Implementing Orthogonally Persistent Java

Alonso Marquez[1], Stephen M Blackburn[2], Gavin Mercer[1], and John Zigman[1]

[1] Department of Computer Science
Australian National University
Canberra, ACT, 0200, Australia
{Alonso.Marquez,Gavin.Mercer,John.Zigman}@cs.anu.edu.au
[2] Department of Computer Science
University of Massachusetts
Amherst, MA, 01003-4610, USA
steveb@cs.umass.edu

Abstract. Orthogonally persistent Java combines the power of abstraction over persistence with Java's rich programming environment. In this paper we report our experience in designing and implementing orthogonally persistent Java. Our design approach is anchored by the view that any system that brings together Java and orthogonal persistence should *as far as possible* avoid diluting the strengths of Java or the principles of orthogonal persistence. Our approach is thus distinguished by three features: complete *transparency* of persistence, support for both intra and inter application *concurrency* through ACID transactions, and the preservation of Java's property of *portability.* In addition to discussing design and implementation, we present results that show that our approach performs credibly.

1 Introduction

An orthogonally persistent system provides an abstraction over persistence, enabling programs to execute with respect to transient and persistent data without distinction. This provides a simple programming model, reducing the complexity of the application code, and thereby offering a substantial software engineering advantage. The Java platform is a standard which encompasses a programming language, a virtual machine, and an execution environment. The *'write once run anywhere'* philosophy behind the development of the Java platform is a key to its success and a motivator for our approach to orthogonal persistence for Java (OPJ).[1]

The practicality of orthogonal persistence as a powerful tool for managing complex data is increasingly being recognized by industry. Commercial developments such as GemStone/*J* [GemStone Systems 1999], PJama [Atkinson et al. 1996] and the Java Data Objects (JDO) standard [Sun Microsystems 1999] indicate the importance of Java as a catalyst for this interest.[2]

[1] We use the term *orthogonal persistence for Java* (OPJ) in a generic sense, encompassing any attempt to apply orthogonal persistence to the Java programming language.

[2] GemStone/*J* is a trademark of GemStone Systems Inc. and Java and PJama are trademarks of Sun Microsystems Inc.

Orthogonally persistent systems are distinguished from other persistent systems such as object databases by an orthogonality between data use and data persistence. This orthogonality comes as a product of the principles of orthogonal persistence [Atkinson and Morrison 1995]:

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

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

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

The impact of these principles on the design and implementation of an orthogonally persistent system is fundamental and pervasive. In this paper we report our experience in applying orthogonal persistence to Java. Our approach is anchored by our view that any system that brings together Java and orthogonal persistence should *as far as possible* avoid diluting the strengths of Java or the principles of orthogonal persistence. As a consequence of this view, our OPJ is marked by three distinguishing features: complete *transparency* of persistence, support for both intra and inter application *concurrency* through ACID transactions, and the preservation of Java's property of *portability.*

After briefly addressing related work, we describe key design issues in section 3, before we discuss major implementation issues in section 4 and conclude with a performance analysis of the three OPJ systems we have built to date.

2 Related Work

Moss and Hosking [1996] present a taxonomy of approaches to developing orthogonal persistence for Java. Their taxonomy explores choices of model, the degree of transparency and the implementation approach. Implementation approaches range from source to byte-code modification, and from compiler modification to runtime system modification. The taxonomy does not explicitly include the key implementation technology used by our approach—user-definable class loaders to transform classes at class loading time.

While there are a number of examples of orthogonally persistent Java implementations, the two most outstanding systems are PJama and GemStone/*J*, both of which take the approach of modifying the Java virtual machine (JVM).

PJama: The PJama system is a joint project of Glasgow University and Sun Microsystems [Atkinson et al. 1996]. It is based on the Sun JDK platform, where the JVM is modified to extend the system semantics to enable persistence. The system consists of a single JVM with an integrated persistent object store. PJama uses a checkpointing mechanism for flushing updates to the store, although there are proposals for transactional mechanisms for PJama [Daynès 2000].

GemStone/J: GemStone Systems has developed GemStone/*J* [GemStone Systems 1999], a system targeted at Java server solutions. Multiple JVMs use a transactional mechanism to concurrently operate with respect to a centralized store. To implement the system, Gemstone has developed their own Java compliant JVM which directly supports persistent mechanisms. Technical details of the GemStone/*J* implementation are not disclosed.

