

pobj: A Lightweight Persistent Objects Library and Its Application to Persistency in Titanium/Java

Gilad Arnold Amir Kamil

January 4, 2005

Abstract

Persistent objects are useful for applications that require data structures to be maintained across multiple executions. This paper describes `pobj`, a lightweight facility for providing persistent objects. The library offloads the actual backing store management to `lladd`, an open-source implementation of the ARIES recovery algorithm, and memory management to external libraries. This layered approach allows `pobj` to be used to implement transparent persistency in Titanium, a language based on Java. `pobj`'s combination of ease of use and flexibility make it ideal for a wide range of programmers and applications, even though it does not perform as well as other persistency tools.

1 Introduction

Persistent memory objects provide a useful abstraction for many application programmers. Ideally, they can offer transparent backup of object data, automatic reconstruction of dynamic memory, atomicity with respect to complex update procedures, and other properties associated with transactional systems.

Transactional facilities for storing virtual memory data in persistent backing stores have been commonplace for a decade. Most facilities in use, however, either suffer from limitations with respect to ease of use and flexibility or rely on a heavyweight implementation to provide persistency. Facilities in higher-level languages generally require a significant amount of programmer involvement despite the

static analysis tools and runtime checks available to such languages and rely on low performance, heavyweight backing stores to achieve persistency..

In this paper, we describe the architecture and implementation of `pobj`, a lightweight library for persistency that is both easy to program and flexible enough to allow fine-grained control over its operation and integration with other libraries. In addition, we leverage the `pobj` library to provide transparent persistency in the Titanium programming language [11], a high-performance dialect of Java.

2 Background

2.1 Design Goals

The main focus of the design of the `pobj` layer was usability and flexibility, with performance only a secondary concern. In particular, the `pobj` layer was designed to provide the following:

- Fine-grained persistency at the granularity of objects.
- Lightweight and flexible library that can serve as a building block for persistency in a higher-level language.
- General purpose interface that is easy to integrate with other libraries such as garbage collectors.
- Ease of use, requiring minimal programmer intervention to obtain persistency.

- Flexibility, allowing fine-grained programmer control over persistent operations.

The Titanium persistency extension was designed with transparency as its main feature without sacrificing flexibility. The goals for the extension were as follows:

- Provide a simple mechanism for specifying persistent objects.
- Automatically reflect operations on persistent objects to the backing store with no programmer intervention.
- Integrate with prior Titanium language features such as garbage collection and threading.
- Support the basic functionality required to allow persistency to be useful, including persistent arrays.

2.2 Related Work

Persistent memory and objects have been popular areas of research over the past decade. Here, we review some of the related work that has been done on persistency.

One of the classic works in the area of persistent virtual memory was *Recoverable Virtual Memory (RVM)* [10], developed as part of the Coda file system at Carnegie Mellon University. RVM allows users to map regions of memory to backing files and supports transactional operations on such regions. Unlike `pobj`, RVM is completely oblivious to the contents of the actual memory and does no pointer swizzling, so regions that contain pointers are not relocatable. In addition, RVM forces the programmer to manually manage mapping of regions and allocation of memory within regions.

Persistent object stores are widespread, with many available implementations. One example is the *Thor* [6] object store developed at the Massachusetts Institute of Technology. Thor enforces type-safety on objects in its store by treating them as black boxes accessible only through their methods. Other object stores generally also maintain some form of type information concerning objects

and their members. `pobj`, on the other hand, maintains only the minimal information necessary to restore the topology of a set of objects on recovery.

Multiple mechanisms for persistency exist in Java. The simplest is the `Serializable` interface [1] that allows a program to write objects to permanent storage. This method, however, creates ordering dependencies between saving objects and recovering them and provides no transactional support. An alternative persistency mechanism is *Java Data Objects (JDO)* [9]. JDO stores objects in relational databases through SQL queries, potentially introducing a large performance overhead over directly interfacing with a persistent backing store. Also unlike the Titanium persistency extension, object and transaction manipulation are not transparent in JDO.

3 Architecture

The `pobj` library is a customizable intermediate layer between the user application and an underlying memory manager and implements persistency of dynamic memory objects by linking to a transaction-capable storage back-end. Thus, `pobj` is completely memory manager independent and reuses the underlying memory facilities for any dynamic allocation operations. Though `pobj` uses the well-known `libc` memory calls (`malloc`, `free`, etc.) in the default settings, we have been able to run it on top of a sophisticated garbage collecting memory manager (see §5.3.3). This key feature is unique compared to previous approaches such as RVM [10], in which memory layout (and therefore memory management) is strictly integrated with persistency management. Our current implementation uses the `lladd` [2] library, an experimental open-source implementation of the ARIES algorithm for recoverable storage management [7], a WAL-based transactional backing store system with automatic crash recovery. Nonetheless, `pobj` could equally link with any generic backing store API that supports transactions and automatic recovery.

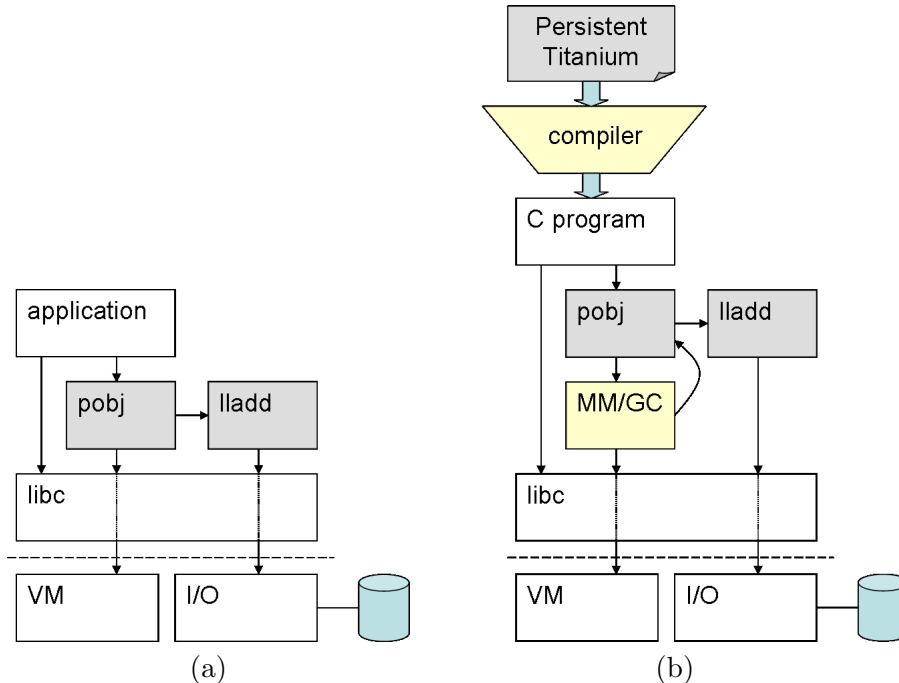


Figure 1: Software block diagrams showing the integration of the `pobj` component with (a) ordinary dynamic memory management of `libc`, as well as (b) garbage collecting memory manager of the Titanium runtime.

