

# Early Experiences with Persistent Java™

Mick Jordan

mick.jordan@Eng.Sun.COM

*Sun Microsystems Laboratories  
2550 Garcia Avenue  
Mountain View, CA 94043 USA*

## Abstract

This paper reports on initial experiences with an orthogonally persistent variant of the Java platform, called Persistent Java (PJava). We review and reflect on the design of PJava and discuss compatibility with Java. The features and limitations of an initial prototype are discussed. The experiences gained in running four distinct applications on PJava are described in detail.

## 1 Introduction

Persistent Java (PJava) [1], [2] is an experimental persistent programming environment for the Java programming language. PJava provides orthogonal persistence [3], an approach that provides equal rights to persistence for all data types with the absolute minimum of additional programming effort. PJava attempts to support a wide range of applications, ranging from simple, almost completely transparent, uses of persistence to sophisticated applications that require extensible transaction models. A major goal of PJava is for arbitrary Java code to participate unchanged in PJava applications.

We report on our early experiences with the first prototype of PJava, called PJava<sub>0</sub>, which is based on the Sun Microsystems™ Java Development Kit (JDK), release 1.0.2 [4]. PJava<sub>0</sub> was designed and implemented by the PJava team at Glasgow University, with support provided by Sun Microsystems Laboratories under its Collaborative Research program.

We begin by providing a brief overview of the design of PJava, followed by a description of the PJava<sub>0</sub> prototype. We then reflect on some of the design choices of PJava and comment on how they relate to the specification of the Java platform, which has been evolving in parallel with the PJava effort. Following a discussion of the limitations of PJava<sub>0</sub> we describe some common pitfalls that beset programmers who are unfamiliar with orthogonally persistent systems.

We then discuss four applications that have been run in the PJava environment, focusing on suitability, ease of

programming, performance and extra functionality.

## 2 PJava Design

In this section we provide a brief overview of the PJava design. PJava provides orthogonal persistence for Java, according to the following principles:

- **Orthogonality:** The principle of orthogonality states that all values whatever their type have equal rights to persistence - longevity or brevity. PJava applies this principle to all Java types, *including* Class types.
- **Persistence Independence:** The principle of persistence independence states that all code should have the same form irrespective of the longevity of the data on which it acts. PJava accepts arbitrary, unmodified Java code.
- **Persistence Identification:** The principle of persistence identification says that there should be a straightforward and consistent mechanism for determining the longevity of values. In PJava the mechanism is persistence by reachability from *root* objects explicitly registered with the class `PJavaStore`.

PJava applications are associated with a store, represented to the programmer as an instance of the `PJavaStore` class. An application is initiated by associating the PJava interpreter with a store and a class to invoke; the latter may be omitted if the application has previously registered the initial class with `PJavaStore`. Objects are candidates to become persistent if they are reachable, directly or indirectly, from root objects that are explicitly registered with `PJavaStore`. By default, successful termination of a PJava application causes all objects reachable from the roots to be atomically and durably updated in the store. Applications that terminate exceptionally make no changes to the stable store. It is possible to call `PJavaStore.stabilizeAll` during program execution in order to achieve a global stabilization. It is the responsibility of the application to ensure

that a stabilization reflects a semantically consistent state.

An application can register callback methods with `PJavaStore` that will be called prior to the execution of the initial class. These can be used for a variety of application specific tasks, for example, verifying global consistency constraints.

Objects in a store are persistently bound to their associated class, including its bytecodes. Class name lookup is always performed first in the store and only if that fails are the normal mechanisms of the Java system employed.

In addition to global stabilization, PJava provides a rich and extensible transaction interface through the `TransactionShell` class. To simplify the programmer's task, a variety of standard transaction models are provided as specializations of this class, for example, traditional flat transactions and nested transactions. The extensibility is provided as a collection of modular building blocks, for example lock management, that can be composed to form new transaction kinds.

The nature of a PJava store is implementation dependent. However, it will usually be a file or disk partition. A PJava interpreter may execute with a store, called *store* mode, or without a store, called *non-store* mode.

### 3 Reflections on the PJava Design

In this section we report on our experiences with those aspects of PJava that are independent of the `PJava0` implementation.

#### 3.1 Programming Interfaces

From the programmer's perspective, the bulk of the PJava design is concerned with the extended transaction model. Since this is not implemented in `PJava0`, we cannot report experiences with it. The only other class that is visible to programmers is `PJavaStore` and our experiences with that have led to some redesign. The most significant change relates to support for multiple stores. Initially, there was a notion that a PJava application might be able to access multiple stores, represented by multiple instances of the `PJavaStore` class. For the reasons described in section 7.1, this has been abandoned. Other minor usability changes have been made.

#### 3.2 Persistence Identification

PJava identifies persistent objects by reachability from named roots registered with the `PJavaStore` class. This requires additional code to be written and also requires that this namespace be managed by the applica-

tion. Another approach, which would be more transparent, would be to define the roots as the set of `Class` instances in the virtual machine, this set being implicitly defined from the classes loaded by the interpreter over time.<sup>1</sup> In this approach, class fields declared as `static` would be viewed as fields of the corresponding `Class` object, and would provide an implicit persistent root table. By associating appropriate semantics for the `static transient` combination<sup>2</sup> of modifiers, some of the design issues discussed in [1] surrounding static initialization could be avoided. On the other hand, failure to use `transient` appropriately would cause data to be made quietly persistent. As we shall see later, the PJava design choice turns out to be very pragmatic for `PJava0`.

#### 3.3 Compatibility with Java

We would like the PJava interpreter to execute "normal" Java applications unchanged. By normal, we mean those applications that make no use of the `PJavaStore` or `TransactionShell` classes. In other words a PJava interpreter running in non-store mode should pass all of the standard compliance tests. A minimal requirement for this is that PJava accept standard class files. In fact, `PJava0` accepts unmodified class files generated by a normal Java compiler and requires no pre-processing or post-processing steps. When run in non-store mode `PJava0` preserves the semantics of the language and the standard packages. At the time of writing we have not tested this compatibility extensively. However, some significant applications such as the Java compiler and the Jeeves HTTP server [5] execute correctly under the `PJava0` interpreter.