3 Design

Our design was heavily influenced by the ideal of constructing an orthogonally persistent Java system that is true to the principles of orthogonal persistence and yet does not compromise Java's strengths. This objective raised three significant design issues: the method of *persistence identification*, the approach to *concurrency control*, and the preservation of *portability*.

3.1 Persistence Identification

The third principle of orthogonal persistence states that 'the choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system' [Atkinson and Morrison 1995]. This is widely interpreted as requiring that the persistence of objects be defined *implicitly* through reachability from some root or roots. We interpret this principle more fully to mean that the roots should also be defined implicitly, an interpretation that is concordant with the first principle of orthogonal persistence, which states that 'the form of a program is independent of the longevity of the data which it manipulates' [Atkinson and Morrison 1995]. We achieve this by making class variables ('**`static`**' variables) implicit roots of persistence. Our approach has the significant consequence of making persistence truly transparent, as a Java program requires *no* modification to become orthogonally persistent (see figure 1). By contrast, other systems have required programs to associate explicit textual labels with each root of persistence, which is at odds with the principle of persistence independence.[3]

```
public class Simple {
    static int count = 0;
    public static void main(String argv[]) {
        System.out.print(count + " ");
        count++;
    }
}
```

Fig. 1. Depending only on whether it is loaded by a 'persistence-enabled' class loader, this simple program will exhibit either persistent ('0 1 2 . . . ') or non-persistent semantics ('0 0 0 . . . ').

[3] Although PJama for a long time depended on explicit roots of persistence [Jordan and Atkinson 1999], it now supports both explicit and implicit modes of reachability.

3.2 Concurrency Control

Strong support for concurrency is crucial to our goals. While non-transactional modes of concurrency control (such as the use of shared variables) provide rich, cooperative programming environments, it is very difficult to effectively intermix such approaches with persistence [Blackburn and Zigman 1999]. For this reason, we use transactions, which are the conventional choice for concurrency control in persistent systems (e.g., databases). Transactions deliver coherency in the face of concurrency through the use of *isolation*, in contrast to the more liberal *cooperative* approach implicit in most non-transactional models of concurrency control. Unfortunately, the combined impact of the principle of persistence independence [Atkinson and Morrison 1995] and strict ACID transactional semantics [Haerder and Reuter 1983] means that an orthogonally persistent system that implements only simple ACID transactions will face *extreme* limitations on concurrency (the execution of each application must be confined to a single ACID transaction, thereby curtailing inter-application concurrency) [Blackburn and Zigman 1999].

In order to avoid having to dilute transactional semantics, compromise the principles of persistence, or forgo our goal of strong support for concurrency, we use the *chain-and-spawn* transaction model [Blackburn and Zigman 1999]. This model provides an environment which observes both strict transactional semantics and the principles of orthogonal persistence, while permitting intra and inter application concurrency.

3.3 Portability

Our goal of maintaining Java's 'write once, run anywhere' philosophy has led us to develop a portable design. Portability issues arise in two key aspects of the design: the mechanism by which Java's semantics are extended to include support for persistence, and the portability of the storage layer.

Semantic Extension The behavior of normal Java programs must be extended in order to incorporate the semantics of orthogonal persistence. Such extensions include: the automatic faulting of objects from storage into memory on demand; the writing of modified objects back to disk when necessary; the transformation of objects between storage and heap formats; and the addition of state information for each object recording whether it has been updated, etc. Moss and Hosking [1996] review some of the wide range of ways of making such semantic extensions to Java. One approach (not explicitly mentioned by Moss and Hosking) is that of using Java's class loader mechanism to semantically extend programs at class loading time. This approach has the advantages of not requiring modification to the Java Virtual Machine (JVM), so it can be used with any JVM, and not depending on compiler modifications or post-processing, so it can be used with any Java class. All that is required for a class to become orthogonally persistent is for it to be loaded by an appropriate class loader.[4] This feature combined with our approach to

[4] Specifically, we create `PersistentClassLoader`, a subclass of `ClassLoader` which implements semantic extension mechanisms for persistence. We then ensure that all user-defined classes are loaded by instances of `PersistentClassLoader`. A program invoked as '`java Class [args]`', would execute with persistence semantics if invoked as '`java PersistentClassLoader [optional store args] Class [args]`'.

persistence identification (section 3.1) means that a compiled Java class (such as the one in figure 1) can be orthogonally persistent (or not) depending solely on the class loaders used to load classes for that program—every other aspect of the execution environment, including the class file and the choice of JVM, remains unchanged.