3.1 High-Level Design

Figure 1 depicts two schemes in which `pobj` is used to provide a persistent objects abstraction. In Figure 1(a), `pobj` is used directly on top of standard `libc` memory management calls, propagating similar allocation semantics to the user application (i.e., allocation and explicit deallocation). Figure 1(b) demonstrates the use of `pobj` as a persistent object facility for compiled Titanium programs, reusing the Titanium runtime allocator and garbage collector as its underlying memory manager. `pobj` provides up-call hooks for the memory manager, thus permitting garbage collection to work with both persistent and non-persistent objects.

3.2 Object Images

In the design of `pobj`, we have chosen to decouple the in-memory image of a dynamically allocated object from its persistent, on-disk counterpart. This choice was guided by our desire to allow ordinary random memory accesses (either read or write) to

still be applicable to persistent objects. It also allows zero read latency in the presence of heavy concurrency, and is in line with some further simplifying assumptions and design choices. Hence, we maintain a mapping function between in-memory, volatile images of objects and their on-disk, persistent counterparts. This enables flexible persistent image update operations to take place and supports on-the-fly adjustment of the persistency property of each allocated object. An important outcome of this approach is that the `pobj` layer can handle both persistent and non-persistent objects within the same memory *region* and dynamically switch between the two modes. This is opposed to the strict, region-oriented distinction between persistent and non-persistent memory blocks that is implemented in RVM [10].

3.3 Simplifying Assumptions

Aiming at a cost-effective, flexible, and portable software component, we have made several simplify-

ing assumptions that have influenced the functionality, semantics, and interface of the resulting library. While some of these were inherent to our approach for persistency abstraction, others were induced by our short-handed implementation process and could otherwise be avoided given more rigorous requirements. One such assumption is factoring out logical consistency, isolation, and locking policies on managed memory objects. While these issues could be addressed at several possible levels of strictness, we found that resolving them would impose numerous limitations on usage patterns (e.g. enforcing an inadequate locking policy as per application concurrency model) and performance (due to locking overhead). In addition, the resulting component would be much more complex, having to deal with fine-grained locking schemes and deadlock resolution. Instead, we preferred to pass that responsibility to the application and only protect the integrity of the library’s own internal structures. Similar assumptions were also made (and similarly motivated) in previous work [10]. This approach implies that an in-memory object image and its persistent counterpart are never assumed to be compatible, nor is it assumed that their contents satisfy any consistency invariants. Such properties must be enforced by the user application as required.

Another restrictive decision we’ve made is the requirement for an authentic object handle to be passed with each call for persistent memory object manipulation. Although this requirement could be eliminated by using a more extensive memory block resolution scheme (via radix tree lookup, for example) we found it to be a fairly acceptable limitation for all usage cases that we’ve considered, and it is no different than similar policies taken by well-known memory manager interfaces ¹.

¹One example of this is the interface for `realloc` and `free` within the `libc` memory manager, which require an authentic, `malloc` returned object handle to be passed as an argument.

3.4 Topological Memory Image Reconstruction

A major outcome of our object-based persistency scheme is the *topological* nature of `pobj`’s persistent memory image. Specifically, though we stick to *binary* intra-object compatibility of in-memory and persistent images, it is often the case that reconstructed memory would yield a completely different layout of objects in the virtual address space, since a restored persistent object is allocated as if it is being created for the first time. This implies a *logical* inter-object compatibility scheme, and requires some form of *pointer swizzling* [8] to take place when objects are reconstructed from the backing store. Since we must provide a pointer resolution mechanism that is sound and complete, as opposed to other scenarios in which conservative schemes can be deployed (e.g. the dead object detection of garbage collectors [3]), we require a shallow but strict *object typing* to be carried out by the user application. Such typing allows `pobj` to distinguish reference fields from other fields that do not require any adjustment upon recovery and must be carried out for every allocated persistent object in order to guarantee the completeness of our recovery mechanism.

A key property of the topological reconstruction scheme is the preservation of transitive object reachability. In order for reconstructed memory to be meaningful for subsequent executions of an application, persistent dynamic memory structures must be reachable through some *anchor references*. These are found in the form of statically allocated reference variables such as global variables and static local variables in C or static object fields in Java. By providing a means for tying such references to memory objects, `pobj` is capable of reconstructing a complete memory image whose reachability edges are isomorphic with those of the memory image from the previous execution. This approach gives rise to further potential procedures that can take place during memory reconstruction such as automatic deallocation of objects that are not reachable from any anchor reference using simple mark-and-sweep.

3.5 Interface

The interface methods that are exported by `pobj` can be split into six basic categories based on their associated functionality.

3.5.1 Initialization

This includes `pobj_init()` to initialize the library before use, either upon first-time execution (construction of the backing store) or subsequent executions (recovery and reconstruction). Initialization can be parameterized with non-default sets of memory manager calls that are used for external (persistent objects) and internal (`pobj` control structures) memory allocation.

3.5.2 Memory Allocation

This includes `pobj_malloc()` and its variants for transient (i.e. non-persistent) object allocation and for use of ad-hoc memory manager calls, and `pobj_free()` and `pobj_finalize()` for use in explicit object deallocations and deallocations by an external garbage collector. Allocation returns a handle to a (persistent or transient) object, which can then be treated as any other memory object, conforming to the ANSI-C standard with respect to object bounds.

