Providing Orthogonal Persistence for Java™ Extended Abstract

Malcolm Atkinson[1] and Mick Jordan[2]

1 Introduction

A significant proportion of software requires persistent data, that is data that outlives the period of execution of individual programs. This is a simple consequence of the fact that most software is written to support human activities and many human activities continue for days, weeks, months or even tens of years. The design of an aircraft or the construction and maintenance of a large artefact, are typical examples. Over these longer periods it is common to find that a system must evolve to accommodate changing requirements, using additional programs and data, as well as (possibly transformed) earlier programs and data.

Two approaches dominate current provision of persistent data:

- File systems, and
- Database systems.

These have both proved effective for building and operating many applications. As we have argued elsewhere [ABC+83, AM95] these technologies have drawbacks which we believe can be overcome by developing and using orthogonal persistence. In short, orthogonal persistence delivers additional safety and consistency checks and is expected to reduce significantly the cost of building and maintaining sophisticated Persistent Application Systems (PASs) by automating data management tasks and using a consistent model encompassing both short-term execution and long-term data.

The basic strategy is to define and support *one* model for data and program that defines both structure and behaviour for program execution, data (and program) storage and system development. The goal is one computational model, which supports and describes all of the activities on and operations of a PAS. In contrast, the two conventional systems, identified above, have separate models for different parts of the system and for different time-scales. This dichotomy generates complex interactions between subsystems that have to be mastered by programmers. It may also result in unspecified or inefficient operational behaviour.

™ Java is a registered trademark of Sun Microsystems in the USA and other countries.

[1] University of Glasgow, Glasgow G12 8QQ, Scotland

[2] Sun Microsystems Laboratories, Palo Alto, California, USA

E. Jul (Ed.): ECOOP'98, LNCS 1445, pp. 383-395, 1998.

Nevertheless, these contemporary systems have strengths, which remain important. Consequently, we view orthogonal persistence as complementary to those systems; another technology capable of supporting the construction and maintenance of new PAS or the extension of an existing PAS. This requires that orthogonally persistent systems should be capable of inter-working with other contemporary technology.

We believe that there are some applications for which orthogonal persistence is the obvious choice of implementation technology, and that as it develops these will become a substantial proportion of all PAS. We are currently developing one such application, which is a integrated set of tools to support distributed collaborations of programmers building large Java applications[3].

Orthogonal Persistence was proposed twenty years ago [Atk78]. It has since received much research attention concerning both its definition and its implementation. This development is summarised in [AM95], which also provides an entry to the literature on orthogonal persistence research. One effect of that research has been to alert others to the benefits of orthogonality and persistence based on reachability. A prominent example is the latest definition of the ODMG model for orthogonal persistence [ODMG97].

Several strategies are possible for developing the common uniform model:

1. Attempting a design of the persistent computational model from scratch,
2. Evolving an existing database model to encompass computation,
3. Evolving an existing programming language model to accommodate persistence, and
4. Eliding two independently developed models.

The Napier88 research is an example of the first approach [MBC+90, MBC+94], the development of SQL3 is an example of the second, and the Java binding for the ODMG model [ODMG97] is an example of the forth. In contrast, the development of the JDBC™ interface [HCF97], the standard binding to relational systems, though of much practical value, does not fit into any of these strategies since there is no attempt at developing a common model.

The advent of Java provided a “green field” development site for pursuing strategy 3. Java is particularly well suited to this because of its type safety and automated space management. It was also hoped that the fact that PASs built in Java were inevitably exploring new technology would free the architects of the new PAS to consider a new approach to persistence.

Our work developing PJama [ADJ+96, AJD+96] is just one example of the strategy 3 for Java. Several OODB vendors have developed persistence for Java, more or less compliant with the goals of persistence and with the ODMG binding. We will review

---

[3] See the paper by Michael Van Der Vater in these proceedings.

™ JDBC is a trademark of Sun Microsystems in the USA and other countries.

their efforts below. Some vendors have also essayed automated mapping between Java and relational database systems, addressing some of the goals of orthogonal persistence.

Before a more detailed look at the PJama technology, it is useful to define the conceptual framework underpinning orthogonal persistence.

2 Persistence Defined

The term Persistent Programming Language (PPL) was coined to describe programming languages that complied with the three principles which are restated below [ABC+83]. There is a tendency to use that term more loosely now, but we still consider these principles essential. But adhering to them completely is not easy.