This method of semantic extension introduces a complication relating to Java's reflection mechanism, which provides methods for examining classes and objects and creating or modifying objects. In order to preserve the transparency of persistence, the application's use of the reflection mechanisms should show no trace of the semantic transformations performed by the class loader. Therefore, the transformations must also suitably transform the semantics of the reflection operations.

A similar complication arises with user defined class loaders, which could potentially bypass the bytecode transformation process. Again, a clean solution is provided by the use of semantic extension. The `defineClass()` method of `java.lang.ClassLoader` can be semantically extended to apply bytecode transformations just prior to the loading of each class. Thus persistence semantics will be added to all classes, even those semantically extended by other class loaders. Of course this solution only addresses issues relating to our *mechanism* for semantic extension. We do not address the more abstract problem of semantic extensions which may clash with orthogonal persistence (such as other persistence mechanisms). This problem may arise whenever Java's semantics are extended, whether through the use of a class loader, by modifying the VM, or any other by any other means.

Storage Interface The architecture of our OPJ implementation embodies a clean interface between the Java runtime and the underlying store. By making such a strong separation, we gain portability with respect to the underlying store.[5] We use the PSI storage interface [Blackburn and Stanton 1999], which while providing strong transactional semantics, allows efficient implementations by virtue of the runtime system having direct access to the store's cache.

4 Implementation

The design for OPJ outlined in the previous section raises some challenging implementation issues. We have made three major implementations of such a system and experimented with a number of the many implementation possibilities. Common to all of our systems is a mechanism for performing the semantic extension of classes at class loading time [Marquez et al. 2000]. The specifics of these mechanisms are incidental to the goals of this paper and are omitted for brevity.

The semantic extension of a class to support orthogonal persistence involves a few basic transformations, including: the insertion of 'read barriers', which will ensure that an object has been faulted into memory from storage before it is used; the insertion of 'write barriers', which ensure that updates to heap objects are propagated to storage; and the addition of mechanisms for transforming objects between heap and object store

[5] In fact we have successfully developed bindings to three stores: a major commercial DBMS product, the SHORE object database [Carey et al. 1994], and most recently, to our own object database [He et al. 2000].

representations. While each of these semantic transformations is fairly simple (such as the insertion of a call to a read barrier method prior to each `getfield`), together they raise major implementation issues.

4.1 Structural Choices: Shells and Façades

In our experience, one of the most important implementation issues is the choice of what structure to use for representing *unfaulted objects*. Unfaulted objects exist as a consequence of our use of eager 'swizzling' strategies.[6] There are a variety of swizzling strategies and the choice of strategy has a significant impact on the design of the system [Moss 1992]. For the purposes of this discussion, four broad options exist: *eager swizzling*, where all references are swizzled as soon as an object is faulted into memory; *lazy swizzling*, where references are swizzled when they are first traversed; *no swizzling*, where references remain in store format and a conversion is made each time such a reference is traversed; and *eager swizzling to handles*, where handles hold OIDs (store-level object identifiers) and an in-memory pointer to the referenced object is established the first time the handle is traversed.

Java's strong typing and our goal of portability severely restrict the approaches that we can take to swizzling. The reference fields in a Java object must contain references of the appropriate type, which precludes the simple overloading of a reference field with an OID—ruling out lazy swizzling and no swizzling. We have therefore experimented with the remaining choices: eager swizzling to objects and eager swizzling to handles. These choices lead to two very different mechanisms for representing unfaulted objects, *shells*, and *façades* , respectively.

Shells The *shell* approach uses an 'empty'[7] object (which we call a shell) for each unfaulted object. When an OID is swizzled for the first time, an object of the appropriate type is created and the OID is replaced by a reference to that object. All subsequent swizzles of that OID will result in the translation of the OID to a reference to the same Java object. The first time the shell object is accessed, the corresponding object is read from the store and used to initialize the shell. The initialization process will usually involve swizzling references, in which case the process recurses.