3.5.3 Persistency Control

This set includes `pobj_persistify()` and `pobj_unpersistify()` to switch back and forth between persistent and transient modes of an object.

3.5.4 Object Typing and Persistent Image Updates

This large set includes explicit methods for object *typification* (`pobj_ref_typify()`, `pobj_ref_flag()` and `pobj_ref_unflag()`), methods for dumping the in-memory object images onto their persistent on-disk counterparts (`pobj_update_range()` and its overloaded macros), numerous methods that bundle setting of object fields and typing of primitive fields (`pobj_memcpy()`

and `pobj_memset()`, `pobj_set_int()` and respective variants for other types). This class also contains more sophisticated update mechanisms such as `pobj_update_recursive()`, which performs a BFS-like traversal of a recursive data structure and dumps all changed objects to the backing store. The latter can also be called with a special flag that allows persistification of transitively reachable non-persistent objects.

3.5.5 Static (Anchor) Reference Manipulation

This includes setting and dumping (with implicit typing) of static pointers with `pobj_static_set_ref()` and `pobj_static_update_ref()`.

3.5.6 Transactional Contexts

This set includes `pobj_start()` and `pobj_end()` to begin and commit a (possibly nested) backing store transaction. The current implementation delivers a conservative nesting scheme by which only a single true level of nesting is propagated to the backing store and all others are interpreted as depth counters.

The above set of methods allows great flexibility for applications using `pobj` for their persistency needs. In particular, it supports various trade-offs between fine-grained atomicity and recoverability and runtime performance. These are further discussed in §6.1.2. Figure 2 demonstrates a simple allocation of a list of persistent strings using deferred update methods. The allocation is transparently mediated to the backing store and automatically recovered upon subsequent executions.

4 Implementation

In this section, we describe the major implementation choices that were made in `pobj` and their conformance to the architectural guidelines in §3.

```

Node *list = NULL;

void add_line (char *line) {
    int len = strlen (line);

    pobj_start ();

    Node *node = (Node *)
        pobj_malloc (sizeof (Node));
    char *str = (char *)
        pobj_malloc (sizeof (char) * (len + 1));
    pobj_ref_typify (node, node_ref_fields);

    strcpy (str, line);
    node->str = str;
    node->next = list;

    pobj_update (str);
    pobj_update (node);
    pobj_static_set_ref (&list, node);

    pobj_end ();
}

int main (int argc, char **argv) {
    char line[256];

    pobj_init ();
    while (get_line (line, sizeof (line)))
        add_line (line);
    print_list ();
}

```

Figure 2: Example C program that utilizes `pobj` to construct a continuously growing list of strings from standard input, using deferred update methods.

4.1 Object Wrappers

Each object allocated with `pobj` is wrapped with a `pobj`-specific header and a trailer consisting of reference field type flags. The per-object header carries a minimal data set just enough to allow transient objects to be understood by `pobj` methods: the object's size and a pointer to the object's repository entry or a null value if such an entry does not exist. This information is also sufficient to access the trailing, variable length type flags. For the latter, we safely assume that reference fields are both word-sized and word-aligned so that reference type flags only cost less than 3.2% additional space per object. The two words used in the object header induce only a constant space overhead per object.

The use of per-object type flags is only a temporary solution and is soon to be replaced with a single-word type descriptor in the object header that points to a predefined type inside a *type repository*. This way we hope to eliminate the current linear space and time overheads associated with object allocation and typification, respectively (see §7).

4.2 Repositories

The `pobj` library initializes and maintains two global repositories per each application process. Both repositories are implemented as extensible arrays to allow quick access in constant time and are stored to the backing store segment-by-segment.

Persistent objects repository. An item in this repository maps a persistent memory object to its on-disk counterpart. It also contains per-object information that is sufficient to allow object reconstruction upon subsequent executions, namely the object's size and ID (the pointer value of the object handle returned by its latest allocation). This list is dumped to the backing store at the resolution of a single item, inducing a small overhead during object allocation, deallocation, and persistification.

Static (anchor) references repository. An item in this repository records the absolute location of a static reference variable along

with its last known pointer value held. The use of absolute pointers for this case is guaranteed to be satisfactory for C as static variables are known to reside at fixed offsets within the data segment of the program's binary image. This is not necessarily the case, however, for abstractions in other languages (e.g. mainstream Java, where dynamic class loading is often used). Therefore some further extensions may be required to support these special case conventions.

Both repositories are exclusively locked for write access but are open for concurrent reads. This way, they are guaranteed to be consistent at all times and correspond to the current state of allocated persistent objects and tied static references. It is possible, however, to read dirty (either stale or unstable) repository data, causing severe malfunctions of the library and possibly harming the coherency of the backing store. We rely on the user to take care of this problem, as it is analogous to dangling pointer dereferencing with plain dynamic virtual memory.

It is interesting to note that a transient object differs from a persistent object only by not having a repository item associated with it. Hence, the process of *persistification* corresponds to the allocation of a new repository item, along with a mapping to a newly allocated backing store record, and its association with an already existing memory object. Similarly, it is easy to *unpersistify* an object by applying the reverse procedure.

4.3 Reconstruction

The following is a description of the actions that are carried out during the reconstruction of an application's persistent dynamic memory upon subsequent executions of the program. These actions are triggered by invoking `pobj_init()`, which automatically determines whether a previous storage for the program already exists.

Recover storage. This part is carried out automatically by the underlying transactional storage management library (`lladd`).

Bootstrap. This reads the storage boot record, allowing subsequent loading of the repositories.

Restore repositories. This loads the persistent objects repository and the static references repository into memory, segment-by-segment.

Reconstruct heap objects. This involves the allocation of space and loading of persistent object images while constructing a temporary conversion table between old object IDs (handles) and new ones returned by recent allocation.

Adjust object reference fields. Using the temporary conversion table, adjust any reference field within any restored object to the new object handle.

Tie static (anchor) references. Again using the conversion table, tie any known static reference to the new object handle returned by the recent allocation.

For the last two parts it is assumed that pointers to persistent objects satisfy the authentic handle invariant (see §3.3). Otherwise, pointers may be nullified during adjustments, resulting in partial loss of topological properties. Further improvements can be made to override this behavior (see §7).