Compliance in store mode is a more subtle issue. The extent to which the specification of Java or the virtual machine applies to store mode is unclear, since the specification is not attempting to define an orthogonally persistent system. Certainly, the PJava behavior differs in some cases and could be interpreted as being non-compliant. We will discuss these cases below. It must be noted, however, that the PJava design predates the existence of the detailed language specification [6], which was not published until August 1996.

##### 3.3.1 Static Class Data

Currently, the specification is silent on the `static transient` combination of modifiers. Indeed the

- 
1. This would require a separate mechanism to remove unused classes.
  2. This was called out as illegal in an earlier language specification. The current version does not, but the Java compiler in JDK 1.0.2. still reports it as an error.

meaning of transient is only defined in general terms. However, the natural interpretation would be to truncate the reachability analysis at `static transient` variables in the associated class instances.

### 3.3.2 System Properties

The properties table maintained by the `System` class is effectively global static data. The semantics of this data in the face of persistence is unclear and can cause compatibility problems. Certain properties are clearly transient in the sense that they relate to the execution of a *particular* virtual machine and to external values that might change between executions. Currently the rule that static data be persistent by default causes the properties table to be made persistent, which means that these transient properties are not reinitialized. However, although it is a dubious programming practice, some applications, for example the Java compiler (see 8.3), use the table for global application-specific data, which might well need to be persistent. Since there is no way in the language to indicate that a particular property should be transient or persistent, there can be no automatic solution to this problem. Although the `Properties` interface could be extended to indicate longevity for given keys, it might be preferable to explicitly define the table as transient and require that applications make alternative arrangements for communicating data.

### 3.3.3 Summary

Overall, these issues are of only minor concern, and we conclude that a very high degree of compatibility has been achieved in PJava.

## 4 PJava<sub>0</sub>

PJava<sub>0</sub> is an initial prototype that is based on version 1.0.2 of the JDK from Sun Microsystems. PJava<sub>0</sub> currently runs only on the SPARC<sup>TM</sup> architecture under Solaris<sup>TM</sup>. In order to produce a prototype in a reasonably short time, as few changes as possible were made to the virtual machine implementation. PJava<sub>0</sub> will be made available by Sun to interested research groups in the near future.

PJava<sub>0</sub> provides a modified virtual machine. This is contentious because it violates the *run anywhere* guarantee. Applications that exploit PJava's persistence features will not run on standard virtual machines. Others [7] are attempting to provide persistence as an extension, ideally written entirely in Java. However, it is hard to see how to achieve the degree of orthogonality that PJava aspires to without modifying the virtual machine. Ideally we would like to see orthogonal persistence as part of the Java platform.

The implementation of PJava<sub>0</sub> is based around a cache of loaded objects and a separate buffer pool that manages pages transferred from the disk. The buffer pool itself is implemented on top of Recoverable Virtual Memory (RVM) [8]. The garbage collected heap in the Java Virtual Machine (JVM) has been left mostly unchanged. New persistent objects are copied from the heap to the object cache as the first part of a stabilization. Objects are copied between the object cache and the buffer pool as required. Object references are converted (swizzled) between machine addresses and persistent identifiers as part of this process. To reduce this overhead, not all references are immediately unswizzled on object load, but are translated on demand. Unswizzling is handled in a thread-safe manner.

A special area of the store called the *bootstrap region* is set aside for the `PJavaStore` class, those classes that it depends on and all the instances and associated classes reachable from the root table at the time the store is created. The bootstrap region is loaded in one step during startup, the idea being to speed up the loading of objects that will very likely be accessed.

The PJava interpreter runs in two modes: store mode and non-store mode. In non-store mode the interpreter behaves almost exactly like a standard Java interpreter, with the extra capability to create a store during execution and register objects as persistent roots. In store mode an extra argument “-store pathname” is passed to the interpreter. The pathname argument indicates the name of a persistent store against which to run. In PJava<sub>0</sub> a store is implemented as a file. The interpreter opens the store as part of its initialization process and makes the persistent roots available via an instance of the `PJavaStore` class.

The bi-modality implied by the “store” argument has proven to be an inconvenience. In a future release the interpreter will automatically switch to non-store mode if the file given as argument does not exist. Access will also be provided in the `PJavaStore` class to the pathname, which can then be used as the name of the store to create.

## 5 Performance and Reliability of PJava<sub>0</sub>

The overall experience with PJava<sub>0</sub> has been very positive, both from a performance and a reliability perspective. This is particularly so given the relatively short time frame in which it was implemented and the fact that it is based on a relatively complicated and undocumented virtual machine implementation.

As a testament to this, the Forest benchmark, described in section 8.2, ran correctly on its first execution.<sup>1</sup> Fur-

thermore, the performance was quite a bit better than the version which runs over ObjectStore/C++ [9], a commercially supported object-oriented database.

## 6 Limitations of PJava<sub>0</sub>

In this section we describe some of the limitations of the PJava<sub>0</sub> implementation that cause it to fall short of the goal of fully orthogonal persistence. The limitations either result from the constraints of the original JDK virtual machine design or from time and manpower considerations on PJava<sub>0</sub>.

### 6.1 Class Loading

The Java Language Specification [6] defines the rules governing class loading in considerable detail. It defines specific phases of the loading process, taking pains to define exactly what, in source code, causes each phase to be completed. The specification explicitly permits different class files formats but is insistent that bindings between class files be symbolic. The specification permits some flexibility in the time at which references between classes are resolved. PJava<sub>0</sub> loads and resolves all required classes, transitively, on a store stabilization, so that a store is always self-contained.