The shell approach is relatively easy to implement. An OID field must be added to each object so that the appropriate store object can be read in when the object is first traversed. The same field can serve the purpose of identifying the object's state with respect to initialization and update (the presence of a valid OID in the OID field can be used to indicate that the object remains uninitialized, another value is used to indicate that the object has been updated). The semantic transformation required for the shell approach is therefore quite trivial—a read barrier is inserted before each `getfield` byte code. The read barrier simply checks to see whether the object to be read has been initialized, and faults it from storage if necessary. Unfortunately shells have one potential drawback—the uninitialized shells may consume a large amount of memory.

[6] The term 'swizzle' refers to the translation of a reference from persistent to transient forms.

[7] By 'empty' we mean that the object has not been initialized with state from the store.

Façades A less memory-intensive alternative to shells is to use *façades*, which have the additional significant advantage of being a mechanism for removing read barriers. A façade is a lightweight representation of an object which masquerades as the real object until such time as the real object is required. The façade must behave like the object that it represents up until the point at which it is first accessed, and once accessed, it must transparently replace itself with the real object. This requires the façade and the real object to be type equivalent and for all classes to be fully 'virtualized' (i.e. all non-private fields made private and external accesses to those fields transformed to use accessor methods such as `getFoo()` and `setFoo()`).

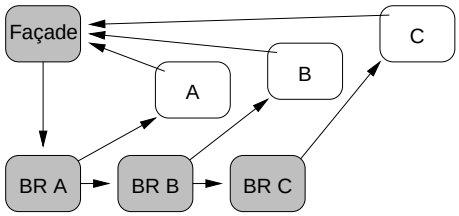


Fig. 2. In order for a façade to be transparently replaced by a real object, all pointers to the façade must be redirected to point to the real object. For this reason, façades must maintain *back references* to all referring objects.

Façades may only be accessed via method invocations (remembering that all classes are fully virtualized), so trapping access to façades simply requires implementing a faulting mechanism in each of the façade's methods. When a façade method is invoked, it creates a corresponding real object, initializes it by faulting in the store object, uses back references to replace all existing references to the façade with references to the real object, and then finally calls the corresponding method on the real object and returns the result. All subsequent method invocations with respect to the object incur no read barrier penalty, as all pointers now refer directly to the real object rather than to the façade.

The semantic transformation required for the façade approach is more complex than that required for shells. Figure 3 illustrates the transformations of a simple two-level class hierarchy. First each class C must undergo a virtualization transformation. Each virtualized class C_v is used to define an interface C_i, and two concrete classes, $C_{v'}$ and C_f, which implement C_i. C_f is the façade class for the 'real' class $C_{v'}$. In the example in figure 3, the implementation $B_{v'}$ of B_i inherits its functionality and fields from the implementation $A_{v'}$ of interface A_i.

Because a façade is never used to represent the state of the real object, each façade can be much smaller than the corresponding real object, containing only enough state to perform the faulting and replacement of references operations.[8] Thus the façade ap-

[8] The number of back-references maintained by a façade can be bounded by adopting a policy of eagerly faulting objects once the number of references to the façade reaches some threshold.

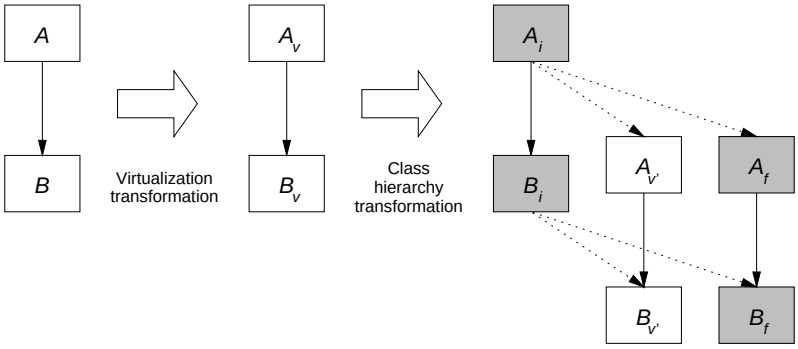


Fig. 3. The use of façades requires a two-stage transformation in order for façades and real objects to be type equivalent. Both transformations occur transparently and automatically at class loading time.

proach may offer a significant space advantage over the shell approach. Furthermore, the façade approach only incurs a read barrier on the first access to each object. Unfortunately these advantage may be offset by the costs associated with full virtualization.