By the end of this procedure, the heap is fully reconstructed with reference fields adjusted to new object handles that were returned by the recent object allocation session. Normal execution can be resumed from that point without any manual intervention.

4.4 Recursive Updates

Having reference-wise semi-typed objects in `pobj` allows us to implement procedures that are useful for recursive data structures manipulation. In particular, we provide an iterative method for dumping all of a recursive data structure's objects to their persistent images, using a BFS-like coloring to detect topology cycles. For each processed object, the procedure reads its current contents, compares it with its current in-memory version (possibly using

fast checksum calculation to reduce CPU and I/O overhead), and dumps the memory image if it has changed since the last update.

In its default behavior, the procedure does not process transient objects encountered during its recursive traversal, as these objects will not be reconstructed on subsequent runs. However, by passing in a flag to this procedure, it is possible to replace this step with a persistification of such encountered objects, resulting a fully persistent data structure when the call terminates.

5 Transparent Persistency in Titanium/Java

One of our main design goals for the `pobj` layer was to facilitate the integration of persistency into a higher-level language. In order to drive the design of the layer and as a proof of concept, we implemented language-based persistency in the *Titanium* programming language.

The Titanium persistency extension differs from other persistency schemes in Java in its emphasis on *transparency*. The extension was designed to require minimal programmer intervention in order to obtain persistency, automatically determining when persistent operations occur and issuing the required backing store operations. Also unlike the Java persistency mechanisms, the Titanium persistency extension is tightly integrated in the language instead of relegated to the library API.

5.1 Titanium Overview

Titanium [11] is a single program, multiple data dialect of Java developed at UC Berkeley. It is designed for high-performance scientific computing on the major supercomputers currently in use, including vector machines and clusters of multiprocessors. Titanium has a global memory space abstraction, where all data can be directly referenced by any processor, both for portability and for ease of programming.

We chose Titanium as our base language for multiple reasons. As a research project at Berkeley,

the compiler source code was easily available to us, as well as the developers for helping us in modifying the compiler. In addition, the current compiler implementation translates Titanium code to C, making it easy for us to implement persistency by generating calls to `pobj` functions. The Titanium compiler also includes global analyses that would be useful in optimizing Titanium programs that use persistency (see §5.5.3). Finally, since Titanium is for the most part a superset of Java 1.4, the persistency additions we made to Titanium could just as well be done to Java.

5.2 Persistency Semantics

As a first attempt at persistency in Titanium, we decided to implement a semantics in which persistency is completely known at compile-time. Not only did this restriction make it easier for us to implement persistency, it also allowed us to add persistency without adversely affecting non-persistent operations.

5.2.1 Persistent Types

In order for operations on persistent types to be statically determinable, we added a new type hierarchy separate from the usual hierarchy rooted at `java.lang.Object`. This new tree is rooted at `ti.lang.Persistent`, and all objects of all types in this hierarchy are persistent. As show in Figure 3, types in the persistent hierarchy do not extend and therefore are not assignable to types in the non-persistent hierarchy². Thus, operations on types in the normal hierarchy are always on non-persistent objects, and on types in the persistent hierarchy are always on persistent objects.

In order to avoid programmer confusion on recovery, fields of persistent objects that reference non-persistent objects must be declared `transient`. These fields are not restored on recovery.

²Interfaces are somewhat problematic in this scheme. Our current semantics forbid the implementation of interfaces by persistent types, but an alternate semantics in which a separate hierarchy of persistent interfaces exists is also possible.

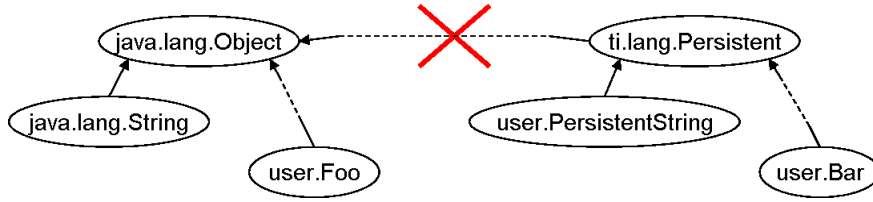


Figure 3: Two separate type hierarchies exist, one for persistent types and one for non-persistent types. A type in one hierarchy cannot be assigned to a type in the other.

5.2.2 Persistent Arrays

In order for persistency to be useful in Java, it must be possible to implement persistent analogs to the non-persistent types in the Java API. Persistent strings and vectors are two examples that are especially necessary. In order to allow such types to be implemented, persistent arrays must be supported.

Since arrays are special types in Java and are not re-implementable by a programmer, a separate mechanism for supporting persistent arrays was necessary. As such, we added a `persistent` type qualifier that can be used to declare an array as persistent. For example, the declaration

```
int[] persistent x =
    new int[4] persistent;
```

can be used to declare and allocate a persistent array of integers.

Persistent array types cannot be assigned to non-persistent array or object types, in order to preserve static knowledge of persistent operations.

5.2.3 Persistent Operations and Transactions

All operations on persistent objects, including allocations and field writes, are guaranteed to be atomically reflected to the backing store without any programmer interference. A mechanism for aggregating transactions, however, is useful for two reasons:

- **Performance.** Persistent operations at the minimum require a write to a log file on disk, so aggregating multiple operations into a single write reduces disk I/O.

- **Consistency.** Logically atomic operations on data structures that require multiple writes could leave a structure in an inconsistent state if a crash occurs in the middle of such an operation. Consider, for example, insertion into a doubly-linked list:

```
oldnode.next.prev = newnode;
oldnode.next = newnode;
```

In order to maintain consistency of the list, these two operations should either both occur or both not happen.

In order to allow manual aggregation of operations by a programmer, we added a `transaction` statement to Titanium. This statement corresponds to a single transaction in the backing store, so either all persistent operations contained within are committed or uncommitted with respect to the store. (No guarantees are made, however, about operations on non-persistent objects.) An insertion into a list could now look like this in order to guarantee consistency:

```
transaction {
    oldnode.next.prev = newnode;
    oldnode.next = newnode;
}
```

the programmer does not have to worry about manually opening and closing transactions, and it is impossible for a transaction commit to be forgotten.

Semantics for nested transactions currently reflect the semantics of the `pobj` implementation: nested transactions commit when the outermost transaction does.