2.1 Orthogonal Persistence

Persistence is the provision of arbitrary life times for data. They should continue to exist for as long as there are programs that can or will use them. On the other hand it is an almost essential engineering requirement that their space may be recycled soon after they are last needed.

Orthogonality requires that this range of life times is available for all types of data, irrespective of their type and other properties. We have found that whenever this orthogonality is not supported fully, programmers have to perform extra work translating between transitory and persistent forms of data. This increases program complexity (consequently increasing the frequency of errors) as well as execution costs.

Many systems are not orthogonally persistent because of the difficulty of implementing persistence for all types. For example, threads in general require the state of a thread be captured and stored outside its execution context so that it can be reconstructed in some other context on another occasion (see below).

2.2 Persistence by Reachability

Some mechanism is needed to determine which data are to be preserved. Generally a system provides some mechanism for identifying a data item, `A`, as persistent, e.g. by indicating that it is a persistent root. It is then required that all objects that are directly reachable from `A` (that is, referenced by pointers in `A`) should also be persistent. This rule applies recursively and guarantees the absence of dangling persistent pointers. It avoids programmers having to explicitly identify each persistent value (as happens in some systems) and is a natural extension to the retention rules that govern life times on a garbage collected heap. Overall, programmers find it easier to understand.

2.3 Persistence Independence

This principle requires that the semantics of programs be unchanged by the introduction of persistence, other than that some data may outlive a single program execution. It is a consequence of this rule that the strength of type checking is

undiminished, all previously automated mechanisms are still automated and no explicit mechanism for transferring or locking data is used.

The primary goal of this rule is re-usability — code that has been written for other contexts can be used in the persistent context (on long-lived or transient data) and code developed in the persistent context can be re-used in 'conventional' execution contexts.

This principle cannot be obtained totally. Some code is needed to identify persistent roots for example, or to arrange intermediate atomic checkpoints. However, it can be a minimal proportion of the code. We find users of PJama writing less than ten lines of persistence specific code in applications with tens of thousands of lines of code and hundreds of classes. The rest of the code has the usual platform independence of all other Java code.

3 Matching Java and Persistence

There are several challenges to providing an orthogonally persistent platform for Java. Here we choose to split them between the semantic issues (discussed in this section) and implementation issues (discussed in the following section).

3.1 Orthogonality for PJama

The first decision is "How enthusiastically should orthogonality be pursued?" We decided for PJama that we sought total adherence to this principle (though we haven't achieved it yet—see section 6). This means that every instance of *every* class[4] should have the same rights to persistence. This is relatively straightforward for arrays and instances of user-defined classes and for all of the core classes that are written entirely in Java. However, many of the core classes, and some user-defined classes, have part of their implementation in some other language, typically C. This presents a technical difficulty; it becomes difficult, if not impossible, to keep track of the use of pointers. This will be overcome in the latest versions of Java by the near universal adoption for all native methods of the Java Native Interface (JNI), which allows such issues and automatic locking to be systematically implemented. We simply assert that we will not support any user-defined C code that isn't JNI compliant.

This still leaves a few core classes, such as `Thread`, those classes in the `AWT` packages, etc. which are so intimately inter-related with the Java Virtual Machine (JVM) that they require special treatment. Their code has to be manually modified by the PJama implementation team, to capture sufficient state when an instance is to be promoted to the persistent form and to restore that state when an instance is faulted back in.

[4] We assume readers are familiar with the standard Java language [GJS96].

3.2 Treatment of class `Class`

The target of complete orthogonality described above ineluctably leads to permitting instances of class `Class` to persist. They are one of the core classes that specialises the class `Object`, therefore they should be allowed to persist. However, there is a more important reason why they should persist. The instances of class `Class` contain all of the meta data describing objects and all methods describing their behaviour. To preserve objects without preserving this information, to enable their correct usage in the future, would be futile. The instances without this meta data would simply be a bag of bits with arbitrary interpretation and open to misuse. We therefore ensure that, whenever an instance of a class `C` or an array of instances of class `C` is preserved, we also preserve the `Class C`.