Java permits multiple instances of a given named class to co-exist in the same virtual machine, provided that each is loaded by a different *classloader*. A fresh virtual machine contains only the system classloader, which cannot be replaced. The classloader used to load a class plays a role in the dynamic type-checking (`instanceof`) operator. Two instances of class *C* loaded by different classloaders are not be considered to be the same type. Note that they may have different definitions anyway, but this is irrelevant since Java uses name equivalence for type-checking purposes.

PJava<sub>0</sub> maintains a single class dictionary, in the `PJavaStore` class, which acts as the persistent table of classes loaded by the system class loader. The associated classloader for a class is currently *not* recorded in this table. Therefore, confusion and possible internal errors may occur if an application attempts to stabilize a store containing multiple class instances. This limitation will be removed in a future release.

### 6.2 Threads

Orthogonal persistence requires that instances of class `Thread` can be made persistent. This is not possible in PJava<sub>0</sub>, in part owing to the complexity of the thread implementation in the JDK. Further, applications are

limited to using *cooperating* threads which are under the control of a manager that has sufficient knowledge to determine when it is safe to stabilize the store. This is due to the fact that, although `PJavaStore.stabilizeAll` is a synchronized method, thus serializing stabilize calls from multiple threads, there is no easy way to stop only the application threads. It is possible to enter single-threaded mode, but then the stabilization cannot succeed because more than one thread is needed for output. So, currently, threads are not stopped and therefore, while stabilization is occurring, other, possibly damaging, thread activity may be occurring. This could be repaired by appropriate locking in the virtual machine, but at some cost. It seems preferable to defer this issue to the design and implementation of the implicit locking required by the PJava extensible transactions model.

### 6.3 Extensible Transactions

The PJava extensible transaction model is not implemented in PJava<sub>0</sub>. Applications are therefore limited to periodic checkpoints, with no ability to rollback. Nonetheless, a wide variety of useful applications can be written with this limitation.

### 6.4 Object Cache Replacement

In the current version of PJava<sub>0</sub>, it is not possible to replace objects in the object cache. The cache is used for all objects brought into memory from the store, and for all newly allocated objects that are to be made persistent on a stabilize. The cache is also of a fixed size determined on interpreter startup. Therefore, there is a fixed limit to the amount of data that can be processed in one execution, which is clearly a major limitation for long-running applications that are associated with large stores. Work is underway to remedy this problem in the PJava<sub>0</sub> prototype.

In the applications that have been written to date, this problem typically affects the store loading phase, where a large amount of data is read from the file system and placed in the store. Since the cache size can be set on interpreter startup it is usually possible to workaround the fixed limit. Applications that process the resulting stored data are much less likely to access the entire database and can therefore usually execute with the default cache size. PJava's object cache architecture is a benefit here, since only those objects that are actually needed are transferred from the page buffer pool to the cache.

### 6.5 Native Code

In the PJava design [1] it is argued that native code

---

1. Having first been debugged in a non-persistent version.

should not be permitted in PJava applications because of the risks to data consistency that arise from the unchecked access to memory that is possible in native code. However, pragmatically, application writers must decide the appropriate trade-off between safety, functionality and performance. As more functionality migrates to the Java core, for example, Remote Method Invocation (RMI) [10], and just-in-time (JIT)<sup>1</sup> compilers become widespread, we expect the need for native code to diminish. In the meantime, native code implementors must follow some additional conventions to ensure that their code is compatible with the implementation of the PJava virtual machine. This is mostly concerned with ensuring object residency before accessing fields and methods. The situation is more complex when the native code contains state that must be made persistent in order to save a persistent version of the associated Java class. PJava<sub>0</sub> provides no support for native state which means that such classes cannot be made persistent. One workaround is to ensure that the state is instead declared as fields of the Java classes, but this idiom causes problems when the state is a C pointer, since Java has no way to declare such a type. It also causes portability problems between machines with 32 and 64 bit addressing. Hiding Java object references by declaring them as integers is more dangerous since it subverts the normal PJava handling of non-resident objects.

## 6.6 External State

Objects that contain references to external state pose a general problem for orthogonally persistent systems. Typically, such external state is associated with native code or core features of the language platform, for example operating system file descriptors (e.g. declared as integers). Such data is not marked `transient` in JDK 1.0.2 and therefore interacts badly with the PJava design choices of persistence by reachability and of `static` fields being persistent by default, since this data is usually meaningless when the object is reloaded in a subsequent execution. PJava<sub>0</sub> has not modified the core Java classes to solve this problem, although it does honor the `transient` modifier. This problem also besets the Java Object Serialization system [11], that will become part of the Java core in JDK 1.1 and, fortunately, in that release, such data will be marked as `transient`.

Of course, this is only half of the solution. There needs to be a way for the class to recreate the transient state when the object is reloaded. In the original PJava design this was handled via class-specific callbacks that were

---

1. N.B. JIT compilers must cooperate with the persistence mechanisms in PJava.

registered with the `PJavaStore` class. Subsequently this was altered to be similar to the mechanisms provided for serialization, by invoking specially named methods that are provided as part of the class definition. However, these mechanisms are not yet implemented in PJava<sub>0</sub>.

A particularly significant piece of external state is the code that actually implements the native methods. Some of this code is bundled in the interpreter. The remainder exists in dynamically loaded libraries that are loaded by calls to `System.loadLibrary` and the typical idiom is to make the call in a static initializer of the class. In PJava static initializers are only run once, when the class is first loaded, that is, when the `Class` class instance is created. So, on subsequent executions, the external library is not loaded, leading to an exception. The correct idiom, for a PJava program, would be something like the following:

```
static transient boolean loaded =
    loadLib();

static boolean loadLib() {
    System.loadLibrary("mylib");
    return true;
}
```

This idiom assumes that transient static variables are automatically reinitialized when a class is first faulted in from the persistent store. Alternatively `loadLib` could be called explicitly in a callback.

