. At this point *p1* and *p2* denote separate instances of *Person*, and *p1.father* is set to *p2*. This structure is not clear from the display, which shows the objects referenced from the current object in the form of a nested list. In particular, *this.p1.father* denotes the same object as *this.p2,* a fact that can only be deduced from the list by noticing that they both have the same address. Presumably the need to make such deductions is the reason for including addresses in the display, which seems an odd mixing of abstraction levels for a language with automatic memory management. Also, in this user interface class names used in the *Variables* pane are not linked to their definitions; it would be useful to able to click on a class name and bring up its definition directly.


Metrowerks Java (Thread 0x66)
Stack
TrivialApplication::main
Test::test
Variables
Source : Graham's PowerMac:Data:Code:C...Test:TrivialApplication.java
Line: 26
Source

```
this : Test                 0x0558DFF8
  p1 : Person               0x0558E010
    age : int               31
    name : String           "Graham"
    father : Person         0x0558E030
      age : int             63
      name : String         "Edward"
      father : LPerson      0x00000000
  p2 : Person               0x0558E030
```

```
        Person p1, p2;

        void test() {

            p1 = new Person( "Graham", 31 );
            p2 = new Person( "Edward", 63 );

            p1.father = p2;

            Object obj = new Object();
```

Figure 1. CodeWarrior Java debugging window

2 Professional Release 1 at the time of writing.

In contrast, the OCB browser does not display any address information. Indeed, since it is implemented as a standard Java program it cannot obtain addresses. OCB allows objects to be distinguished on the basis of identity, so that for the example above it would be obvious to the user that *this.p1.father* and *this.p2* denote the same object, since they would be represented by the same independent window.

The Java debugger *jdb*, part of Sun's Java release, can also be used to browse object states at a program break-point. It makes no attempt to hide address information, and is command line based. In common with the CodeWarrior debugger, the states of the objects reachable by a thread can only be displayed if the running program has been compiled with a 'debug' flag set.

The O_2 object-oriented database system provides a graphical browser which allows the state of persistent and transient O_2 objects to be displayed. Its facilities are similar to those of OCB, except that it does not allow the display of inherited attributes to be controlled in the same way—OCB's style of handling this is described in Section 3.2. Indeed the authors do not know of any other browsers for inheritance-based languages which provide similar control[3].

2.2 Research Systems

Two previous object browsers with which the authors were involved are those for the persistent languages PS-algol [DB88] and Napier88 [KD90]. Similarly to OCB these can be invoked either from a running program or as a stand-alone application. The Napier88 browser displays object graph structures in the form of icons linked by directed edges.

Various other systems have provided graphical object browsers with which OCB shares some similarities in display style, for example Smalltalk-80 [GR83], Trellis/OWL [OHK87] and Cedar [Tei84].

2.3 OCB Design Aims

The main differences between OCB and the related work described above arose from the following specific design aims identified for OCB:

- to provide portability by implementing in Java;
- to allow control from running Java programs through a class interface and callback methods which allow the programmer to specify actions to be performed in response to user interaction;
- to support the visualisation of object sharing and identity, and to allow simple navigation between related objects and classes;
- to allow the graphical display format to be customised for specific classes, including the temporary hiding of superclass fields and methods.

3 OCB User Interface

3.1 Instances and Classes

This section uses the augmented definition of *Person* shown in Figure 2.

[3] OCB's style is not directly applicable to O_2 anyway, since O_2 supports multiple inheritance in contrast to Java's single inheritance.

```
public class Person implements Cloneable {
    private int      age;
    public String    name;
    public Address   address;
    public Person    mother, father;
    public Person[] children;
    public static int numberOfPeople=0;

    Person() {…}
    Person( String n, int a ) {…}

    public Date dateOfBirth() {…}
}

public class Address {
    public String street, town;
    public int  number;

    Address( int n, String s, String t ) {…}
}
```

Figure 2. Definition of class *Person* used in example

Figure 4 shows how OCB displays an instance of the class *Person*. The *Instance* pane on the left displays details of the instance's fields, while the *Class* pane on the right displays details of the class.

In the *Instance* pane the values of any fields with primitive types are displayed next to the field names, while the values of object type fields are represented by boxes containing the appropriate class names. The user may also customise the OCB display by defining alternative textual representations to be used for instances of particular classes, as described in Section 4.3. In the example a customised representation is used for strings, as illustrated for the *name* field.

The *Class* pane displays details of methods, constructors and fields, and the values of static fields. It also displays the superclass and any interfaces implemented by the class. Class and interface names are identified by underlining.

The various modifiers applicable to class members are indicated by coloured squares displayed next to member names. In the default colour coding public members are indicated by transparent squares and private members by red squares, hence the only visible squares in the example are for the *age* field. Colour coding is not used for the *static* modifier, since static members are already displayed in separate regions from non-static members. To assist the user in remembering the colouring scheme, a panel showing the current scheme is displayed whenever the user clicks a mouse button over one of the coloured squares, as illustrated in Figure 3.

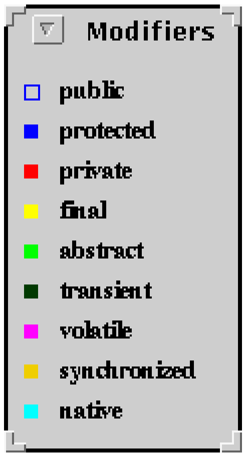
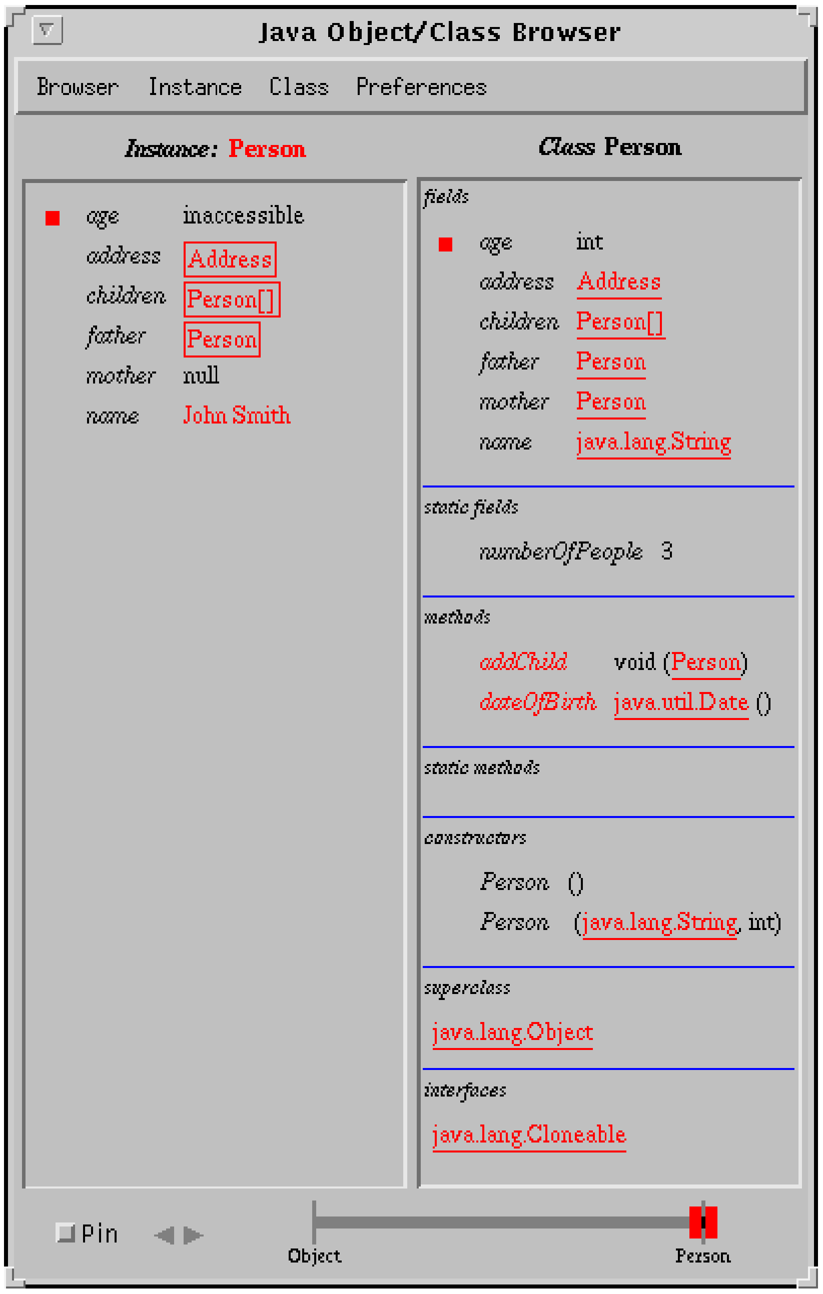


Figure 3. Modifier colour code prompt panel



Instance: Person

🟥	*age*	inaccessible
	address	Address
	children	Person[]
	father	Person
	mother	null
	name	John Smith

Class Person

fields

🟥	*age*	int
	address	Address
	children	Person[]
	father	Person
	mother	Person
	name	java.lang.String

static fields

numberOfPeople 3

methods

addChild	void (Person)
dateOfBirth	java.util.Date ()

static methods

constructors

Person	()
Person	(java.lang.String, int)

superclass

java.lang.Object

interfaces

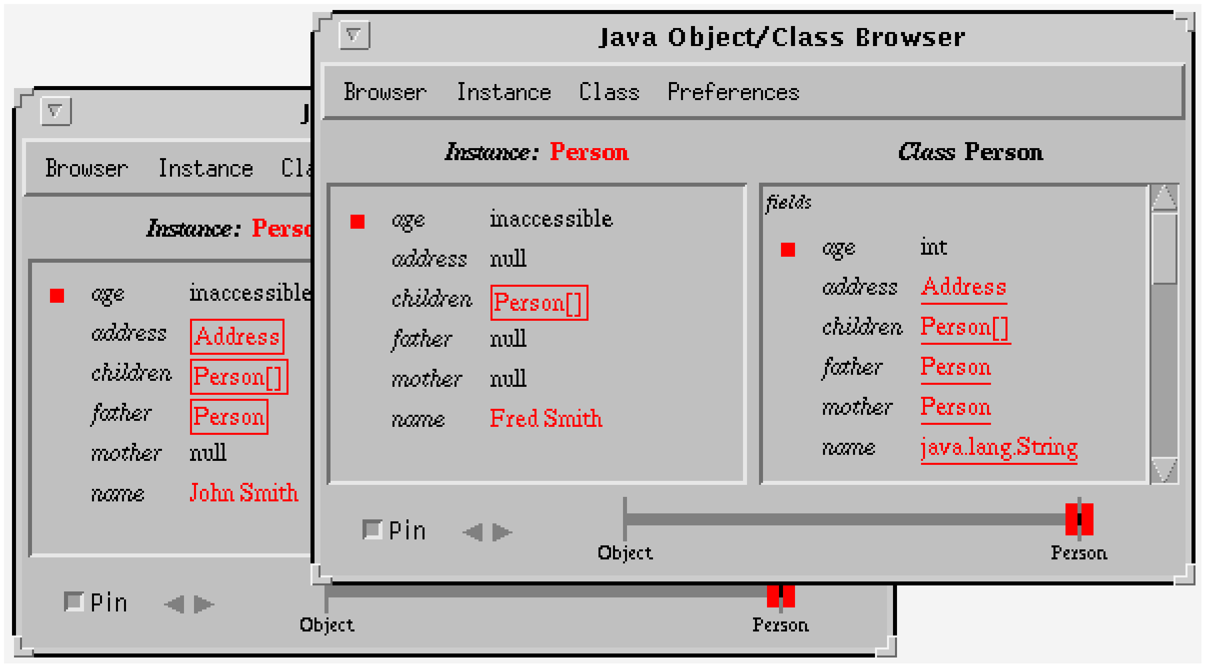
java.lang.Cloneable

Figure 4. Display of an object and its class

The user may click on various parts of the display to navigate an inter-object or inter-class link. Display regions which are sensitive in this way are differentiated by being displayed in red, as well as by the cursor which changes to a hand icon when it passes over them. Sensitive regions are used to denote values of object type fields, both in standard boxed format and customised string format, and class names which are underlined.

When the user clicks on an object value, the corresponding object and its class are displayed in turn. When the user clicks on a class name, the corresponding class is displayed. In the latter case, the OCB continues to display the current instance if the new class is the same as, a superclass of, or an interface implemented by the current class. Otherwise the new class is displayed on its own in the *Class* pane with no corresponding instance in the *Instance* pane.

The user may control whether or not a new window is created when new information is displayed. The default behaviour is that no new window is created, with the new information replacing the existing information within the current window. Alternatively, the user may pin a window by checking the *Pin* checkbox in its lower-left corner. While this is checked, new windows will be created whenever new information is displayed. New windows are initially unpinned. Each time a link is followed in an unpinned window the currently displayed object and class are pushed onto an internal stack. The user may navigate backwards and forwards in the stack using the arrow buttons next to the *Pin* checkbox. This style of browsing combines that used by web browsers such as Netscape Navigator [Net97] with the pinning mechanism of Sun Microsystems' Open Look graphical user interface [Sun89]. Figure 5 shows the display obtained by pinning the first *Person* instance and then clicking on its *father* field.



Instance: Person

age	inaccessible
address	null
children	Person[]
father	null
mother	null
name	Fred Smith

Class Person

fields

age	int
address	Address
children	Person[]
father	Person
mother	Person
name	java.lang.String

Instance: Perso

age	inaccessible
address	Address
children	Person[]
father	Person
mother	null
name	John Smith

Figure 5. Example of pinning

There is a one-one correspondence between pinned windows and the identities of the objects they are displaying. This means that the user can detect object sharing simply. For example, in Figure 5, if the *father* field of another object referring to *Fred Smith* is selected, the window displaying the *Fred Smith* object is brought to the front, rather than a new window being created.

Each window also contains menus which allow the user to select particular objects or classes to be browsed, adjust the colour coding used for modifiers, and to load and save preferences.

3.2 Viewing Inherited Members

In the examples shown earlier, only the fields and methods defined in the instance's most specific class were displayed, omitting members inherited from superclasses. For example in Figure 4 the methods inherited from class *Object* are not shown (and *Object* defines no instance fields). This is satisfactory if the methods inherited from *Object* are not of interest to the user, since the display remains relatively compact and focused on the application-specific class *Person*. The default behaviour of OCB is thus to display only the members of the most specific class, hiding any members defined in superclasses. These hidden inherited members may then be revealed under user control.

Alternatively a user may wish to view an instance through a superclass view, for example viewing a *Student* as a *Person*, requiring members defined in the most specific class to be hidden. Both requirements are met by allowing the user to set a *top* class and a *bottom* class to be applied to an OCB view, subject to the constraint *top*   *bottom*   *C*, where *C* is the most specific class of the displayed instance, and   means "is an ancestor class of, or is equal to".

The effect is, informally, to hide any members which are defined outside the part of the class hierarchy bounded by *top* and *bottom*. More precisely, a member of *C* is displayed if and only if it is defined or over-ridden in a class *X* such that *top*   *X*   *bottom*. Furthermore, if a member which is displayed is over-ridden in a subclass of *bottom*, it is displayed in the form applicable to *bottom* rather than to *C*. To illustrate this, consider the (somewhat contrived) class definitions in Figure 6:

```
public class Animal {
   public String name;
   public int age;
}

public class Person extends Animal {
   public Person spouse;
}

public class Student extends Person {
   public Student spouse;
   public int number;
}
```

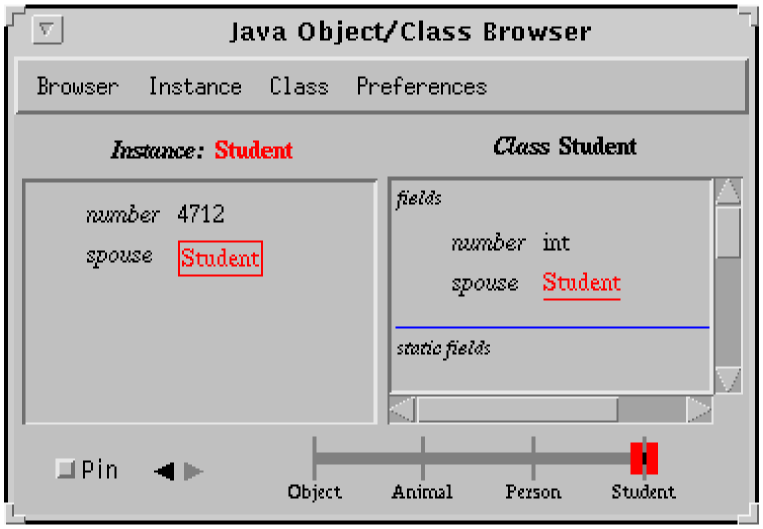
Figure 6. Example class hierarchy

This gives the class hierarchy *Object*   *Animal*   *Person*   *Student*. Table 1 defines the effects of setting *top* and *bottom* to various classes for a displayed instance of class *Student*.

top	***bottom***	***result***
Student	*Student*	The fields *spouse* and *number* are displayed, but no methods. This is the default setting (Figure 7).
Object	*Student*	The fields *name*, *age*, *spouse*, *number* are displayed, as are all of the methods inherited from *Object* (Figure 8).
Animal	*Student*	The fields *name*, *age*, *spouse*, *number* are displayed, but no methods (Figure 9).
Animal	*Person*	The instance is viewed as a *Person* and methods inherited from *Object* are hidden; the fields *name*, *age* and *spouse* are displayed, while *number* is hidden. The value of the *spouse* field is displayed as a *Person* rather than as a *Student* (Figure 10).

Table 1. Effects of various settings for *top* and *bottom*

The values of *top* and *bottom* can be set by manipulating the double-thumbed slider at the bottom of the display. The allowed positions on the slider range from class *Object* at the left, to the most specific class of the displayed instance at the right. The left thumb controls the setting of *top* and the right thumb the setting of *bottom*. Initially both are set to the most specific class. The example in Figure 7 shows both *top* and *bottom* set to *Student*: only members defined in that class are displayed.



Instance: Student

number 4712
spouse Student

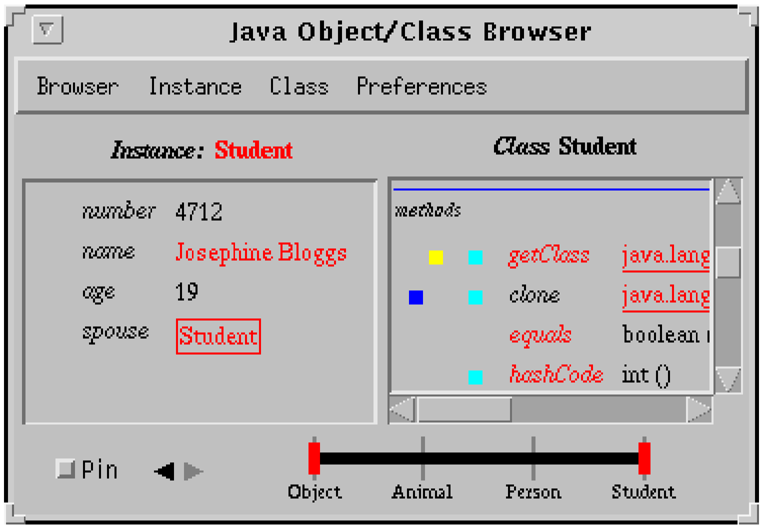
Class Student

fields
number int
spouse Student

static fields

Figure 7. Viewing members of class *Student* only

The example in Figure 8 shows *top* set to *Object* and *bottom* remaining set to *Student*. This makes visible all of the members available in class *Student*, i.e. those defined in any of its ancestor classes.



Instance: Student

number 4712
name Josephine Bloggs
age 19
spouse Student

Class Student

methods
getClass java.lang
clone java.lang
equals boolean
hashCode int ()

Figure 8. Viewing all members

The example in Figure 9 shows *top* set to *Animal* and *bottom* remaining set to *Student*. This filters out the members inherited from class *Object*.

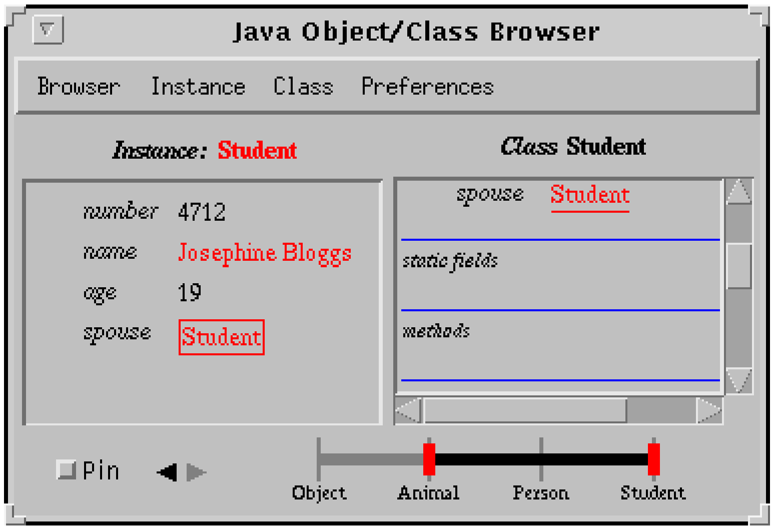


Figure 9. Viewing only members defined in and below class *Animal*

The example in Figure 10 shows *top* set to *Animal* and *bottom* set to *Person*. This filters out members inherited from *Object*, and also those defined in *Student*, effectively viewing the *Student* instance as a *Person*. Note that the value of the *spouse* field is now displayed as a *Person*.

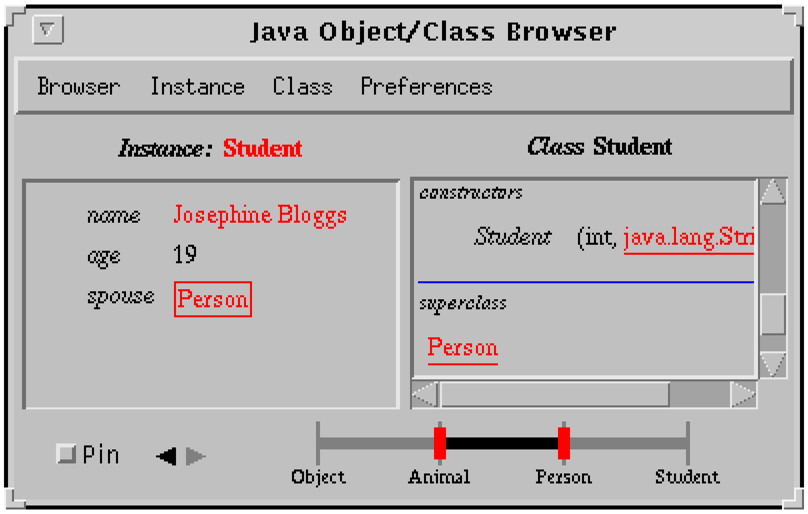
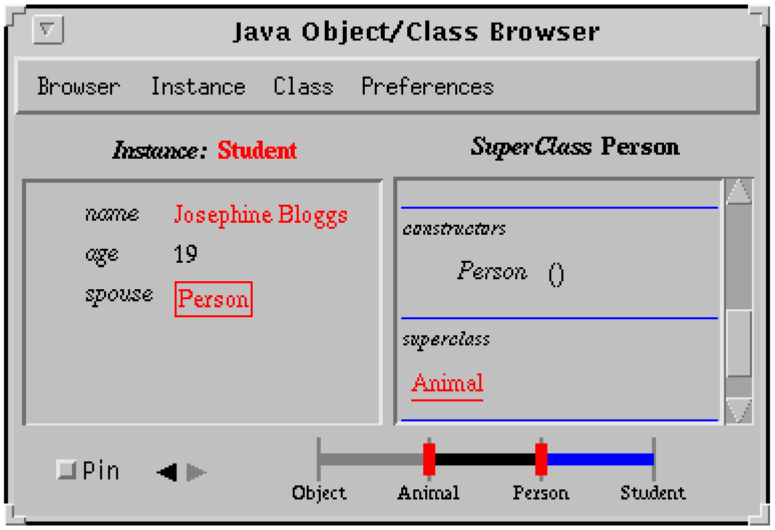


Figure 10. Viewing instance of *Student* as a *Person*

The example in Figure 11 shows the view obtained when the superclass link to *Person* visible in Figure 10 is selected. The instance display remains the same, while the superclass *Person* is displayed subject to the same filtering out of *Object* members. Since it would now result in an inconsistency if the user were allowed to set *bottom* to *Student*—because class *Person* is being displayed—the right slider thumb cannot now be moved past *Person*[4].



Instance: Student

SuperClass Person

name	Josephine Bloggs
age	19
spouse	Person

constructors

Person ()

superclass

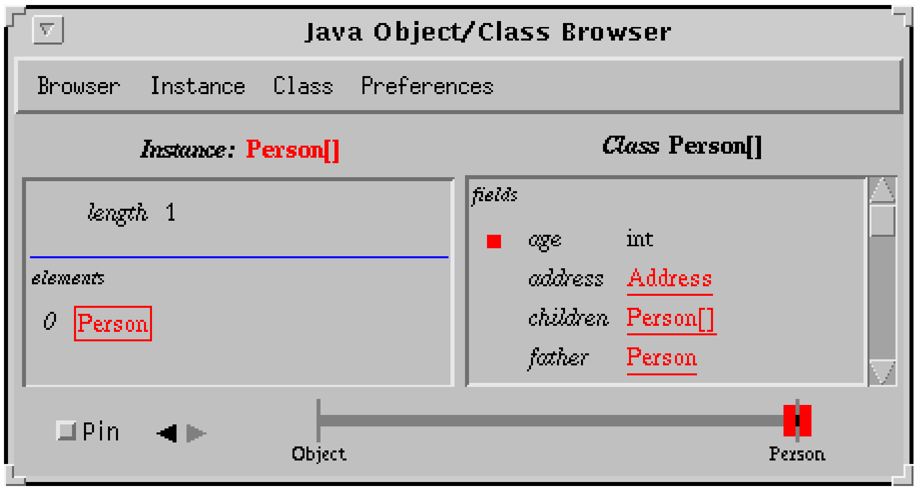
Animal

Figure 11. Viewing a superclass of *Student*

In the current implementation, the initial settings for *top* and *bottom* when an object is first displayed are the object's most specific class. As discussed in Section 6, it is planned in a future version of OCB to allow the default settings to be specified on a per-class basis.

3.3 Arrays

For array objects the *Instance* pane displays the *length* field and a scrolling list of the array components. The *Class* pane displays the class of the array elements. The example in Figure 12 shows the OCB display after the user has clicked on the box containing *Person[]* in the *children* field of Figure 4.