An alternative way of considering this is that we ensure that whenever we store data we also store all other data that is needed to enable consistent interpretation of that data. If we recognise that our computational model needs to be consistent, and that during a normal execution each instance of class `C` is bound to `C`, then we see that persistence independence requires that this precise binding continues unbroken. The requirement to preserve a class `C` in order to interpret its instances correctly extends to all of the classes used to define that class `C`.

The alternative semantics, in which only the class name is preserved with instances, we consider unsatisfactory for three reasons:

1. It enables the semantics of the language to be broken with long-term data, as during the next activation of the JVM that name may be bound to a totally different class (by accident or malevolence).
2. It fails to provide persistence for static variables, which, like all other data should be afforded the privileges of persistence.
3. It is inefficient and does not permit the mutual optimisation of program and data.

3.3 Static Initialisation and Static Variables

Java classes are permitted static variables and methods. Programs may exploit the values in static variables. If these variables are not allowed to persist, then data in this context is not treated orthogonally and the semantics of operations on static variables differ in the persistent context from that in the transient context. This would break two of the persistence principles, and so we chose to make static variables persistent if the class is persistent. For example, if a static variable holds the extent of a class (i.e. its current set of instances), this extent will persist.

This raises the issue of when static initialisation should occur. In standard Java initialisation occurs when a class is loaded, typically the first time it is required during a JVM execution that involves that class. In the persistent case, we choose to initialise the class the first time that the class is loaded into the persistent system. There after it retains its state as it receives successive values. This would yield a persistent extent in the above example.

Other systems choose to rebind to classes and to re-initialise them for each JVM execution. This excludes the persistent use of static variables.

3.4 Persistent Roots

In PJama we have chosen to introduce a set of persistent roots in a `Dictionary` structure which binds names, held as `String`s, to values, held as `Object`s. Any data can then be reached via those `Object`s. Two simpler approaches might have been taken:

1. There could have been a single, pre-ordained root (c.f. PS-algol [ABC+83] and Napier 88 [MBC+94]) of the most general type, `Object`. Typically, programmers then agree to a convention that assigns an associative naming structure to the top value in this root. The advantages of simplicity and generality, which the single root implies, are outweighed by the unreliability of depending on a convention.
2. The static variables of classes that become persistent could act as persistent roots. The classes themselves would not become persistent initially because there would be no instances forcing them persistent in the absence of persistent classes. Therefore a bootstrap mechanism, e.g. explicitly stating that a class should persist, would be necessary. This then becomes equivalent to the named binding scheme, except that the naming scheme has been predefined.

We therefore chose explicit persistent roots. These are managed and a few other operations are made available via the interface `PJStore`. Our particular implementation of this interface is `PJStoreImpl`. This nomenclature has been chosen to encourage alternative implementations. To use persistence in its simplest form, with each persistent JVM execution behaving as a transaction, with atomic durable checkpoints provided by the method `stabilizeAll`, it is only necessary to use these two classes and our modified JVM, `opj`.

3.5 Transient Data

By default the PJama system will make *all* data structures that are reachable (directly and *indirectly*) from persistent roots persistent. All other data structures live their transient lives on the JVM's standard, garbage collected heap. There are two reasons why a programmer may wish to modify this:

1. It is known a priori that this data will never be re-used, and hence economies can be made by not saving it; and
2. The data is intrinsically transient, and must be replaced with equivalent but revised data during subsequent executions.

Standard Java provides the keyword `transient` to denote the former case. The effect of `transient` is to avoid capture of state onto long-term storage and to ensure that if this data is read in any subsequent (different) execution of the persistent

JVM, then that execution will encounter the items marked `transient` re-initialised to their standard default values. PJama honours this semantics[5].

The second case is more interesting, as it draws our attention to the fact that we are trying to define more carefully a computational model that describes long-term computation. In any long-term computation, we must recognise that the context of the computation may change, and that the computation may have to adapt to that change. A mechanism is therefore required to announce and identify the change, and to allow the computation to attempt to reconstruct an equivalent (external) binding. Examples are, connections to sockets, connections to databases, connections to files, and connections to windows. This recognition of an external world with which we interact and an attempt to accommodate its inherent autonomy is essential. However, it forces programmers to face this inevitable complexity.

We provide a set of major events (`PJActions`) e.g. `StartUp`, `Recovery`, `Stabilize` and `ShutDown`, and allow programmers to register `PJActionHandlers` to deal with these events, typically by attempting to reconstruct an external binding. Some standard handlers come with the system, so that programmers, dealing with standard connections to the external world, are provided with a standard rebinding and need not consider this issue.