A related problem is that the library search path that is used by `System.loadLibrary` is made persistent by the default rules for static variables. This has the unfortunate property that stores cannot be moved into an environment where the library search path is different. The temporary workaround in PJava<sub>0</sub> is to force the search path to be reinitialized, which is straightforward, since the relevant code is in the core virtual machine. A more robust solution, that would be more consistent with the PJava model of persistent class consistency, would be to load the native code into the store and make it a persistent object. Subsequent attempts to load the library would look first in the store, as is done for persistent Java classes.

## 6.7 The Abstract Window Toolkit (AWT)

AWT is an important part of the Java core but the implementation, although structured for portability, is heavily platform dependent and contains a considerable amount of native code. It suffers all of the problems referred to in the previous two sections. Since AWT is a key part of the Java environment it is obviously rather common to

attempt to make an AWT class persistent by reachability from some other class. In PJava<sub>0</sub> this is explicitly checked for and the stabilization aborted, since there is no prospect of the class working correctly on a subsequent execution. Unfortunately this restriction leads to some unnatural programming idioms. It is very natural and convenient to encapsulate application-level persistent data in a subclass of an AWT class. In PJava<sub>0</sub>, the programmer must place the application data in a separate class, make this reachable from some persistent root and then associate it with the AWT class by a level of indirection.

A mechanism is needed to separate the persistent and portable aspects of an AWT class from the transient and platform dependent pieces. It would then be possible to mark the latter as `transient` and make persistent enough information to enable the class to be reinitialized on a subsequent reload, using the callback methods of PJava. This mechanism might be somewhat similar to those in place for mobile agents in Visual Obliq [12].

## 6.8 Persistence Independence

It should be clear from the preceding sections that the PJava design of explicit root registration is a very pragmatic choice. It completely avoids the problems that would arise if all loaded classes were the implicit roots of persistence, since many of these contain transient data not marked as such, or other state that is problematic. In the PJava design the application has complete control over which classes are made persistent.

## 7 Common Pitfalls

Perhaps the most significant consequence of the principles of orthogonality is the fact that code and data are bound together in a persistent store. While this provides a high degree of consistency, contributing to application reliability, it runs counter to the normal experience of most programmers. It also requires additional tools to support the evolution of software. We consider these two issues separately below.

### 7.1 Binding Code and Data

The lack of orthogonal persistence in everyday programming languages has meant that a significant part of learning a new programming language concerns the facilities for input/output. While, it would be incorrect to say that orthogonal persistence completely removes the need for such facilities, one of its goals is to substantially remove the need to save and restore data structures in ad hoc formats between separate executions of an application. Today, however, this is the norm and is so

ingrained that programmers have some difficulty in believing that there might be alternative.

Early programming languages were consistent with this separation of code and data, since they provided separate facilities for data definition (records, arrays etc.) and behavior (procedures), with a more or less loose connection between the data and the code that operated on it. However, since the advent of abstract data types and, more recently object-oriented programming, the focus has shifted towards the integration of code and data (state and behavior). Nevertheless, persistence mechanisms generally continue to maintain the separation. For example, ObjectStore does not store C++ [13] code in the database. To underline how pervasive the separation mindset can be, we must admit to spending one frustrating PJava<sub>0</sub> debugging session wondering why the application behavior was not changing despite having changed and recompiled the code!<sup>1</sup>

One characteristic of the separation of code and data is the ability to open multiple databases from within one application, and the inability of PJava to do this has been commented on. However, it should be clear that if these databases contain code bound to data there is a conflict with the language semantics. There may be multiple, possibly different, versions of the same-named class, a situation that cannot exist according to the language definition.<sup>2</sup>

A much better analogy for the Java programmer is the notion of a store as a, more or less, consistent persistent virtual machine. Multiple stores are then handled as distinct virtual machines which are able to communicate, for example, by RMI. This model is more faithful to the object-oriented paradigm by focussing on active objects with behavior rather than passive data.

### 7.2 Schema (Type) Evolution

Since the code is bound to the data in the store, it becomes more difficult to modify it, for example to fix a bug. In addition to the need to recompile, the new code has to be installed in the store and all affected objects rebound.<sup>3</sup> This seems onerous and makes separation of code and data seem an attractive simplification. However, the separation has a cost. Typically a consistency check (schema validation) has to be carried out every time a store is accessed. The same is true for the input

---

1. But only in the file system, not the store.

2. Except in different classloaders. Certainly not for the basic classes such as `Object` or `Thread`.

3. A more complex solution is to support schema *versioning* in which objects and types can coexist in multiple versions in the same store.

phase of Java Object Serialization. In contrast, except during schema evolution, PJava needs to make no checks, thus maximizing performance when the application is actually in use.

The lack of schema evolution tools is particularly felt by application developers since their main task is to change code, and this lack has been noted by the users of PJava<sub>0</sub>. Once an application is deployed, the need for evolution occurs less frequently, corresponding to well defined release points. Further, when it does eventually occur it is common for the changes to be significant enough to cause problems even for applications using ad hoc persistence. Ultimately we expect the reflection capabilities available in PJava to actually ease such transitions, and even increase the rate at which applications can evolve without compromising reliability.

Since PJava<sub>0</sub> provides no schema evolution mechanisms, this means that stores must be completely rebuilt in the face of change. Clearly this is unacceptable in the long term. As a short term solution we investigated the provision of a simple schema evolution tool at SunLabs this summer. We made some progress towards supporting simple changes, such as only modifying the code of methods. However, the current solution suffers problems of scale that cannot be fixed without redesigning part of the PJava<sub>0</sub> implementation. It is clear that it is essential to keep schema evolution (or schema versioning) in mind when designing a persistent object system. The constraints under which PJava<sub>0</sub> was built prevented this foresight.