Instance: Person[]

Class Person[]

length 1

elements

0 Person

fields

🟥	*age*	int
	address	Address
	children	Person[]
	father	Person

Figure 12. Display of an array

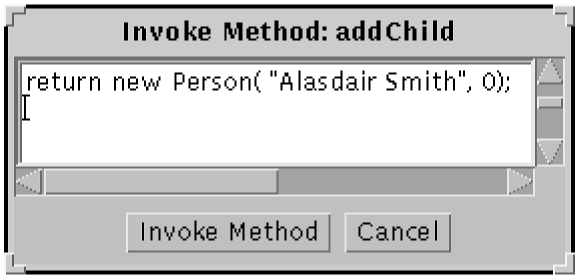
4 Although not visible in the monochrome screen-dump, the section of the slider to the right of *Person* is now drawn in a different colour to indicate this.

3.4 Invoking Methods

As well as passively displaying the current state of objects and their classes, OCB can also be used to interact with objects by invoking their methods. A method is invoked by clicking on the appropriate name in the *methods* region of the class pane. Depending on the method signature, parameter values for the method call may be required. OCB then displays a dialogue requesting the user to enter code defining the parameter values. For each parameter the user provides a fragment of code which, when executed, will produce a suitable parameter value. For example when the *addChild* method shown in Figure 4 is clicked on, a dialogue requesting a single parameter is displayed. The text area initially contains the code

```
return new Object();
```

which is then edited by the user, as illustrated in Figure 13.



```
return new Person( "Alasdair Smith", 0);
```

Figure 13. Parameter value dialogue for method invocation

When the user presses the *Invoke Method* button, OCB compiles and executes the code to create the required parameter value, which is then used to invoke the *addChild* method. If the code entered is invalid, or its execution does not produce an instance of *Person*, an error message is displayed. For methods with multiple parameters the dialogue contains a separate code area for each one.

In the current implementation, the parameter creation code entered by the user can only refer to classes which already exist and are accessible through the normal classpath. As described in Section 6, it is planned in a future version of OCB to allow the user to define new classes at the point of method invocation which can be used in creating the parameter values.

3.5 Browser Menus

Several OCB facilities are accessed via the menus. The *Browser* menu simply allows the current window to be closed. The *Instance* menu contains an entry which gives access to persistent objects; the details of the operation depending on which persistent Java system is being used. For PJama it displays a dialogue listing the names of the current persistent roots. When one is selected the corresponding object is displayed.

The *Class* menu provides access to persistent classes in a similar way; it also contains an entry which allows the user to enter any fully qualified class name, in response to which the corresponding class is displayed. Finally the *Preferences* menu contains entries which allow the user to alter the colour coding used to display modifiers, and to load and save colour and class customisation settings from a file or from the persistent store.

4 Program Interface

4.1 Invoking OCB

A Java program may create an OCB window by instantiating the class *ocb.OCB*, which implements the interface *ocb.OCBInterface* defined in Figure 14.

```
package ocb;
import java.awt.Container;

public interface OCBInterface {
    public void displayObject( Object anObject );
    public void displayClass( Class aClass );
    public Object getDisplayedObject();
    public Class getDisplayedClass();
    public void clear();
    public void clear( String paneName );
    public void close();
    public Container getOCBContainer();
    public void addCallback( Callback cb );
    public void removeCallback( Callback cb );
    public Callback[] getCallbacks();
    public void addCustomDisplay( Class theClass, DisplayAsString customDisplay );
    public void removeCustomDisplay( Class theClass );
}

public interface DisplayAsString {
    public String objectToString( Object theObject ) throws WrongClassException;
}

public interface Callback {
    public boolean callback( Object theObject, OCBInterface ocb );
    public boolean callback( Class theClass, OCBInterface ocb );
    public boolean callback( Object theObject, String name, OCBInterface ocb );
    public boolean callback( Class theClass, String name, OCBInterface ocb );
}
```

Figure 14. Interface *ocb.OCBInterface*

The constructors provided by class *ocb.OCB* are shown in Figure 15:

```
package ocb;
import java.awt.*;

public class OCB implements OCBInterface {

    /** Creates an OCB in a new frame of default size. */
    public OCB() {…}

    /** Creates an OCB in a new frame with the given position and size. */
    public OCB( Point position, Dimension size ) {…}

    /** Creates an OCB in the given container. */
    public OCB( Container parent ) {…}

    …
}
```

Figure 15. Constructors of class *ocb.OCB*

The first constructor creates an OCB which displays its information in a new *Frame* (an independent window), as does the second. The third constructor creates an OCB which displays information within a given existing *Container*. Once an instance of class *ocb.OCB* has been created, objects and classes may be displayed by invoking its *displayObject* and *displayClass* methods.

4.2 Callbacks

The user may register callback methods with an OCB instance to be called whenever an object, class or member is selected. When used in a persistent Java system these callbacks persist between sessions. The example in Figure 16 shows the registration of a callback which writes out a message whenever a class member is selected in the OCB:

```
import ocb.*;

public class MyCallback implements Callback {
    public boolean callback( Object theObject, OCBInterface ocb ) { return true; }
    public boolean callback( Object theObject, String name, OCBInterface ocb ) {
        return true; }
    public boolean callback( Class theClass, OCBInterface ocb ) { return true; }
    public boolean callback( Class theClass, String name, OCBInterface ocb ) {
        System.out.println( "class member " + name + " selected" );
        return false;
    }
}

public class Test {
    public static void main( String args[] ) {

        OCB myOCB = new OCB();

        myOCB.addCallback( new MyCallback() );

        Person john = new Person( "John Napier", 447 );
        myOCB.displayObject( john );
    }
}
```

Figure 16. Registering a callback

The *ocb.Callback* interface provides methods to be called when an object, class, object member or class member is selected. Each method returns a boolean which specifies whether other registered callbacks should also be called on this occasion.

4.3 Customising the Display

The user may associate a customised display method with a particular class. This is then used by OCB whenever it displays a field value of that class. The method takes as its parameter the class instance and returns a string representing that instance, which is then displayed in place of the default boxed class name.

Customised display methods are registered as instances of a class that implements the interface *DisplayAsString*:

```
public interface DisplayAsString {
    public String objectToString( Object theObject ) throws WrongClassException;
}
```

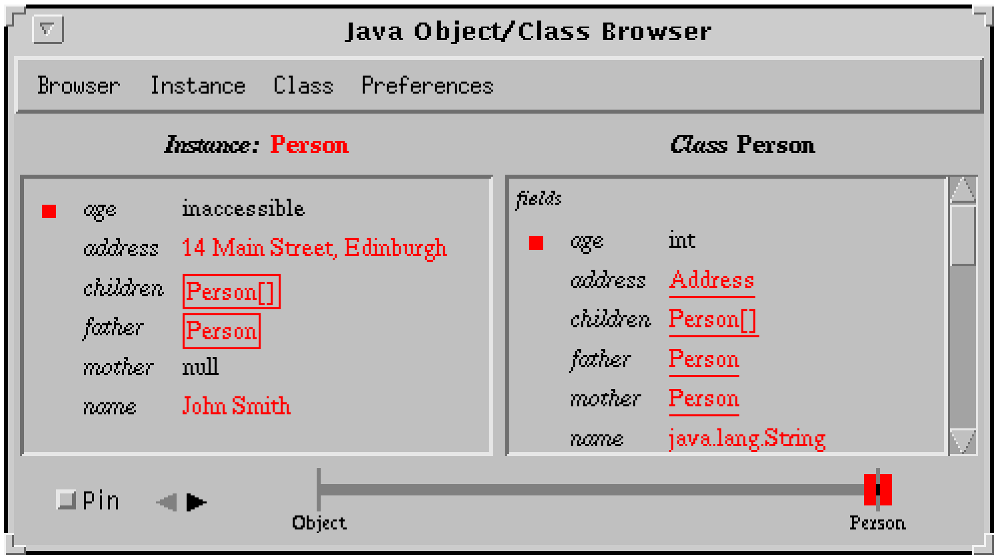
Figure 17 shows how a custom display method for class *Address* could be registered with an *OCB* instance.

```
import ocb.*;

public class AddressDisplayer implements DisplayAsString {
    public String objectToString( Object theObject ) throws WrongClassException {
        if ( theObject instanceof Address ) {
            Address objectAsAddress = (Address) theObject;
            return String.valueOf( objectAsAddress.number ) + " " +
                objectAsAddress.street + ", " + objectAsAddress.town;
        } else throw new WrongClassException( "class Address expected" );
    }
}
```

```
public class Test2 {
    public static void main( String args[] ) {

        OCB myOCB = new OCB();

        try {
           myOCB.addCustomDisplay( Class.forName( "Address" ), new AddressDisplayer() );
        } catch (ClassNotFoundException e) {
           System.out.println( "Couldn't get class for Address" );
        }

        Person john = …   // Create Person instance as in Figure 2.
        myOCB.displayObject( john );
    }
}
```

Figure 17. Customisation of OCB

Figure 18 shows the display of the *Person* object with the customised display of the *Address* field.



```
age       inaccessible
address   14 Main Street, Edinburgh
children  Person[]
father    Person
mother    null
name      John Smith
```

```
fields
age       int
address   Address
children  Person[]
father    Person
mother    Person
name      java.lang.String
```

Figure 18. Example customised OCB display

The motivation for this somewhat cumbersome mechanism is to allow OCB customisation without the need to alter existing classes. A second simpler mechanism may be used if display by OCB can be taken into consideration at the time of class definition: where a class overrides the method *toString*, which is inherited from class *Object*, that method will be used to produce the custom string. Figure 19 shows the definition of such a subclass of *Address*. Instances of this class will be displayed in the customised form without the need to register with the *OCB* instance.

```
public class CustomAddress extends Address {
    CustomAddress( int a, String s, String t ) {
        super( a,s,t );
    }
    public String toString() {
        return String.valueOf( number ) + " " + street + ", " + town;
    }
}
```

Figure 19. Class with customised display method

5 Implementation

OCB is implemented completely in Java and requires release JDK 1.1 or later. It uses the core reflection classes in *java.lang.reflect* to discover details of the classes and objects passed to it. The graphical display is constructed using the *awt* toolkit.

In common with many Java programmers, the use of *awt* was often a frustrating experience, particularly due to the inconsistencies between implementations on various platforms. The core reflection classes, in contrast, were found to be particularly easy to use. They appear to be well designed and to provide all the functionality needed to display object states. It could be argued, though, that for full reflection the classes should allow the retrieval of method source code, although this would present significant implementation difficulties. A security mechanism would also be required in order to suppress source code in situations where public access to it was not desirable.

The core reflection classes do not, however, support the introduction of new code into the running system, the need for which was described in Section 3.4. This requires dynamic access to the Java compiler, that is, an executing Java program needs to be able to invoke the compiler, passing it a source program representation and receiving compiled classes in return. Some techniques for achieving this are described in detail in [KMC+97]. Briefly, given the core reflection classes currently provided this involves either forking an operating system process to perform the compilation, or relying on the availability of a Java compiler implemented in Java which can be invoked directly. The details of the former option depend on the platform, and it may not even always be possible. The latter option depends on access to non-core classes which may not always be present.

The need for dynamic compilation means a degree of platform dependence. Currently OCB fails gracefully by interrogating its environment before attempting to perform compilation; if it detects that compilation is not possible it simply disables dynamic method invocation. A more satisfactory solution would be for compilation support to be added to the core reflection classes. For example a class *Compiler* could be provided, removing the need for ad-hoc solutions:

```
public class Compiler {
    public Class[] compile( Source source, Class[] imports ) throws …
    …
}
```

Here the method *compile* takes a source code representation, which could be a string or a more structured representation, and an array of classes used by the code, and returns an array of compiled classes.

At the time of writing (September 1997) OCB has been fully implemented as described in this paper, with the exception of customising colours, and the loading and saving of customised settings. OCB is freely available at the following URL:

`http://www-ppg.dcs.st-and.ac.uk/Java/OCB/`

6 Further Work

6.1 Static Visualisation

Several relatively minor enhancements are planned for the next version of OCB:

- to allow customisation of the default settings for the *top* and *bottom* class when an instance is first displayed: allow a top/bottom pair to be specified for a particular instance or for all instances of a particular class;

- to allow arbitrary new classes to be defined at the time that an object method is invoked: these classes could then be used in the creation of the parameter values to be passed to the method. For example a subclass of *Person* called *Child* could be defined dynamically and a new instance of it passed to the *addChild* method.

- to allow the user to update public instance fields in a similar manner to method invocation, and to provide a facility to ‘bookmark’ objects for convenient re-visiting.

The issue of assisting the user in discerning the structure of linked object graphs was raised earlier. The currently implemented tool does support this to a limited extent, in that when pinned displays are used there is a one-to-one mapping between objects and OCB windows. However this becomes unsatisfactory when more than a few objects are displayed simultaneously. One possible enhancement is to provide an additional ‘overview’ window containing iconic representations of the objects encountered, connected by edges showing the inter-object references, in the style of the PS-algol and Napier88 browsers [DB88, KD90]. Other possibilities include options to display all persistent roots, and to display the results of queries executed over the persistent store.

The class customisation mechanism described in Section 4.3 supports only customised string representations for particular classes. It would be relatively simple to extend this to allow the specification of graphical representations which would be displayed in place of the standard boxed class name representation, perhaps by renaming the interface *DisplayAsString* to *CustomiseDisplay*, with the methods:

```
public interface CustomiseDisplay {
    public String objectToString( Object theObject ) throws WrongClassException;
    public Image  objectToImage( Object theObject ) throws WrongClassException;
}
```

In this case the *objectToImage* method would be called if the *objectToString* method returned *null*. Another possibility would be for instances of the customised class appearing in fields to be displayed using nested OCB windows, so that the entire field value would be displayed within the parent window. This option could be indicated by having both methods return *null*.

As mentioned in the previous section, a possible addition to the core reflection classes would be the ability to obtain source code from a given method representation. If this were supported the OCB display could then be adapted to show the source of each method in the class pane. Clearly this would raise security issues with regard to controlling which users were allowed to access source code; it would not be acceptable to allow universal access.

One of the issues considered in the design of OCB was whether the display of an object and its members should respect the access modifiers specified in its class definition, for example whether a *private* member should be visible. More generally, should the user be able to discover or affect any aspect of the Java system that they would not be able to do by writing an appropriate program? However, it turned out to be a non-issue: because OCB is implemented solely in Java, by definition it can only perform actions permitted by the language rules. While giving a clean and relatively safe tool—a user cannot do anything using OCB that they could not do anyway—this is too restrictive for debugging purposes. For example it would be useful if all the members defined in a certain package were accessible by OCB while the package was under development. This suggests a refinement of the core reflection facilities to support over-riding of language security mechanisms in limited circumstances.

6.2 Dynamic Visualisation

The current functionality of OCB is to display a snapshot of the state of an object at one particular moment. It could also be enhanced to provide an active monitoring capability, whereby an object could be polled regularly by an OCB instance, with any changes being reflected in an updated display window. This would make OCB more useful as a debugging tool.

As mentioned earlier, the display of persistent objects and their classes is only part of what is necessary for the visualisation of a persistent Java system: the programmer may also need to examine the states of the threads executing at a particular time. Since threads are Java objects it is possible to pass a thread to the current OCB implementation for display. However this does not reveal very much interesting information since most of the thread’s state is not publicly accessible.

An obvious avenue for development of OCB is to extend it with symbolic debugging capabilities, allowing threads to be started and stopped, break-points set etc. Clearly given the current core classes this would have to be implemented using a non-standard JVM. More interestingly, perhaps the core reflection classes could be augmented to allow sufficient safe introspection into the dynamic state of the system for such facilities to be implemented in “100% Pure Java™”. There is considerable existing work on Meta-Object Protocols to support this style of reflection in other languages [KRB91, MJD96].

6.3 Implementation

The efficiency of the implementation could be improved by caching the results of various calculations internally. Each time an object is displayed the structure of its class is traversed, involving a significant number of calls to the core reflection classes. The window layout is then calculated on the basis of that structure. Some results from both of these stages are dependent only of the class of the object and could thus be cached on a per-class basis. Similarly, it is quite common for the user to press the *back* arrow to return to the previously viewed object. In the current implementation this results in a complete recalculation, whereas the complete display state could be cached, keyed by the identity of the object. Where OCB is used with a persistent Java system, both forms of cache could persist across multiple user sessions.

7 Acknowledgements

This work is partially supported by the EPSRC under grant GR/J 67611 “Delivering the Benefits of Persistence to System Construction” and by a gift from Sun Microsystems. We thank Malcolm Atkinson, Laurent Daynès, Mick Jordan, Dave Munro, Michael Van De Vanter, the anonymous reviewers, and the participants of the PJW2 workshop for their helpful suggestions.

8 References

[ADJ+96] Atkinson, M.P., Daynès, L., Jordan, M.J., Printezis, T. & Spence, S. “An Orthogonally Persistent Java™”. SIGMOD Record 25, 4 (1996) pp 68-75.

[DB88] Dearle, A. & Brown, A.L. “Safe Browsing in a Strongly Typed Persistent Environment”. Computer Journal 31, 6 (1988) pp 540-544. URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1988.html#safe.browsing*.

[DHF96] Dearle, A., Hulse, D. & Farkas, A. “Operating System Support for Java”. In Proc. 1st International Workshop on Persistence for Java, Glasgow (1996).

[GN96] Garthwaite, A. & Nettles, S. “Transactions for Java”. In Proc. 1st International Workshop on Persistence for Java, Glasgow (1996).

[GR83] Goldberg, A. & Robson, D. **Smalltalk-80: The Language and its Implementation**. Addison Wesley, Reading, Massachusetts (1983).

[KD90] Kirby, G.N.C. & Dearle, A. “An Adaptive Graphical Browser for Napier88”. University of St Andrews Technical Report CS/90/16 (1990). URL: *http://www-ppg.dcs.st-and.ac.uk/Publications/1990.html#napier.browser*.

[KMC+97] Kirby, G.N.C., Morrison, R., Connor, R.C.H. & Stemple, D.W. “Linguistic Reflection as a Paradigm for Program Generation in Java”. Submitted for publication (1997).

[KRB91] Kiczales, G., des Rivières, J. & Bobrow, D. **The Art of the Metaobject Protocol**. MIT Press, Cambridge, Massachusetts (1991).

[MCK+96] Morrison, R., Connor, R.C.H., Kirby, G.N.C. & Munro, D.S. “Can Java Persist?”. In Proc. 1st International Workshop on Persistence for Java, Glasgow (1996) pp 34-41. URL: *http://www-ppg.dcs.st-and.ac.uk/ Publications/1996.html#java.persistence*.

[Met97] Metrowerks Inc “Metrowerks CodeWarrior”. (1997). URL*: http://www.metrowerks.com/*.

[MJD96] Malenfant, J., Jacques, M. & Demers, F. “A Tutorial on Behavioral Reflection and its Implementation”. In Proc. Reflection 96, San Francisco (1996) pp 1-20.

[Net97] Netscape Communications Corporation “Netscape Navigator”. (1997). URL*: http://www.netscape.com/*.

[OHK87] O’Brien, P.D., Halbert, D.C. & Kilian, M.F. “The Trellis Programming Environment”. In Proc. International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’87), Orlando, Florida (1987) pp 91-102.

[Sun89] Sun Microsystems **Open Look™ Graphical User Interface Functional Specification**. Addison-Wesley, Mountain View, California (1989).

[Sym96] Symantec “Symantec Visual Café”. (1996). URL*: http://www.symantec.com/*.

[Tei84] Teitelman, W. “A Tour Through Cedar”. IEEE Software, April (1984) pp 44-73.

Analysing, Profiling and Optimising Orthogonal Persistence for Java[*]

Position Paper for the
Second International Workshop on Persistence and Java
California, August 1997

Quintin Cutts
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland
quintin@dcs.gla.ac.uk

Antony L. Hosking
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA
hosking@cs.purdue.edu

August 22, 1997

Abstract

Persistent systems manage main memory as a cache for efficient access to frequently-accessed persistent data. Good cache management requires some knowledge of the semantics of the applications running against it. We are attacking the performance problems of persistence for Java through analysis, profiling, and optimisation of Java classes and methods executing in an orthogonally persistent setting. Knowledge of application behaviour is derived through analysis and profiling, and applied by both a static bytecode transformer and the run-time system to optimise the actions of Java programs as they execute against persistent storage. Our prototype will unify distinct persistence optimisations within a single optimisation framework, deriving its power from treatment of the entire persistent application, consisting of both program code and data stored in the database, for *whole-application* analysis, profiling and optimisation.

Keywords: persistence, Java, bytecode, program analysis, dynamic profiling, optimisation

1 Introduction

Orthogonally persistent programming languages [Atkinson and Buneman 1987; Atkinson and Morrison 1995] provide improved support for the design, construction, maintenance and operation of applications that manage large bodies of long-lived, shared, structured data. In spite of this, there is continued mainstream resistance to languages with orthogonal persistence due to a perception that they cannot deliver performance to match that of traditional programming languages. We believe that performance problems associated with persistence can be dealt with through extension of traditional program analysis and optimisation techniques to encompass optimisation of persistent programs, as well as new techniques based on execution profile feedback [Hölzle and Ungar 1994; Grove et al. 1995], specialisation, customisation and other partial evaluation [Chambers and Ungar 1989; Chambers et al. 1989; Chambers and Ungar 1990; Chambers 1992; Dean et al. 1995; Dean et al. 1995; Consel and Danvy 1993; Jones et al. 1993], and dependence analysis and loop restructuring [Wolfe 1996]. We also believe that analysing, profiling and optimising in a persistent setting has benefits even for generic optimisations not directly related to persistence.

1.1 Orthogonal persistence

The language principles of *transparency* and *orthogonality* have been repeatedly articulated [Atkinson and Morrison 1995; Moss and Hosking 1996] as important in the design of persistent programming languages, enabling the full

[*]Java is a trademark of Sun Microsystems, Inc.

power of the persistence abstraction. Transparency means that access to persistent objects does not require explicit calls to transfer them between stable store and main memory. Thus, a program that manipulates persistent (or potentially persistent) objects looks similar to a program concerned only with transient objects. To support this, the program's compiled code or interpreter, and the persistent run-time system, contrive to make objects resident in memory on demand, much as non-resident pages are automatically made resident by a paged virtual memory system.

Treating persistence as *orthogonal* to type encourages the view that a language can be extended to support persistence with minimal disturbance of its existing syntax and store semantics. Thus, programmers need to add little to their understanding of the language in order to begin writing persistent programs. A common way to achieve orthogonal persistence is by treating persistent storage as a stable extension of the dynamic allocation heap. This allows a uniform and transparent treatment of both transient and persistent data; persistence is orthogonal to the way in which objects are defined (i.e., their types), allocated, and manipulated in the heap.

Implementation of transparent, orthogonal persistence requires two underlying mechanisms: *residency checks* to trigger retrieval of non-resident objects from stable store to main memory for read access, and *update checks* to track their modifications (including acquisition of write locks as necessary for concurrency control) for eventual propagation back to stable storage. The application of these checks constitutes a *read barrier* and *write barrier* for persistence, respectively, since execution is suspended until the checks and any consequent action is complete. Efficient implementation of these mechanisms, and sensible object caching policies, are the keys to performance for persistence.

1.2 Performance

Cattell [1994] (p. 268) mentions two performance tenets for an object data management system (ODMS):

T26: "*Minimal access overhead.* An ODMS must minimise overhead for simple data operations, such as fetching a single object."

T27: "*Main-memory utilisation.* An ODMS must maximise the likelihood that data will be found in main memory when accessed. At a minimum, it should provide a cache of data in the application virtual memory, and the ability to cluster data on pages or segments fetched from the disk."

We are addressing both of these issues, which can significantly affect performance: reducing access overhead leads to persistent programs whose performance approaches that of their non-persistent counterpart, since the persistent program will have negligible overhead when operating entirely on memory-resident data; improving main-memory utilisation reduces I/O which is the main performance barrier for persistence.

1.2.1 Minimal access overhead

There is an inherent tension between the principle of orthogonality and efficient implementation of that model. The abstraction of orthogonal persistence obscures the performance disparity between fast cache/main memory and slow secondary storage. Thus, naive implementations of orthogonal language designs can lead to inefficiencies. For example, a given object reference in an orthogonal persistent program may target either a resident or non-resident object. Before the object can be manipulated through that reference a residency check must be performed to make sure the object is available in memory. A naive implementation would encode each object reference using its target object's disk-based persistent identifier (PID). Every time an object reference is traversed, the PID must be mapped to a pointer to the target object in memory (with a call to make the object resident if it is not already). Residency checks on transient or already-resident persistent objects are unnecessary, so long as those objects remain resident. Eliminating the check and using a direct memory pointer to refer to such objects is more efficient, since repeated object access can be achieved through fast main-memory addressing as opposed to slow PID translation. We are developing global data flow analyses (both intra- and inter-procedural) to discover and eliminate redundant residency and update checks and to influence conversion of PIDs to direct pointers (i.e., "swizzling").

1.2.2 Main-memory utilisation

Persistence transparency precludes explicit control by the programmer over the physical transfer of objects between main memory and persistent storage. Rather, objects are automatically retrieved as needed by the program and cached in memory until evicted by the cache replacement policy. However, I/O latencies are so high that the timing of fetch requests, the way in which objects are clustered for storage and retrieval, and the policy for object replacement all have a significant impact on the performance of a persistent program.

Note that persistent systems face a much more difficult task of cache management than file-based systems. A persistent store contains highly-structured complex objects that are traversed in relatively random orders with respect to their storage on disk, rendering the low-level clustering and prefetching strategies of traditional file systems ineffective.

We are devising techniques for automatic derivation of strategies for prefetching, replacement, and clustering of objects, based on static program analysis, dynamic profiling, and direct consideration of the schema and physical structure of the target database.

2 Optimisation

Our goal is to devise, implement, and evaluate optimisations for the orthogonal persistence for Java (OPJ) prototype [Atkinson et al. 1997; Atkinson et al. 1996] being developed jointly by Sun Laboratories and the University of Glasgow. We divide candidate optimisations into those that are:

enabled by persistence: optimisations that target generic program improvement and are enabled by analysis, compilation and execution in a persistent setting; and

enabling for persistence: optimisations specifically targeting reduction of the persistence-specific overheads of execution

We now describe in more detail the optimisation strategies we plan to explore.

2.1 Persistence-enabled optimisations

Java's dynamic, late-binding, object-oriented nature provides many opportunities for optimisation, such as those pursued for Self [Ungar and Smith 1987] and other OO languages: e.g., specialisation, customisation, method splitting, cloning and inlining, which all act to reduce the overhead of dynamic method invocations. Similar optimisations have been applied in other settings such as Modula-3, based on "whole-program" analysis [Fernandez 1995; Diwan et al. 1996; Diwan 1997].