4.2 Other Implementation Concerns

Object Packing and Unpacking When an object is faulted into memory, it must be transformed from a store object into a Java heap object. We refer to this process as *unpacking*, and it involves creating a Java object of the appropriate type and then initializing each of its non-`transient` fields, swizzling references as necessary. The transformation from heap object to store object at commit time is referred to as *packing*, which is simply the reverse transformation. We have experimented with two quite different approaches to packing and unpacking. Our first implementation used C++ pack and unpack methods. We took this approach on account of the perceived performance benefits associated with C++, and in the hope that C++-implemented map structures (used for swizzling) would be faster than Java implementations of the same structures. We quickly found that it was *critical* to minimize the number of traversals of the Java/C boundary, as each traversal is extremely expensive. We also found that the use of Java's reflection mechanism to establish the structure of the transformed object was very expensive.

We have subsequently re-implemented the pack and unpack methods in Java. We did this by including the automatic generation of per-class pack and unpack methods in our semantic extension transformations that occur at class loading time. The major advantages are that this approach minimizes the number of traversals of the Java/C boundary, and that it avoids the use of reflection mechanisms at runtime by 'hard-coding' the appropriate instructions into methods at class loading time. We cannot completely avoid the use of JNI, as the calls to read and write objects from the underlying store are necessarily calls to C functions. The use of Java for packing and unpacking depends a great

deal on the Java libraries efficiently implementing the map structures which are used in the swizzle process.

Transactional Isolation The requirement of *isolation* for any ACID transaction means that all transactions concurrently executing within a single JVM must be isolated from each other's uncommitted actions. The implementation of fine-grained locks to ensure isolation between transactions within a single JVM could be very expensive. Fortunately, the Java class loader mechanism provides a powerful feature in this respect—namespace isolation. Namespace isolation ensures that computations running in distinct class loaders are strictly isolated. Class loaders do not share instances or classes (system classes being the exception), so neither instance nor class variables are visible across namespace boundaries. We make use of the class loader mechanism to cheaply implement transactional isolation by binding each transaction to a class loader.[9] To our knowledge, no other OPJ system incorporates a mechanism for enforcing strict isolation semantics.

System Classes User-defined class loaders, for security and boot strapping reasons, are not able to intercept the loading of system classes. As a consequence, system classes cannot be semantically extended at class loading time. The key impact of this limitation is that we cannot insert read barriers to trap accesses to fields within system classes. The simple (although unsatisfying) solution that we have adopted is to eagerly load objects referred to by system classes. The impact of this is particularly clear in the case of arrays. Because arrays are system classes, we cannot trap accesses to the fields of non-primitive arrays so we eagerly fault *all* unfaulted objects referred to by the array. For a more detailed discussion of this problem, the interested reader is referred to [Marquez et al. 2000].

Thread Persistence Our current OPJ implementations do not make the execution state of threads persist. Because Java threads are first class objects, this limitation is a violation of the principle of data type orthogonality, which states that 'all data types should be allowed the full range of persistence, irrespective of their type' [Atkinson and Morrison 1995]. However, we believe that it will be possible to implement persistent threads within our OPJ framework by using techniques similar to those used to implement thread migration [Hohlfeld and Yee 1998]. Thread migration depends on thread state being encapsulated and then revived after *transmission*, which has much in common with problem of encapsulating and reviving thread state after *storage*.

5 Results

We have used the design outlined in section 3 as the basis for three distinct implementations. In the process, we have explored two major dimensions of the implementation

[9] Note that while we achieve semantic extension by creating a new class loader *class*, we achieve isolation by creating new class loader *instances*. The mechanisms we use for achieving semantic extension and transactional isolation are thus orthogonal.

space: the use of shells versus façades for unfaulted objects and the use of C++ versus Java for object packing and unpacking. We will now analyze the performance impacts of these choices and compare our approach to OPJ with two versions of PJama.

5.1 Experimental Setting

OO7 Benchmark OO7 is an object database benchmark suite [Carey et al. 1993]. The suite consists of a large number of operations including both *traversals*, which are navigational, and *queries*, which are query-like retrievals. The OO7 schema is modeled on that of a CAD application. We have implemented the OO7 benchmark in Java. We used `Chain()` and `Checkpoint()` in ANU-OPJ and PJama respectively at OO7's commit points. The non-persistent Java implementation builds the OO7 database in memory before each set of benchmark operations are executed.