One problem that must be addressed by any schema evolution mechanism for PJava is what type compatibility rules to enforce. PJava takes a strong, compile-time centric, view of type compatibility which is reflected in the approach to class loading described in section 6.1. The language specification, on the other hand, specifies a weaker set of consistency rules to be applied at class load time. In particular, sets of classes that would not compile together can pass these weaker rules. Ironically, this weakening is justified to support a limited form of class evolution that does not require recompilation. It is to be hoped that evolution can be put on a firmer footing in the context of PJava.

In summary, it is clear that schema evolution is a very high priority feature to provide in future releases of PJava.

## 8 Example Applications

Since PJava has only been available for a short period and to a limited group of users, we do not have a extensive set of substantive applications to report on. How-

ever, the following set of applications cover a fairly wide spectrum and provide useful insights. For each application, we provide a brief description and then discuss it from four perspectives, suitability for PJava, ease of programming, performance and extra functionality.

### 8.1 Oscar

Oscar is an application from the domain of Geographical Information Systems (GIS), authored at Glasgow. Originally written to test the capabilities of Java in this domain, it was subsequently modified to run under PJava.

Oscar consists of three main phases:

1. Open a file of NTF data (National Transfer Format), as available from Ordnance Survey (GB). This data is a representation of carriageways (roads) in the UK.
2. Read the file a line at a time (where a 'line' of data can actually span several physical lines in the file). Each line represents one NTF structure. Each line is parsed and a Java object corresponding to the NTF structure is built. This continues until an end-of-data indicator is encountered.
3. AWT is used to display the data in a meaningful manner. Rudimentary GIS features are available, such as clicking a road to get information about it. It is possible to alter the display colors for different kinds of roads.

In normal usage Oscar expects to load 4 *tiles* NW, NE, SW and SE, where a tile is one file of NTF data, representing a 5km square area. These tiles are displayed via an offscreen image and can be shown in a number of scales (in pixels) defaulting to 800x800. The viewing area can display one complete tile and the user scrolls around to view the rest of the quadrant.

The four tiles comprise quite a lot of data, with each tile containing between 8,000 and 20,000 objects. For the tiles used in the experiment this translates to reading data for approximately 60,000 objects and building the corresponding Java objects.

#### 8.1.1 Suitability for PJava

Oscar is representative of the classic set of applications for which persistent object systems were invented.

#### 8.1.2 Ease of Programming

Converting Oscar to PJava was easily accomplished, requiring the addition or amendment of 23 lines of code in only 4 compilation units and one new class to load the tiles into the store. The limitation of no persistent AWT classes in PJava<sub>0</sub> made the conversion slightly more onerous.

### 8.1.3 Performance

Loading the tiles from the NTF format files was taking between 3 and 8 minutes per tile, depending on the amount of data in the tile and the load on the system. Loading all four would normally take about 20 minutes. While the interpreted implementation of Java in the Sun JDK undoubtedly inflates this time over a compiled implementation, the need to read the entire file clearly provides a limit on scalability.

The performance benefits of converting to PJava were substantial. The time to display a tile now measures in seconds - on average about 20 seconds to process the tile from the store, draw its data into an image buffer and display that buffer. Subsequent redispays are much faster since the objects have been loaded into the cache.

Since all the NTF data from one file has to be read into memory in one go, the Java version could run out of memory. This is an example of the *big-inhale* effect. In PJava, once the data has been converted to persistent objects, only the subset that is needed for a particular display need be loaded from the persistent store. This permits large sets of tiles to be loaded and saved incrementally in a single store. We must stress that no work is required on the part of the programmer to effect the storing of the data, beyond registering the root objects.

### 8.1.4 Extra Functionality

Although it would have been possible to save the map from road kinds to display colors in the Java version, using some ad hoc persistence mechanism, it would have involved extra programming and a mechanism to connect the color data and the NTF data. Since this structure had already been constructed in the program domain, making it persistent required no extra work under PJava.

## 8.2 Forest

Forest is an application from the domain of Computer-Aided Software Engineering (CASE). The first prototype was written in C++ using the ObjectStore object-oriented database.

Forest integrates the three essential activities of software development: authoring, versioning and configuration management and system building. It adopts the Vesta [14] approach to configuration management, which is characterized by a repository of immutable, versioned, *source* objects, that are combined with modular system build descriptions to generate *derived* software artifacts.

### 8.2.1 Suitability for PJava

Our experiences in implementing the Forest prototype

with ObjectStore/C++ were positive [15]. However, dissatisfaction with C++, the lack of orthogonality in ObjectStore/C++, and general enthusiasm for Java, lead us to the SunLabs collaboration with Glasgow to develop Persistent Java. We believe that CASE tools are an ideal application for persistent object systems and that, in such a framework, the Vesta approach is an optimal choice. In the long term, we believe that the extensible transaction model of PJava will greatly assist in the development of collaborative software development tools.

The complete Forest environment has not yet been translated into PJava. However, we have converted a benchmark that we developed as part of the evaluation of the prototype based on ObjectStore. The benchmark simulates a number of users exercising the versioning and authoring system by checking out components, editing them and checking them back in. Following Vesta, the logical unit of checkout is an entire tree of objects. Each step in the process is logically a separate transaction. Each run of the benchmark performs a set number of checkouts, passed in as a parameter. The particular set of objects that is accessed is chosen pseudo-randomly. Since ObjectStore is a commercial product a comparison between it and PJava<sub>0</sub> is a useful datapoint.