5.2.4 Recovery

Recovery of persistent objects must begin at set of statically addressible roots. Our Titanium language extension defines all `static` variables that reference persistent objects to be recovered, as well as all transitively reachable persistent objects. Fields of persistent objects that reference non-persistent objects or are otherwise `transient`³ are zeroed out on recovery.

5.3 Implementation

Since the Titanium compiler generated C code, persistency can be implemented in Titanium by directly mapping operations on persistent objects to appropriate calls in the `pobj` layer.

5.3.1 Object Layouts and Types

In the Titanium runtime implementation, each object has a reference to a statically allocated type descriptor. Since descriptors are statically allocated and are at the same memory location on every program run, descriptors do not have to be persisted. The Titanium compiler, however, does have to prevent the descriptor reference from being zeroed out on recovery. This is done by neglecting to inform the `pobj` layer that it is a reference, so that the layer treats it as an integer and recovers it.

No changes to object layouts were required for the persistency extension.

5.3.2 Operations

Allocations of persistent objects and arrays use the `pobj` allocation calls in order to allocate space, as described in §5.3.3. Allocations of non-persistent objects and arrays are unaffected.

Reads on persistent objects are implemented the same way as reads on non-persistent objects, as reads do not affect the backing store.

Writes to persistent objects are implemented in two ways, one for references and one for primitives:

³Actually, at the current moment, transience of persistent references and primitive fields is not supported, due to lack of support from the `pobj` layer.

- **References.** References are set using `pobj_set_ref()`.
- **Primitives.** Primitives are set by doing a normal assignment followed by a `pobj_update()` on the affected object. This allows primitives of arbitrary width (which Titanium supports through *immutables*) to be written in the same way.

Writes to non-persistent objects also are not affected.

Writes to static variables of persistent type use `pobj_static_set_ref()`. Writes to non-persistent static variables are not affected.

5.3.3 Memory Management

One of the main benefits of using the `pobj` layer over something like RVM is that it integrates cleanly with memory managers. In the Titanium implementation, we integrated `pobj` with Titanium's Boehm-Weiser garbage collector [3] without any changes to the collector itself.

The Titanium runtime uses multiple different allocation functions in order to obtain space from the garbage collector. These include functions that work only on a single garbage collector page, functions that span pages, functions that don't clear memory on allocations, and functions that do clear memory. With `pobj`'s adhoc allocation support, these functions could be used as necessary by passing them to `pobj` when allocating a persistent object.

Garbage collection itself also integrates well with the `pobj` layer. The Boehm-Weiser collector supports finalization when deallocating an object, and by passing `pobj`'s finalizer to the collector, the collector informs the `pobj` layer on all object collections. Thus, the `pobj` layer can deallocate the space in the backing store corresponding to collected objects. Garbage collection on non-persistent objects is unaffected.

5.4 Bugs and Limitations

Due to time constraints, limitations in the `pobj` layer, and limitations in the `lladd` library, the fol-

lowing limitations exist in the Titanium persistency extension:

- The extensions only work with the sequential and smp backend of the Titanium compiler. This is due to a lack of support for wide pointers in the `pobj` layer, and a lack of distributed support in `lladd`.
- Persistent Titanium arrays are not supported.
- Some semantic checks are not currently done by the compiler. For example, it does not enforce that persistent array types not be assigned to non-persistent array types.
- Persistent operations in static initializers are not currently supported. This is because recovery occurs after initializers are run. Such operations can be supported by deferring their persistification until after recovery is done.
- Transience of persistent references and primitives is not supported. This is due to lack of support from `pobj`.

5.5 Generalized Persistency Semantics

The persistency semantics we implemented in Titanium limit the interactions between non-persistent and persistent data types. Functions and containers originally written to operate on non-persistent objects cannot operate on persistent objects, and predefined non-persistent types cannot be made persistent. Here we present alternate semantics that allow such interactions to occur and discuss the implementation changes necessary to support them.

5.5.1 Persistency by Type

The separation of non-persistent and persistent type hierarchies discussed above is somewhat artificial, required only to avoid runtime persistency checks. By replacing the implementation of writes in the Titanium compiler with a runtime persistency check and branch to the appropriate persistent or non-persistent write, this separation requirement can be implemented. Persistency could then be obtained

by implementing an interface instead of by extending a type.

There are a couple of flaws in this set of semantics. Performance of operations on non-persistent objects suffers greatly, with a single memory write now requiring an additional memory read and a branch. The semantics are also not completely well-defined with respect to inheritance: what happens to the fields inherited by a persistent type from a non-persistent supertype? Primitive fields do not pose a problem, but non-persistent, non-transient reference fields do, violating the previous constraint that persistent objects cannot have such fields. Either this constraint needs to be relaxed, resulting in possible programmer confusion when such fields are not recovered, or such inheritance must be made illegal.

5.5.2 Persistency by Qualification

A more general scheme that allows predefined types to be persistent is to use a `persistent` qualifier to denote an object's persistency, similar to that discussed in §5.2.2 for arrays. However, assignment of persistent types to non-persistent types would be allowed, and the runtime would use the scheme described in §5.5.1 for writes.

An important semantic issue arises with the use of qualifiers: do qualifiers get propagated to reference fields? For example, does the backing array of a persistent `Vector` also become persistent? Without this ability, such a type would be useless. The solution appears to be *persistence by reachability*, in which all objects reachable (through non-transient fields) from a persistent object also are persistent. As such, the compiler would have to determine when an object becomes newly reachable and accordingly make it persistent.

As in §5.5.1, this scheme forces programs to pay a penalty for each non-persistent write, reducing performance.

5.5.3 Qualification Inference

The Titanium compiler contains optimizations that leverage global analyses in order to infer various desirable properties of references in a program. For

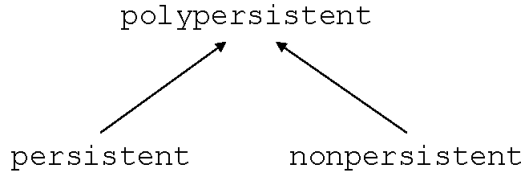


Figure 4: Partial lattice of qualifiers that can be used to infer persistency of references.

example, it can infer when a particular pointer always references an object that is local to the current processor and thus generate code that takes advantage of this fact. The results of this inference rival that of code heavily annotated by hand, with no work required from the programmer [4].