Each execution of the benchmark consisted of ten iterations of each operation. The elapsed time for the first iteration is regarded as a *cold* time (the cache was cold), while the times for all subsequent iterations excluding the last are averaged to produce a *hot* time. To help measure transaction initiation and commit costs, OO7 allows the operations to be executed using *many* transactions (i.e., a commit for each iteration), or *one* transaction (i.e., a single commit at the end of the set of iterations). The size of the OO7 database is configurable, but three sizes are commonly used: *small*, *medium*, and *large*. Due to limitations in our OO7 implementation and the underlying systems, it was only possible to use the small configuration, and seven of the operations could not be executed on one or more of the systems (*q4, i, t2b, t2c, t5do*, and *t5undo*). To gain a quantitative impression of the overall performance of each system, we take the geometric mean of the results for each of the operations.

Systems Used We measured six systems:

label	*description*
ANU-OPJ-S	ANU OPJ using shells with C++ packing.
ANU-OPJ-S-J	ANU OPJ using shells with Java packing.
ANU-OPJ-F	ANU OPJ using façades with C++ packing.
PJama-1.2	PJama version 1.2.1.
PJama-1.6	PJama version 1.6.4.
Java	Non-persistent Java.

All of the Java systems used JDK 1.2.2, and with the exception of PJama-1.2 and PJama-1.6, all used the Hot Spot just in time compiler (JIT). PJama-1.6 uses a JIT as described in [Lewis et al. 2000]. All of the ANU OPJ systems used version 1.1 of the SHORE object database [Carey et al. 1994]. The benchmarks were executed on a Sun Ultra-170 with 128MB of RAM and separate hard disks for the persistent store and log.

5.2 Performance Analysis

Figure 4 provides a picture of the overall performance of the six systems. We see that all of the ANU OPJ implementations perform worse than PJama-1.6 and PJama-1.2

when cold, which suggests that the cost of starting a transaction in the ANU systems is greater than for either of the PJama systems. The large performance differential in the cold results between the persistent systems and Java is unsurprising as the Java system does not incur the I/O overhead associated with faulting objects into the store. The ANU systems perform much better when hot, with ANU-OPJ-S only slightly lagging PJama-1.6 and performing about five times better than PJama-1.2. The particularly good performance over the shorter operations suggests that fast commit times are a factor in this result. The hot 'one' results (figure 4 (c)) show the performance of four of the systems during normal processing (i.e. excluding startup and commit). This indicates that the performance of PJama-1.6 is close to optimal (*cf* Java). The ANU systems are all appreciably slower, but still significantly faster than PJama-1.2.

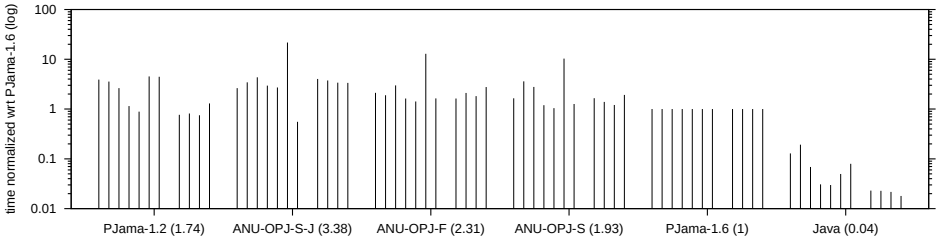
Shells versus Façades A comparison of ANU-OPJ-S and ANU-OPJ-F indicates that the use of façades produces a slow-down in the order of 20%. The success of the façade approach depends heavily on two factors: the ability of the JIT to inline `final` field access methods, and the average size of unfaulted objects being appreciably bigger than façades. These results suggest that the JIT may not be inlining as aggressively as we had hoped, and that the small size of most OO7 objects limits the memory saving that the façade approach can offer over the use of shells.

Java versus C Packing and Unpacking By comparing ANU-OPJ-S and ANU-OPJ-S-J we see that the use of Java for the management of OID to Java object mappings as well as packing and unpacking operations leads to a performance degredation compared to the case where C++ is used. However, while the hot-one times indicate that there is a significant slowdown in runtime performance, the slowdown is much less for the hot-many times. This is probably because the use of Java leads to more efficient packing and unpacking by avoiding the use of Java's expensive reflection mechanisms.