### 8.2.2 Ease of Programming

The Forest environment contains a fairly large set of object types; indeed an open-ended set, since developers can define new types using the environment itself. Therefore, the orthogonality of the persistence mechanism has a direct impact on the ease with which the system can be programmed. The ObjectStore/C++ version fell quite a bit short in this regard and in the end, in order to achieve adequate performance, our code had become quite ObjectStore specific. In particular, with ObjectStore, persistent objects can only be accessed inside transaction boundaries, and new persistent objects must be allocated with syntactically different forms of the C++ new operator. Paying attention to clustering, which must be managed at allocation time, turns out to be very important owing to false lock conflicts that can arise because of ObjectStore's choice of page-level locking and data transfer.

In contrast, the PJava version, which, like Oscar (8.1), was originally developed and tested as a non-persistent application, required only two changes. The first, isolated to one module, was the registration of a single object as the persistent root. The second change was the insertion of the calls to the `stabilizeAll` method to checkpoint the store at the transaction boundaries.

The addition of the calls to the `stabilizeAll`

method to achieve the checkpoints, if done in the obvious and simple way, does cause the code to become persistence-specific. Had the PJava transaction model been implemented, we would have used a sequence of flat transactions as we did in the ObjectStore version. Since the PJava transaction model supports the composition of user transaction code<sup>1</sup>, with hindsight we could have made the body of the benchmark persistence-independent by defining a simple subclass of `TransactionShell` that provided only the durability (D) aspect of transactions (by virtue of global stabilization), and composing this with the classes implementing the benchmark proper.

### 8.2.3 Performance

An important aspect of the ObjectStore version of the benchmark was measuring how the system behaved as concurrent users were added. Unfortunately, PJava<sub>0</sub> does not yet support concurrent transactions, therefore performance comparisons are limited to the single user case. Even then, architectural differences between the two systems make like comparisons difficult. However, they do provide some insights into the costs of architectural decisions. ObjectStore employs a client-server architecture, with the server acting as the concurrency control point between the clients, which are regular operating system processes. In general, objects (pages) are transferred from the server to the client machine for processing locally, and then back to the server on a transaction commit. Typically, the clients reside on separate machines in a network. However, it is possible to execute the clients on the same machine as the server, and configure the server to use shared memory rather than sockets for client-server communication, and we used this mode for the comparison. One final difference is that code is not stored in an ObjectStore database; instead a schema (type) check is carried out every time a client starts a transaction. Since the type check does not extend to code behavior, this is a weak consistency check, but substantially reduces the need for schema evolution.

PJava<sub>0</sub>, on the other hand, is not client-server in the ObjectStore sense and, as noted earlier, has an architecture that is essentially a (consistent) persistent virtual machine. The clients share the same address space (and protection domain) with the PJava<sub>0</sub> system. This is acceptable because Java is a safe language. Contrast this with Thor [16], which wishes to support clients written in unsafe languages and is devising ways of dealing with

the performance problems that result from crossing protection boundaries [17]. Although it would be possible for multiple PJava interpreters to execute in read-only mode from the same persistent store, concurrent updates can only be achieved using a single interpreter and the basic Java concurrency mechanism, namely threads. The limitations on PJava<sub>0</sub> prevent us from benchmarking the multi-user simulation.

We used the same data for the benchmark that we reported on in our previous paper [15]. The initial store is about 13Mb in size and consists of about 2500 text files in 216 (versioned) directories. The benchmark accesses all the versioned directories and creates about 0.5Mb of new leaf objects. It must be stressed that the PJava version is not a direct port of the C++ version, but the general structure of the code is the same. Also, the C++ code is compiled into machine code, whereas the PJava code is interpreted by bytecodes.

The total elapsed time for the single-user version of the benchmark running on ObjectStore/C++ version 4.0 was 207 seconds, and for PJava was 16 seconds, both times averaged over four runs and rounded to the nearest second. The tests were run on a SPARCcenter™ 2000, with 196Mb of memory, and the stores on a locally attached disk. The ObjectStore figure is consistent with the value that we reported previously, although we are now using the production version rather than the beta release.

We were not expecting to find an order of magnitude difference in favor of PJava, and began to search for an explanation. One surprising fact is that there appears to be only about a 10% improvement running the ObjectStore/C++ version on the same machine as opposed to a client-server combination. This suggests that the shared memory communication is providing little benefit.

As an experiment, we altered the benchmark to execute as a single transaction. In this case, the ObjectStore time came down to 53 seconds and the PJava time reduced to 12 seconds. Evidently transactions are considerably more expensive in ObjectStore than are stabilizations in PJava. Given that ObjectStore provides concurrency control and rollback, this is not surprising, although the cost seems rather high.

In the ObjectStore/C++ variant, the initial scan of the store to locate all the versioned directories took 31 seconds, as compared to 3 seconds in PJava, a substantial difference that demands an explanation. ObjectStore performs schema validation on every transaction.<sup>2</sup> We tested the cost of this by repeating the initial scan trans-

---

1. The lack of first-class functions (methods) in Java limits the composition to classes that implement the `Runnable` interface.

---

2. If the database has not been changed by another client since the last transaction, this test is very fast.

action and found that it took only 14 seconds. Therefore 17 seconds can be attributed to schema validation, a cost that PJava need not pay because of the consistent binding between code and data.

In conclusion the performance difference between ObjectStore/C++ and PJava<sub>0</sub> clearly merits more study. However, the results are very encouraging given the prototype nature of PJava<sub>0</sub>, and demonstrate that the strong consistency guarantees of PJava have a clear performance benefit.

#### 8.2.4 Extra Functionality

Forest was designed from the outset with a persistent object system as the implementation infrastructure, and many of its capabilities simply could not be realized (efficiently) without it. Indeed, a recent port of a limited set of its functionality using file-system storage served to remind us of this fact.

### 8.3 Java Compiler

The Java compiler in the Sun JDK was one of the first medium-sized applications to be written entirely in Java. It consists of over 300 classes. The compiler is ordinarily executed once per compilation unit, although it can compile multiple compilation units in one run.