3.6 Stable Hash Coding

Once we embark on developing a consistent computational model we encounter many deviations from consistency. Most of these are artefacts of the fact that the original design or implementation considered only a limited scope. The `hashCode` method of many classes in Java is a typical example. It has exhibited three problems:

- It is not stable. That is, the repeated application of `hashCode` method returns different hash code values on different occasions. For example, it is common to use the object's address in forming the hash code, but this changes when an object is saved by one Persistent Java Virtual Machine (PJVM) execution and re-instated by another.
- It is not sufficiently accurate or sufficiently uniform. The volumes of data that can accumulate in a persistent system are much larger than occur in a single execution of a JVM. Clustering of the values returned by `hashCode` means that it becomes a poorer approximation to equality than would be expected for the number of bits allocated and that long collision chains occur. The first of these flaws may mean data is brought from disk and read for the precise equality check unnecessarily and the second means that long chains have to be traversed on by faulting them, typically from different pages, on disk. Both have much more deleterious effects on a persistent computation than they have on conventional computations.

---

[5] At present we are forced to endure a work-around for **transient** instance fields because the keyword **transient** has been misused in the `Object Serialization` core class, to mean "this field has a special serialisation".

- The Java definition is ambiguous about whether `hashCode` relates to identity or content equality. Large-scale systems supported by persistence, involve many application programmers. They must interpret the data consistently, but such ambiguities militate against this.

3.7 General Requirement for Stable Definitions

Hash code calculation is just one example of the need to review computational definitions once a longer-term model is sought. We have made a start, but it seems that the scale, growth and complexity of the Java core classes, may mean that we have only scratched the surface of this issue. It is, in our opinion, one which should be of greater concern to those specifying languages such as Java. The requirements for definitions that are stable over many contexts leads to improved quality. It certainly is consistent with aspirations to "compile once and run anywhere" and to "serve the Enterprise".

3.8 Managing Evolution

Any long-lived system has to adapt to meet new requirements and to improve processes. This requires mechanisms that permit change. However, this change has to be managed in a consistent way and implemented efficiently. Our choice of keeping all of the class `Class` instances of relevance in the persistent store has both benefits and drawbacks.

On the one hand having both the class information (program and meta data) and the instances together means that there is enough information to make consistent changes and to ensure proposed changes are consistent. On the other hand, loading methods from the persistent store means that even the smallest program correction requires technology to replace corrected parts, which is avoided by those who load from class files every time.

4 Implementation Architecture

PJama has developed an architecture, which presents persistent data structures to the JVM so that they look to the interpreter identical with those that are found on the garbage collected heap. However, these are held in the object cache. Incremental algorithms fault in and evict these objects via the buffer-pool management system [DA97]. On a stabilisation (invoked by the program, at end of program or end of transaction) the updated data in the object cache is examined. Pointers to instances on the heap, indicate the roots of object graphs that must be promoted to the persistent object store. The mutated instances are then written back. This is implemented by building a special version of the JVM (PJVM) and by modifying several core classes: `ClassLoader`, `Thread`, etc. Many optimisations of the simple scheme presented here are necessary to achieve acceptable performance. Some of these are described in [DA97].

We allow only one PJVM to execute against a persistent store and use the concurrency and locking within Java to service concurrent loads. Multiple read-only

use is also allowed. We also restrict the PJVM to execute against only one persistent store at a time. These restrictions significantly simplify the requirements for persistent application programmers to understand new semantics and simplify our implementation. Only evaluation against serious applications will determine whether this is an appropriate trade-off.

5 Other Approaches

Several other approaches have been taken to providing persistence for Java[6]. The ODMG committee has defined a standard binding which respects both the principle of orthogonality and that of persistence by reachability. This leaves much flexibility over its implementation.

Several of the implementations have significantly restricted the use of core classes, even requiring that different `Hashtables` or proprietary collection classes be used. Such a lack of orthogonality and persistence independence militates against re-usability.

A variety of implementation architectures are in use; some implemented entirely in Java. This has the advantage of achieving acceptable platform independence at considerable cost to performance and (potential) orthogonality (it is difficult to see how to deal with a significant proportion of the core classes in with this approach).