Java's model of network code distribution dictates a standard class file format for network transmission, with code taking the form of architecture-neutral bytecode instructions for the Java Virtual Machine (VM) [Lindholm and Yellin 1996]. In such a setting Java source code is never transmitted to clients: they see only the VM bytecodes, which they may interpret with a virtual machine, or translate to native code using a "just-in-time" (JIT) compiler for direct execution. Without any control over the initial server-side compilation phase that translates source code to bytecodes, the only opportunity for clients to influence code quality is at the time of JIT compilation after receipt of the Java bytecodes. However, there is an inherent tradeoff to optimisation during JIT compilation, since the goal is usually to begin executing code as soon as possible after it has been received, whereas many optimisations depend on a time-consuming analysis of the code. Indeed, Self-93 [Hölzle and Ungar 1996] retreated from many of the more expensive analyses and optimisations of Self-91 [Chambers 1992] simply because they were too expensive for fast turnaround of JIT compilations.

It is here that persistence can provide assistance, allowing more aggressive off-line analyses and optimisations of both bytecode and native code. Analysis and optimisation phases are greatly simplified when all code, data, profiles and other measurements are retained within a persistent store. Advantages accrue from the accumulation of large bodies of types, analysis results and both unoptimised and optimised bytecode and native code in the persistent store. JIT compilers can take advantage of such analysis to generate more efficient code, but without the overhead of on-line analysis. In this sense, persistence changes the game because there is no longer a need to trade off performance for responsiveness.

OPJ calls for Java classes (including code) to be interned as part of the persistent store. When loading a Java class, OPJ first checks to see if the class is already available in persistent storage; if so then the interned code can be used instead of loading the external bytecoded code representation. Needless to say, interned persistent code can take whatever form is convenient: standard bytecodes, extended bytecodes, or native code. Whatever form is assumed for interned code, the expensive analyses that drive candidate optimisations can be performed on the stored code during periods when the persistent system is inactive. Moreover, the interned body of code can approximate the notion of ”whole-program” that has been exploited for optimisation in other settings. Naturally, open-world assumptions must apply where necessary, either to avoid recompilation of vast quantities of code when new classes are interned and existing classes are modified, or to trigger re-optimisation and re-compilation if appropriate. Such interaction between optimisation and system evolution is a promising area for further investigation, as is the concept of “active” for a persistent system.

Another interesting twist is that in a persistent setting code can be specialised with respect to the stored data (both the instances and the instantiated classes). Thus we are presented with a classic opportunity for partial evaluation of code with respect to the database schema, its physical structure, and the stored instances. The schema consists of the set of types of all objects actually stored in the database, as opposed to described in the code. Where there are differences between the types described in the code and the database schema we have an opportunity for optimisation based on type hierarchy analysis [Dean et al. 1995; Diwan et al. 1996; Diwan 1997], which bounds the set of procedures a method invocation may call by examining the object-oriented type hierarchy for method overrides. It is these overrides that result in dynamic method invocations at so-called polymorphic call sites. Although the program’s type hierarchy may imply a polymorphic call is necessary, the database schema’s hierarchy might indicate that only a monomorphic call is required, since objects of only one of the types possible in the polymorphic call can actually be allocated or encountered in the database. Thus, the indirect polymorphic call can be converted to a direct monomorphic call.

Specific to Java there are remaining interesting problems for optimisation that also arise out of its inherent dynamism, where the body of code in the system can evolve over time. On the one hand, persistence helps by enabling complex, long-running analyses in a dynamic object-oriented environment, while on the other hand, static analysis can become obsolete at any instant. It seems that this is very different from prior settings for optimisation.

2.2 Persistence-enabling optimisations

These are our primary focus. Whereas persistence-enabled optimisations take and extend to a persistent setting previous work in the area of optimisation of non-persistent aspects of execution, persistence-enabling optimisations directly address the performance of persistence features. Persistence optimisations strive for minimal access overhead for persistent data and improved main-memory utilisation as described above. Minimising access overhead means that simple operations can be performed on persistent data with minimal overhead (ideally, there should be little or no performance penalty for manipulating resident persistent data compared to transient data). Good main-memory utilisation means maximising the likelihood that data will be found in memory when accessed, through caching of data in the application virtual memory and clustering of related data on pages or segments fetched from disk.

Persistence optimisations are driven by information about the *co-residency* of particular objects. We write $i \rightarrow_p j$ to indicate that whenever an object i is resident so also j will be resident, with probability p. We write $i \rightarrow j$ if $p = 1$. If $i \rightarrow j$ and i is made resident then references to j can be swizzled to direct memory pointers. Residency check elimination assumes that the run-time system will respect co-residency assumptions. By default, we assume $i \rightarrow i$

(once resident an object will stay resident so long as “live” swizzled pointers to it exist). Thus, residency checks are idempotent, and redundant checks can be eliminated. Further, given $i \rightarrow j$, references from i to j can be traversed without checks. Similarly, update checks and lock acquisition for transaction concurrency control are idempotent within transaction boundaries, and can be optimised in similar fashion.

Co-residency is transitive only if all weights p have value 1. One can think of adjusting the co-resident reach of a given “handle” on a persistent data structure by combining weights and applying a threshold. Suppose $i \rightarrow_p j \rightarrow_q k \rightarrow_r l$, specifying that j should be co-resident with i with weight p, k with j with weight q, and l with k with weight r. Assume $0 \leq p, q, r \leq 1$. Then, given a ”handle” on i, and some threshold t, j will be co-resident if $p > t$, k if $pq > t$ and l if $pqr > t$. That way, a given handle can modulate its ”reach” using the threshold combined with the weights on the edges of the data structure.

Note that when executing code that is compiled to take advantage of co-residency assumptions the object cache manager is required to guarantee residency of certain objects. In a multi-threaded environment, these guarantees require careful management to avoid severe performance degradation. Section 2.4 considers this issue in more detail.

Swizzling can also be driven by co-residency information. If $i \rightarrow j$ holds then we might as well swizzle references to j contained in i. Not only are checks on those references redundant, but the link can be followed with minimal overhead. Prefetching and clustering can be driven similarly: when fetching i we might as well issue a request for (i.e., prefetch) j at the same time; if j is also clustered with i then further I/O is unnecessary.

As implied earlier, co-residency information can be acquired in several ways. Static data-flow analysis can approximate the ”storage profile” of a piece of code which can be refined through dynamic profiling. The information might be encoded as constraints that must be obeyed by the run-time system before the code (compiled in light of the constraints) can execute, or more globally, a collection of related types can be annotated by the system to indicate the global storage profile of their instances, with code optimised in light of those global annotations [Moss and Hosking 1995]. Other static residency information can be gleaned from knowledge of the object-oriented execution paradigm of Java: the target of any dynamic method invocation (i.e., the “this” argument) must be resident in order to dispatch the method. Thus, the bodies of those methods can access the fields of the target object without residency checks. This observation led to elimination of 86-99% of residency checks in a prototype persistent Smalltalk system [Hosking 1997]; we expect similar improvements for OPJ programs.

To sum up, the principal objective is to focus on the unique setting persistence gives for optimisation, both persistence-enabled and persistence-enabling. There is a strong connection among persistence optimisations such as residency-/update-/lock-check elimination, clustering, prefetching, and swizzling, centered on co-residency analyses and dynamic profiling. Prior work in these areas has not recognised the commonalities that arise in each of them, and our goal is to unify the approaches in a common framework of program analysis and execution profiling. Recasting previous optimisations (e.g., from Self) in this framework is a by-product of this objective.

2.3 Analysis and optimisation framework

In order to understand our approach it is necessary to consider the current architecture of OPJ. Without changing the standard Java language, the OPJ team has made extensions to JavaSoft’s Java Development Kit (JDK) implementation of the VM to support transparent, reachability-based persistence for Java. Stable storage is currently provided as a layer below the standard JDK VM and its garbage-collected volatile heap. Attempts to access non-resident objects are trapped by the OPJ VM. The resulting object faults are serviced by calls to the storage layer to fetch, swizzle as necessary, and cache a copy of the target object in memory, before execution can proceed.

We are focusing on persistence optimisations, but intend also to consider the synergies between persistence and generic optimisations, particularly those able to take advantage of analysis and profile information stored along with the persistent store schema (e.g., the instantiated types in the store) and the store’s physical organisation (e.g., clustering, indexes, etc.).

We plan to analyse and optimise to an extended VM bytecode set from the standard bytecodes. Some simple analyses and static optimisations might be performed by an extended VM itself when the code is interned (e.g., replacing slow bytecodes with their quick forms as the current JDK VM already does), but more complicated analyses and transformations will take place off-line. Short of native-code compilation there is much we can do to improve the performance of OPJ. We will modify the OPJ VM to incorporate additional non-standard bytecodes for use in optimised interned code. These bytecodes will isolate and expose the costlier operations of persistence. Static data-flow analysis and optimisation, allied with execution profile feedback, will determine where these bytecodes are redundant and so can be eliminated.

As a concrete example, consider residency checking. Currently, OPJ residency checks are performed on every ”unhandle” operation in the VM. By removing the checks from the internal unhandle operation and exposing them as separate bytecodes we can eliminate those found to be redundant. (Similar strategies can be applied for clustering, prefetching, and elimination of unnecessary update checks and lock requests [Hosking and Moss 1991; Moss and Hosking 1995; Hosking 1995; Hosking and Moss 1995; Hosking 1997].)

2.4 Profiling and run-time support

In addition to static analysis we will also perform significant dynamic profiling, since Java's execution allows extreme forms of dynamism (e.g., injection of classes unknown at compile-time). The results of static analysis will be both updated code sequences and auxiliary data structures capturing both analysis information and frameworks for dynamic profiling. In the persistent setting it is possible for the analyser itself to plant profiling code and attach appropriate data structures to a targeted class object to record profile information associated with that class. This is in line with previous approaches to dynamic persistence optimisations [Cutts et al. 1994].

As well as profiling localised to certain code regions, profiling of a more global nature will also be performed. For example, the cache manager, thread scheduler and lower-level store mechanisms maintain a global view of data access spanning multiple executing threads. Static code/class analysis alone cannot capture this global behaviour.

For any particular optimisation, some adjustment will be required within the OPJ VM. Particular examples are implementations for new bytecodes, and adjustments to the cache management mechanism to take into account static residency assumptions used to drive optimisation. For example, static analysis may identify redundant residency checks on the basis of guarantees about the on-going residency of the target object. That is, there is an expectation that a resident object will not be unceremoniously evicted by the cache manager. In essence, residency checks pin objects in memory for some range of code, and the cache manager must agree to maintain those objects as resident so long as the thread executing the pinning code is active.

Excessive pinning of objects will prevent effective cache management. This is especially so in the context of pre-emptive thread scheduling, where a thread could be pre-empted in the middle of a pinning range. If the pre-empted thread is pinning large amounts of data then other running threads will face a congested cache. To avoid these problems we plan to allow the cache manager to steal pinned objects from suspended threads, on the understanding that stolen pins will be restored when the thread is resumed. Analysis of the existing cache manager and thread scheduler of the OPJ VM [Daynès 1997] has shown that they can be modified to support this additional functionality efficiently. Briefly, rather than overloading execution with explicit pinning mechanism, residency checks will implicitly pin objects. So long as there exist live references from a thread stack to an object, the cache manager will avoid stealing that object, and then only if the thread itself is inactive. If it absolutely must steal from the thread then it faces two choices: make the thread inactive until such time as memory becomes available to allow the object to be pinned and the thread resumed; or arrange for access to the object to be trapped transparently with respect to the code (techniques based on operating system primitives for virtual memory page protection are one approach [Hosking and Moss 1993]). Of course, one key question is how to discern live references. Bytecode analysis can help here by capturing liveness information per code range for use at run-time, but we can assume no such information is available for native methods. Since the current object cache manager must work with native methods anyway, we see no reason why analysis-guided pinning assumptions cannot also be incorporated into its mechanisms.

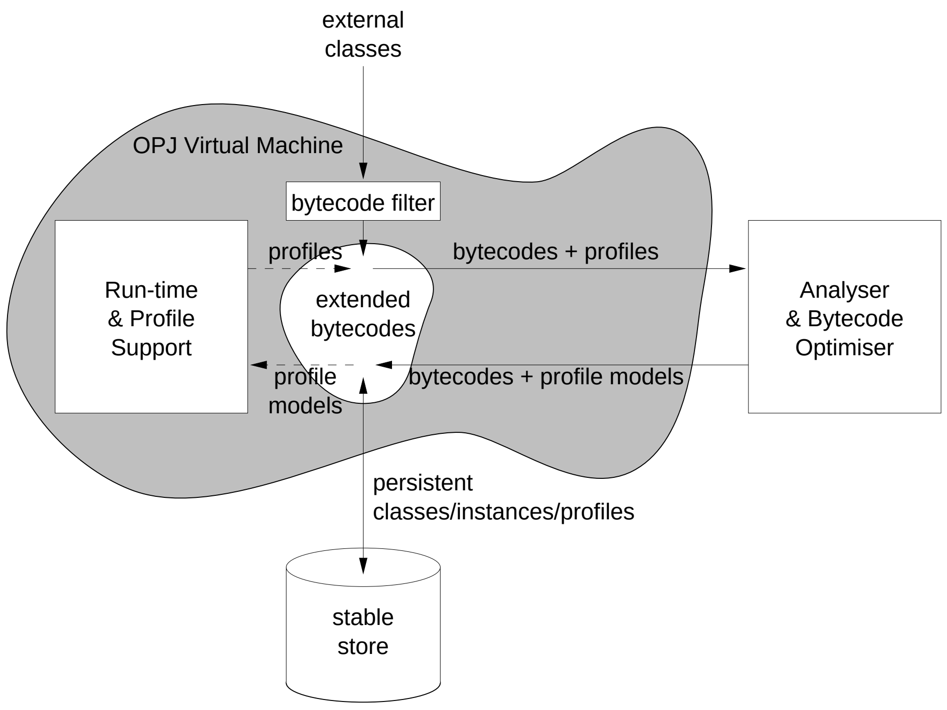


Figure 1: Prototype architecture

3 Prototype

We are building a prototype bytecode analyser based on the optimisation framework of Diwan [1997], and collaborating with the OPJ team to support the necessary extended bytecodes and run-time functionality to permit persistence optimisations. The basic architecture of the prototype is illustrated in Figure 1. As external classes are interned by the OPJ virtual machine, their bytecoded methods are filtered and augmented with the extended bytecodes for persistence (e.g., residency checks, update checks, etc.). These extended bytecodes simply expose persistence functionality that is currently buried inside the bytecode implementations of the current OPJ system. Assuming that the OPJ VM fully implements these extended bytecodes (i.e., as other than no-ops) then it can excise the buried persistence mechanisms and run with the extended bytecode set. The filtering process does not perform optimisation; it simply adjusts the bytecode for execution on the (extended) OPJ VM.

Analysis and optimisation takes place as necessary, on-line in response to requests by the run-time system when it discovers execution hot-spots that might benefit from optimisation, and off-line during periods of system quiescence. The analyser/optimiser is being coded in Java and will take advantage of a privileged interface to the persistent store allowing it to navigate persistent classes and to peruse and update their bytecoded methods. The analyser will communicate *profile models* to the run-time system for annotation; these consist of auxiliary data structures on which to hang execution statistics that may later serve more focused analysis and optimisation. Examples of these auxiliary structures include control-flow graphs for execution frequency annotation, and type graphs and instance structure graphs for pointer traversal annotation. Annotation may be performed by the insertion of profiling code, or through *a priori* contracts between the run-time system and the analyser. Note that all of this information can itself persist in the store for subsequent use in later executions.

4 Conclusion

We have briefly described a prototype bytecode analysis and profiling framework that we plan to implement for the prototype OPJ system. The intention is to support optimisations that both enable, and are enabled by, persistence. Construction of the initial prototype analyser and attendant modifications to the OPJ virtual machine for residency check elimination are expected to be nearing testing by the time of the workshop, where we will report on our experiences thus far. Areas we expect at that time to be more concrete include:

- the interface between the OPJ system and the analyser
- dynamic profiling results to expose system hotspots, and the potential benefits of candidate optimisations
- a detailed description and design of the pinning architecture within the OPJ virtual machine, including its interaction with the thread scheduler and object cache manager
- specification of the new, internal bytecodes for persistence mechanisms

Acknowledgements

We are grateful to Amer Diwan for assistance in understanding his Modula-3 analysis and optimisation framework, on which we model our own Java bytecode analyser. For details see Diwan et al. [1996] and Diwan [1997].

References

ATKINSON, M. P. AND BUNEMAN, O. P. 1987. Types and persistence in database programming languages. *ACM Comput. Surv. 19,* 2 (June), 105–190.

ATKINSON, M. P., DAYNÈS, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent Java. *ACM SIGMOD Record 25,* 4 (Dec.), 68–75.

ATKINSON, M. P., JORDAN, M. J., DAYNÈS, L., AND SPENCE, S. 1997. Design issues for persistent Java: A type-safe object-oriented, orthogonally persistent system. See Connor and Nettles [1997].

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *Int. J. Very Large Data Bases 4,* 3, 319–401.

CATTELL, R. G. G. 1994. *Object Data Management: Object-Oriented and Extended Relational Database Management Systems.* Addison-Wesley.

CHAMBERS, C. 1992. The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages. Ph.D. thesis, Stanford University.

CHAMBERS, C. AND UNGAR, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Portland, Oregon, June). 146–160.

CHAMBERS, C. AND UNGAR, D. 1990. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (White Plains, New York, June). *ACM SIGPLAN Notices 25,* 6 (June), 150–164.

CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, Oct.). *ACM SIGPLAN Notices 24,* 10 (Oct.), 49–70.

CONNOR, R. AND NETTLES, S., Eds. 1997. *Proceedings of the Seventh International Workshop on Persistent Object Systems* (Cape May, New Jersey, May 1996). Persistent Object Systems: Principles and Practice. Morgan Kaufmann.

CONSEL, C. AND DANVY, O. 1993. Tutorial notes on partial evaluation. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan.). 493–501.

CUTTS, Q., CONNOR, R., KIRBY, G., AND MORRISON, R. 1994. An execution driven approach to code optimisation. In *Proceedings of the 17th Australasian Computer Science Conference* (Christchurch, New Zealand). 83–92.

DAYNÈS, L. 1997. Private communication.

DEAN, J., CHAMBERS, C., AND GROVE, D. 1995. Selective specialization for object-oriented languages. See PLDI [1995], 93–102.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming* (Åarhus, Denmark, Aug.). Lecture Notes in Computer Science, vol. 952. Springer-Verlag.

DEARLE, A., SHAW, G. M., AND ZDONIK, S. B., Eds. 1991. *Proceedings of the Fourth International Workshop on Persistent Object Systems* (Martha's Vineyard, Massachusetts, Sept. 1990). Implementing Persistent Object Bases: Principles and Practice. Morgan Kaufmann.

DIWAN, A. 1997. Understanding and improving the performance of modern programming languages. Ph.D. thesis, University of Massachusetts at Amherst.

DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. 1996. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (San Jose, California, Oct.). *ACM SIGPLAN Notices 31,* 10 (Oct.), 292–305.

FERNANDEZ, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. See PLDI [1995], 103–115.

GROVE, D., DEAN, J., GARRETT, C., AND CHAMBERS, C. 1995. Profile-guided receiver class prediction. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Austin, Texas, Oct.). *ACM SIGPLAN Notices 30,* 10 (Oct.), 108–123.

HÖLZLE, U. AND UNGAR, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Orlando, Florida, June). *ACM SIGPLAN Notices 29,* 6 (June), 326–336.

HÖLZLE, U. AND UNGAR, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst. 18,* 4 (July), 355–400.

HOSKING, A. L. 1995. Lightweight support for fine-grained persistence on stock hardware. Ph.D. thesis, University of Massachusetts at Amherst. Available as Computer Science Technical Report 95-02.

HOSKING, A. L. 1997. Residency check elimination for object-oriented persistent languages. See Connor and Nettles [1997], 174–183.

HOSKING, A. L. AND MOSS, J. E. B. 1991. Towards compile-time optimisations for persistence. See Dearle et al. [1991], 17–27.

HOSKING, A. L. AND MOSS, J. E. B. 1993. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, Dec.). *ACM Operating Systems Review 27,* 5 (Dec.), 106–119.

HOSKING, A. L. AND MOSS, J. E. B. 1995. Lightweight write detection and checkpointing for fine-grained persistence. Tech. Rep. 95-084, Department of Computer Sciences, Purdue University. Dec.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.

MOSS, J. E. B. AND HOSKING, A. L. 1995. Expressing object residency optimizations using pointer type annotations. In *Proceedings of the Sixth International Workshop on Persistent Object Systems* (Tarascon, France, Sept. 1994), M. Atkinson, D. Maier, and V. Benzaken, Eds. Workshops in Computing. Springer-Verlag, 3–15.

MOSS, J. E. B. AND HOSKING, A. L. 1996. Approaches to adding persistence to Java. In *Proceedings of the First International Workshop on Persistence and Java* (Drymen, Scotland, Sept.), M. P. Atkinson and M. J. Jordan, Eds. Sun Microsystems.

PLDI 1995. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (La Jolla, California, June). *ACM SIGPLAN Notices 30,* 6 (June).

UNGAR, D. AND SMITH, R. B. 1987. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, Florida, Oct.). *ACM SIGPLAN Notices 22,* 12 (Dec.), 227–241.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley.

Persistent Servers + Ephemeral Clients = User Mobility

Alan Dearle
University of Stirling, Stirling, Scotland
al@cs.stir.ac.uk

Abstract

A large group of computer users are now mobile; they either make use of more than one computer or carry lap-top computers with them. User migration is often hindered by inadequate programming models and architectures. This paper describes an architecture which permits the user's environment to migrate with them. A corner-stone of this architecture is the ability of persistent Java systems to save and restore the state of active computations. This concept is extended to permit computations to be restored on different machines thus permitting a user's environment to migrate. The architecture also addresses the difficult issue of channel mobility between two migratory applications. It is therefore general enough to support arbitrary distributed mobile computations.

1 Introduction

Many computer users make use of more than one computer, for example, it is common to have a computer at work and another at home. Sometimes within the workplace a user may use more than one computer, perhaps in different rooms, buildings or even countries. Users in these situations have been forced to accept that the data they wish to manipulate may be unavailable on the local machine or perhaps available but the appropriate software to manipulate it is not. Software can add to the problem by encapsulating data, making it difficult or impossible to access from other machines. Examples include: data held in editor buffers, CAD designs, electronic appointment programs and electronic mail.

Many people have adopted ad-hoc working practices to accommodate mobility, for example, carrying floppies or a lap-top containing a cached version their current work. Ironically, the use of a lap-top introduces another form of mobility which must be accommodated - that of machine mobility. A common solution to the problem of making data globally available is to store and manipulate data on a central server. The situation is typified by electronic mail. Some users of electronic mail read their mail using tools such as elm, xmh or mailtool which both execute and manipulate mail files on a central server. Alternatively, tools such as Eudora which run on a local workstation may be used. The former solution forces users onto the central resource whereas the latter utilises the local machine but at a cost. Since mailers such as Eudora maintain both read and unread mail folders on the client[†], a user moving to another machine and wishing to access mail has considerable difficulty.

A more desirable situation would be for a user to approach an arbitrary machine and be able to continue performing their work regardless of where they worked last. This paper presents a first step towards the realisation of this ideal. A number of developments have made such an approach possible: the ubiquity of the Internet, the widespread adoption of Java, and the maturity of persistent technologies.

[†] Pop based mailers such as Eudora do allow mail to be left on the server but the users forfeit the ability to organise mail into folders.

The paper is organised as follows: first some terminology and a characterisation of different kinds of mobility is made. This is followed by a statement of the requirements for supporting mobility and a description of an architecture designed to satisfy these requirements. The architecture has two main components, platforms and servers which are described in Sections 3 and 4 respectively. A design to cope with the problems of environment binding and communication channels between mobile entities is addressed in Section 5. Section 6 describes an initial experiment which we have conducted to prototype, test and develop our ideas. Some related work is discussed in Section 7 and Section 8 concludes.

2 Terminology and Requirements for Mobility

2.1 Terminology

In this paper we will consider the mobility of three classes of entity: *people*, *machines* and *processes*. We assume that a mechanism for finding data on the network exists; a naming mechanism such as that provided by Uniform Resource Locators (URLs) will suffice.

We define a *user* to be a person who uses a computer; users are mobile: they move from home to their place of work, from city to city and from continent to continent. We define a *view* to be that which a user sees when they sit down at a computer screen be it connected to a PDA, PC, workstation, network computer, or mainframe. Users may own multiple views but only make use of one at a time. A view is implemented by a *platform*. A platform is a collection of hardware and software that combine to implement the view. We will separate a platform into two components the *platform software* and the *platform hardware*. The platform software consists of:

- active threads and/or processes[‡] that implement the view,
- the code being executed by the threads,
- the code that implements the software environment (e.g. the Java-virtual machine, dynamic libraries etc.) and,
- the data representing entities visible in the view.

The platform hardware consists of a computation environment on which to run the platform software i.e. a CPU and main memory, a screen with a pointing device and perhaps a keyboard. The platform hardware may contain persistent storage but need not. In cases where persistent storage is not available on the platform, it is provided by a *server*. A server contains non-volatile storage and may be used as a general purpose repository. There are no requirements for a server to support any form of view. If a device which is used as a server also contains the ability to operate as a platform or vice-versa, it will be considered to be two different entities. Clearly, there is an opportunity for optimisation in this case.

2.2 Characterising Mobility

The framework described above is shown in Figure 1 which shows the three classes of entity that may be mobile: users, views and platforms. When users move from location to location they require their view to move with them. The view includes the user interfaces to applications which may be executing on the platform, on the associated server, or elsewhere on the network. When a view migrates from one platform to another, either the threads and

[‡] We will use the term thread to mean thread/process.

data implementing the view must also migrate, or those threads must be notified of the location of the new view.

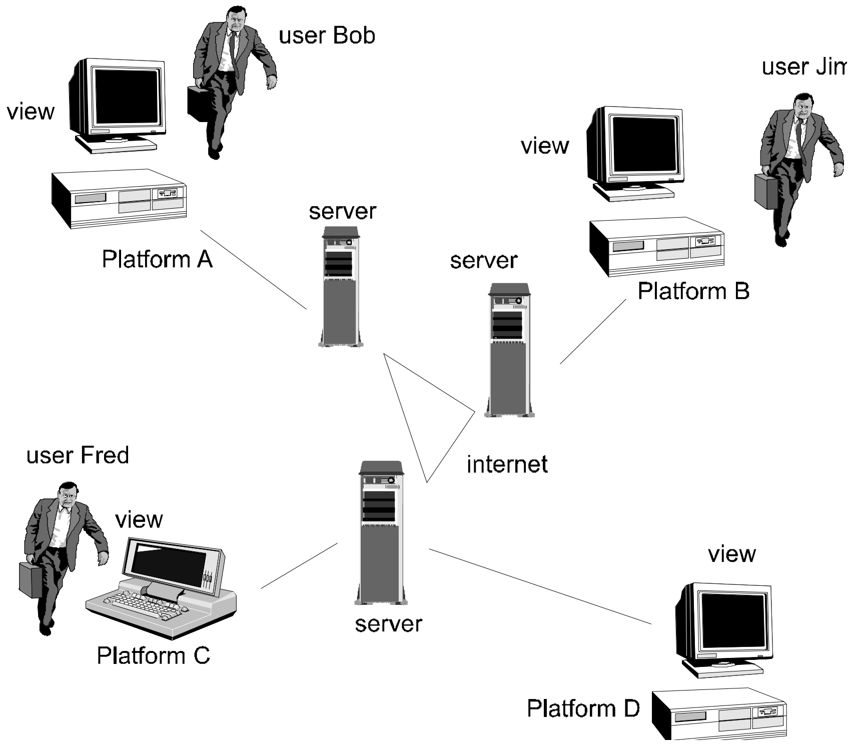


Figure 1: Users, Views, Platforms and Servers.