#### 8.3.1 Suitability for PJava

Superficially, a compiler does not appear to be an obvious application for PJava. If the source code itself is kept in the persistent store, as is the case in Forest (8.2), then the compiler necessarily becomes a suitable application. However, in its normal mode of reading and writing a file system, its requirements for orthogonal persistence appear limited. But delving a little deeper reveals several benefits from the PJava environment.

In general, every time the Java compiler runs, all of its classes are loaded from the file system, although, if there are errors, then the classes that handle code generation are not loaded. Small variations in the set of loaded classes may also occur depending on the nature of the source code being compiled. However, on average most of the classes are loaded. In principle, every time a class is loaded, it is verified. In practice, classes loaded from directories in an application's classpath are not verified, even though this is a trade-off between performance and security. In addition, the standard optimizations are performed on the bytecodes every time they are loaded. Evidently it should be possible to exploit the persistence of classes in PJava to reduce these costs, since the verification and optimization occurs once when the class is first loaded.

When a compilation unit contains errors, the compiler loads the error messages dynamically from a *properties* file.<sup>1</sup> This file is expected to be found at a special pathname relative to another, standard system property called `java.home`. While this approach provides some flexibility for tailoring the content of the error messages, for example, for internationalization purposes, it introduces the potential for inconsistency and adds a surprising amount to the compilation time (see 8.3.3). By binding the properties table with the compiler classes in the persistent store, we remove the potential for inconsistency and avoid the per-compilation overhead of loading the table.

#### 8.3.2 Ease of Programming

Modifying the compiler to load the properties table and save it as a persistent object was simple. The code that already existed to read the table was moved from the main part of the compiler into a new class, `Install`, that is invoked to perform initialization. This permits the error message table to be reloaded at a later date, if required. It should be noted that the compiler could be modified to encapsulate the error messages as code, to completely remove the dependency on an external file. However, modification of the messages would then require a custom interface to display and edit the table.

Loading the classes into the persistent store is also carried out at initialization time. To achieve this it suffices to create a dummy persistent instance of the `Main` class and register it as a persistent root. The PJava rules for persistence by reachability of classes then force all necessary classes to be made persistent.

#### 8.3.3 Performance

The time to load the error messages table from a file was 4 seconds, which effectively reduces to zero in the PJava version.

If the `Install` class operates on an existing, minimal, store, then the classes specific to the compiler will be faulted in on demand during a compilation. If the `Install` class creates a new store, then all the classes will be stored in the bootstrap region, which will be loaded in one step on startup. The rationale for the existence of the bootstrap region is to minimize the faulting overhead for classes that will be loaded early in an application's lifetime. Since the compiler contains a fairly large number of classes, it serves as a testbed for the effect of this optimization in the PJava<sub>0</sub> implementa-

---

1. The standard class `Properties` is a map from `String` to `String` values and has built-in persistence support through a defined file format.

tion.

We measured the time the compiler took to compile the `java.lang` classes, one compilation unit at a time, in four configurations:

1. using the standard JDK 1.0.2. interpreter (`java`).<sup>1</sup>
2. using the `pjava` interpreter in non-store mode (`pjava`).
3. using the `pjava` interpreter in store mode without the compiler classes in the bootstrap region (`pjava-s`).
4. using the `pjava` interpreter in store mode with all classes in the bootstrap region (`pjava-sb`).

The compilation time in seconds, running on a SparcStation™ 10, averaged over four runs are as follows:

**Table 1: Compilation time for `java.lang`**

<code>java</code>	<code>pjava</code>	<code>pjava-s</code>	<code>pjava-sb</code>
380	392	332	346

We see that the PJava interpreter adds a small overhead to the compilation time. As expected, in store-mode, the time is reduced because it is faster to fault in the resolved classes that reload them from the class files. However, storing all the classes in the bootstrap region performs worse than this, although still better than non-store mode. We believe that a possible explanation for this is that there are more classes in the bootstrap region than are strictly needed by the compiler. Owing to a bug in the current version of PJava<sub>0</sub>, we are unable to store just the transitively reachable classes. Instead, if any class is used from a package, we store all the classes in that package. However, it seems unlikely that fixing this bug will cause `pjava-sb` to perform significantly better than `pjava-s`.

The time to compile with class verification enabled was 620 seconds for the standard JDK compiler, thus verification accounts for an additional 240 seconds. This cost is incurred once in the PJava version during the initialization phase and so compiling with verification enabled should be identical to the figures in Table 1. Unfortunately, owing to another bug in PJava<sub>0</sub> related to class verification, we have been unable to prove this.

#### 8.3.4 Extra Functionality

We did not attempt to add extra functionality to the compiler, although a number of possibilities suggest them-

---

1. We used a non-optimized version of the interpreter for compatibility with the non-optimized `pjava` interpreter.

selves. For example, the ability to encapsulate arbitrary configuration data, for example compiler options, could be arranged through a customization interface.

## 8.4 Jeeves

Jeeves [5] is an HTTP server under development at Sun. Jeeves is entirely written in Java and provides for extensibility through *servlets*, which may be loaded from anywhere on the Internet. Servlets, like applets, execute in a controlled environment (sandbox). Unlike applets, servlets have no user interface component.

### 8.4.1 Suitability for PJava

Jeeves is a highly suitable application for PJava. Jeeves maintains a considerable amount of state that controls its behavior in the external file system. Although, currently, the webmaster is required to edit some configuration files manually, the intent is to replace this with an applet interface that will allow the server to be configured from any web browser capable of running Java. If running under PJava, the server configuration data could be completely encapsulated and made durable in the face of crashes.

More interesting is the potential for storing user data in the persistent store associated with the server rather than the external file system. It is already common for OODB vendors to offer web front-ends to their databases, it being relatively straightforward to transform simple objects into an HTML representation.

The PJava design includes a proposal for a new URL protocol that provides type-checked access to objects in a persistent store [18]. This protocol could easily be recast into an HTTP-based servlet URL. However, it cannot easily be implemented without additional reflection facilities such as those proposed for JDK 1.1 [19].