Some implementations re-use existing persistent object-store servers and allow multiple JVMs to interact with servers. Some implementations require a pre-processing pass over the Java source before using a standard compiler. Any implementation that requires access to source will reduce code re-use. To avoid this some implementations depend on post-processing the class files to produce new class files with modifications that provide persistence. Both these approaches suffer from two disadvantages:

- They perturb the Java program build process, and
- The output of diagnostics and debugging tools no longer corresponds to the source that the programmer is working with.

They both have the significant advantage that they should work with any compiler and with any implementation of the JVM. A variation that gives up on the second of these is to generate new byte codes and use an extended version of the JVM to interpret them to deliver persistence. PJama takes advantage of this technique, as we believe it gives a significant performance advantage. But we do not perturb the build process, as this transformation occurs as the class file is first loaded.

Our decision to include class `Class` in the orthogonal set does not appear to be replicated, though some products permit classes to be explicitly placed in the store.

---

[6] Specific products and papers are deliberately not identified here. To make explicit comparisons at this stage, when so many products and projects are evolving rapidly, would be injudicious and unfair.

There are also a group of products that map automatically between Java and relational (or object-oriented) databases. These appear to still be in an early state of development and consequently may have significant orthogonality failures or performance problems. However, they may work well for a carefully selected set of "Enterprise classes".

In a few cases, application programmers are being asked to manually achieve the effect of a write barrier; they have to write code to explicitly notify that an object has been updated. In some implementations they have to explicitly mark classes as constructing instances capable of being made persistent. They also have to mark methods (whole classes) as handling potentially persistent objects, presumably so that the read barrier for object faulting can be selectively inserted. Such requirements to adorn source code with annotations for persistence have pathological effects. It means that code sharing cannot take place at the class file level and that there is virtually no chance of code being re-usable in any other (persistent) context.

The total set of available implementations from both commerce and research indicates the importance attached to persistence for Java. It also presents consumers with a huge and confusing variety and with a potentially very difficult set of choices. It is certain that there are different kinds of application, and some approaches to persistence may be optimal for one kind, while other approaches match other applications. What is needed is a better categorisation of persistent applications and systematic evaluation against the criteria that that categorisation generates. We have begun such an evaluation, but it requires much broader input from the object-oriented research community.

6 Status and Future of PJama

Our close collaboration with Sun, and access to the developing versions of the JDK, have made it possible for us to release versions of PJama compatible with the JDK to run on Solaris SPARC, Solaris x86, Windows NT and Windows 95. These have typically been available on the web site, http://www.sunlabs.com/research/forest/ within days of the corresponding JDK release. The tracking of Java development has taken, and continues to take, a significant amount of labour. However we consider it important, if we are to have our research properly evaluated.

We strongly encourage downloads of the system for evaluation and research purposes. For licensing reasons, each user organisation has to register with SunLabs. So far, there are about 100 registered user sites, as well as our own usage, and no one has been refused access to the system.

The system uses the standard Sun Java compiler, but a modified interpreter. Versions of this interpreter are available for the platforms mentioned above. We supply an off-line disk garbage collector and an off-line tool for substituting persistent classes. A recent achievement has been the provision of persistent RMI.

The facilities we deliver fall short of our goals at present, in the following respects:

- We have not yet achieved full orthogonality. The major limitation is the failure to deal fully with AWT peers, and with Threads. The reasons for this are, in the first case, the labour involved in dealing with so much C, which should become largely unnecessary when JNI is used universally. In the second case, it is currently hard to get at the complete state of a Thread. However, we currently support all of the Swing Set JFC.
- Our store is currently based on RVM [SMK+94]. This is limiting the maximum store size to 2 Gbytes and restricting us to a no-steal policy for buffer management. The largest store constructed so far had a population of 160 million objects.
- We do not yet have either concurrent or incremental disk garbage collection.
- The class substitution technology does not yet support reformatting existing class instances.