Clearly the former solution makes better use of caching and network bandwidth. In this paper it is therefore assumed that the threads and data implementing the view migrate with the user. There is no requirement for the applications with which the user is interacting to migrate although it may be expedient for them to do so. When the threads implementing a view migrate, the connections to networked entities must also migrate. For example, if the view includes a traditional window implementing a Unix interaction with a remote host, the input and output to and from that host must migrate with the view.

A platform may migrate with a user, for example, when a user carries a lap-top to another site. When this occurs, the applications running on that platform and the view that it presents also migrate. However, network connections to the platform may be severed and need to be re-connected at another site. This situation is analogous to the process which occurs when platform software migrates. When a platform migrates, it may be expedient for the platform to employ the services of a local server at the new site.

As described above, a view is implemented by a collection of persistent threads each of which operate on some cached data. Figure 1 is refined in Figure 2 which shows the composition of views, platforms and servers.

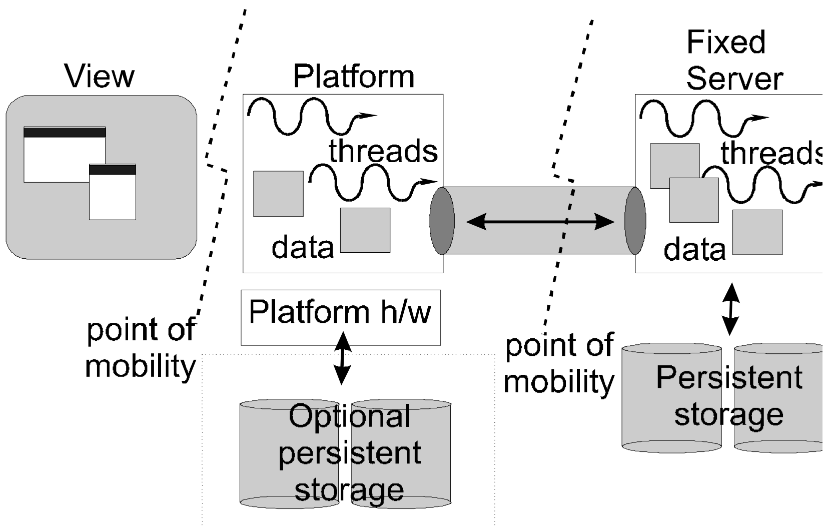


Figure 2: Points of Mobility

Figure 2 shows the two different points of mobility that may be considered:

1. view mobility over platforms and,
2. platform mobility over servers.

View mobility is supported by platforms and servers, platform mobility is supported by servers. The persistence of threads and data in the platform may be implemented either by the platform, if it is equipped with persistent storage (e.g. in the case of a PC), or co-operatively between the platform and the server if it is not (as is the case in a NC).

2.3 **Requirements for User Mobility**

A user may move from machine to machine, for example, in Figure 1 user *Jim* may move from platform B to platform C connected to a different server. When the user moves, the user's view should also move, permitting the user to continue with whatever work was being performed at the last platform.

The ability to migrate a view requires that the platform software be capable of migrating to a different platform hardware instance. Since the platform software consists of active threads and the data representing entities visible in the view, migrating the view requires two forms of migration:

1. data migration, and,
2. thread migration,

or rephrased,

view mobility = thread mobility + data mobility.

Each of these forms of mobility introduces other problems discussed below.

2.4 **Requirements for Platform mobility**

Consider platform C, in Figure 1, as a mobile device it may be disconnected from the network and reconnected elsewhere. Here the view of user Fred is (may be) maintained by the mobile device, however the mobility of platforms highlights some other problems, namely:

1. *environment mobility*, and the special case of this,
2. *channel mobility*.

Environment mobility is concerned with bindings between threads and the external environment. For example, a thread may make use of a printer or some input device. Mechanisms must be provided so that threads running on a mobile platform may bind to services which appear in their environment. This situation also arises when a view is migrated to another platform. It is clear that different kinds of bindings are required even for one class of device. For example, in the case of a printer, a user may wish to print confidential or personal documents on a secure printer at a fixed location and in other cases any printer may satisfy their needs. This illustrates the need for both static and dynamic binding mechanisms to support environment mobility.

Channel mobility is a special case of environment mobility. A thread running on a platform may open a communications channel with a another thread running on another platform or server. If the platform is taken off line and moved to another location, the channel will be lost. Like environment mobility, this situation also arises when views are migrated since threads are migrated to other platforms. To make movement transparent, software that maintains the channel across movement must be provided. This may be achieved in one of two ways:

1. implementing software at both ends of the channel to manage the transparent connection/reconnection, and,
2. using a server as a *connection proxy*.

These ideas are expanded in Sections 5.1 and 5.2 below.

2.5 **Overall architecture**

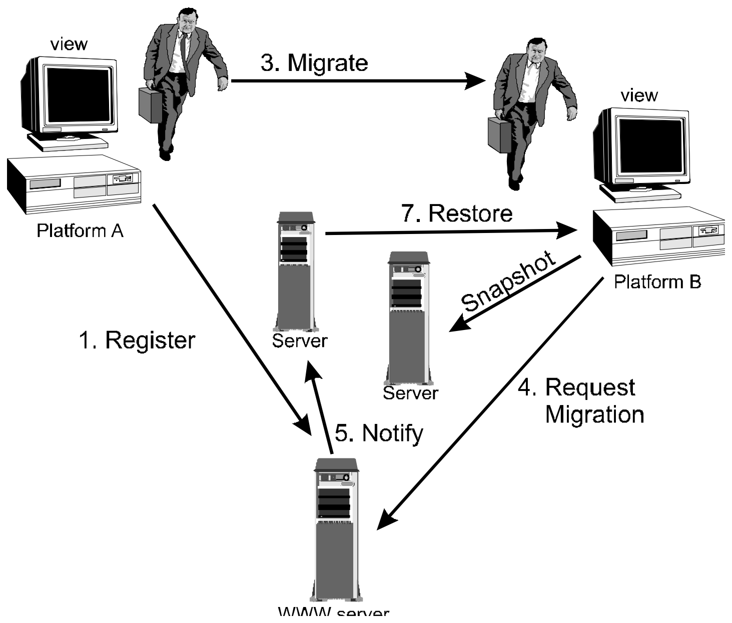


Figure 3: Re-establishing a View

Before view is made visible, the owner of the view must be authenticated. This requires the following:

1. users must identify themselves,
2. users must present authentication, and
3. the location of the view specified by the user must be established.

The first two stages above are identical to a conventional login session on a Unix machine. The third step is necessary in order to locate the user’s specified view. These three steps may be easily achieved if a smart card with modest memory is available. Using smart cards it is possible to record the identity of the user and where they were last active. Many of the Network Computers that are currently available have smart card interfaces built into them which could be used for this purpose. However, since smart card devices are not ubiquitous, we shall consider other ways of identifying the user and their views. Figure 3 shows one method of doing this using the World Wide Web.

Each user is assumed to have a home which is capable of recording the identity of the last hardware platform on which each view was last made. This functionality may be implemented via a simple *cgi* script located on a Web server which is capable of saving and retrieving locations. In the worst case, a user might type in the location of their home during the authentication process in order permit the system to locate it. Other possibilities are to use a search engine or global identifiers to locate a user’s home.

When a view is established on a hardware platform, the user's home is contacted to register the server as manager of the view (1). Periodically the view is snapshotted to the server to provide resilience and to permit future migration (2). Following user migration (3), the new platform contacts the user’s home to register and request view migration (4). The Web server requests the platform which last implemented the user’s chosen view to migrate the software platform to the new hardware platform (5). These requests will typically be sent via the server supporting the platform as described in Section 5.2 below. In practice many of these requests will not be forwarded to the platform and will be handled by the server implementing the platform’s persistent storage. In either case, the closure of threads and data implementing the view are migrated to the new hardware platform for restoration (6) and the cycle begins again.

3 **Platforms**

A platform must be capable of:

1. authenticating a user,
2. loading a view from the platform/server identified during the authentication sequence, and
3. saving the state of the view to persistent storage provided by the platform or the server that supports it.

The second and third activities are intimately related to each other and require a protocol that defines:

1. how to identify the persistent state implementing a view,
2. how persistent state is preserved,
3. what format the persistent data is in, and
4. how to transport that state to and from persistent storage and between platforms and servers.

The persistent state that implements a view may easily be tagged using the user-id combined with a view identifier. This scheme is used in Grasshopper to identify login sessions and similar schemes have been used elsewhere [1]. The next question is how state is preserved; in general, there are 4 approaches to saving state in Java systems:

1. manually writing save and restore code in every application/applet,
2. perform saving and restoration using (Java) serialisation,
3. providing persistence at the (Java) virtual machine level [2], and

4. providing persistence at the address space level.

The first approach is the traditional solution to persistence: write flattening code by hand for every object class in the system. Whilst this is possible for simple data structures it becomes unmanageable in complex applications and has been estimated to account for 30% of all application code. The only merit of this approach is that code may be written that is highly optimised for the data types.

The second approach has become popular since the introduction of Java object serialisation [3] which permits an arbitrary graph of objects to be marshalled into a stream. Whilst this approach would be appear to be a panacea it is not without its problems some of which are fundamental and others accidents of implementation. The first problem, which may be argued to fall into either of the above two categories, is that not all fields of objects are written to the stream. In particular, private fields are not serialised due to a perceived security breach. The second problem with this approach is that active context (i.e. active threads) is not saved. Knowing that threads are not preserved across serialisation will inevitably force programmers to write code in a certain way. Whether this is detrimental to coding remains to be seen. The third problem is that Java object serialisation is not an efficient method of making data persistent due to the fact that, like pickling, it is an all or nothing approach. There is no concept of saving only that data which has been modified since a particular point or time such as the start of a transaction. One can attempt to avoid this problem by selectively serialising objects. However, using such an approach it is easy to lose referential integrity which must be avoided. A final problem with serialisation is that it cannot be effectively used to save anything other than entire object closure. In many persistent systems the object closure may include the entire persistent store and perhaps even large portions of the Internet. If this approach is to be followed, techniques such as Farkas' OCTOPUS mechanism [4] or the use of weak pointers are also required.

The next approach to saving state is to provide persistence at the (Java) virtual machine level. This is the approach followed by Atkinson's PJava group [2]. Whilst we have argued elsewhere that the last approach is better, this approach has many merits in the application domain described in this paper. It also addresses many of the shortcomings of the serialisation approach described above. Providing persistence at a level lower than the Java language level permits the (reflective) type system to be broken and consequently all fields of objects may be saved to persistent storage rather than only the public ones. Secondly, the runtime state associated with threads may also be saved to persistent storage and later restored[†]. Since the runtime system has access to object implementations, it is easy to save only those objects that have been modified since a previous checkpoint or transaction start. This approach helps solve the closure problem described above although does not address the problem with respect to network transmission.

The final approach to providing persistence is to provide it at the virtual address space level; this approach is followed in the design and implementation of Grasshopper and has many desirable properties which we have described elsewhere [5]. One benefit of this approach is the ability to make all data in the address space persistent, including the state of threads and the stacks supporting them. Curiously, this does not assist in the transmission of state to another machine since data must be in an architecturally neutral format to support heterogeneity.

The above techniques are all capable of gathering some approximation to the persistent state of a computation. The format of the data in each case is different. Using the manual approach

[†] Current implementations do not support this functionality.

to saving persistent data produces persistent data in an ad-hoc format. This is a hindrance to its use in a general purpose system. Clearly some standard representation is required to enable the data to be saved and restored. In this respect, object serialisation is clearly the best approach. However, as described above, it is deficient in that it does not capture the dynamic state of computations. This is also true of the Aglet approach described below [6]. The most complete solution, that of persistence at the address space level, suffers from problems with heterogeneity, leaving the Pjava approach being the most promising.

The last problem is how to transport state between platforms and between platforms and servers. Once a format for the persistent data has been agreed, this may be easily achieved using one of the stream abstractions provided by Java that use TCP/IP.

4 Servers

Servers are responsible for four tasks in the architecture:

1. implementing the home of users,
2. providing persistent storage for non-persistent client platforms,
3. providing channel proxies, and
4. provide caches for client platforms.

The task of providing a home for users is the simplest of the four tasks. This requires the ability to record and recover the identity of the last hardware platform which implemented a view. As described above, this may be done simply and efficiently using existing Web tools and protocols. A server providing a user’s home may also be required to provide persistent storage for (passive) data owned by a user. This is the traditional file/object server task often associated with the role of a server.

In addition to the file/object storage role, servers are required to provide persistent storage for the views implemented by the hardware platforms they support. This role is similar to that played by servers in support of early Sun diskless workstations (e.g. Sun 3/50). In this architecture the servers are likely to be required to support a cluster of diskless network computers. The state of the platforms is periodically checkpointed to the server which is responsible for saving the view on non volatile storage. This state may be requested by the platform following a crash or by another platform during the re-establishment of a view.

Servers also implement channel proxies discussed in Section 5.2. These provide a fixed location for communications with migratory hardware and software platforms. The fourth role of servers is that of cache manager. It is likely that clusters of network computers would often be running similar if not identical collections of code. Since servers support persistent storage and act as proxies for communications channels, it is natural for the servers to implement code and data caches.

5 Channel and Environment Mobility

5.1 Managing connection/reconnection

As described above, channel mobility may be implemented in two ways:

1. implementing software at both ends of the channel to manage the connection/reconnection, and,
2. using the server as a *connection proxy.*

In order to accommodate the connection and reconnection of channels we introduce a new abstraction called a *half session*. The primary purpose of a half session is to implement a communication channel which provides a reliable stream abstraction that can be disconnected and reconnected to different platforms and servers. As shown in Figure 4, on each platform or server implementing a relocatable channel, a half session is used to manage the connection. Thus there is a half session managing each end of a relocatable channel. The name, inspiration, and thinking behind half sessions is motivated by the seminal work of Strom and Yemini [7].

Half sessions present stream abstractions which are an extension of the interfaces presented by java.io.InputStream and java.io.OutputStream [8]. In addition to the methods provided by these interfaces, the half session abstraction provides methods for the re-establishment of the stream with an alternative client or server should a half session object be migrated. Clearly the re-establishment method must communicate with its peer half session in order to re-establish the channel.

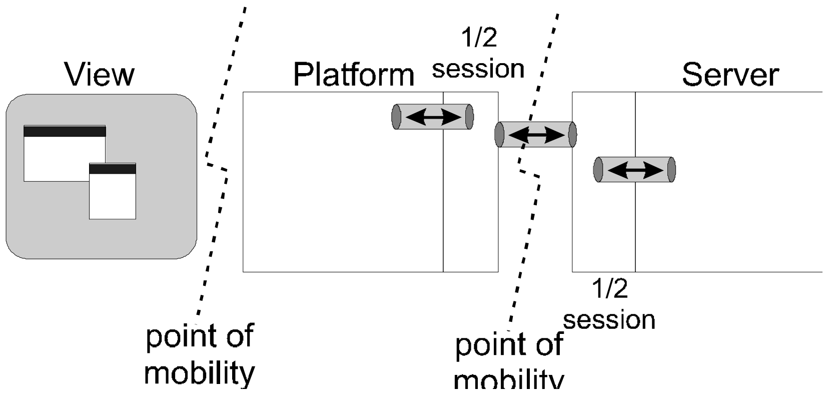


Figure 4: Half Sessions

5.2 Channel Proxies

The half session abstraction permits two threads running on different servers or platforms to communicate with each other and permits migration of either end of the channel. However, views may be required to communicate with legacy systems which do not implement the channel abstraction. This is the case where a user is interacting with a legacy application running on a Unix system, for example a shell. Since legacy code does not support the half session abstraction, some additional mechanism must be provided to permit the platform (hardware or software or both) to migrate. This may be achieved by the use of the server as a fixed proxy for communication. Using this scheme, the fixed server communicates with the remote party on behalf of the platform. This communication is achieved using a traditional socket interface. The platform in turn communicates with the server using the half session abstraction permitting the platform to be relocated without the knowledge of the remote party which is only aware of the server.

The above scheme may be implemented in Java using an implementation of the java.net.Socket class which uses the half session objects described above rather than standard input and output streams. In the implementation of this class, all data is routed via the server using the half session abstractions rather than using direct communication with the remote party. This may all be achieved transparently to the client.

5.3 Agents and Channels

In addition to managing socket like streams, we also wish to support agent-style computation. This model is now well known [6, 9, 10] and requires autonomous computations capable of moving between different nodes in the network carrying code and data with them. Agents may be used to perform a number of tasks including scheduling meetings between different users and gathering information from a number of sites for example to arrange a trip. Consider this last example, a user may wish to travel between Stirling and California. Such a trip may involve agents visiting different sites containing information about train and flight times. After initiating agents to arrange a trip, the user may move to a different site. However, the agents should report back to the user not back to the site from which they originated.

The management of agents leaving from and returning to a view may be handled using a mechanism similar to channel proxies. All agents are routed via the server using the half session abstractions. This ensures that agents have a fixed location to which they may return. The server is responsible for holding agents attempting to return to views that are currently inactive.

5.4 Binding to Services

A final aspect of mobility that must be addressed is binding to the external environment. Binding to external services may either be dynamic or static, for example, an application running on a platform may wish to make use of a printer. In this case any postscript printer available locally may be suitable and the binding is dynamic. In other circumstances, for example when a user wishes to make use of a file/object server, only the file server containing the user's files would be appropriate. Here the binding between the application and the external service is static.

In both cases the external services are provided by servers, the only issue is how platforms bind to the servers. Clearly, if either the hardware or software platform is permitted to migrate, some indication of the (re)binding regime must be specified when the binding to the service is initially established. In order to support (re)binding activities, servers are required to provide an associative lookup mechanism like that provided by CORBA [11] and Grasshopper nameservers.

6 An Initial Experiment

As an initial experiment we have implemented an instance of this architecture to support a single application – a ubiquitous mailer. This system was constructed as a demonstrator and mimics the functionality of a ubiquitous mailer being implemented by DEC. The mailer is a Java applet which is uploaded from a server running the Grasshopper operating system. All persistent state is held on the Grasshopper system and loaded on demand to the mailer applet. The applet (800 lines) is essentially a mail viewer and contains code to authenticate the user, view mailboxes, compose mail messages and snapshot its state. Most mailer operations involve communication with the server. All interactions between the mailer and Grasshopper are made using HTTP. Since HTTP behaves in a connectionless manner, the mailer may move from platform to platform transparently to the server.

In the system described in this paper the snapshotting and recovery of platforms is automatic. In the mailer system, the state of the mailer is persistified (sic) by a thread with application specific knowledge. The thread sleeps on a timer and on awakening sends any volatile state of the mailer including messages currently being composed, lists of new mail etc. to the server using HTTP.

7 Related Work

7.1 Migratory Applications

In [9] an architecture designed to support migratory application in the language Visual Obliq is described [12]. Single user migratory applications are supported at the language environment level and may migrate from node to node whilst maintaining the state of their user interface. Almost no requirements are made of the application programmer which is the ethos behind orthogonal persistence and the thinking behind this paper. The basic building block of the system is the concept of an *agent*, a computation that may hop from site to site carrying with it a *suitcase* containing the agent’s persistent memory. When the agent executes a *hop* instruction, the suitcase and the computation’s closure is migrated to the new site. When an agent arrives at a site, it is given a *briefing* which may include advice for the agent and site specific information. The above mechanisms have been used to construct migratory applications containing a *MigrateTo(Host)* command which causes the remote host to be contacted and if it will accept the application, it checkpoints the state of its user interface and performs a hop instruction. This work is complementary to our own and has coloured our own thinking.

7.2 Aglets

As described in [6] an *aglet* is a mobile Java object capable of visiting different hosts on a network. Aglets are autonomous, they each contain an active thread of execution and are capable of reacting to messages sent to them. Like an applet, the class files for an aglet can migrate across a network. Unlike applets, when an aglet migrates it also carries its state. An applet is code that can move across a network from a server to a client. An aglet is a running Java program (code and state) that can move from one host to another on a network. Each aglet executes in a *context* which provides a uniform execution environment independent of the capabilities of the host. The aglet context serves to isolate the aglet from the platform.

Aglets have an *onCreation* method which is executed when the aglet is created or migrated to a new context. Aglets also contain a *dispatch* method which takes an URL as a parameter and may be used to migrate the aglet to a new context. When *dispatch* is called, the byte code and state of the aglet is preserved using standard Java object serialisation and transmitted across the network using the Aglet Transport Protocol (ATP) [6]. Aglets can be reactivated at the new site using the *onCreation* method.

Since aglets use Java object serialisation to export their state, the execution state of the threads owned by the aglet are not serialised. Therefore when an aglet is migrated or deactivated, any state resident on stacks and the program counters of running threads are lost. This is a consequence of the JVM, which does not permit direct access to run time state. Before an aglet is serialised, the host informs it that serialisation is immanent (via the *onDispatch* method) so that it may store any information it will need to continue its execution in object variables.

Aglets are complementary to the ideas in this paper in that they offer potential technology for implementing the architecture described in this paper. Whether or not the loss of dynamic state is too much of a programming restriction will remain to be seen.

8 Conclusions

This paper presents some initial thoughts on how persistent technology in general and Java in particular may be used to provide mobile users with a ubiquitous environment. A generally applicable architecture to support mobile users has been described. We have implemented a restricted prototype in order to validate these ideas. The architecture is realisable using technology which is currently available. Techniques with which user views may be located and restored on a different hardware platform have been described as have techniques for dealing with the difficult problems of inter-platform communication. These techniques both address the need to interact with other mobile computations and with legacy systems. They are also general and may be applied to mobile distributed applications unlike [9]. A number of engineering problems remain, in particular, the best way to save and restore closures of Java objects. We have suggested several approaches which may be used. The best approach will require further investigation.

9 References

1. Rosenberg, J. and L. Keedy, *Security and Persistence*. Workshops in Computing. 1990, Berlin: Springer-Verlag.
2. Atkinson, M.P., *et al. Design Issues for Persistent Java: a type safe, object oriented, orthogonally persistent system*. in *7th International Conference on Persistent Object Systems*. 1996: Springer-Verlag.
3. Javasoft, *Object Serialization Specification*.
4. Farkas, A. and A. Dearle. *Octopus: A Reflective Language Mechanism for Object Manipulation*. in *Proceedings of the Fourth International Workshop on Database Programming Languages*. 1994: Spinger_Verlag.
5. Dearle, A., *et al.*, *Grasshopper: An Orthogonally Persistent Operating System.* Computer Systems, 1994. **Summer**: p. 289-312.
6. Lange, D. and D. Chang, *Programming Mobile Agents in Java: A White Paper*, . 1996, IBM Corporation.
7. Strom, R. and S. Yemini, *Optimistic Recovery in Distributed Systems.* ACM Transactions on Computer Systems, 1985. **3**(3): p. 204-226.
8. Chan, P. and R. Lee, *The Java Class Libraries An Annotated Reference*. 1996, Reading, Massachusetts: Addison-Wesley.
9. Bharat, K. and L. Cardelli, *Migratory Applications*, 1996, DEC, Systems Research Center.
10. White, J.E., *Telescript Technoloogy: The Foundations of an Electronic Marketplace*, . 1994, General Magic Inc.
11. OMG, *The Common Object Request Broker: Architecture and Specification*, 1991, OMG.
12. Cardelli, L., *A Language with Distributed Scope.* Computing Systems, 1994. **8**(1): p. 27-59.

ObjectStore PSE: a Persistent Storage Engine for Java

Gordon Landis, Charles Lamb,
Tim Blackman, Sam Haradhvala,
Mark Noyes, and Dan Weinreb
Object Design, Inc.

Object Design, Inc. markets and sells *ObjectStore PSE for Java*, a lightweight *P*ersistent *S*torage *E*ngine for Java. PSE is an entry-level product for users who require persistent Java object storage in client, server, servlet, or applet environments. Typical PSE applications run the gamut from applets that need to store user configuration information locally, servlets that need to store user access history, GUI-based user-directed web spider applications that need storage for their results, and financial applications that need to store up to tens of megabytes of cached data on a client host.

This paper describes the design goals of PSE, as well as some of the implementation details, user and customer experiences with it, and future directions.

Goals

The goal of this project was to build a persistent storage and database access system for Java, with the following characteristics:

Transparent access to persistent data.

- *Fetch/Store*. The operations to fetch persistent data from a database and store modifications back to the database should be automatic and largely transparent.

Tight integration with the Java language and environment.

- *Object identity*. Object identity should be preserved across the transient/persistent boundary. That is, it must not be possible to get two different representations of the same object, nor to get two different references that denote the same object yet are not equal.
- *Memory management / persistence*: Java is a garbage-collected language, so an object *exists* if it is reachable from another (reachable) object. Similarly, an object *persists* if it is reachable from another (reachable) *persistent* object. We call this *persistence by reachability*, or *transitive persistence*.
- *Single type system*. There should be a single type system for both transient and persistent storage. It should not be necessary to translate between the types stored in the database and the types used in the runtime environment. Rather, the type system of the Java language should be directly supported by the persistence mechanism (including the ability to directly store built-in Java types).
- *Multi-threaded applications*. Java encourages multi-threaded programming so it is important for PSE to support thread safe operations. At a minimum, the entry-points must be safe against simultaneous calls from different threads. Furthermore, threads must be able to cooperate in their accesses to persistent data, or act independently from one another.
- *Compatibility with Java environment.* The persistence mechanism must not interfere with Java's goals of portability to any VM or platform, simplicity of code distribution, or applet execution.
- *Support for reuse of existing class libraries.* The development environment and tools should enable existing class libraries to be made persistent with little or no source modification.

Capable of supporting multiple back-end storage systems with different scalability characteristics.