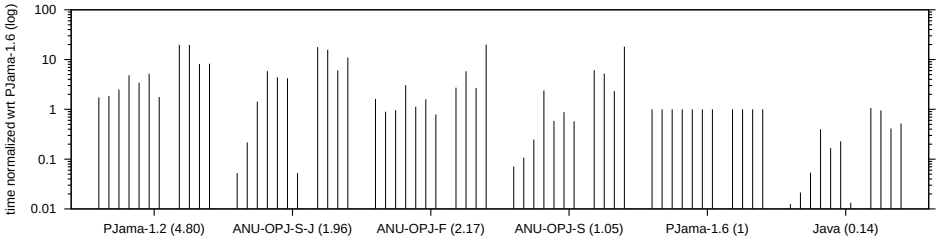
Overall Performance These results indicate that our OPJ implementations are lagging the state of the art. However, a number of important factors moderate this result. First, unlike each of the other systems, our OPJ systems are *fully transactional*, implementing fully ACID semantics and supporting inter-transactional concurrency rather than just implementing an atomic checkpoint mechanism. Second, with sufficient resources, we believe that we could make significant improvements to the performance of our systems—the results presented in this paper are the outcome of a very limited implementation effort. Finally, we do not exclude the value of virtual machine modifications—our approach could happily exploit suitably targeted JVM enhancements without sacrificing its portability.

An obvious source of performance improvement for our systems would be the optimization of read and write barriers. Hosking et al. [1998] have demonstrated significant performance improvements by using code analysis to detect and remove redundant barriers. This approach could be incorporated into the byte code transformations we perform at class loading time.

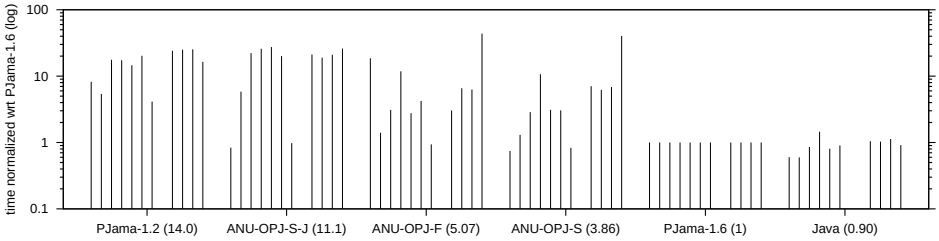
One of the most difficult constraints that we have had to work with is Java's prohibition of user-defined class loaders modifying system classes (section 4.2). The key



(a) OO7 parameters: *cold* and *many*. Reflected in these results: read IO, transaction startup and commit (including write IO), runtime overheads.



(b) OO7 parameters: *hot* and *many*. Reflected in these results: transaction startup and commit (including write IO), runtime overheads.



(c) OO7 parameters: *hot* and *one*. Reflected in these results: runtime overheads.

Fig. 4. The performance of each of the six systems. The results are plotted on a log scale, and each impulse reflects the average execution time for a particular operation on a given system *normalized* with respect to the time for PJama-1.6 for the same query. A result lower than one indicates better performance than PJama-1.6. Each set of results is grouped as follows: queries *q1, q2, q3, q5, q6, q7, d* (left) and traversals *t1, t2a, t3a, t6* (right). The geometric means of normalized results for each system appear in brackets.

impact of this limitation is that we can not insert read or write barriers with respect to field accesses within system classes. A relaxation of this policy would allow us to greatly improve the performance of our systems. A more radical step would be to extend Java with native support for a generic semantic extension language, such as the one we have developed for this project. Our semantic extension language allows extensions to be specified at a very high level, and for the extensions to be applied to classes according to inheritance relationships [Marquez et al. 2000]. Native support for the language would open the possibility of semantic extensions being applicable to system classes. Such an feature would have application well beyond the implementation of OPJ.

Finally, our object faulting mechanisms are expensive because we have no means of directly moving an object image into the Java heap. Instead, a new Java heap object must be created and each of its fields must be separately initialized. An extension to Java that allowed this process to be sped up would improve our 'cold' performance results.

6 Conclusions

We have designed and built an orthogonally persistent Java system, strongly motivated by the desire to remain true to the principles of orthogonal persistence without sacrificing Java's strengths. The product of this effort is a working system with credible performance that is distinguished in three key respects: complete *transparency* of persistence, support for both intra and inter application *concurrency* through ACID transactions, and preservation of Java's property of *portability*. These are achieved by: making class variables implicit roots of persistence, using the chain-and-spawn transaction model, and by making use of Java's provision of a user-definable class loader mechanism.