Titanium’s inference system can also be used to determine when variables only reference non-persistent types, and thus be used to generate code that does not require the check and branch discussed in §5.5.1. Three qualifiers must be added to the compiler⁴:

- **persistent**: the referenced object is always persistent
- **nonpersistent**: the referenced object is always non-persistent
- **polypersistent**: the referenced object may be persistent or non-persistent

These qualifiers form a partial lattice with **polypersistent** at the top, as in Figure 4. The optimal inferred solution is that which maximizes the number of **nonpersistent** qualifiers in a program, which can be computed easily [5].

With the qualifiers introduced above, the compiler can eliminate the persistency check and branch for **persistent** and **nonpersistent** references. A program which makes only limited use of persistence would thus pay little penalty for non-persistent accesses.

⁴The **nonpersistent** and **polypersistent** qualifiers need only be internal to the compiler and not exposed to the programmer.

6 Evaluation

We assess the **pobj** layer using two main criteria:

- **Performance**. We evaluate performance compared to other persistency libraries and discuss programmer optimizations to increase performance.
- **Usability**. We compare the difficulty in using **pobj** versus other options.

6.1 Performance

In this section, we examine the performance of the **pobj** layer. We compare it to other persistency implementations and determine the performance benefits of certain **pobj** features.

6.1.1 Relative Performance

In order to quantify the effects of **pobj**’s generality on performance, we measured it against RVM and manual encoding of data structures to disk. We used a microbenchmark that generates a list of integers, saves the list to disk, and subsequently recovers the list from disk. Unfortunately, an apples to apples comparison was not possible, due to the limited functionality provided by RVM and manual encoding. The three implementations and the measurements on them were as follows:

- **pobj**. The list is generated in the context of a single transaction, and each node is allocated as persistent. Save time is computed as the amount of time required to recursively update all nodes and to commit the resulting changes. Recovery time does not include normal initialization time.

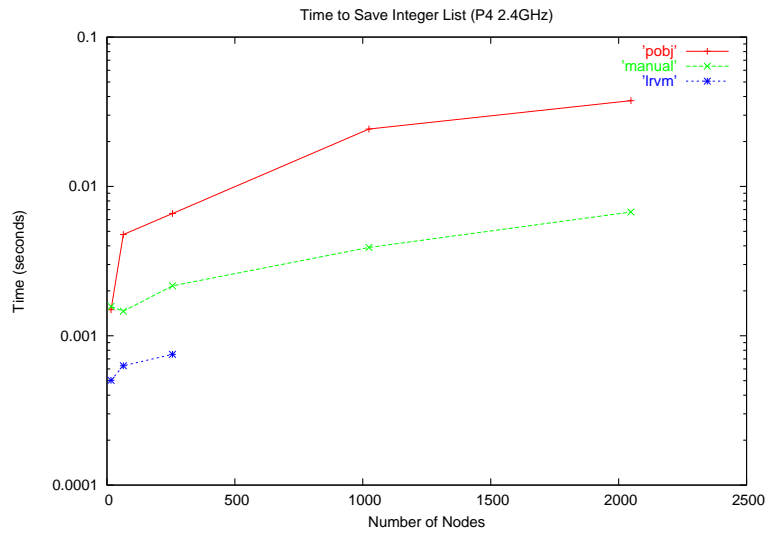


Figure 5: Time to write a simple list data structure to disk using `pobj`, LRVM, and manual writes.

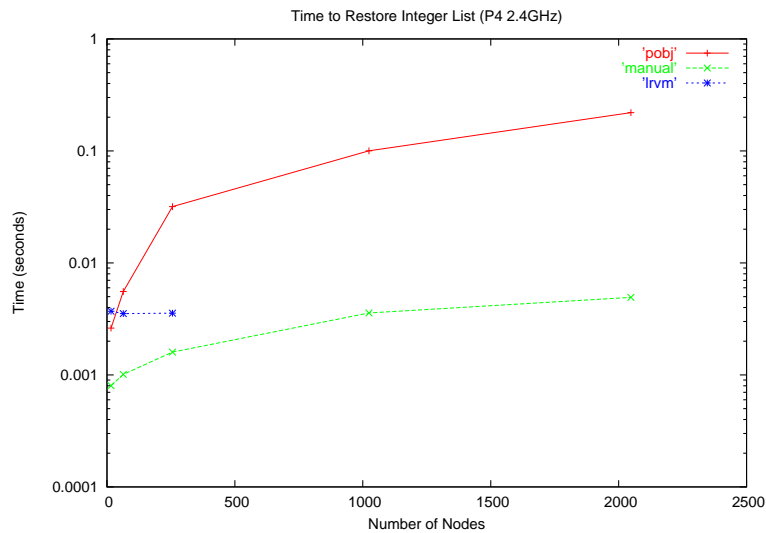


Figure 6: Time to restore a simple list data structure from disk using `pobj`, LRVM, and manual reads. Initialization time for `pobj` and LRVM is not included in their restoration times.

- RVM. A single region is mapped into memory, and nodes are sequentially allocated on this region. Pointers in nodes are kept swizzled in memory. Save time is computed as the amount

of time required to commit a transaction ranging over all nodes. Recovery time does not include initialization of the RVM library.

- **Manual encoding.** List nodes are augmented by a list ID, and nodes contain both a reference to the next node and the next node’s ID. Nodes are encoded on disk as the tuple (NodeID, IntVal, NextID). Upon recovery, each node’s `next` reference is regenerated according to the saved `NextID`, using an array to map IDs to nodes. Save time consists of the time required to write each node’s tuple to disk. Recovery time includes both regenerating nodes from their tuples and adjusting their `next` references.

As Figure 5 indicates, RVM performs about 8 times as fast as manual encoding when saving the list, and manual encoding performs about 5 times as fast as `pobj`. RVM’s fast performance is to be expected, since it does no pointer swizzling (the in-memory swizzled pointers result in slower pointer accesses, however, which was not benchmarked), and its reflection to disk consists of a single sequential dump of the mapped region. There are a few reasons for `pobj`’s slow performance compared to manual encoding:

- It writes over twice as much data to disk, since it writes out each object on allocation and on update.
- The `lladd` backing store has been measured to perform poorly relative to manual disk writes [2].
- The performance measurement includes the time it takes for `pobj` to determine which objects have been updated.