There are currently three back-end storage systems in use with this API: PSE, PSE Pro, and ObjectStore. These span the range from a simple, low-end, single user persistent store, to a high-end multi-user client-server DBMS. They share a common front end, such that the same user code (both Java source and compiled byte-code) can access either a PSE database or an ObjectStore database without modification or recompilation. This paper focuses on the PSE and PSE Pro engines.[1]

The primary design goals for the PSE storage engine are:

- *Small footprint*: The target size for the uncompressed runtime .zip file is 300k bytes. This allows applets to download the PSE classes quickly. Minimum database size as well as storage overhead must also be small.
- *Portability*: To maximize its portability, PSE must be pure Java, and must not rely on specialized Java environments (such as a modified VM). Nothing should prohibit PSE's use in a variety of application scenarios, including applets, applications, and servlets.
- *Random Access to Persistent Data*: PSE must provide random access to persistent data, and must only read or write those objects which are read or modified during a transaction. This differs from Object Serialization, which is an "all or nothing" access method.
- *Atomic Transactions*: Even at the low end of the storage system functionality spectrum, it is a requirement that PSE must support ACID transaction semantics: atomic, consistent, isolated, and durable[2]. This allows applications to maintain data integrity as well as provide simple "undo" capabilities through a transaction abort function. PSE must protect persistent data from system failure.
- *Multi-user*: PSE need only support coarse-grained locking for multi-user access; more fine-grained concurrency is provided by other storage systems in the ObjectStore product suite.

Note that it was not a goal to make PSE a full-function multi-user database system; rather, the goal was to make an easy to use, small-footprint storage environment, which would be API compatible with the full-function ObjectStore DBMS.

The following sections describe the implementation of PSE, and the level to which it has succeeded in meeting these goals.

API Transparency

For an entry-level database system such as PSE to be widely accepted, it must be easy to use, and it must impose as few implementation burdens as possible on the programmer. Ideally, programming for persistent storage should not be much different from programming for transient storage. In particular, the user should not have to explicitly read objects in from the database before using them, or write objects out when they are changed; instead, the database system should perform these tasks automatically. In most cases, the user should not even have to be aware of the fact that the objects being manipulated are being read from and written to a database.

We considered three main implementation options for such a transparent interface to persistent storage:

- changes to the Java Virtual Machine itself
- pre-processing of the Java source files
- post-processing of the class byte-code files

[1] Although there are small differences between them, for the purposes of this paper, PSE and PSE Pro may be considered the same.

[2] Transaction durability is one area where PSE and PSE Pro differ slightly. While both guarantee the durability of updates in the event of a process crash, only PSE Pro guarantees durability in the event of an operating system crash.

The first approach, changes to the Java Virtual Machine, was considered in some detail. While there could be a number of highly desirable aspects to such a solution (assuming it could be implemented in a way that did not compromise the performance of applications that do not make use of persistent storage, and could cover all of the corner cases), we nonetheless rejected it as unfeasible. Given JavaSoft's understandable reluctance to make sweeping changes to the VM, especially changes that affect the fundamental operations of object access and update, we felt that it was not practical to depend on changes to the standard Java VM definition. Even if it were possible to define a set of changes that was simple to implement, and that did not adversely affect the performance of mainstream Java program execution, it appeared that it would take a long time and a great deal of effort to overcome the inertia to change generated by the need for a stable VM definition on which to base silicon implementations. We also rejected the option of making our own, non-standard VM changes, both because we were concerned about the performance impact on transient code, and (perhaps more importantly) because this would violate our goal of portability.

The second approach, a source-code pre-processor, was also considered. It was rejected because it appeared to entail many of the same difficulties as post-processing. In fact, because the class file format is simpler than the source code format, a pre-processor is a larger undertaking than a post-processor. A pre-processor would also violate our goal of supporting the re-use of existing class libraries in their compiled form (that is, without access to the Java source code).

Given that the Java portability and code-distribution benefits are based upon the standardization of the byte-code format and structure, a post-processing technique seemed most consistent with the goals of Java and of PSE. Finally, we felt that post-processing might have wider utility, if compilers from other languages into Java byte-codes were to become available. This technique is feasible because the Java byte-code format is stable (unlikely to change much over time), simple (far simpler than .o file formats, for example), and well specified.

The transparent access to persistent storage provided by PSE (and the ObjectStore Java interface) is achieved by a class file post-processor called *osjcfp* (for ObjectStore for Java Class File Post-processor)[3]. The classfile post-processor accepts a program's .class files as input, and produces new annotated .class files as output. By setting the CLASSPATH appropriately, the annotated .class files produced by the post-processor form the basis of the user's program.

Class file post-processing performs two main functions.

- *Schema Generation*: It examines the fields of the targeted class files and determines the field types and locations. Using this information, it then generates auxiliary "ClassInfo" files (for example, the post-processor would generate a *PersonClassInfo.class* file to correspond to a persistent *Person.class*). ClassInfo files are registered with PSE at run-time, and contain methods that allow PSE to interrogate the number and type of fields in the corresponding class.[4] A *ClassInfo* file has no corresponding source file; the post-processor generates the byte code directly.
- *Code Annotation*: In order for PSE to know when data is being read from or written to the database, byte codes are inserted into a class's methods. These byte codes cause the *COM.odi.ObjectStore.fetch* and *COM.odi.ObjectStore.dirty* methods to be invoked, to read data from the database into the object's fields, or mark an object as modified. These annotations ensure that a persistent object is fetched from the database before any attempt is made to read the object; and that any updates to the object are noted so that they may be

[3] Our implementation supports manual code annotation as well, but most customers prefer the simplicity and safety of the post-processor.

[4] Some of the ClassInfo code that the post-processor currently generates can be simplified or even replaced by the use of the introspection API provided in the JDK 1.1.

written to the database by *Transaction.commit*. Optimizations are employed to deal with cases such as repeated use of the same object within a single method, or within a loop.

The class file post-processor can make a class *persistence-capable* (both of the above operations are performed), or it can make a class *persistence-aware* (only the code annotation phase is performed). A persistence-capable class may have both transient and persistent instances: persistence is determined on an instance-by-instance basis, based on reachability (described in the next section). A persistence-aware class, on the other hand, can manipulate persistent objects, but cannot itself have persistent instances[5].

The calls to *fetch* and *dirty* that are inserted by the post-processor check the object's state to determine what (if any) actions need to be taken. If the object in question is persistence-capable, but not actually persistent, then no action is required by either *fetch* or *dirty*. If the object is persistent, then it can be in one of three states:

- *Hollow:* This means that the contents of the object have not been read in from the database. Both *fetch* and *dirty* need to read the data from the database to initialize the fields of the object.
- *Active:* This means that the contents of the object have been read from the database, but have not been modified in the current transaction. While *fetch* does not need to do anything, *dirty* needs to mark the object as modified so that its contents will be written back to the database when the transaction commits.
- *Modified:* This means that the contents of the object have been read in from the database, and have been updated in this transaction. Neither *fetch* nor *dirty* need to do anything.

For instances of most classes, *fetch* and *dirty* check the state of the object by inspecting fields of the object that were inserted by the classfile post-processor. These fields, defined on the base class *COM.odi.Persistent*, store object state information[6]. When an object is read from the database by the *fetch* method, the object's state is initialized, and the object moves from the *hollow* state to the *active* state. At this point, any objects that are referenced by the object (and that have not already been accessed previously) are allocated in the Java virtual memory space, and are initialized to the *hollow* state. Hollow objects are placeholders for persistent objects, but their fields are not filled in from the database until they are actually needed. Any attempt to access a field of one of these hollow objects in the *fetch* operation being called on that object in turn.

Persistence-capable types that are not classes, such as array types and primitive types (e.g., int, boolean, long, etc.), are dealt with specially, as are their corresponding wrapper classes (*java.lang.Integer*, *java.lang.Boolean*, *java.lang.Long*), and the class *Java.lang.String*. All of these types are automatically persistence-capable. However, because they do not inherit from *COM.odi.Persistent*, their object state must be inferred differently. In the case of the primitive types, only their values are stored in fields of other objects, so they do not themselves have identity or separate representation in the database; they are read in when their containing object becomes *active*. The immutable types (the primitives wrapper classes, and *Java.lang.String*) do have separate identity in Java, but because they cannot be modified once they are created, PSE gives them a simpler treatment, and reads them in when any object that points to them is

---

[5] Note that a class only needs to be persistence-aware if it *directly* manipulates the fields of a potentially persistent instance. Any class, even one that has not been post-processed at all, can manipulate persistent instances by calling their methods. In this case, because only the class itself directly manipulates its instances' fields, the object is strictly encapsulated, and code that uses the class need not be aware that instances may be persistent.

[6] In the current release of PSE, the classfile post-processor inserts *COM.odi.Persistent* into the inheritance hierarchy of any persistence-capable class, so that these fields will be available. While this is generally automatic and transparent to the user, it nonetheless violates our goal of a single type system. An updated version of the PSE classfile post-processor will be available shortly that does not require this, but instead provides persistence support through the *COM.odi.IPersistent* interface.

initialized. Thus, there can never be *hollow* (or *modified*) instances of one of these types, there can only be *active* instances.

Arrays have identity and are mutable, and so have all the same possible states as class instances. However, because they cannot inherit from *COM.odi.Persistent* (or implement *COM.odi.IPersistent)*, their state fields must be stored out-of-line, in a separate data structure that PSE maintains. Note that this means that access to a persistent array object is more expensive than access to a normal class object, because the *fetch* and *dirty* methods must look up the array in a separate table to determine its current state.

Two other classes bear special mention: *java.util.Hashtable* and *java.util.Vector*. These utility classes can be useful in a variety of database applications, so we generated persistent versions named *COM.odi.util.OSHashtable* and *COM.odi.util.OSVector,* which are shipped with PSE. We re-implemented and renamed these classes, rather than simply post-processing them with the same package and name, for three reasons. First, these classes are used both in the Java VM and in the PSE implementation in ways that would result in bootstrapping difficulties if they were persistence-capable. Second, these classes are widely used both in the Java VM itself and in application code, so the added expense of using persistence-capable classes in transient contexts could have had a performance impact. Finally, while we could have simply post-processed the classes to make them persistent, we chose instead to modify the implementations to make them perform better in persistent contexts.

Memory Management

Another aspect of transparency is the handling of memory management and object management. Java is a garbage-collected language. At the very least, therefore, a persistence interface should not interfere with the (transient) garbage collection built into the Java environment. Ideally, such a system should also extend this garbage collection support by implementing transitive persistence (or persistence by reachability), and garbage collection of persistent space.

PSE supports a model of *persistence by reachability*. When an object is stored in a database, the transitive closure of all persistence-capable instances that are reachable from that object is also stored. If a persistent object refers to other persistence-capable objects, then PSE will migrate the referenced object into the database when the transaction is committed. For example, if a linked list consists of one or more chained *ListElement* instances (and *ListElement* has been made persistence-capable of osjcfp), then by storing the head of the chain in the database, all of the *ListElement*s on the chain will also be stored.

An object becomes persistent, therefore, by one of the following methods:

- *Root Value*: If an object is referred to by a database root, it becomes persistent. Roots are the mechanism for naming objects in a PSE database. All root objects are persistently reachable, so these objects comprise the roots of the persistent reachability graph. There will generally be at least one root in a database since without any, there would be no way to navigate to any of the stored objects.
- *Reference from another persistent object*: If an object that is already persistent is modified to refer to an object that is not yet persistent, then the referenced object will become persistent when the transaction is committed.
- *Explicit migration:* An object can be explicitly migrated by a call to *COM.odi.ObjectStore.migrate*.

Two special cases must be considered when inspecting the fields of an object to compute transitive closure of reachable objects. First, references through fields that are declared with the Java *transient* modifier are ignored when the transitive closure is computed. The value of a transient field on a persistent object is set to its default value when the object is first read from the database (hooks are available that enable the application to reconnect a persistent object to other objects in the transient environment when an object is fetched). The second special case is static fields. These are also ignored, because we treat static fields as fields of the Class object (which is an element of the transient runtime environment) rather than as fields of instances of the class. A warning is issued by the class-file post-processor if a static field is declared to be of a persistence-capable type, telling the user that if the static field is intended to denote a persistent object, then it is necessary to manually initialize the field, for example by making it a root object.

If a persistent object refers (through a field that is neither static nor transient) to an object that is not persistence-capable, then a runtime exception is thrown by *Transaction.commit*.

PSE must maintain the correspondence between an object in the Java Virtual Machine, and an object stored in the database. In general, the application programmer should not have to think about the fact that there are two copies of the object (several cases where this fact does become apparent through the PSE API are described in the next section, ***Transactions***). A single persistent object should never be manifested in the application space as more than one Java object; nor should a single Java object be stored in the database as more than one persistent object. Therefore, if a persistent object is referenced by two different paths, the resulting references obtained must be equal. For example, if the same object is stored as the value of two different roots, *root1* and *root2*, the following expression will evaluate to true:

```
    db.getRoot("root1") == db.getRoot("root2")
```

Similarly, if a transient object is reachable from two different persistent objects, only one persistent copy of it is stored in the database. PSE accomplishes this with an internal *Object Table*, which maintains a bi-directional mapping between a persistent object in the database and the Java object representing the persistent object. When a persistent object is referenced, before a new Java object representing that object is created for the application, the object table is checked to see if one already exists. If an entry in the Object Table is found using the object's persistent ID as a key, the corresponding element in the table is returned to the user rather than instantiating a new object. If the object is not found in the Object Table, then a new hollow object is created and inserted into the table.

The object table has entries for all persistent objects that have been accessed in the current transaction. In early releases of PSE, therefore, the object table became an obstacle to effective garbage collection of the Java VM. In the current release of PSE, however, the object table is implemented with weak references (on those Java VM implementations that support them).[7] This allows the object table to be garbage collected, and any entries for objects to which the user program is not holding references are cleaned up.

There are two places where our goal of transparency and object identity is not fully met, however. The first is a relatively minor issue, alluded to in the previous section: PSE/PSE Pro does not preserve identity for certain objects that are instances of the Java wrapper classes, because it is more efficient to store these objects as values rather than as objects. Only *Long*s and *Double*s are stored as separate distinct objects. Because identity is not preserved, programs that use object identity to compare wrapper class objects, while perhaps not good programming style, work differently when used with persistent objects. For example, this method is incorrect if it is applied to persistent Integer objects:

```
boolean comparePersistentIntegers(Integer x, Integer y) {
  return (x == y);
}
```

Instead, it should be written as:

[7] A weak reference is a Java reference to an object that allows the referenced object to be garbage collected, provided that there are no other references (except weak ones) to that object. It is a concept taken from Smalltalk implementations. The JDK 1.1 for Windows and Solaris, and the SDK 2.0 from Microsoft, support weak references without the use of non-Java (i.e. native) code.

```
boolean comparePersistentIntegers(Integer x, Integer y) {
  return x.equals(y);
}
```

The second limitation in PSE's transparency implementation is more serious: the current release does not include a garbage collector for persistent space. This means that users must explicitly delete objects to remove them from the database. A persistent GC is under development and will be released shortly.

Transactions

All access to persistent storage is mediated by a transaction. By default, once the transaction ends, any persistent objects become inaccessible. Any attempt to use a reference to a persistent object outside of any transaction will cause *NoTransactionInProgressException* to be thrown. At the start of the next transaction, of course, persistent objects become accessible again. An application can then navigate back to any objects of interest by starting from a database root; or it can simply continue to refer to any persistent objects that were read in the previous transaction (provided of course that the application held on to references to these objects). In this second case, we say that the references to persistent objects have been *retained* across the *Transaction.commit*. In either case, the objects will be read back in from the database if they are needed. This behavior is a natural extension of the notion of object identity: a reference to a persistent object will continue to refer to the same object from one transaction to the next.

PSE also provides options for not retaining references (that is, for discarding object identity for persistent objects at a transaction boundary, by explicitly flushing the PSE object table); as well as for retaining not only the references to persistent objects (the object identity), but also the contents of persistent objects. If references are not retained, then any attempt to use a reference to a persistent object in a subsequent transaction will cause *ObjectNotFoundException* to be thrown (of course, it will still be possible to re-navigate to the object from a database root). If on the other hand both the references and the contents of persistent objects are retained, then the contents of any objects that were readable in the previous transaction remain readable even after the transaction commits.

If a Java application or applet has multiple threads, then they may need to either cooperate, or act independently from one another. PSE supports both modes of operation through a transaction *session.* If a thread needs to access a PSE database, it must call *COM.odi.ObjectStore.initialize*. There are two different overloadings of the initialize method: the first overloading creates a new session, which acts independently from all other threads. That is, the thread is *isolated* from threads in other sessions, in the sense that the PSE concurrency control mechanism guarantees that any transaction in this session will be unable to see changes-in-progress from other sessions' transactions, and vice-versa.

The second overloading of the *ObjectStore.initialize* method takes a *java.lang.Thread* argument, and causes the calling thread to join the session of the argument thread. In this case, the threads are said to cooperate, and they can see changes in progress to persistent data. In fact, one thread could start a transaction and make some changes, the other thread could then call *Transaction.commit* to commit the changes to the database.

Example

The following simple program consists of two classes, a *UserConfig* class, which represents a simple user configuration, and a *Browse* class, which makes use of the *UserConfig* to retrieve and update a user's configuration. The *UserConfig* class has two fields, a vector of *java.lang.String*, and an *int* that holds an offset into the vector of *String*s. Several methods are defined on *UserConfig*, including a default constructor for the class, a method that returns the last URL string in the vector, and a method that sets the last URL string in the vector. (A more complete definition might have other methods, to iterate over the entire history, prune selective portions, re-sort, etc. These functions are omitted for simplicity.)

```
import COM.odi.*;
import COM.odi.util.OSVector;

class UserConfig {
  int lastVisitedURLOffset = -1;
  OSVector history = new OSVector();

  String getLastVisitedURL() {
    if (lastVisitedURLOffset == -1) {
      return null;
    }
    else {
      return (String) history.elementAt(lastVisitedURLOffset);
    }
  }

  void setLastVisitedURL(String URL) {
    history.addElementAt(URL, ++lastVisitedURLOffset);
  }
}

public class Browse {
  static Database database;

  public static void main(String argv[]) {
    ObjectStore.initialize(null, null);

    database = Database.open("users.odb", ObjectStore.OPEN_UPDATE);

    Transaction.begin(ObjectStore.UPDATE);

    UserConfig config = findConfig(System.getProperty("user.name"));
    System.out.println("Last URL is " + config.getLastVisitedURL());

    <do some stuff>

    config.setLastVisitedURL(newURL);

    Transaction.current().commit();
  }

  static UserConfig findConfig(String name) {
    try {
      return (UserConfig) database.getRoot(name);
    }
    catch (DatabaseRootNotFoundException e) {
      UserConfig result = new UserConfig();
      database.createRoot(name, result);
      return result;
    }
  }
}
```

The program first initializes PSE and opens a database; then it calls *findConfig()* to look up or create a new *UserConfig* object representing the currently logged in user. Next, the program calls *UserConfig.getLastVisitedURL()* to retrieve and display the last element of the vector in the *UserConfig*

object. After doing some other unspecified operations, the *Browse.main()* method updates the "last URL" by calling *UserConfig.setLastVisitedURL()*. The final act of *Browse.main()* is to call *Transaction.commit()*, which writes any changes back to the database.
This program illustrates a few of the fundamental features of PSE: roots, API transparency, and transactions. Once a root has been retrieved, all subsequent access to persistent data is completely transparent. In the example above, the *UserConfig* class does not require any keywords, or source code modifications in order to store instances persistently in the database. Nor does the code that reads and writes the elements of a *UserConfig* (specifically, the *getLastVisitedURL()* and *setLastVisitedURL()* methods) require any special user annotations (e.g. fetch or dirty calls). The code creates and operates on persistent instances of *UserConfig* just as it would on transient instances.

The example illustrates the use of transactions as a scoping mechanism for all access to persistent data. Transactions are atomic all-or-nothing units of work. A transaction is started by calling *COM.odi.Transaction.begin()* with a transaction type (Update or Read-only) as argument. Persistent data may not be read from or written to the database unless a transaction of the proper type is in progress. Transactions are committed by calling *COM.odi.Transaction.commit()*, or they are aborted (rolled back) by calling *COM.odi.Transaction.abort()*. Transactions may not be nested in PSE. For simplicity, the example above has only a single transaction, which spans virtually the entire program execution. In a more realistic example, of course, smaller segments of processing would be bracketed by *Transaction.begin()* and *Transaction.commit()* calls.

If the *Browse* class accessed any of *UserConfig*'s fields directly, it would have to be made persistence-aware. As it is written, however, all accesses to persistent data are through method calls on *UserConfig* so only *UserConfig* needs to be post-processed. That is, because the *UserConfig* class encapsulates all access to its fields, and only exposes methods, there is no need to make any user of this class persistence-aware. Users of the class need not know whether a particular instance is transient or persistent, or even whether the class itself is persistence-capable.

Performance

In this section we present a simple implementation of the ***oo1*** benchmark[8] using ObjectStore PSE for Java. Briefly, oo1 has two classes, *Part* and *Connection*, which represent parts in a design and the connections between them. Each part has 3 connections to other parts. In each *Part* instance, we used an instance of the OSVector class to reference the 3 inbound and outbound connections. Each Part instance has the following fields: *id* (int), *type* (String), *x* and *y* (int), *time* (int), and *from* and *to* (OSVector) fields. Each Connection instance has a *type* (String), *length* (int), and *source* and *target* (Part) fields. For each Part, 90% of the Connections are randomly selected among the 1% of the parts with id values "closest," while the remaining 10% of the Connections are made to any randomly selected Part. We ran the benchmark with 1000, 2000, and 5000 Part objects (3000, 6000, 15000 Connection objects, respectively).

A Pentium Pro 200 Mhz machine with 32mb of memory was used. Two runs were made, one with the Sun JDK 1.1.2, and the other with the Sun JDK 1.1.1 JIT. For each of the three benchmark runs, the entire database was created in a single transaction. The create times shown below are from the start of the transaction to the end of the commit call, and include the database creation time.

Size	Create Time (secs) JDK 1.1.2, no JIT	Create Time (secs) JDK 1.1.1 with JIT
1000	13.8	9.25
2000	28.2	16.8

[8] R.G.G. Cattell, J. Skeen, "Object Operations Benchmark," ACM Transactions on Database Systems, 17(1), March 1992.

5000	88.0	60.7

The "lookup" part of the benchmark builds a random array of N part id's, where N is the total number of parts in the database. Then, within a single transaction, it looks up each of those parts in the database, recording the time spent doing so. Since each part is stored in an OSHashtable, the lookup is a simple OSHashtable.get() call using the part id. It repeats the N lookups 10 times so that the effects of caching may be seen. The table below shows the "cold" time, which is the average time for the first N lookups, as well as the "warm" time, which is the average time per lookup for the fastest set of N lookups

Size	Cold time per lookup (msec)		Warm time per lookup (msec)	
	JDK 1.1.2	JDK 1.1.1 with JIT	JDK 1.1.2	JDK 1.1.1 with JIT
1000	1.31	.796	.141	.094
2000	1.38	.852	.148	.094
5000	1.54	1.03	.150	.103

Customer Feedback

PSE was made generally available in October 1996. It can be downloaded for free from Object Design's web site (www.odi.com). At this writing it receives more than 80 downloads per day. It is bundled with the Netscape Communicator 4.0 browser, the Symantec Cafe environment, the Asymetrix Supercede IDE, Borland's Latte environment, Natural Intelligence's Roaster IDE as well as other Java-based products. Because it is also distributed through other sources from which we do not receive download information (e.g. www.microsoft.com) we do not have complete information on the total number of copies in the field.

The deployed applications using PSE that we are aware of include:

- *US Open Golf Results*: During the US Open live commentary and results of each hole were stored in a PSE database as audio byte streams. These results were made available via telephone to press and other interested parties. This was a multi-threaded application.
- *Web Spider Applications*: Several user-directed web spider applications that use PSE to store results are known to exist. These programs generally query a user for a starting URL and then go out seeking and displaying information from that URL and the links it contains. The user can save information from the web page, or the URL itself, in PSE.
- *Servlet Applications*: PSE is being used in servlet applications for applications ranging from storing user configuration information to storing newspaper content for redistribution via the web. The size of the data ranges from a few thousand bytes to tens or hundreds of megabytes.

Because PSE was designed to be an entry-level persistence engine, it was an important goal that it be easy to use, and require little support and maintenance. The number of downloads that we have seen, and the high level of traffic on the pse-java-discussion majordomo list (set up for the purpose of discussing PSE usage and issues), together with the low level of support events (fewer than 5 per day) indicate to us that this goal has been achieved, and that most users are able to use the product successfully with little or no assistance.

There have been, however, several problem areas in the product, which have led to user difficulties and support events. These include:

- *classpath and classfile re-writing*: The most common source of confusion for new users centers around the use of post-processed classes in the development and runtime environments. Because post-processor reads in .class files and writes out new ones into another directory hierarchy, the runtime classpath must specify the annotated .class file hierarchy ahead of the original "source" hierarchy (otherwise the classes will not appear to be persistence-capable or persistence-aware at runtime). Some users have tried to correct this by copying the annotated .class files back into the source hierarchy, which unfortunately can cause further problems if they later modify, recompile, and attempt to re-post-process these files. To address this program, we are considering modifications to the post-processor to be able to annotate .class files "in-place".

- *classfile post-processing mechanics*: The classfile post-processor technology requires that all .class files for a program be post-processed at once. (Although this is not a strictly correct statement of the requirement, for many simple applications this is the effective result.) If all of the classes are not postprocessed *en masse*, then persistence-aware classes may not be aware of all of the persistence capable classes that they may encounter, and runtime errors can result, because persistent objects may be accessed without the appropriate *fetch/dirty* calls. We're considering the addition of runtime checks to detect if a user has made this mistake.
- *integration with RMI:* Both PSE and RMI are most easily used by employing inheritance from a special base class (*COM.odi.Persistent*, in the case of PSE, and *java.rmi.server.UnicastRemoteObject*, in the case of RMI). This means that using the two systems together requires some extra work. It is possible to make a class persistence-capable in PSE without inheriting from the Persistent base class, but only by manually annotating the class (the automated generation of persistence-capable classes by our post-processor currently inserts this base class). It is also possible, of course, to use RMI without inheriting from UnicastRemoteObject, but again this requires extra coding to accomplish. We are in the process of updating our post-processor to implement persistence through the IPersistent interface, rather than the Persistent base class, to simplify integration with RMI and other packages.

Future

Future enhancements to the product might include the following:

- *Associative Queries, Indexes*: While PSE supports the JGL (Java Generic Library from ObjectSpace, Inc.) it provides no built-in support for queries or indexes. Many of our users who have asked us for this believe that it is important enough to warrant the significant increase in footprint that it implies.
- *Schema Evolution:* Applications that have deployed with PSE to date are relying on dump/reload mechanisms for dealing with schema evolution. Built-in support for schema evolution is one of our most common enhancement requests.
- *Persistent Garbage Collection:* We intend to release an updated version of PSE that includes a persistent garbage collector in the near future, as discussed above in the section on ***Memory Management***.

Conclusions

The simplicity and flexibility of Java makes it a compelling choice for new application development in a wide variety of product domains. Our goal in the development of PSE was to extend the Java environment with a persistence capability that is easy to use and flexible in terms of underlying storage system implementation. The most challenging aspects of this development have been in building our implementation of transparent persistence in such a way that it melds cleanly with the existing Java environment and tools (including unmodified Virtual Machines), in the creation of a single API that could serve as the front end for a set of storage systems of quite different implementation and capability. Despite these challenges, we feel that we have been largely successful in our fundamental goals.