While performance results indicate that our system is not competitive with the latest release of an OPJ system based on a custom JVM, it seems that much of the performance differential will be susceptible to erosion by improvements in JVM technology and improvements in our implementation approach. The gap might be further narrowed by the provision of suitable hooks in JVMs. We are therefore optimistic that the novel approach to OPJ outlined in this paper will make a genuine contribution to the goal of bringing orthogonal persistence to popular use through Java.

Bibliography

[Atkinson et al. 1996] 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 and Morrison 1995] ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent systems. *The VLDB Journal 4*, 3 (July), 319–402.

[Blackburn and Stanton 1999] BLACKBURN, S. M. AND STANTON, R. B. 1999. The transactional object cache: A foundation for high performance persistent system construction. In R. MORRISON, M. JORDAN, AND M. ATKINSON Eds., *Advances in Persistent Object Systems, Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3) Sept. 1–3, 1998, Tiburon, CA, U.S.A.* (San Francisco, 1999), pp. 37–50. Morgan Kaufmann.

[Blackburn and Zigman 1999] BLACKBURN, S. M. AND ZIGMAN, J. N. 1999. Concurrency—The fly in the ointment? In R. MORRISON, M. JORDAN, AND M. ATKINSON Eds., *Advances in Persistent Object Systems, Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3) Sept. 1–3, 1998, Tiburon, CA, U.S.A.* (San Francisco, 1999), pp. 250–258. Morgan Kaufmann.

[Carey et al. 1993] CAREY, M. J., DE WITT, D. J., AND NAUGHTON, J. F. 1993. The OO7 benchmark. In P. BUNEMAN AND S. JAJODIA Eds., *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26–28, 1993*, Volume 22 of *SIGMOD Record* (June 1993), pp. 12–21. ACM Press.

[Carey et al. 1994] 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.

[Daynès 2000] DAYNÈS, L. 2000. Implementation of automated fine-granularity locking in a persistent programming language. *Software: Practice and Experience 30*, 4 (April), 325–361.

[GemStone Systems 1999] GEMSTONE SYSTEMS. 1999. GemStone/J. http://www.gemstone.com/.

[Haerder and Reuter 1983] HAERDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15*, 4 (Dec.), 287–317.

[He et al. 2000] HE, Z., BLACKBURN, S. M., KIRBY, L., AND ZIGMAN, J. N. 2000. Platypus: Design and implementation of a flexible high performance object store.

In *Proceedings of the Ninth International Workshop on Persistent Object Systems, Lillehammer, Norway September 6–9, 2000* (2000).

[Hohlfeld and Yee 1998] HOHLFELD, M. AND YEE, B. 1998. How to migrate agents. http://www.cs.ucsd.edu/users/bsy/pub/migrate.ps.

[Jordan and Atkinson 1999] JORDAN, M. J. AND ATKINSON, M. P. 1999. Orthogonal persistence for Java? – A mid-term report. In R. MORRISON, M. JORDAN, AND M. ATKINSON Eds., *Advances in Persistent Object Systems, Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3) Sept. 1–3, 1998, Tiburon, CA, U.S.A.* (San Francisco, 1999), pp. 335–352. Morgan Kaufmann.

[Lewis et al. 2000] LEWIS, B., MATHISKE, B., AND GAFTER, N. 2000. Architecture of the PEVM: A high-performance orthogonally persistent Java virtual machine. In *Proceedings of the Ninth International Workshop on Persistent Object Systems, Lillehammer, Norway September 6–9, 2000* (2000).

[Marquez et al. 2000] MARQUEZ, A., ZIGMAN, J. N., AND BLACKBURN, S. M. 2000. Fast, portable orthogonally persistent Java. *Software: Practice and Experience 30*, 4 (April), 449–479.

[Moss 1992] MOSS, J. E. B. 1992. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering SE-18*, 8 (August), 657–673.

[Moss and Hosking 1996] MOSS, J. E. B. AND HOSKING, A. L. 1996. Approaches to adding persistence to Java. In M. JORDAN AND M. ATKINSON Eds., *First International Workshop on Persistence and Java* (Drymen, Scotland, September 16–18 1996). Available online at: `http://www.dcs.gla.ac.uk/~carol/Workshops/PJ1Programme.html`.

[Sun Microsystems 1999] SUN MICROSYSTEMS. 1999. Java Data Objects Specification, JSR-12. http://java.sun.com/aboutjava/communityprocess/jsr (July), Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043.