Alternatively an applet could be transferred to the client browser and set up a custom communication mechanism using RMI, or some similar mechanism, to access the objects in the associated persistent store.

### 8.4.2 Ease of Programming

At the time of writing we have not modified Jeeves to store its configuration data in the persistent store. However, we expect the modifications to be very similar to those carried out for the Java compiler properties table, since Jeeves also uses property tables.

We have experimented with a simple persistent counter servlet, which requires only that the counter be registered as a root and that the store be stabilized on each request from a client browser, resulting in two extra lines of code.

Jeeves exploits multi-threading to service many requests concurrently. Each servlet executes in its own thread, and servlets are, in general, unaware of each other. It is therefore quite likely that several threads might attempt to stabilize the store at the same time. In PJava<sub>0</sub>, this must be prevented by serializing the stabilize calls. Since the interface between Jeeves and a servlet is already handled through a method in a special class, the cleanest way to achieve this is to wrap this method in a `TransactionShell` that enforces serial stabilization.

### 8.4.3 Performance

Running an internal benchmark, we have observed a slowdown of 6% when running Jeeves under the `pjava` interpreter in non-store mode. It would be informative to compare the time taken to serve up HTML from the host file system against the time to serve it from the persistent store, and we hope to report on this in a later paper.

### 8.4.4 Extra Functionality

One piece of extra functionality that PJava<sub>0</sub> could provide, which would otherwise require special programming, is transparent caching of objects through the mechanisms of the object cache. Caching objects from the external file system would still require periodic checks for external modifications. One interesting possibility would be to provide a file-system emulation in the persistent store, thus eliminating this requirement.

## 9 Conclusions

The early experiences with PJava have been positive; reliability is good and performance rather better than might be expected from a first prototype. Many useful applications can be written with the simple global stabilization mechanisms available in PJava<sub>0</sub>.

PJava is compatible with Java programs that do not use the persistence mechanisms. In PJava<sub>0</sub>, several issues remain to be resolved regarding state that exists outside the control of PJava. Wider use of the `transient` modifier in the Java core packages would help in this task.

The transition from PJava<sub>0</sub> to a fully-fledged environment, supporting concurrent transactions and the extensible transaction model, and with the ability to evolve the code and data of applications that reside in PJava stores, remains a significant design and implementation challenge. We hope to be able to report our experiences with such a system in the future.

## 10 Acknowledgments

Information on Oscar was provided by Stewart Macneill of Glasgow. Cathy Waite, also from Glasgow, provided helpful input on her experiences with PJava. Finally, I would like to thank the Glasgow PJava team of Malcolm Atkinson, Susan Spence, Laurent Daynès and Tony Printezis for their help in clarifying my understanding of the PJava implementation, and their prompt response to bug reports.

## 11 Trademarks

Sun, Sun Microsystems, Java and Solaris are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. SPARC, SPARCstation and SPARCcenter are trademarks of SPARC International, Inc., licensed exclusively to Sun Microsystems Inc.

## 12 References

- [1] Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system, M.P. Atkinson, M.J. Jordan, L. Daynès, S. Spence, Proceedings of the 7th International Conference on Persistent Object Systems, Cape May, New Jersey, May 1996.
- [2] PJava Design 1.2, Working Document, available via <http://www.dcs.gla.ac.uk/pjava>.
- [3] Orthogonally Persistent Object Systems, M.P. Atkinson and R. Morrison, VLDB Journal, 4(3), pp319-401, 1995.
- [4] The Java Development Kit version 1.0.2, <http://java.sun.com/JDK>.
- [5] Jeeves, <http://www.javasoft.com/jeeves/>
- [6] The Java Language Specification, James Gosling, Bill Joy, Guy Steele, Addison-Wesley, 1996, ISBN 0-201-63451-1.
- [7] Object PSE Pro for Java, Object Design Inc. <http://www.odi.com/products/pse/psepj.html>.
- [8] Recoverable Virtual Memory, H.H. Mashburn and Satyanarayanan, RVM Release 1.3, CMU, January 1994.
- [9] The ObjectStore Database System, Charles Lamb, Jack Orenstein and Dan Weinreb, *Communications of the ACM* 4,10 (October 1991) 50-63.
- [10] Java Remote Method Invocation, Revision 0.9, Sun Microsystems Inc., May 1996.

- [11] Java Object Serialization Specification, Revision 0.9, Sun Microsystems Inc., May 1996.
- [12] Migratory Applications in Visual Obliq, Krishna Bharat and Luca Cardelli, Proceedings of the ACM Symposium on User Interfaces, Software and Technology, Pittsburgh, PA, Nov. 1995.
- [13] *The Annotated C++ Reference Manual*, Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, Reading, Massachusetts, 1990.
- [14] The Vesta Approach to Configuration Management, Roy Levin and Paul McJones, DEC Systems Research Center TR 105, June 1993.
- [15] Software Configuration Management in an Object-Oriented Database, Mick Jordan and Michael Van De Vanter, USENIX Conference on Object-oriented Technologies, Monterey, CA, June 1995.
- [16] B. Liskov et al. The language-independent interface of the Thor persistent object system. In *Object-Oriented Multi-Database Systems*, pp570-588, Prentice Hall, 1996.
- [17] Type-Safe Sharing can be Fast, B. Liskov, A. Adya, M. Castro, Q. Zondervan, Proceedings of the 7th International Conference on Persistent Object Systems, Cape May, New Jersey, May 1996.
- [18] Distribution Strategies for Persistent Java, Susan Spence, Proceedings of the 1st International Workshop on Persistence for Java, Drimen, Scotland, September 1996.
- [19] JDK Preview, Sun Microsystems Inc., <http://www.javasoft.com/products/JDK/1.1/designspecs/index.html>