The gap between manual encoding and `pobj` in restorations is even larger, as Figure 6 shows. This is largely in part due to the simplicity of the list data structure and the use of an array in the manual decoding as opposed to a hash table in `pobj`. RVM performs worse than manual encoding for small list sizes due to the fact that regions must be at least as

large as a memory page, which is more than required for small lists.

In general, `pobj` performs 5 to 10 times slower than manual encoding. This is not a large price to pay for its extra features such as transactions and its relative ease of use. On the other hand, `pobj` is another factor of 8 to 10 slower than RVM. This is the price of flexibility.

6.1.2 Transactions and Updates

The `pobj` layer supports user-defined transactions mainly to allow coarser atomic operations in order to maintain consistency. Transactions provide performance benefits as well by allowing multiple operations to be written to the `lladd` log at once. As Figures 7 and 8 show, aggregating multiple allocation or write operations into a single transaction can provide an order of magnitude increase in performance, even for small objects.

In addition to transactions, the `pobj` layer contains support for performing multiple operations on the same object and then issuing a single update to reflect them. The results in 8 show large performance advantages in this technique, with write costs approaching that of non-persistent objects for large numbers of writes.

6.1.3 Partial Updates

In order for operations on large objects to attain a reasonable level of performance, the `pobj` layer provides a mechanism for updating only the modified part of an object in the backing store instead of rewriting the entire object. This is especially important for operations on persistent arrays, which could be arbitrarily large in size. Figure 9 shows that partial updates perform about 2 to 3 times better than full object updates.

While a threefold increase in performance is nice, partial updates should theoretically perform as much as N times better for an object of size N . As can be seen in Figure 9, this is not the case. This is due to an inefficient implementation of large records in `lladd` that requires an entire record to be written even on a partial update.

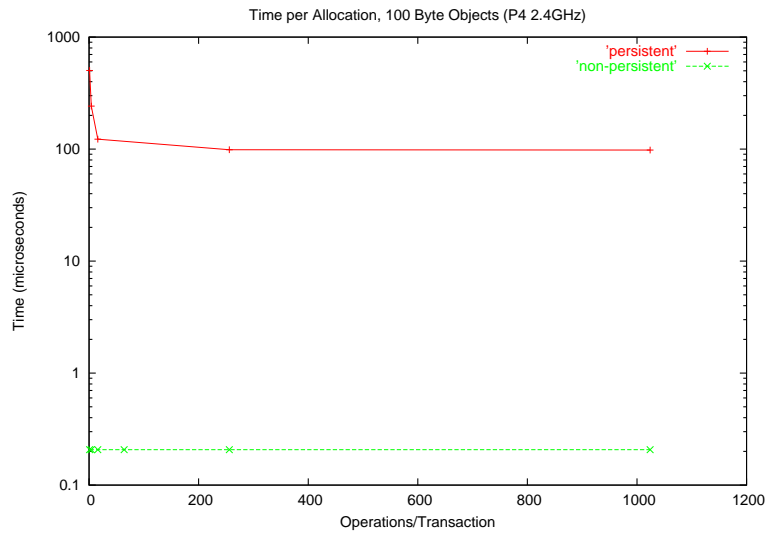


Figure 7: Time to allocate a persistent object and reflect it to the backing store. Aggregating multiple allocations into a single transaction increases performance.

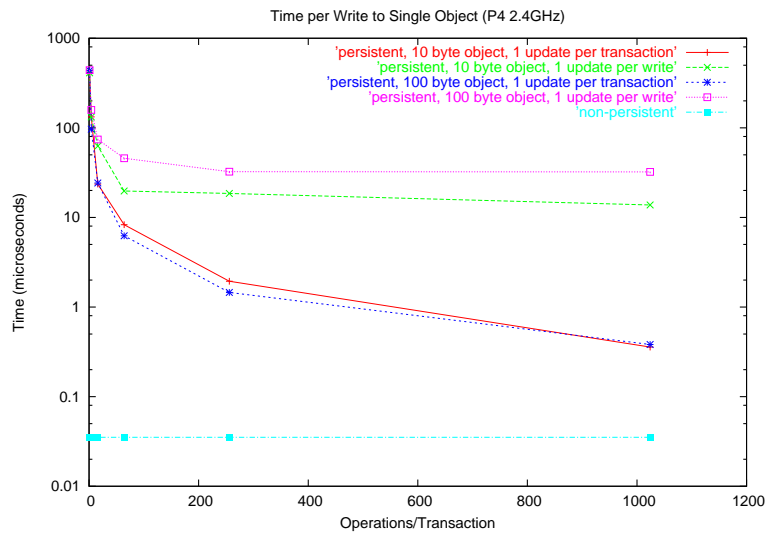


Figure 8: Time to write to a persistent object and reflect the change to the backing store. Aggregating multiple writes into a single transaction increases performance, as does issuing a single update for multiple writes.

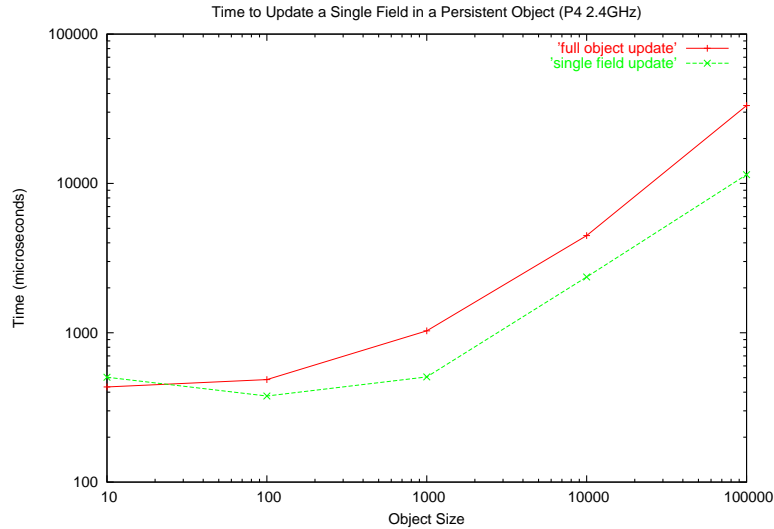


Figure 9: Comparison of writing out an entire object upon a field write and writing out only the modified field.

