# The Design of a new Persistent Object Store for PJama

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

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

June 1997

## Abstract

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

## 1 Introduction

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

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

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

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

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

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

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

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

## 1.1 Design Goals

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

The specific and immediate goals are:

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

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

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

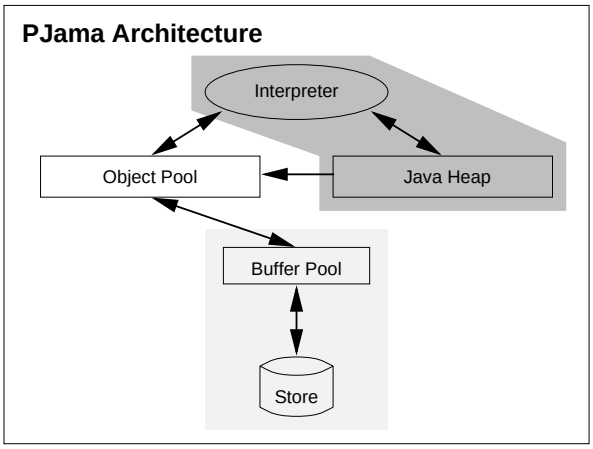


Figure 1: PJama Architecture.

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

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

## 1.2 PJama Architecture

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

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

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

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

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

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

### 1.3 Paper Overview

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

## 2 Store Organisation

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

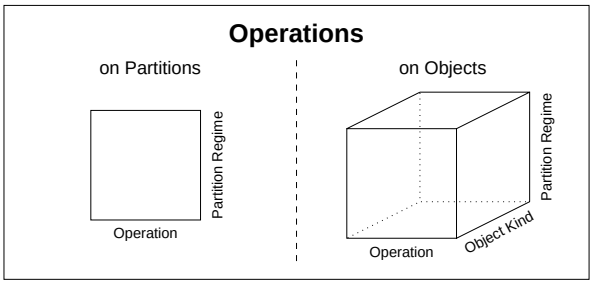


Figure 2: Operation Matrices.

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

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

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

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

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

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

## 2.1 Partition Regimes

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

**Small Arrays** : scalar and object arrays.

**Small Instances** : instances of classes.

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

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

**Large Scalar Arrays** : scalar arrays spanning TU boundaries.

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

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

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

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

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

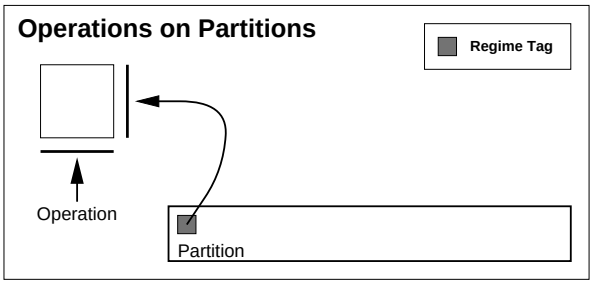


Figure 3: Invoking an Operation on a Partition.

- Separating small objects from large ones avoids many boundary checks upon object-faulting as they are unnecessary for small objects since they are guaranteed not to span multiple TUs.
- When partitions only contain arrays, they do not need to include descriptors and their management structures (see Section 3.6), as arrays have a compressed-type encoding in their header.
- Partitions containing only large scalar arrays[10] can be very large, since they do not need to be scanned to identify intra-partition references during garbage collection. Their reference counts (see Section 3.3) determine whether they are garbage.

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

## 2.2 Invoking Operations on Partitions

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

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

## 2.3 Object Kinds

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

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

**Instances** of any class, apart from `Class`.

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

**Scalar Arrays** : arrays of any scalar type.

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

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

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

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

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

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

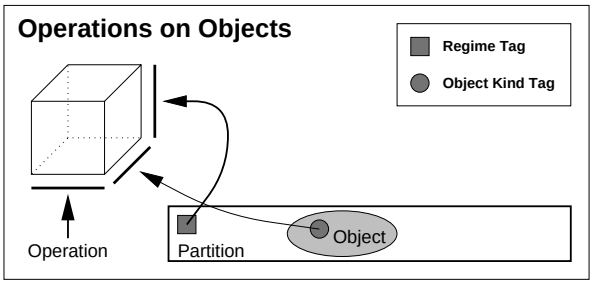


Figure 4: Invoking an Operation on an Object.