Java(TM) Persistence via Persistent Virtual Storage

Maynard P. Johnson, Steven J. Munroe, John G. Nistler, James W. Stopyro
IBM Corporation, Highway 52 & Northwest 37th St., Rochester, MN 55901

Ashok Malhotra
IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Hts., NY 10598

Abstract

Persistent Java(TM) objects are important for using Java in business applications. Storing Business objects in a Relational Database (RDB) has proven unsatisfactory. Direct use of a RDB prevents full use of Object Oriented (OO) technology, while schema mapping objects to RDB rows is a difficult problem. An alternative is persistent virtual storage where the persistent and runtime forms of objects are the same. This improves efficiency and allows a simple intuitive persistence program model but requires a large address space. Single Level Store (SLS) large address architecture (48-bit and 64-bit) systems have been commercially available since 1980 (IBM System/38 and AS/400) and recent papers describe similar storage semantics on commodity 64-bit processors. Our prototype implements a Java Virtual Machine (JVM) that simulates large address architecture for 32-bit systems (AIX, Windows95, WindowsNT), then adds a subsystem to create a shared address space (SAS) with persistence. The SAS subsystem provides SLS storage semantics for non-SLS systems. The implementation is targeted towards typical business applications. For performance testing we devised a Business Object Benchmark (BOB-C), based on the well known TPC-C(TM) data processing benchmark, and made some measurements. These are discussed later in the paper.

1.0 Introduction

Currently the Java language [Gosling] directly supports only temporary local objects. Persistent Java objects are important to applying Java to business applications. Traditionally, persistent data for business applications is stored in a RDB, but this has proven to be unsatisfactory. Direct application use of the RDB prevents the application of OO technology to core business problems (using polymorphic behavior to build extensible frameworks). The obvious approach is to support persistent objects by mapping objects to RDB rows. In this model, temporary object instances have their images freeze dried and written to one or more rows. These images can be later resurrected from the freeze dried form into another temporary object. This approach makes it difficult to support complex object relationships and forces harsh compromises between the simplicity of the program model and performance of the application. Maintaining object identity across freeze/thaw cycles is also a challenge. Similar problems apply to storing objects in files.

Persistent virtual storage offers an interesting alternative to object mapping. This technique allows the persistent form of the objects to (closely) match the object's runtime form. It is also efficient as:

- Data is accessed and shared "in place", minimizing data movement overhead
- Object data and references are accessed directly with load and store instructions
- The object's virtual address is persistent, eliminating object identity to virtual address mappings and lookups

The result is a programming model where persistence is as simple to program and as fast to access as temporary objects.

Such technology has been commercially available for over 17 years in single-level storage (SLS) systems, starting with the IBM System/38 and now the AS/400 [AS/400]. SLS systems require large address architectures (starting with 48-bit and now 64-bit addressing). Recent papers [Chase] have demonstrated that the SLS storage model can be applied to commodity 64-bit processors.

Several researchers have developed persistent objects systems on large address SLS systems [Rosen1, Rosen2, Malhotra] and have demonstrated that SLS allows a simple, natural implementation of object persistence.

But the majority of programming platforms are still non-SLS 32-bit. This paper describes a prototype effort to provide the same simplicity of persistent programming that SLS provides to 32-bit non-SLS platforms. Essentially, we implemented a Java Virtual Machine (JVM) to simulate a large (> 64 bit) virtual address, then added a SLS paging subsystem (SAS) to support persistent virtual storage. This JVM executes standard JDK 1.0.2 class files and runs on several popular 32-bit platforms. Persistent storage and associated locks are sharable across all instances of this JVM running on a system. We use this Shared Address Space attribute of our JVM to name our project and prototype "JavaSAS." Details of the implementation are discussed in section 2.

Our implementation was targeted towards standard business applications. In this arena, performance and scalability are key issues that must be addressed, measured, and proven. Currently available Java benchmarks (Caffeine marks) do not measure total system performance under transaction loads (including display and file I/O) that are required for business applications, nor do they address the scaling issue (both total number of persistent and number of concurrently active objects). We implemented a new OO Business Object Benchmark (BOB-C) based on the Transaction Processing Performance Council benchmark C [TPC-C] specification. Since BOB-C reports the same results as TPC-C, we can directly compare OO (BOB-C) and procedural (TPC-C) implementations for popular platforms. A figure of merit for an OO implementation is simply the BOB-C/TPC-C performance ratio. More importantly, it allows for direct comparison between competing object persistence implementations when available for platforms publishing TPC-C results.

The results for BOB-C running on JavaSAS are very encouraging. We have successful BOB-C runs for AIX (RS6000 model 250, 60MHz 601 PowerPC, 128MB RAM), Windows95 (Gateway2000, 200MHz MMX Pentium, 64MB RAM), and Windows/NT (IBM model 750, 166MHz Pentium, 80MB RAM). However, for the following reasons, we can not claim fully qualified TPC-C results:

- We have not yet implemented full roll-back and recovery
- We are running on desktop workstations that do not support the terminal simulators specified in the TPC-C specification
- We are running on desktop workstations that do not have sufficiently large real memory and disk capacity for full scale (item, stock, and customer) object populations for multi-warehouse configurations
- Our results have not been audited by the Transaction Processing Performance Council auditors.

We have tested with multiple terminals simulated by running multiple threads displaying to a single shared test window, as well as with multiple processes each with its own text display window. As text display in windowed systems cause significant CPU loads, we implemented a non-display option to serve as stand-in for a client/server configuration. We have built a fully populated (~2.35 million Java object) BOB-C warehouse and run upto 40 processes against it. We also run multiple BOB-C Warehouse configurations with the item, stock, and customer populations scaled down to fit our available disk space (normally 700MB to 1GB). Even with these restrictions, we have achieved impressive and (to some) counterintuitive results.

2.0 Implementation

The JavaSAS VM integrates the simple and effective object persistence and sharing semantics of single-level store (SLS) with the Java bytecode interpreter and runtime. The initial focus is to bring SLS persistent programming model to 32-bit Java client platforms. Our goal is to make the SLS programming model available for standard 32-bit desk-top systems and the huge pool of programmers that support them. Our implementation involves extensions to the Java VM and runtime environment:

- SAS Persistence Enabling
  - A SAS storage simulator.
  - Extend the JVM to recognize SAS storage references and call the SAS simulator for translations.
  - A Java SAS class loader for activating persistent Java classes.
  - Java lifecycle extensions for creating persistent objects in SAS storage.
- Persistence Model
  - Persistence is orthogonal to type: instances of any class can be temporary or persistent depending on their method of creation. Transient objects are created, as usual, with the Java **new** operation, persistent objects are created with factory methods discussed later.

- Persistent objects are not garbage collected but must be explicitly deleted via a special method on a static class[1].

2.1 SAS Simulator

SLS is an address space hungry technology. A 32-bit address is just not large enough to persistently map all the code and data required for interesting commercial applications. Our solution is to extend the Java VM to simulate a large address architecture (104-bits, following the AS/400 object pointer) for 32-bit platforms. This requires:

- Simulating real memory as a page pool allocated in shared virtual memory.
- Simulating virtual (104-bit) to real (32-bit) address translation as an inverted page translation table.
- Simulating the address translation look-aside buffer (TLB) mechanism as a local hash table.

Obviously on a 64-bit architecture we would not need to simulate address translation. For 64-bit systems the transform from 104-bit to 64-bit is trivial and the 64-bit address can be used directly to access storage.

The 104-bit virtual addresses are stored in reference fields as 128-bit quadwords (which we call SAS addresses). The SAS address space is partitioned into persistent shared global and temporary private local spaces. The vast majority of the SAS address space is reserved for persistent shared storage. SAS addresses are used for all Java references, whether persistent or temporary. The internal structures of loaded (persistently activated) classes and persistent storage heaps use SAS addresses throughout.

References to temporary local objects are also stored as quadword SAS addresses. A portion of the SAS address space is reserved for temporary local objects and supports a trivial translation from SAS to 32-bit addresses (the high order bits have a special value and the low order bits are the virtual address). Temporary SAS addresses always point into the local temporary heap of a specific Java VM instance. This quadword object reference format supports interoperation with future 64-bit JavaSAS implementations. This also ensures that persistent Java object references are context independent.

The SAS simulator function is split into a client instance for each JavaSAS VM and a server instance to manage shared resources for each system. Client and server instances run on the same machine. On a particular system JavaSAS VMs (SAS clients) share a single page pool managed by a single server. This allows efficient sharing of objects between

1. Data is the life blood of commercial enterprises and they require strict controls over access and deletion. Thus, it is unacceptable that data could be deleted by accident (some other object is deleted) or when some other object is updated (pointer set to null). For this reason, JavaSAS does not implement persistence by reachability but requires explicit deletes which can be audited. In some enterprises such as insurance companies or hospitals the law requires that data never be deleted, just marked inactive.

VM instances. To translate a SAS virtual address to a 32-bit process-local address the SAS client first searches a (thread scoped) TLB hash table, then searches the (shared) inverted page table. If the SAS client cannot translate a SAS address the page request is sent to the SAS server.

The SAS server will locate the page on backing store, read it into the shared page pool, and update the shared page translation table. If the requested page does not exist the SAS client is notified and the client throws an address exception. The SAS server manages the page pool LRU, monitors reference and dirty page information, and insures that dirty pages get written out.

SAS addresses are not used on the Java runtime stack (operand stack and local variables). The Java runtime stack is always local to the Java VM instance and accessed via 32-bit addresses. Object references on the stack are 32-bit pointers to TLB entries (which we call SASRefs). All SASRefs can be referenced by their 32-bit address or looked up in the Local Hash Table using the corresponding SAS address. Each SASRef contains the SAS address, corresponding starting 32-bit virtual address in the page pool, and ending address. So, in most cases, only one level of indirection is required to access fields in persistent objects.

Thus, the implementation uses a standard Java stack and introduces no new bytecodes. As we shall see later, it needs to do a bit more work to interpret some of the bytecodes.

The SAS client maintains SASRefs (TLB equivalent) for each Java thread. The SAS client also manages local heap storage for temporary/local Java object instances. Temporary instances are referenced by temporary SASRefs (TempRefs). All SASRefs are subject to garbage collection when the object reference goes out of scope. Temporary object heap storage is freed via mark and sweep garbage collection. Temporary object handles can’t be freed until all the SASRefs and temporary heap (both local object and Copy-On-Write) COW storage) have been scanned for references.

The Java VM supports both local/temporary and persistent/shared (SAS storage) objects. The Java VM primarily uses load/store methods of the SASRef C++ class in its implementation, allowing the VM to largely ignore the details of temporary vs. persistent objects.

References to temporary objects stored in persistent objects require special handling. The Java VM detects the case of storing a temporary reference into a persistent object and the SAS client marks the referenced temporary handle “keep forever” (or until that Java VM instance terminates). Note that this is not a recommended programming practice, since the reference is transient. Temporary SAS addresses are tagged so we can detect if the reference was created in another JVM. There is no problem with storing a persistent reference in a temporary object.

SASRefs to permanent storage (PermRefs) maintain page reference counts for the shared page pool. The page reference count is incremented when the SASRef is created and decremented when the SASRef is garbage collected. Shared SAS pages cannot be removed (stolen) until the page reference count is zero. Thus page removal from the shared pool is indirectly driven by the SASRef garbage collector.

Java class static data fields require special handling. Static fields are scoped to the Java VM instance (static is shared by all threads within a VM but not between VM instances). SAS supports this semantic by simulating copy-on-write (COW) storage. When a Java class is loaded into persistent storage static fields are allocated as separate page(s) within the activated image, initialized, and then persistently marked as a COW page(s). COW pages are not read into the shared page pool but instead are copied directly into the local heap of the VM. SASRefs to COW storage point directly to the local Java VM instance heap and are managed as a VM wide resource

Synchronization also requires attention as it is integrated with Java language semantics. Synchronization must have the same scope as the object instance. For JavaSAS we co-located locks with the objects in the shared persistent store. Specifically, "test and set" locks were stored with the persistent object handles and used an operating system "sleep" for back-off. While this is crude by any standard, it did allow Java applications to use Java language synchronization for both local and shared objects. For a production system we would provide a shared memory lock with thread control.

2.2 Java VM

We modified the Java VM so bytecodes that reference object (instance, static, and array) storage can support SAS addressing. When a SAS reference is loaded (by the getfield bytecode) they are passed to the SAS Simulator for translation. The current strategy is to store the full (128-bit) SAS address in objects and translate them to short (32-bit) references for the operand stack and local (automatic) storage.

This partitions the Java bytecode into four categories:

- Bytecodes that are unaffected
- Bytecodes that load or store object references
- Bytecodes that load or store from persistent objects
- Bytecodes that reference the loaded class structures

Most bytecodes operate on simple data types on the operand stack or local storage or local branches and are unaffected.

Bytecodes that load (128-bit SAS address) object references will need to translate to local (32-bit) references. Bytecodes that store (SAS address) object references will need to perform the reverse translation (32-bit local to 128-bit SAS). This impacts the following subset of the Java bytecodes:

- field puts and gets of object references (putfield and getfield byte codes)
- object reference array loads and stores (aaload and aastore byte codes)
- all persistent array loads and stores (Xaload and Xastore byte codes)
- static puts and gets of object references (putstatic and getstatic byte codes)

The load/store bytecodes are exposed to the (hopefully) rare case of object instances or arrays spanning page boundaries. The SAS simulator does not guarantee that adjacent SAS pages are contiguous in the page pool. While the Java VM runtime should take care in space allocation to avoid page crossings, eliminating page crossing is impossible without imposing draconian restrictions. Since we wish to support large (>4KB) objects and arrays, we handle loads/stores beyond the first (SAS) page referenced. The put/getstatic bytecodes are not impacted because copy-on-write storage is always allocated contiguously.

Page crossing impacts all bytecodes that take an object reference as an operand and load (get) or store (put) data or access the class. If the object is temporary and local, then no further action is required (since local/temporary objects are allocated contiguously by the runtime). Otherwise, if the object is in SAS storage and the field is not in the first page of the object, then (logically) the field/entry offset must be added to the base SAS address and translated to an offset in a valid page within the page pool. The SAS Simulator optimizes this case by chaining secondary SASRefs off the primary SASRef.

In our current implementation the total (excluding quick variants) number of affected byte codes is 35.

- ldc1, ldc2, ldc2w
- iaload, l,f,a,b,c,s,d
- iastore,l,f,a,b,c,s,d
- get/putstatic, get/putfield
- invoke virtual/nonvirtual/static/interface
- new, newarray, anewarray, multianewarray
- arraylength, athrow, checkcast, instanceof

Bytecodes that use object references to invoke methods, reference object fields, or check casts must deal with the internal details of Java handles. In the JavaSAS VM implementation, handles contain the (128-bit) SAS addresses of the instance data and class. The getfield bytecode translates the SAS address of the Java Object handle, but the translation of the instance data and class address is delayed until the handle is used. For invoke (method) bytecodes we translate the instance data and class SAS addresses to SASRefs immediately and store then in the called methods frame. This insures that invocation and field references to these are preresolved.

Invocations and field references using another object reference (not “this”), must resolve the instance data and/or class SAS addresses each time. We are looking at a combined SASRef and local handle to allow instance data and class SASRefs to be cached in the handle. The down side is that it would muddy the line between the Java VM and SAS.

2.3 JavaSAS Class Loader

The JavaSAS Class Loader takes a standard Java class file and loads it into SAS storage. A SAS loaded Java class is (persistently) activated in SAS storage and is persistent, shared and paged like all SAS storage. A SAS loaded class is capable of supporting both temporary and persistent Java object instances.

This impacts the Java class dictionary (which tracks Java class loads and is used to bind class implementations). The class dictionary must be at the same persistence and scope as the classes themselves, so a class name to class address map is created and maintained in SAS storage.

2.4 Java Lifecycle Extensions

Currently Java supports only temporary local objects via the **new** operation. For backward compatibility we preserve the function of **new** unchanged. Applying the **new** operation to a SAS loaded class returns a reference to temporary object in the local Java environment. For SAS loaded classes we add operations to create persistent objects in SAS storage. Once a persistent object instance is created, it behaves exactly like a temporary object, independent of its persistence scope.

To locate related persistent objects close to one another in the vast SAS address space, Java developers need a containment/placement abstraction -- put this object over there! -- which we call Containers, and additional **new** operations. The **new** operations work with the container abstraction to insure that objects instances are created into the correct persistence scope and physical location. This raises some design challenges since the Java language and runtime time does not support metaclass programming. The Class class can not be subclassed! The two possibilities are:

- Generic Factory Class
- Extend java.lang.Object and java.lang.Class

The current prototype supports persistence via a Factory class. The Factory provides static methods to create instances of the specified class. The class is specified as either a (package qualified) class name string or a Class reference. The Factory methods delegate the work of instance creation to either java.lang.Class (newInstance method for temporary objects) or the appropriate Container instance. Containers are special classes with native methods for storage allocation.

The factory methods we have in our prototype are:

- **newPermInstance(Class classRef)** - Used primarily to create Containers and Container Heaps.
- **newInstanceIn(Class classRef, Container containerRef)** - To support placement of new objects for locality of reference or security reasons. Creates an object instance into the specified Container.

- **newInstanceWith(Class classRef, Object nearRef)** - For generic programming of composite objects. Create an instance into the same Container as the existing nearRef.
- **newInstanceNear(Class classRef, Object nearRef)** - For generic programming of composite objects with a strong suggestion for physical placement. Create an instance into the same Container and as close as possible (same page) to the existing nearRef.

To complete the persistence framework we need some additional methods associated with existing persistent objects:

- **destroyInstance()** - Used to destroy a persistent object instance. Redirects to a deallocate method on the appropriate Container.
- **getInstanceContainer()** - Returns a reference to this instance's Container. Used by the implementation of newInstanceWith, newInstanceNear, and destroyInstance.

The Container allocate methods are implemented as native methods integrated with the Java VM. These native methods allocate storage from the Container heap, initialize the Java object handle, invoke the default constructor for the class, and return a reference to the new instance. Since the Container heap was created in persistent/shared SAS storage, any contained object instance is persistent and shared.

This is an effective work around for the lack of metaclass programming and provides a convenient interface for additional **new** operations. We were able to create persistent versions of simple (non-composite, non-array) classes via the Factory methods **newInstanceIn()** or **newInstanceNear()**. For example, we used java.util.Date extensively in our TPC-C implementation. However arrays and existing complex (composite) java class libraries are still awkward and require special Factory methods or arbitrary restrictions.

Neither the Java (1.0.2) language nor java.lang.Class provides a mechanism to programmatically create arrays of an arbitrary type. Arrays of primitive types were handled by writing a special method for each primitive type, such as **newlongArrayNear()**, **newdoubleArrayNear()**,... and implementing specific native Container methods to support them. Arrays of Java Classes required a special **newArrayOfNear()** method with special native method support. **newArrayOfNear()** accepts a (non array) Class reference for the base type and number of elements as arguments and returned an Object[] that has to be downcast to the correct type

Another problem concerns the management of compound objects. A compound object is composed of several sub-objects linked by object references. For example, each Person object may reference an Address object. Or simpler yet, any object containing an array, since in Java all arrays are objects. For example, a Person object includes first and last name fields stored as an character arrays.

The SAS aware Java application will use **newInstanceNear()** to create any sub-objects and arrays. This insures that the sub-objects have the same persistence and scope as the parent object. However the current JDK class libraries and the majority of Java programmers are not SAS aware. The SAS unaware programmer will naturally use the simple **new** operator to create sub-objects. The result would be a shared persistent object containing references to local temporary objects, which may not deliver the desired result.

For example, java.lang.String and java.util.HashTable are both composite objects. Strings are used pervasively so we implemented special native Container methods (**allocString-Near()**) to insure both the String and the internal char array are created at the same persistence and scope. We could not afford to write special Container methods for each interesting class so we simply avoided using these classes for persistent objects.

Now that we have this working and understood it may be appropriate to pursue class library and/or language changes with JavaSoft. We would like to leverage the larger world of Java programmers who are not necessarily persistence aware. To achieve this we need to make this programming model pervasive in the industry as a simple extension to Java.

Being able to extend java.lang.Class would be very useful. It already supports **newInstance()**. It should be simple to add **newPermInstance(), newInstanceIn()**, **newInstanceNear()**, **newInstanceWith()**, and **newArrayOfNear()** to the runtime implementation of Class. This implies that the Container abstraction and a default (temporary, local, garbage-collected) are introduced and shipped with the JDK. java.lang.Object would also need to be extended to support the **destroyInstance()** and **getInstanceContainer()** methods. We have used this framework extensively across several projects and platforms and believe that it is generally applicable and complete. Similar Container and new-near abstractions are integrated into the San Francisco Business Object Framework design [Andrews].

2.5 Exceptions

The single level store paradigm is unique in that it provides access to persistent storage via machine level (load and store) instructions. The traditional file based persistence model requires explicit calls to system APIs. But what happens if the requested operation cannot complete? The file system returns a status variable that reflects the success or failure of each operation: open, close, read, or write. A SLS system supports the equivalent of read and write at the machine instruction level which makes a return code strategy impractical. Instead an exception strategy must be used.

This is a good fit for Java since Java supports and makes extensive use of exceptions. Java supports exceptions in three classes:

- Errors
- Runtime Exceptions
- Exceptions

Errors and RuntimeExceptions are normally thrown directly by the Java VM. Errors reflect conditions where the requested resource is: not valid (VerifyError), inconsistent (NoSuchFieldError), or VM has run out of resources (OutOfMemoryError). RuntimeExceptions reflect a runtime violation of some constraint (IndexOutOfBoundsException, ClassCastException), arithmetic error (ArithmeticException) or the security policies of the environment (SecurityException). Java programmers are not expected to program for these exceptions but can (via try/catch) if they wants to. Just plain Exceptions are part of the API and must be explicitly handled (via a catch clause) if the interface specifies the exceptions (via throws on the method declaration). IOException is an example of this class.

The JavaSAS persistence model introduces some new conditions to be handled:

- ObjectDestroyed -- The object no longer exists. The runtime detection of a dangling reference.
- ObjectAccessDenied -- a form of SecurityException, The object is read-only and the method attempted a store, or an attempt was made to call a method on an object the client has a reference to but is not authorized to.
- StorageReadFailure -- The object exists but the Java/SAS VM can not read the media containing the object. This may be a permanent read error (damaged media) or a temporary condition (the link to the remote object storage is down).
- StorageWriteFailure -- The object exists but the Java/SAS VM can not write to the media containing the object. This may be a permanent write error (damaged media) or a temporary condition (the link to the remote object storage is down).

The Container abstraction introduces more conditions in this category, including:

- ContainerFull -- The physical storage space allocated to this container heap is exhausted.
- ContainerDamaged -- The container is in an inconsistent state and requires a recovery action before it can be used.

These exceptions all seem to fit into either the Error or RuntimeException classification. This relieves the Java programmer from explicitly programming for these conditions and the system default action (terminating the thread) is appropriate in most cases. However, a Java class that implements recoverable resources (like a journal or transaction manager) can catch and program recovery actions.

In the current prototype we have implemented SASException and ContainerException as a subclasses of RuntimeException. All exceptions introduced by the JavaSAS package or the JavaSAS VM are subclasses of SASException or ContainerException.

3.0 Some Observations

For this implementation we had to make some fundamental decisions. The first such decisions was that temporary and persistent instances should have the same format and share a single class image. This required that reference fields be the same size and alignment for temporary and persistent object instances. From our experience with the AS/400, we chose a reference size of 128-bits (16-bytes) with a 104-bit Virtual Address.

We also decided to use a page-based shared memory pool for currently accessed persistent storage pages. This implied that large objects that cross page boundaries would be discontiguous in the page pool. We also decided that references on the operand stack and local variables should be 32-bits. We resolved this problem by enforcing a level of indirection between the operand stack and object fields. Thus, the implementation for each getfield/putfield bytecode must perform the following actions:

- load the current address of the object into the page pool from the reference look-aside
- add any offset to this address
- and check for page boundary crossings
- load/store the target field
- loads/stores of object references must also translate 128-bit references to 32-bit addresses in the page pool and back again

The design decisions above make the JavaSAS object instances larger and object field references slower than the equivalent standard JVM. So the question is “how does the performance and storage size compare to a more traditional (two-level store) persistence model?”. In the following sections we present some informal back-of-the-envelope comparisons.

3.1 Storage impact of large pointers

The first observation is that while a large pointer size (16-bytes vs. 4-bytes) does impact the “foot print” of persistent objects, two other changes have a more significant impact. First, Java Strings are stored as 16-bit Unicode characters. Second, persistent Java objects include Java storage handles and the full java.lang.String object structure (not just the characters).

To test this we simply plug in the population requirements of the BOB-C (TPC-C) specification, the field sizes of the various BOB-C objects, and internal object structure requirements of Java into a spreadsheet. A BOB-C warehouse requires the following object population:

- 1 **Company** and 100,000 **Items** independent of the number of Warehouses
- 1 to N **Warehouses**, where each has:
    - 10 **Districts**
    - 100,000 **Stock** corresponding to the Companies Items
    - 30,000 **Customers**
    - 30,000 initial **Orders**
    - 300,000 initial **Orderlines**
    - 9,000 initial **NewOrders**
    - 30,000 initial order **Histories**

- Each Warehouse, District, and Customer has an associated **Address**

This allows direct comparison between the RDB storage requirements of a TPC-C Warehouse and persistent Java implementations with various reference sizes. The resulting storage requirement for a fully populated BOB-C Warehouse are:

- The Base (RDB equivalent) is 77MBs
- Java object equivalent with 32-bit references requires 245MBs (a 219% increase over the base
- Java object equivalent with 64-bit references requires 267MBs (a 247% increase over the base)
- Java object equivalent with 128-bit references requires 350MBs (a 356% increase over the base)

Note the 200+% storage increase for the 32-bit Java equivalent. This reflects: the additional object runtime overhead (i.e. object handles), use of java.lang.String to replace fixed and variable text, use of Unicode to replace ASCII, use of java.util.Date to replace system date and time and additional object references to avoid some lookups. The biggest contributors are: Unicode, Java Strings and Java Dates. ASCII strings would save approximately 62MBs and implementing Date as a simple long would save approximately 18MBs.

However, a 32-bit persistent object reference does not support the scale required for a large business application using SLS technology and is not a valid comparison. A 32-bit JVM would be forced into a Two Level Store (TLS) implementation. TLS implementations do not need to store the complete Java structure, but object references are more complicated. For example, only the characters of String need to be stored, but the Java national language support strategy implies that Unicode should be preserved.

