Toward Assessing Approaches to Persistence for Java™

John V. E. Ridgway
Craig Thrall
Jack C. Wileden
{ridgway,thrall,wileden}@cs.umass.edu

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

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.

1

- 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 pro-
cedure 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 benchmark-
ing. 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

| Measure | | Transient | JSPIN | | | | C with Mneme | |
	Cache	Java	Mneme	O³DB	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

Measure	Cache	Transient Java	JSPIN		OPJ	JDBC	C with Mneme	
			Mneme	O^{3}DB			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-ool`.

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

```java
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

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.