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

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

## 2.4 Invoking Operations on Objects

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

## 2.5 Clustering Considerations

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

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

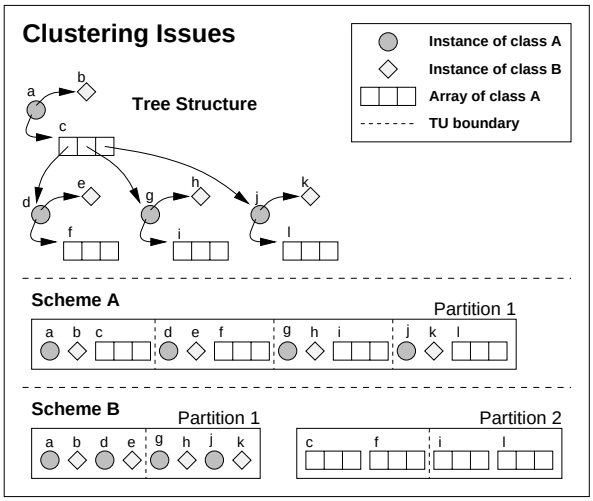


Figure 5: Clustering Issues.

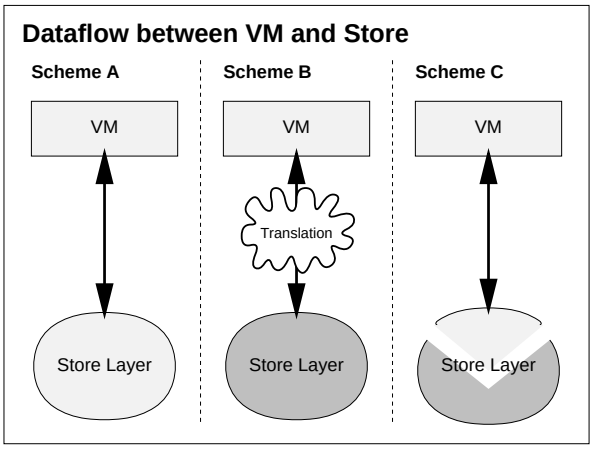


Figure 6: Dataflow between the Virtual Machine and the Store Layer.

in traditional ones [5].

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

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

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

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

## 2.6 Optimising Dataflow

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

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

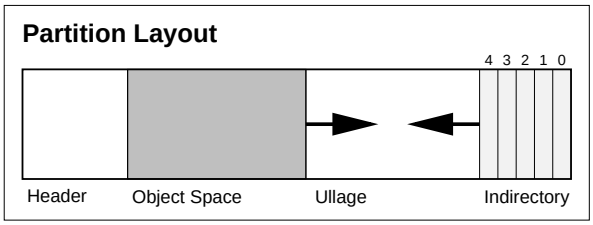


Figure 7: Partition Layout.

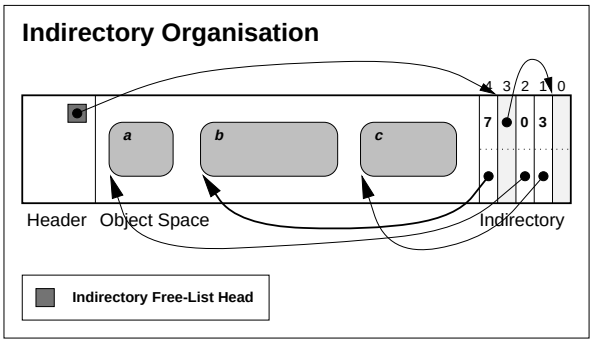


Figure 8: Organisation of the Indirectory.