The plan that we are currently pursuing, includes:

- Development of our proposed concurrent and flexible transaction model [AJD96].
- Replacement of our existing store technology with store technology that is much more flexible and which is much less closely coupled to the particular interpreter technology [PAD+97], which allows a buffer steal policy and which will accommodate several orders of magnitude more data. This store will also support work on incremental and concurrent garbage collection.
- Development of co-ordinated store and class evolution techniques.
- Implementation of instrumentation, visualisation and diagnostic technology, appropriate for large persistent applications. This will be used to develop a better understanding of the factors that affect performance in large persistent object stores.
- Tracking the development of Java. For example, handling JDK 1.2 and HotSpot™. We believe that the move to HotSpot will put us in a position to achieve our goal of complete orthogonality.
- Developing further optimisations, both through code transformations and through better object cache and buffer management algorithms.
- Evaluating our technology via larger, more realistic test loads and by building extensive and long-running PAS. This will also continue to involve an element of comparison with other technologies, both with respect to execution costs and with respect to programming costs. We are concerned to develop an

---

™ JDK, Solaris and HotSpot are registered trademarks of Sun Microsystems Inc. in the USA and other countries.

understanding of the total cost of running an application. Micro benchmarks that look at some particular aspect can be very misleading.

Our initial long-term research goals still remain:

- Develop an industrial strength persistent language and supporting system and exhibit its power,
- Develop a model of persistent computation that encompasses uniformly many more of the activities that occupy application programmers' working life.

7 Conclusions

PJama is a vehicle for research into how to design and build industrial strength persistent programming technology. It is also a useful tool that can be used now to experience the advantages of orthogonal persistence for a popular language.

Many research challenges remain. Perhaps the primary ones are:

- Developing a better understanding of the different nature of applications and a corresponding assessment of the technological choices available.
- Improving performance by combining the technologies of program optimisation with those of database optimisation.

Extending the computational model to coherently and consistently encompass more of application programmers' total technical requirements.

Acknowledgements

This research is supported by a collaborative research grant from SunLabs, by a grant from the British Engineering and Physical Sciences Research Council, and by a scholarship from the University of Glasgow.

Bibliography

ABC+83 Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R., 1983. An Approach to Persistent Programming. *Computer Journal* 26, 4 pp 360-365.

ADJ+96 Atkinson, M.P., Daynès, L., Jordan, M.J., Printezis, T. and Spence, S. An Orthogonally Persistent Java, SIGMOD RECORD, 25, 4, December 1996.

AJD+96 Atkinson, M.P., Jordan, M.J., Daynès, L. and Spence, S. Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system, May 1996, In Proceedings of the seventh international workshop on Persistent Object Systems (POS7).

AM95 Atkinson, M.P. and Morrison, M., Orthogonal Persistent Object Systems, VLDB Journal, 4, 3, 1995.

Atk78 Atkinson, M.P., 1978. Programming Languages and Databases. *In Proc. 4th IEEE International Conference on Very Large Databases* pp 408-419.

DA97 Daynès, L. and Atkinson, M.P., Main-Memory Management to support Orthogonal Persistence for Java, in [JA97] pp 37-60.

GJS96 Gosling, J., Joy, W.N. and Steele, G., The Java Language Specification, Addison-Wesley, 1996.

HCF97 Hamilton, G., Cattell, R. and Fisher, M. JDBC Database Access with Java, Adison-Wesley, 1997.

JA97 Jordan, M.J. and Atkinson, M.P., Proceedings of the second international workshop on Persistence and Java, Sun Microsystems, M/S MTV29-01, 901 San Antonio Road, Palo Alto, CA 94303-4900. Tech. Report SMLI-TR-97-63.

ODMG97 ODMG (Ed. Cattell, R.) The Object Database Standard: ODMG 2.0, Morgan Kaufmann, 1997.

MBC+90 Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. and Atkinson, M.P., 1990. The Napier Type System. In *Persistent Object Systems*, Rosenberg, J. and Koch, D.M. (ed.), Springer-Verlag, Proc. 3rd International Workshop on Persistent Object Systems, Newcastle, Australia pp 3-18.

MBC+94 Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S., 1994. The Napier88 Reference Manual (Release 2.0). University of St Andrews Technical Report CS/94/8.

PAD+97 Printezis, T., Atkinson, M.P., Daynès, L., Spence, S. and Bailey, P.J., The Design of a new Persistent Object Store for PJama, in [JA97], pp 61-74.

SMK+94 Satyanarayanan, M., Mashburn, H.H., Kumar, P., D.C. Steere and Kistler, J.J., Lightweight Recoverable Virtual Memory, ACM Trans. On Computing Systems, 12, 2 pp167-172, May 1994.