A TLS implementation tends to have fat persistent object references to support polymorphic types and to simplify object activation and distribution. TLS persistent object references require a class ID in addition to an object ID or key. This allows an empty object instance to be created and activation (stream internalization or schema mapping) to be delegated to the object implementation. In a distributed environment, a database or node ID is required in addition to object ID and class ID. This allows function shipping requests to be routed to a server without resolving the object identity to its storage location. For open systems environments, these IDs are usually defined as 16-byte (128-bit) DCE UUIDs.

If you apply these assumptions (32-byte primary keys and 48-byte persistent object references: 16 bytes for the Class, 16 bytes for the object and 16 bytes for the server) to the spreadsheet, the storage required for a 32-bit TLS implementation jumps to 267MBs--a 247% increase over the base and only 31% less than the JavaSAS equivalent. The net: the 16-byte pointers are not the largest impact on the storage footprint -- object infrastructure, String and Unicode are -- and the result compares favorably with the footprint of many functionally equivalent TLS implementations.

3.2 Measured Performance

Performance is difficult to measure objectively. The goal would be to directly compare fully configured BOB-C measurements to the equivalent TPC-C on the same hardware configuration. Due to limited resources, we were not able to configure BOB-C object populations equivalent to the published TPC-C configurations.

For example, 166MHz Pentium systems with unconstrained (512+MB) DRAM and large (10+GB) RAID storage configuration have published results over 1000 tpmC(TM)[2]. Such a result implies 80 to 110 terminals and requires ~100 TPC-C warehouses to qualify. Our 166Mhz Pentium desktop system had 80MB and 1.7GB of disk space. This configuration was large enough to build a single full scale warehouse or several fractional (1/10th and 1/100th) scale warehouses. The 80MB of DRAM supported a 16MB shared storage pool and 40 instances of the JVM process (simulating 40 independent users/terminals).

So far we have built and measured a single full Warehouse configuration on AIX, Windows95, and WindowsNT. We have measured 11.5 to 12.6 tpmBOB-C running 10 JVM processes each in a separate command prompt (or X-Term) window. This is not very exciting, but right on the money for one TPC-C Warehouse, ten terminals, and terminal wait/think times per specification. A TPC-C run must perform at least 9 tpmC and not more than 12.7 tpmC per warehouse. For higher tpmC (and tpmBOB-C) numbers, more warehouses and terminals would need to be added to the configuration.

We built four 1/10th scale Warehouses and ran 40 JVM processes/windows. We measured 44.6 to 49.8 tpmBOB-C in this configuration on the AIX and WindowsNT platforms -- again, within the qualifying range with nominal wait/think times.

At this point it was clear that our system’s DRAM would not support the number of JVM processes required to run larger Warehouse/terminal configurations. In an effort to measure the “intrinsic” capacity of our technology, we added a benchmark option to run multiple threads within a single JVM process and set the wait/think times to zero. This allows the JVM to run unconstrained but forces all terminal output to a single window. Since scrolling large volumes of text is a significant load on windowed systems, we added a option to disable the display of transaction screens. In this case the screen image is built in memory, but the final display write is not called.

This mode allows the persistence storage mechanism to push to the maximum rate for that CPU. This option also serves as a stand-in for a client/server configuration where the screen display load would be distributed over a number of client systems. In this mode we measured 98.3 tpmBOB-C running 4 threads over 4 1% scale warehouses, on a 166MHz Pentium with 80MB DRAM under WindowsNT 4.0. Similarly we measured a 96.4 tpm-BOB-C with screens enabled and 176 tpmBOB-C with screens disabled running 1 thread over a single 1% scale warehouse, on a 200Mhz MMX Pentium with 64MB DRAM under

2. *Transactions per minute benchmark C, based on TPC-C

Windows95. The difference is explained by a combination of faster processor cycle times and the smaller database fitting entirely in the available DRAM. When running in a three warehouses configuration on the 166MHz system, measures rise to the 116-124 tpmBOB-C range. This shows a relatively graceful degradation for overcommitting real memory.

4.0 Conclusion

While these results are early and incomplete they demonstrate that SLS techniques are applicable to 32-bit standard platforms. Combining SLS technology with Java VM technology provides both an effective and highly portable persistent object model. The same persistent object model is a natural fit for 64-bit systems supporting Java and persistent virtual storage (SLS).

Clearly, the JavaSAS interpreter has to do more work than a standard, transient-only JVM and would not be suitable for an application that does not require persistence. But for commercial applications that require some form of persistence JavaSAS performs acceptably on 32-bit systems while providing a superior programming model.

While our performance claims are modest to date (120 tpmBOB-C Vs 1000 tpmC), it is early yet with many potential improvements untapped. Our JavaSAS VM needs additional tuning especially in the structure of the class image. Since we use 16-byte pointers within the structure of the loaded class image, any unnecessary levels of indirection add significant overhead. Additional hand tuning of the Bytecode dispatch loop is also warranted, including a hand turned assembler implementation. Other important areas to be explored include JIT technology and compilation to native code.

Of course the key comparison should be with other Java object persistence mechanisms. While there are a number of products and published papers on Java persistence, we could not find any published results comparable to TPC-C (or BOB-C). It is important to agree on an appropriate benchmark so that results are directly comparable.

5.0 Bibliography

[Andrews]
Andrews, D.H., IBM's San Francisco Project: Java Frameworks for Application Developers, *D.H. Andrews Group, 700 West Johnson Ave., Cheshire, CT 06410*, Dec. 1996

[AS/400]
IBM Corporation, *IBM Application System/400 Technology Journal (Version 2)*, 1992

[Chase]
Chase, J. S., Levy, H. M., Baker-Harvey, M., & Lazowska, E. D. (undated), Opal: A Single Address Space System for 64-bit Architectures, *University of Washington, Department of Computer Science and Engineering*

[Gosling]

Gosling J., Joy B., & Steele G., *The Java Language Specification*, Addison Wesley, 1997.

[Malhotra]
Malhotra, A., & Munroe, S.J., Support For Persistent Objects: Two Architectures, *Proc. 25th Hawaii Intl. Conf. on System Sciences*, 1992,

[Rosen1]
Rosenberg. J., & Abramson, D.A., MONADS-PC: A CapabilityBased Workstation to Support Software Engineering, *Proc. 18th Hawaii Intl. Conf. on System Sciences, 1985*, pp222-231

[Rosen2]
Rosenberg, J., Koch, D.M. & Keedy, J.L., A Massive Memory Supercomputer, *Proc. 22nd Hawaii Intl. Conf. on System Sciences*, 1989, pp 338-345

[TPC-C]
Transaction Processing Performance Council, TPC Benchmark(TM) C, Standard Specification Revision 3.2, 27 August 1996, *URL: http://www.tpc.org/bench.descrip.html*

JTool: Accessing Warehoused Collections of Objects with Java[1]

S. Bailey and R. L. Grossman[2]
Laboratory for Advanced Computing
University of Illinois at Chicago

July 25, 1997

Point of Contact: R. L. Grossman
grossman@uic.edu

Table of Contents

Abstract

The purpose of the work describe here is to gain experimental experience with data warehouses for large collections of Java objects. We report on the design, architecture, and early experimental work with a software tool called JTool for creating data warehouses of Java objects. Our primary interest is in building distributed data warehouses containing large collections of Java objects as a basis for the data mining of objects on the web. This work is broadly based upon our prior work with a software called PTool which we have used for the data mining of large collections of C++ objects in clustered computing environments [Grossman 1996 and 1997a].

[1] This research was supported by Grants from the National Science Foundation and the Department of Energy.
[2] Robert Grossman is a also a member of the technical staff at Magnify, Inc.

With Version 0.2 of JTool, we have built Gigabyte size data warehouses of Java objects and showed that JTool scales linearly with the size of the warehouse and the size and complexity of the underlying objects. Unfortunately, due to the overhead of the Java Virtual Machine and to our use of object serialization supported by JDK 1.1.1, querying a gigabyte warehouse of Java objects takes approximately 15 hours (vs minutes using PTool).

1 Introduction

Object warehouses are data management systems designed to support the analysis of collections of objects. Where as databases are designed for transactions environments with many writes, data warehouses are designed for analysis environment with many reads. Data warehouses speed up the analysis of data by precomputing and indexing as much derivative and summary data as feasible. At one end of a spectrum are transaction systems supporting updates of simple normalized relational data; at the other end are object warehouses supporting complex queries on complex objects which incorporate derivative and summary data.

Data mining is the discovery of patterns, associations, and anomalies in large data sets. It is convenient to divide data mining systems into three generations [Grossman 1997b]. Most systems today are first generation systems which support one or more data mining algorithms and which interface to file systems and databases. Second generation data mining systems are characterized in part by incorporating data management techniques in order to handle large data sets and to mine data out of memory. Third generation data mining systems support the mining of distributed data, including web-based data. Broadly speaking, there are two approaches to building third generation systems: one based upon agent-based computing and one based upon network-based computing. With Java emerging as one of the de facto standards for network-based computing, it is reasonable to explore the feasibility of designing third generation data mining systems using Java. This paper describes work in this direction.

The purpose of the work describe here is to gain experimental experience with large data warehouses for collections of Java objects. In this paper, we report on the design, architecture, and early experimental work with a software tool called JTool for creating data warehouses of Java objects. Our primary interest is in building data warehouses containing large collections of Java objects. This work is broadly based upon our prior work with a software called PTool which we have used for the data mining of terabyte size collections of C++ objects in clustered computing environments [Grossman 1996 and 1997a].

With this (second) version of JTool, we have built Gigabyte size data warehouses of Java objects and showed that JTool scales linearly with the size of the warehouse and the size and complexity of the underlying objects. Unfortunately, due to the overhead of the Java Virtual Machine and to our use of object serialization supported by JDK 1.1.1, querying a gigabyte warehouse of Java objects takes approximately 15 hours (vs minutes using PTool). Even so, because of the improvements expected in Java, we feel that the type of light weight data management employed by JTool for collections of Java objects may emerge as a viable foundation for third generation data mining systems, a topic we are currently exploring.

Our goals in the design of JTool are broadly similar to the design goals for PTool [Grossman 1995a and 1995b]:

1. We wanted a lightweight data management tool optimized for the types of queries common in data warehouses. Data warehouses are optimized for read-only queries and precompute and index as much data as possible in order to speed the performance of common queries.
2. We wanted the tool to scale to large collections of objects, to objects containing large numbers of attributes, and to queries which are numerically intensive.

3. We wanted the tool to support hierarchical storage systems incorporating memory, disk and tape and a variety of network protocols.

Broadly speaking, we designed a software tool which met these goals and tested it on a several data mining and data intensive applications with which we have worked in the past. We report on the preliminary results here. We emphasize that our intention was not to design a persistent version of Java - this has been done by others [Atkinson et al. 1996a, Atkinson et al. 1996b, Dearle et al. 1996, Garthwaite and Nettles 1996, and Jordan 1996].

It is well known that object serialization can be very efficient, especially for large collections of objects. A standard approach for dealing with this type of inefficiency is to partition hierarchically collections of objects into extents of increasing large size and manage each level in the hierarchy with a separate cache manager. This is the approach we take in this work. We present some evidence that this approach might prove useful in this context.

2 Related Work

Recently there has been a substantial work devoted to adding persistence to Java. Broadly speaking, there are three approaches. One approach to adding persistence is to provide direct support for persistence by changing the Java compiler or the Java virtual machine. Alternatively, a preprocessor for the Java source code could be used or a post-processor for the Java byte code. This has the advantage that it is the most powerful, but the disadvantage that it requires the most work. Another approach is to use existing data management systems to manage persistent data, either relational, object-oriented, or object-relational databases. This has the advantage that databases are widely available but the disadvantage that moving data between the two systems usually requires additional application code. Finally, the Java development environment itself supports a mechanism called serialization [Sun 1996] for taking a complex object and transforming it into a byte stream, which can then be made persistent. This has the advantage that it is well integrated into the Java environment, but the disadvantage that it can be very inefficient. We now discuss each of these approaches in greater detail.

2.1 Directly Supporting Orthogonal Persistence

In some sense the "right" way to provide persistence to Java is well known and articulated in [Atkinson et al. 83 and Atkinson and Morrison 95] through three principles. The first principle is that persistence should be orthogonal to type in the sense that all data whatever its type should have equal rights to persistence. The second principle is the principle of persistent independence which argues that all code should have the same form independently of how long the data upon which it acts persists. The third principle is the principle of transitive persistence which argues that whether data persists or not should be determined by whether the data is reachable from a persistent (root) object or not.

A difficulty with this approach is that providing persistence compliant with these three principles is not easy and requires changing the Java compiler, the Java virtual machine, preprocessing the Java source code, or post-processing the Java byte code. Moss and Hosking [Moss and Hosking 1996] classify some of the different approaches for providing orthogonal persistence to Java. Despite the effort required, there have been several implementations of persistent programming languages following these principles. A version of the Java language called PJava which supports persistence compatible with these three principles is described in [Atkinson et al. 1996 and Jordan 1996]. Another orthogonally persistent Java has been developed by Garthwaite and Nettles [Garthwaite and Nettles 1996]. Dearle et al. have developed an orthogonally persistent Java on top of their persistent operating system Grasshopper [Dearle et al. 1996]. Related material from this point of view is contained in [Morrison et al. 1996].

Malhotra argues in [Malhotra 1996] that scalability issues necessitate supplementing defining persistent objects through reachability with an alternate mechanism such as using explicit adds and deletes.

2.2 Interfaces to Databases and File Systems

There is embedded SQL interface between Java and relational databases called JDBC [Hamilton and Cattel 96]. This has the standard problems -- the impedance mismatch between the data models in the two different environments requires substantial additional programming -- and the standard advantages -- it is extremely useful due to the large amount of relational data in current systems. Related work is described in [Santos and Theroude 1996].

Most of the commercial object oriented database vendors have developed interfaces between their databases and Java. The ODMG is developing a standard for this interface [Cattel 1996]. Since the ODMG data model and the Java data model are similar, there is not the impedance mismatch problem that occurs with the JDBC binding. Most of these implementations appear to support orthogonal persistence.

An even greater amount of data is contained in legacy file systems than is contained in databases. Gruber [Gruber 1996] argues that this type of data is best accessed with external object faulting mechanisms.

2.3 Object Serialization

A serialization of a complex object is a byte stream representation of it. Alternate terms for this process are flattening and pickling. Once an object has been serialized, it can easily be made persistent, either using a file system or database. A common approach is to store serialized objects in a relational database as a BLOB or string. Another use for object serialization is that it provides an easy means to move complex objects between machines.

A closely related technique is to pack complex data into a string or blob. This is important for some high performance decision support and data mining applications. Accessing complex data with a conventional database might require several database accesses. By packing the data, this can be reduced to one operation. The trade-off is that updating the data is more difficult and expensive.

Perhaps the most important reason for employing object serialization to add persistence to Java is that it is easy: the Java Development Kit JDK 1.1.1 provides direct support for object serialization. The main disadvantage is that currently employing object serialization can be very inefficient. Employing object serialization also violates one of the three principles in [Atkinson et al. 83 and Atkinson and Morrison 95]. See [Atkinson et al. 1996b].

3 Using JTool

3.1 API Objects and Utilities

JTool Version 0.2 has three core objects for creating and manipulating large, persistent object stores: Ref, Store, and JTool. There is also a collection object for managing sets of persistent objects called: PSet. Finally, a utility called JTool_Register is required to use the JTool API with general objects.

3.1.1 Ref

A Ref object is a 64 bit reference to an object within a given name space. By name space we mean a collection of Stores. The Ref is broadly modeled after the ODMG specification of the same name; however, the lack of templates and operator overloading required us to change the syntax and exclude type information in Ref.

Ref Methods:

`Ref();`

> The `Ref()` constructor available to the programmer takes no arguments and constructs a "NULL" Ref. By NULL, we mean that the Ref instance is not pointing to a legal position in the store.

`Object Deref();`

> The `Deref()` method takes no arguments and returns the Object that is located at the persistent address in the persistent name space that the instance of Ref is pointing to. `Deref()` throws an exception if the instance is referencing an illegal address in the name space (e.g.. a Store that does not exist).

`void Persist();`

> The `Persist()` method takes no arguments and has a void return. `Persist()` causes the persistent image of the object to become consistent with the current transient object to which the instance of Ref is pointing.

3.1.2 Store

A store object is a named persistent space into which persistent objects can be allocated and out of which persistent objects can be retrieved and modified. The store is also a collection class and supports API methods which allow functionality similar to that of the ODMG Bag class (i.e. allows insertion of duplicate objects). We have included this functionality in order to be compatible with the PTool-v2.3 API. However, future releases of JTool will most like only include a method to add on object to the root of a Store.

Store Methods:

`Store( String );`

> The `Store( String )` constructor takes a String and opens the persistent space of that name. If the store has not be created before, the constructor initializes the store and adds the new store to the name space.

`void insert_element( Ref );`

> The `insert_element( Ref )` method takes a Ref object as an argument and adds the reference to the Store's collection of objects. This method throws an exception if the Ref cannot be added to the collection.

`boolean Next( Ref );`

> The `Next( Ref )` method takes a Ref object as an argument, attempts to set the Ref to point to the next object in the Store's collection and returns a boolean. This method returns true if

another object was available in the collection and false if the end of the collection has been reached and there are no more objects to return. By “next’ object we mean in reference to previous calls to `Next()` starting with the first object being returned on the first call to `Next()`.

```
void Reset();
```

The `Reset()` method takes no arguments, returns void, and insures that the next call to the `Next()` method will attempt to return the first object in the Store’s collection.

3.1.3 JTool

The JTool object handles persistence transactions on the stores in a given name space.

JTool Methods:

```
static Ref New( Store, Object );
```

The `New()` method is static, takes both a Store and an Object as arguments, attemps to allocate the object into the persistent Store and returns a Ref pointing to the persistent location of the object in the name space.

```
static void FinalizeJTool();
```

The `FinalizeJTool()` method must be invoked at the end of a JTool application to bring the object store into a consistent state.

3.1.4 PSet

The PSet is an untyped collection class and supports API methods which allow it functionality broadly similar to the ODMG Bag class (i.e. allows duplicates to be inserted). The name “PSet” is a carry over from the PTool-v2.3 API and will be mostly changed to PBag in future releases.

PSet Methods:

```
PSet( Store );
```

The PSet constructor available through the JTool API takes a Store as an argument and instanciates the PSet. The Store given in the constructor denotes the particular Store where new PSet “nodes” will be allocated. By “node” we mean an object which contains two Refs. One Ref points to an object in the collection and the other Ref points to the next “node” as in a linked list. It should be noted that the Store where “nodes” are allocated does not have any bearing the Store(s) where objects managed by the PSet are located.

```
void insert_element( Ref );
```

The `insert_element( Ref )` method takes a Ref object as an argument and adds the reference to the collection of objects. This method throws an exception if the Ref cannot be added to the collection.

```
boolean Next( Ref );
```

The `Next( Ref )` method takes a Ref object as an argument, attempts to set the Ref to point to the next object in the collection and returns a boolean. This method returns true if another object was available in the collection and false if the end of the collection has been reached and there are no more objects to return. By "next' object we mean in reference to previous calls to Next() starting with the first object being returned on the first call to `Next()`.

```
void Reset();
```

The `Reset()` method takes no arguments, returns void, and insures that the next call to the `Next()` method will attempt to return the first object in the collection.

3.1.5 JTool_Register (Java Application)

The JTool_Register application calculates and registers the "sizeof" objects which are to be made persistent in a given name space. Therefore, all objects which are going to be persistent must be registered with JTool_Register.

```
Usage:

java JTool_Register <object list>
```

3.2 An Example

The purpose of this section is to give a simple example of working with scientific data using JTool. The example in this section is adapted from [Grossman 1994], where further details can be found. There are three main steps to create and access a store of events: first, a schema for the store is designed, which defines the objects and their attributes; second, a store is created and populated with objects; and third, the store is queried for objects meeting specified criteria.

1. The first step is to define and register the schema for the store by defining the objects and their attributes. Simply defining the relevant Java classes, as in Event.java, Jet.java, and Lepton.java found in the appendix does this. It should be noted that most sub-objects (e.g. as Lepton is to Event) should be declared with a Ref in the main object. All objects that are to be persistent must implement Serializable as per Object Serialization in JDK 1.1.1 which simply means adding the words "`implements Serializable`" to the class definitions. This is necessary since JTool currently utilizes Object Serialization found in JDK 1.1.1 to cast byte streams to objects as discussed in section 4.2.

To register one would execute the following command:

```
> java JTool_Register Event Jet Lepton
```

2.The second step is to populate the store. The statement

```
Store store = new Store("PsiEvents");
```

creates a store with the internal handle a, and the external name PsiEvents, or opens the store for appending if it already exists.

When a persistent object is created in a Store, it must also be part of some kind of data structure that allows access to it in the future. JTool comes with a built-in sequential data structure as part of the Store and an external version call PSet.

To make an object persistent, the standard Java statements

```
Event event;
event = new Event();
```

are replaced by

```
Ref  eventRef;
eventRef = JTool.New( store, new Event() );
```

In order to access the object later, it must be added to the base collection in the store with the statement:

```
store.insert_element(eventRef);
```

For a working example, see the source code Pop.java in the appendix.

Attributes for persistent objects can be accessed in the follwing ways:

```
((Event)eventRef.Deref()).vertex = 10.4;
```

or

```
Event event = (Event)eventRef.Deref();
event.vertex = 10.4
```

assigns 10.4 to the vertex attribute of the Event object.

3.The third step is to query the persistent store of objects. Any other process can open the store with the external name PsiEvents and access its elements using the internal handle b with the statement

```
Store store = new Store("PsiEvents");
```

To loop through all objects in a store:

```
Ref eventRef = new Ref();
while( store.Next( e ) )
{ . . . }
```

For a working example, see the source code Pop.java and Acc.java.

4 Design and Implementation

4.1 Physical Storage Management

Both JTool and PTool achieve scalability by hierarchically grouping objects into extents of increasing size: objects are gathered into segments, segments into folios, and folios into stores. A name space consists of a number of stores. The different size extents are then managed by a multi-level caching and migration system. For more details, see [Grossman 1995a].

Segment. A segment is a physical collection of objects. The size of the segment is currently set at 65K. Segments are the basic unit for transferring objects between disks and memory.

Folio. A folio is a collection of segments and the basic unit of managing segments on secondary and tertiary storage systems, such as disks and tapes. Folios are currently implemented as files. After a network Waddle has been implemented, we will be able to distribute the folios of a Store across an arbitrary number of nodes in a network and create stores which are larger than the maximum file size allowed by a single file system.

Store. A store is a collection of folios.

Name Space. A name space is a collection of stores for which there is a JTool Registry and a JTool Database Map. Complex persistent objects can contain sub-objects, which may reside in other stores as long as all the stores in question are in the same name space.

In summary, we designed JTool to use four storage levels for performing physical data management. With this design we can theoretically create and access very large object stores, and yet manage the store efficiently. This is analogous to multi-level caching schemes, which are common in distributed file systems.

4.2 The Use of Object Serialization

The current release of JTool utilizes the Object Serialization support in JDK 1.1.1 to facilitate the casting of bytes held in the JTool Cache to objects and vice-versa. JTool itself, however, provides a much greater functionality than simple Objects Serialization. A JTool name space functions as a randomly accessible, 64bit addressable, persistent heap as opposed to a flat file. Objects can be much larger than the single file size limit and objects can span multiple Stores.

In order to retrieve or store objects, JTool fetches the appropriate segment into the cache as discussed below, creates a temporary ObjectInputstream or ObjectOutputstream at the appropriate offset in the segment were the objects resides or is to reside, and finally invokes `objectWrite( Object )` or `objectRead()` as necessary.

4.3 Architecture

In this section we will examine the internal architecture of JTool. Most notably we will better describe the Ref and outline some important modules including: The Database Map, Object Registry, Waddle (an unfortunate name retained from the PTool system), and Cache. Finally we will step through an example of allocating a new persistent object using JTool.

4.3.1 The Ref

The Ref, points to objects in the persistent space. Once the `Deref()` method is called or a `JTool.New()` is invoked, the Ref also points to a transient image of the persistent object. The attributes of ref are shown below:

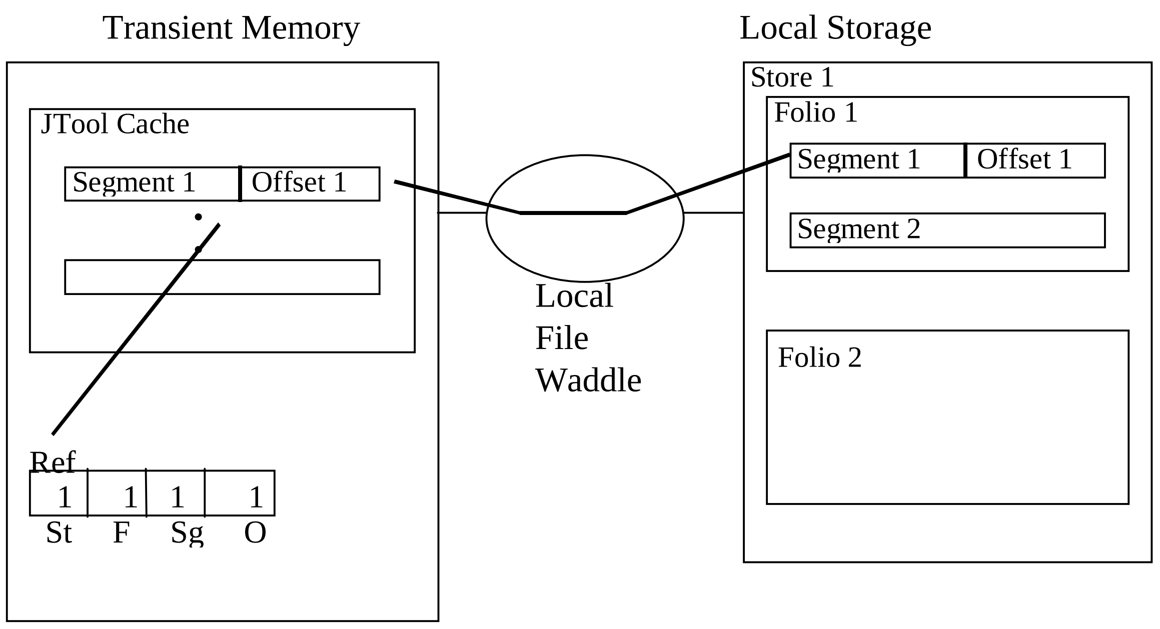
```
class Ref implements Serializable
{
      transient Object object;
      long ppointer;
}
```

The “object” attribute of the Ref is marked as transient for the case when a persistent object A has a Ref to a persisent object B as an attribute, as in:

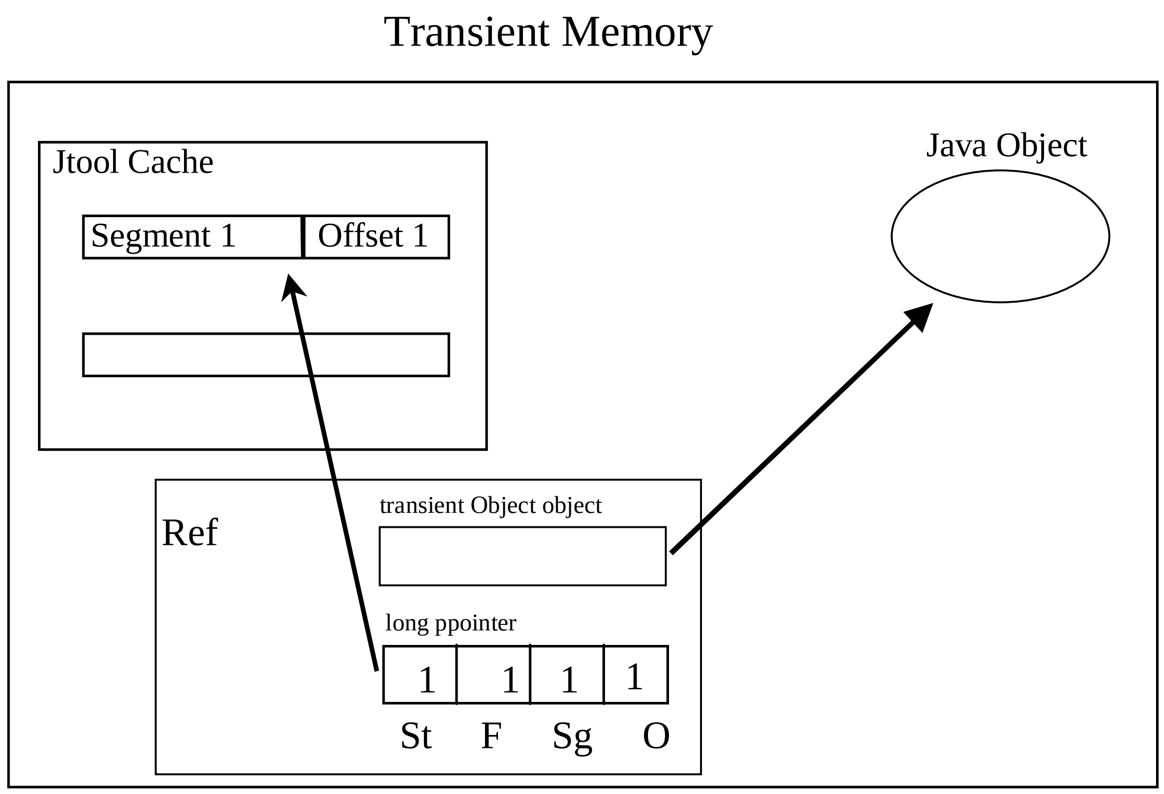
```
class A
{
      Ref b;
}
```

In this circumstance, only the persistent address (i.e. “ppointer” attribute of the Ref for object B) is stored with the object A. The Persist() method of the Ref for object B must be called to make object B persistent. This functionality is desired so that an object can be much larger than the memory limit provided by the transient 32bit addressing space of Java. For example, consider a very large collection of objects as an object itself, if the collection contained 100 Gigabytes of objects it would be required to use Refs since the whole collection cannot fit into transient memory a single time.