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

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

# 3 Partition Organisation

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

## 3.1 Partition Identifiers

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

## 3.2 Partition Layout

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

**Header** : where the information describing a partition is stored.

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

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

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

## 3.3 Indirectory

An indirectory entry contains the following fields.

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

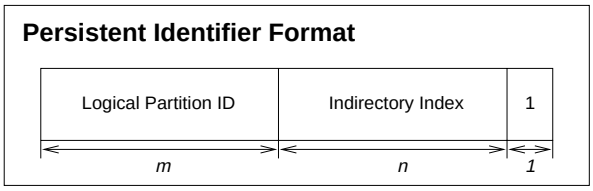


Figure 9: Persistent Identifier Format.

acceptable to limit the maximum partition size to 4GB.

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

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

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

## 3.4 PID Format

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

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

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

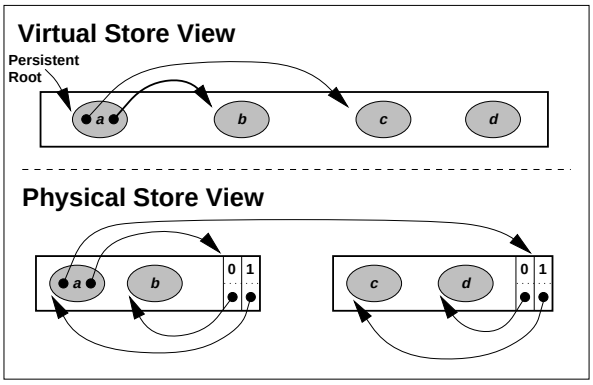


Figure 10: Virtual and Physical Store Views.

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

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

## 3.5 Virtual Store View

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

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

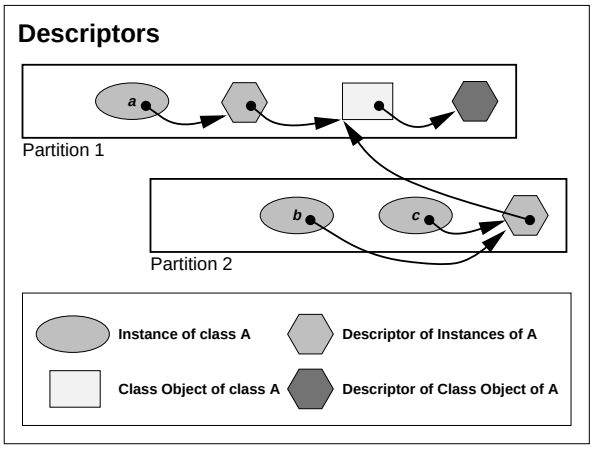


Figure 11: Use of Descriptors.

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

## 3.6 Descriptors

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

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

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

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

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

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

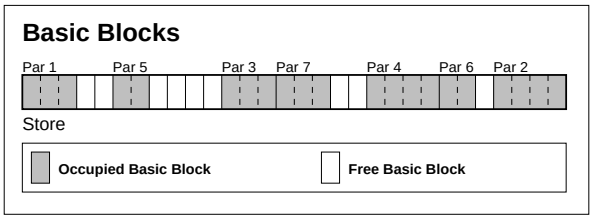


Figure 12: Use of Basic Blocks.

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

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

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

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

# 4 Free-Space Management

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

## 4.1 In the Store

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

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

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

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

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

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

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

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

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

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

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

### 4.2 In a Partition

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

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

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

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

## 5 Related Work

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

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

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

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

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

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

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

# 6 Conclusions and Future Work

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

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

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

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

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

# 7 Acknowledgements

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

# References

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

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

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

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

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

[6] M. P. Atkinson, M. J. Jordan, L. Daynès, and S. Spence. Design Issues for Persistent Java: a Type-Safe Object-Oriented Orthogonally Persistent System. In *Proceedings of POS'7*, Cape May, New Jersey, USA, May 1996.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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