6.1.4 Titanium Performance

The performance of the Titanium persistency extension is on par with C code that directly uses `pobj`. Unfortunately, we could not compare its performance to `Serializable` or `JDO`, as neither is currently supported in Titanium.

6.2 Usability

One of the main advantages that the `pobj` layer has over other persistency libraries is its ease of use and flexibility. Our experience with RVM in particular was not entirely pleasant, with multiple operations required to initialize the library and allocate a region in which to work in. These operations would seemingly fail for no apparent reason; for example, we could not get RVM to map a region into statically allocated space, despite abiding by the documentation’s requirements that the buffer be page-aligned. Combined with the lack of support for relocation, this forced us to manually swizzle pointers between list nodes. In addition, we were unable to successfully recover more than about 400 list nodes from a

persistent region, regardless of how large the region was, limiting the benchmarks we could run.

Persistency through manual dumping of data structures is also difficult to use. Not only must pointer swizzling be done by hand, but the data contained in a structure must be encoded in such a way as to be recoverable. This is not only tedious but error-prone as well⁵.

The `pobj` interface, on the other hand, is quite easy to use. A single call to an initialization functions is required to start the library, and the `pobj` allocation functions can be used as drop-in replacements for `malloc()`, `calloc()`, and `free()`. Only two calls are required to save a data structure to the backing store, as long as the structure has been typed (§3.5.4): one to save its root in a static variable and the other to recursively update all objects reachable from the root. The `pobj` layer does not sacrifice flexibility for ease of use, however. It provides many functions that advanced programmers

⁵For example, checkpointing code in a Titanium heart simulation suffered from many bugs and took the developers months to get it working.

can use to control what gets updated, when updates occur, and when transactions begin and end. In addition, it exports an interface that can be easily integrated with other libraries such as garbage collectors. This combination of usability and flexibility is `pobj`'s main advantage over RVM and other persistency layers.

7 Future Work

In this section, we describe several issues that need to be addressed in the implementation of `pobj` and the Titanium persistency extension. We consider all of them to be beneficial to user applications. However, while some of them are relatively simple technical hacks, others require extensive modifications to current `pobj` and Titanium internals.

Type descriptors. As mentioned in §4.1, replacing the currently used embedded type information with global, user-definable type descriptors would lead to a considerable performance gain, thanks to reduced time and space overheads. Implementing this feature requires an additional types repository to be maintained by `pobj`, similar to the ones that are already in use.

Transient fields. In current `pobj` implementation every field of a persistent object, reference or non-reference, is always restored during reconstruction. Furthermore, any reference field of such an object will always be traversed during recursive update procedure. However, it is sometimes necessary to denote *transient fields* within objects that do not follow this behavior: the contents of such fields is known to be lost (nullified) through reconstruction, and they should not be traversed by recursive update procedures. (This is analogous to transient fields in Java and their role in object serialization.)

Supporting transient fields would require adding an considerable amount of type information to each object, given that type descriptors are not yet implemented. Introducing

them becomes straightforward in the presence of type descriptors, hence we consider it a dependent feature of the above.

Correct swizzling of non-authentic handles.

As previously mentioned in §4.3 and §3.3, this feature would allow correct reconstruction of object reference fields and static references, even if the authentic handle invariant is not met. One way to support it is to construct a temporary radix tree that will allow efficient mapping of arbitrary pointer values to object handles. A handy feature for many usage patterns, it will be implemented as a future extension to `pobj`.

Generalized persistent statics. This relates to the persistification of any static variable / structure, that is not necessarily a static reference. Such a feature would require considerable extensions to the way `pobj` handles persistent image binding, so we consider it to be a long-term extension.

Explicit abort semantics. This relates to the ability to invoke explicit abort methods that restore a set of objects to the state of their last known persistent image.

Generalized Titanium semantics. As discussed in §5.5, generalizing Titanium's persistency semantics would make it possible to use legacy code with persistent objects.

Titanium persistency optimizations. The Titanium compiler currently issues an update for every persistent operation in a transaction. It should be possible for the compiler to detect when multiple operations occur on the same object and thus issue only a single update at the end of the transaction.

8 Conclusion

The `pobj` layer is a useful tool for providing persistency in both C and high-level languages such as Titanium. Though it does not perform as well

as alternative persistency facilities, its ease of use and flexibility make it ideal for programs that don't make heavy use of persistency.

While other mechanisms for providing persistency in a high-level language have existed for years, they have primarily been implemented at the application level over heavyweight databases. The Titanium persistency extension, on the other hand, integrates persistency into the language, allowing persistency to be transparent to programmers. Its implementation on top of the lightweight pobj layer allows it to be integrated with the rest of Titanium's language features without causing any interference. A similar approach can be followed in Java to provide easy persistency to the masses.

References

- [1] Java object serialization specification, 2001. <ftp://ftp.java.sun.com/docs/j2se1.4/serial-spec.pdf>.
- [2] J. Bayer, J. Blomo, J. Kittiyachavalit, and E. Brewer. Lladd: Lightweight library for atomic, durable data. Technical report, UC Berkeley, March 2004. <http://www.xcf.berkeley.edu/~jkit/LLADD.pdf>.
- [3] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.
- [4] B. Liblit and A. Aiken. Type systems for distributed data structures. *Principles of Programming Languages*, January 2000.
- [5] B. Liblit, A. Aiken, and K. Yelick. Type systems for distributed data sharing. *International Static Analysis Symposium*, June 2003.
- [6] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shira. Safe and efficient sharing of persistent objects in thor. *Proceedings of the 1996 ACM SIGMOD Int. Conf. on Management of Data*, pages 318–329, June 1996.
- [7] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.
- [8] J. Moss. Working with persistent objects: To swizzle or not to swizzle. *Transactions on Software Engineering*, 18(8):657–673, 1992.
- [9] C. Russell. Java data objects: Jsr 000012. Technical report, Sun Microsystems, Inc., 2000. <http://java.sun.com/products/jdo/index.jsp>.
- [10] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, 1994.
- [11] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11–13), September–November 1988.