The ppointer attribute of a Ref is a long. These 64bits are further subdivided by JTool into four fields: Store, Folio, Segment, Offset. Currently, each field is 16bits long. These four fields are recovered from the long by bit shifting. The figure below shows how these are used.



After the Deref() method has been called on a Ref object and the appropriate segment and offset has been located in the cache, the byte array is “deserialized” into a Java object and “object” attribute of the Ref is updated to point to the new transient image of the persistent object.



4.3.2 Database Map (JTool.DbMap)

The Database Map maps the global identifiers of Stores to ID numbers used in Refs as the most significant 16bits of the Ref's ppointer attribute for objects in the a particular Store. Any time a new Store is created it is assigned a Store ID. The Store ID and the Store name (i.e. global identifier) mapping is added to the Database Map. The Stores contained in a particular Database Map define a name space. Any object in a particular Store can contain Refs pointing to objects in other Stores as long as the Stores in question are in the same name space (i.e. all the Stores are in the Database Map).

4.3.3 Object Registry(JTool.Registry)

The Object Registry maps class types which can be made persistent to a corresponding image size in the persistent Store. The Object Registry is initialized using the JTool_Register application. This effectively gives JTool a "sizeof" method which it can use to allocate appropriate space in the persistent store. The Object Registry is currently implemented as a Serialized Hashtable.

4.3.4 Waddle

The Waddle module of JTool (Waddle is a historical name from PTool) is responsible for the actual fetching of segments from secondary storage into the JTool cache. Currently, only a local file Waddle is provided with JTool. However, like PTool, we expect to have network, compressed file, and tape Waddles available for future implementations of JTool.

There is one instance of a Waddle for every open Store. The Waddle instance fetches segments from secondary memory by opening the appropriate folio (file), seeking to the appropriate offset in the folio (i.e. where the wanted segment starts), and reading or write the segment as requested.

4.3.5 Run-time Store and Waddle Tables

These are look up tables containing pointers to instances of Store and Waddle based on the Store ID.

4.3.6 Segment Cache

The Segment Cache holds a variable number of Segments (decided at compile time) in main memory during the lifetime of a JTool application. When there is a Cache "miss" (i.e. a particular segment is needed but not in the Cache), the Cache makes a call to the appropriate Waddle based on Store ID and requests the particular segment. The current cache eviction policy dumps the oldest segment (i.e. been in the cache the longest). Furthermore, every segment is flushed back to the persistent folios when evicted. We plan to add a differentiation of "dirty" and "clean" segments in future releases. We recognize the cache implementation as naïve as well as a potentially dominant performance factor.

4.3.7 An Example of Object Allocation

Below is a diagram of main memory after a new object Foo is allocated into Store "test" with the following code fragment. This also assumes Foo has been registered with JTool_Register application.

```
.
.
.
Store s  = new Store( "test" );
Ref pf ;
pf = JTool.New( s, new Foo() );
.
.
.
```

Here are the steps invovled:

Code: `Store s = new Store("test");`

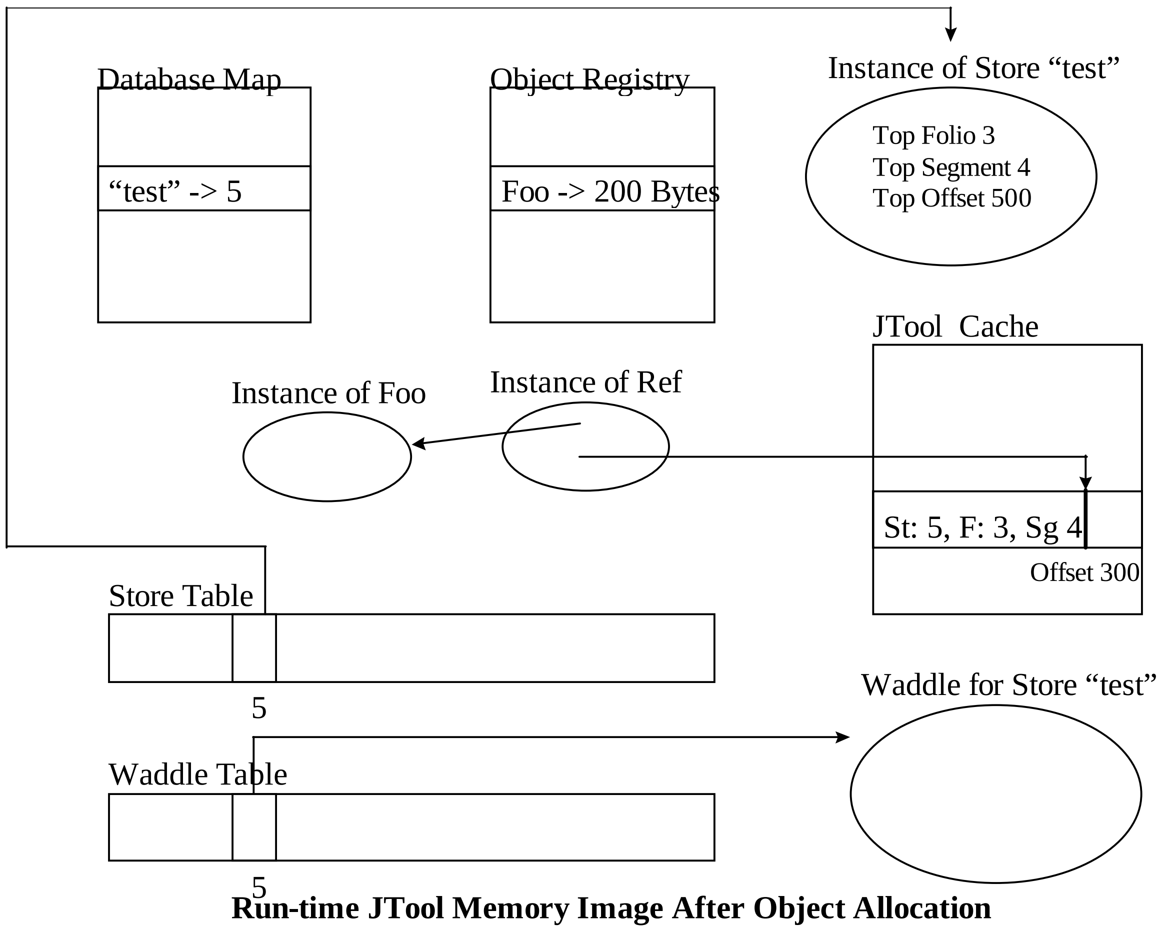
Step 1. Lookup Store ID in Database Map for "test"
Step 2. Initialize Store and Waddle instances and set Store and Waddle Lookup Tables to point to them. In this example, the Store "test" has a Top Folio of 3, Top Segment of 4, Top Offset of 300. These three numbers denotes the current end of the Store; new objects will be allocated after that.

Code: `Ref pf;`

Step 3. Make a Ref variable. Currently it is null.

Code: `pf = JTool.New( s, new Foo() );`

Step 4. Make a new instance of Foo
Step 5. Make a new instance of Ref and set the “object” attribute to point to the new instance of Foo.
Step 6. Look of “sizeof” Foo (i.e. 200 Bytes) in the Object Registry.
Step 7. Set Top Offset of Store “test” to 500 (i.e. allocate 200 Bytes in the Store).
Step 8. Set the “ppointer” attribute of the Ref to [5, 3, 4, 300] (i.e. Store 5, Folio 3, Segment 4, Offset 300).
Step 9. Request the following Segment: Store 5 (“test”), Folio 3, Segment 4.
Step 10. If not present in the Cache, the Waddle for “test” fetches the segment into the Cache.
Step 11. Create a temporary ObjectOutputstream 300 Bytes into the Segment.
Step 12. Call writeObject(Object) on the ObjectOutputstream passing it the new Foo object.
Step 13. Return the new Ref and set pf equal to it.



Run-time JTool Memory Image After Object Allocation

4.4 Status

JTool Version 0.2 was used for the experimental studies reported here. JTool is based upon Version 2.3 of PTool.

5 Experimental Results

5.1 Experimental Facility

Experiments were conducted on a Sun Sparc 20 with 32 Mbytes of RAM running Solaris 2.5.1 and Jdk 1.1.1. The disk on which we populated stores was a 9 Gig Seagate NFS mounted to the SparcStation over standard ethernet. The version of JTool used in the tests was JTool Version 0.2.

5.2 Data Sets

For ease of comparison with our past work mining scientific data, we created a data set containing synthetic data called Events, broadly similar to data arising in high energy physics [Grossman 1995a and 1996]. More specifically, we populated a series of stores using JTool containing Events defined by the class listed in the appendix. The Stores ranged in size from 2 to 1000 Megabytes. The attributes of the events were assigned random values. Each event had 2 Lepton and 1 - 3 Jets as attributes.

We also populated a series of store keeping the number of Events constant but varying the number attributes (Jets) between 5 and 1000.

5.3 Tests

1. First, we measured the access time for examining all entire Event objects (i.e. including its Leptons and Jets) in a store as we varied the Store size.

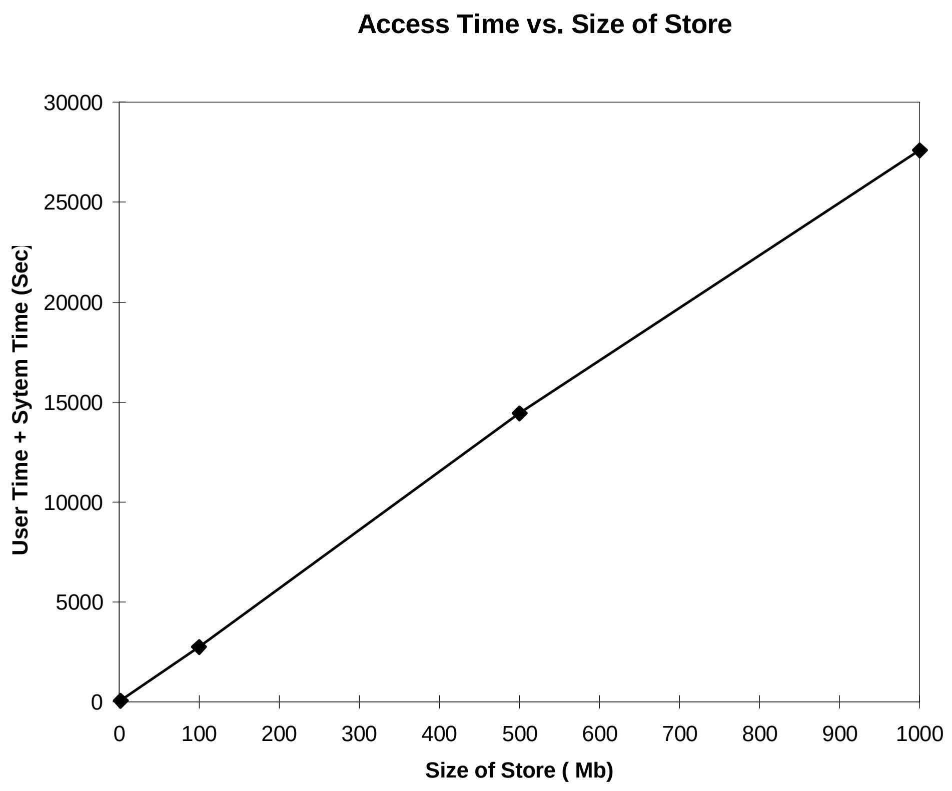
2. Second, we measured the access time for examining all entire Event objects in a store while varying the number of attributes per Event.

The data and tests are designed so that the best performance possible is a linear scale up as the amount of data and number of attributes increases. With inappropriate designs, linear scale up fails after a certain point.

5.4 Summary of Results

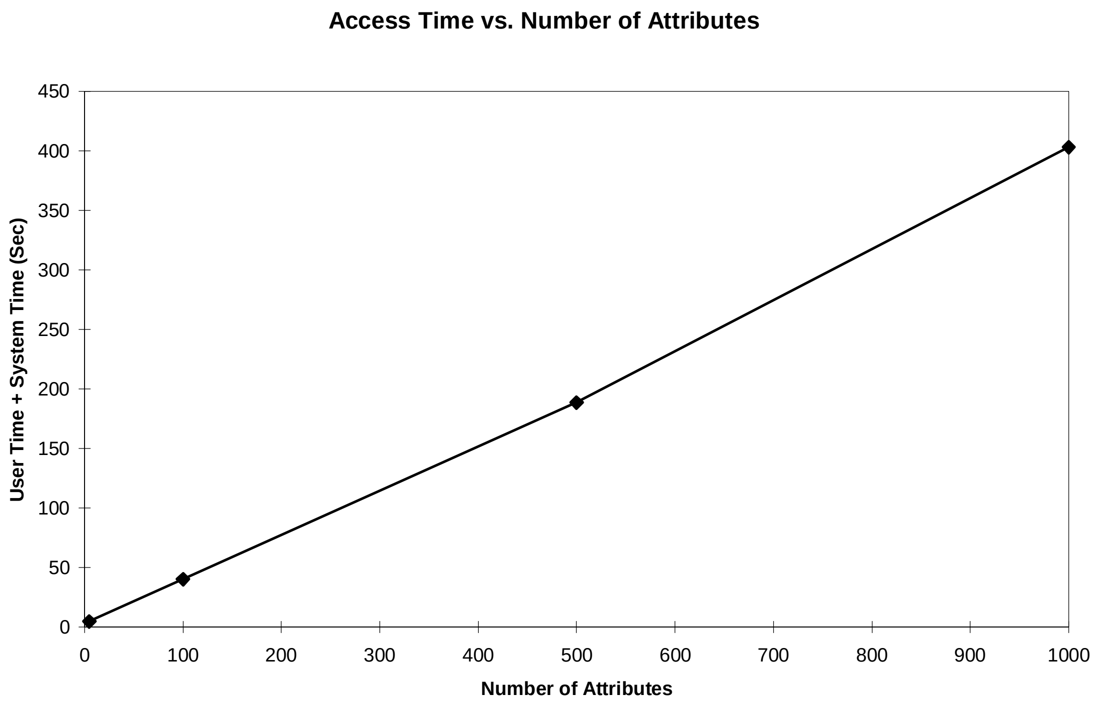
In our first test we found that access times did indeed scale linearly as a function of store size:

	ACCESS TIME (sec)		
STORE SIZE (Mb)	Real Time	User Time	System Time
2	176.13	57.12	1.2
100	5469.1	2726.51	22.12
500	31051.15	14321.51	116.33
1000	51671.77	27378.4	217.11



In our second series of tests we found that access times did indeed scale linearly as a function of the number of attributes:

	ACCESS TIME (Sec)		
Number of Attributes	Real Time	User Time	System Time
5	4.93	4.05	0.7
100	40.56	39.53	0.81
500	191.18	186.95	1.66
1000	534.36	400.37	2.69



6 Summary and Conclusions

In this paper, we have described the design and implementation of a software tool for creating data warehouses of Java objects called JTool and reported on experimental studies showing that JTool scales as designed for data warehouses containing up to one Gigabyte of data and for objects containing up to one thousand attributes.

The current release of JTool (Version 0.2) is our first pass with this design (the prior version had a very different design) and has room for considerable improvement. We are currently experimenting with JTool's caching and migration methods, the developments of alternatives to object serialization, and running additional applications and benchmarks.

Our main interest in JTool is as a data management infrastructure for data mining systems designed to mine collections of complex data distributed over local and wide area networks. With the rapid growth of the net, networked information is growing at a far faster pace than our ability to effectively use it. Much of this information consists of collections of Java objects. Data mining provides one means of automatically discovering patterns and associations in large data sets; without the development of software tools such as JTool, mining collections of Java objects will be more difficult.

7 References

[Atkinson et al. 83] M. P. Atkinson, K. J. Chisholm, W. P. Cockshott, and R. M. Marshall, Algorithms for a Persistent Heap, IEEE Software, Practice and Engineering, Volume 13, pages 259-272, 1983.

[Atkinson and Morrison 95] M. P. Atkinson and R. Morrison, Orthogonal Persistent Object Systems, VLDB Journal, Volume 4, pages 319-401, 1995.

[Atkinson et al. 1996a] M. P. Atkinson, M. J. Jordan, L. Daynes, and S. Spence, Design Issues for Persistent Java: a type safe, object oriented, orthogonally persistent system, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Atkinson et al. 1996b] M.P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence, An Orthgonally Persistent Java, SIGMOD 96.

[Bailey 1997] S. Bailey, A. Goldstein, R. L. Grossman, and D. Hanley, Accessing Warehoused Collections of Objects Through Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Cattel 1996] R. G. G. Cattel, Object Database Standard: ODMG-96, Release 1.2, Morgan Kaufmann, San Francisco, 1996.

[Dearle et al. 1996] A. Dearle, D. Hulse, and A. Farkas, Operating System Support for Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Garthwaite and Nettles 1996] A. Garthwaite and S. Nettles, Transactions for Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Grossman 1994] R. L. Grossman, ``Working With Object Stores of Events Using PTool,'' 1993 Cern Summer School in Computing, C.E. Vandoni and C. Verkerk, editors, CERN-Service d'Information Scientifique 94-06, pp. 66--97, 1994.

[Grossman 1995a] R. L. Grossman, N. Araujo, X. Qin, and W. Xu, ``Managing physical folios of objects between nodes,'' Persistent Object Systems (Proceedings of the Sixth International Workshop on Persistent Object Systems), M. P. Atkinson, V. Benzaken and D. Maier, editors, Springer-Verlag and British Computer Society, 1995, pages 217--231.

[Grossman 1995b] R. L. Grossman, D. Hanley, and X. Qin, ``PTool: A Light Weight Persistent Object Manager,'' Proceedings of SIGMOD 95, ACM, 1995, p. 488.

[Grossman 1996] R. L. Grossman, The Terabyte Challenge: An Open, Distributed Testbed for Managing and Mining Massive Data Sets, Proceedings of the 1996 Conference on Supercomputing, IEEE, 1996.

[Grossman 1997a] R. L. Grossman, S. Bailey and D. Hanley, Data Mining Using Light Weight Object Management in Clustered Computing Environments, Proceedings of the Seventh International Workshop on Persistent Object Stores, Morgan-Kauffmann, San Mateo, 1997.

[Grossman 1997b] R. L. Grossman, Data Mining: Challenges and Opportunities During the Next Decade, submitted for publication.

[Gruber 1996] O. Guber, Transparent Access to Legacy Data in Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Hamilton and Cattel 96] G. Hamilton and R. Cattel, JDBC: A Java SQL API, http://splash.javasoft.com/jdbc, June, 1996.

[Jordan 1996] M. Jordan, Early Experiences with Persistent Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Malhotra 1996] A. Malhotra, Persistent Java Objects: A Proposal, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Morrison et al. 1996] R. Morrison, R. Connor, G. Kirby and D. Munro, Can Java Persist?, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Moss and Hosking 1996] J. E. B. Moss and A. L. Hosking, Approaches to Adding Persistence to Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Santos and Theoude 1996] C. S. dos Santos and E. Theroude, Persistent Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1

[Sun 1996] Sun Microsystems, Inc., Java Object Serialization Specification, Draft Revision, 0.9, http://chatsubo.javasoft.com/current/doc/rmi-spec/rmiTOC.doc.html, 1996.

Appendix A. Event.java

```
import java.io.*;

class Event implements Serializable
{
    int runNumber = 0;
    int eventNumber = 0;

    double vertex = 0.0;

    Ref lepton1 = new Ref();
    Ref lepton2 = new Ref();

    Ref JetSet = new Ref();

    Event()
    {
    }
    Event( Store s )
    {
        runNumber = (int) ( Math.random() * 10000 );
        eventNumber = (int) ( Math.random() * 10000 );

        vertex = (double) Math.random();


        lepton1 = JTool.New( s, new Lepton() );

        lepton2 = JTool.New( s, new Lepton() );
        JetSet = JTool.New( s, new PSet( s ) );


        int numberofJets = (int)( ( Math.random() * 3 ) + 1 );
        for( int x = 0; x < numberofJets; x++ )
        {
            Ref j = JTool.New( s, new Jet() );
            ((PSet)JetSet.Deref()).insert_element( j );
        }

        JetSet.Persist();
    }

    public String toString()
    {
        String JetString = new String();
        Ref j = new Ref();
        PSet ps = (PSet)JetSet.Deref();
        while( ps.Next( j ) )
        {
            JetString = JetString + j.Deref();
        }
        return new String( "Run Number: " + runNumber +
                           "\nEvent Number: " + eventNumber +
                           "\nLepton 1: " + lepton1.Deref() +
                           "\nLepton 2: " + lepton2.Deref() +
                           "\nJet Set: " + JetString +

"\n=========================================\n");
    }
}
```

Appendix B. Lepton.java

```
import java.io.*;

class Lepton implements Serializable
{
    double p[] = null;
    double charge = 0.0;

    Lepton()
    {
        charge = (double) Math.random();
        p = new double[4];
        for( int x = 0; x < 4; x++ )
        {
            p[x] = (double) Math.random();
        }
    }

    public String toString()
    {
        String pstring = new String();
        for( int x = 0; x < 4; x++ )
        {
            pstring = pstring + new String( "\n    p[" + (x+1) + "]: " +
p[x] );
        }
        return new String( "\n  Lepton: " +
                           "\n    charge: " + charge +
                           pstring + "\n" );
    }
}
```

Appendix C. Jet.java

```
import java.io.*;

class Jet implements Serializable
{
    int jet_num = 0, ntrk = 0;
    double q_frac = 0.0, phi_jet = 0.0, eta_jet = 0.0, mass_jet = 0.0,
pt = 0.0;


    Jet()
    {
        jet_num = (int) (Math.random() * 10);
        ntrk = (int) ( Math.random() * 100 );

        q_frac = (double) Math.random();
        phi_jet = (double) Math.random();
        eta_jet = (double) Math.random();
        mass_jet = (double) Math.random();
        pt = (double) Math.random();
    }

    public String toString()
    {
        return new String( "\n Jet: " +
                           "\n    jet_num: " + jet_num +
                           "\n    ntk: " + ntrk +
                           "\n    q_frac: " + q_frac +
                           "\n    phi_jet: " + phi_jet +
                           "\n    eta_jet: " + eta_jet +
                           "\n    mass_jet: " + mass_jet +
                           "\n    pt: " + pt + "\n" );
    }
}
```

Appendix D. Pop.java

```
import java.io.*;


class Pop extends JTool
{
      public static void main( String args[] )
      {

        Store store = new Store( "PsiEvents" );

        Ref eventRef;

        try{
        for(  int a = 0 ; a < 1000 ; a++ )
            {
                eventRef = JTool.New(  s, new Event( s ) );
                store.insert_element( eventRef );
          }
            FinalizeJTool();
        }catch(Exception e ){ System.out.println( "Exception: " + e); }
      }
}
```

Appendix E. Acc.java

```
import java.io.*;

class Acc extends JTool
{
      public static void main( String args[] )
      {
        Store store = new Store( "test" );
        Ref eventRef = new Ref();
      try{
        while(  s.Next( eventRef ) )
        {
            System.out.println( eventRef.Deref() );
        }
        }catch(Exception e){};
        FinalizeJTool();
      }
}
```

About the Editors

Mick Jordan is currently a Senior staff Engineer and Principal Investigator at Sun Microsystems Laboratories. His interests include programming languages, programming environments, software configuration management, and persistent object systems. He has a Ph.D. in Computer Science from the University of Cambridge, UK.

Malcolm Atkinson is a Visiting Professor at Sun Microsystems Laboratories, and has been a Professor at the University of Glasgow since 1984. He has sought to provide better persistence to programmers since working in a CAD and graphics group in Cambridge with Mick Jordan in the early 1970's. He identified the value of orthogonal persistence at VLDB'78, and led a team that built the first orthogonally-persistent language, PS-algol, in 1980. Since then, he has worked on improving the implementation, semantics and programming practices for both persistence and object-oriented systems. Combining persistence with Java is just the next step of the journey.