The Transactional Object Cache as a Basis for Persistent Java System Construction

Stephen M. Blackburn and Robin B. Stanton*

Department of Computer Science
Australian National University
Canberra ACT 0200 Australia
{Steve.Blackburn,Robin.Stanton}@cs.anu.edu.au

Abstract

The promise of Java as the vehicle for widely used, industrial strength orthogonally persistent systems places a renewed emphasis on implementation technologies for orthogonally persistent systems. The implementation of such systems has been held back by a number of factors, including a breadth of technologies spanning database and programming language research domains, and difficulty in capitalizing on the fruits of the mainstream database research community.

In this paper we present PSI—a practical storage abstraction that separates database and programming language concerns and facilitates the adoption of mainstream transactional storage technology within orthogonally persistent systems. We argue for PSI as the basis for persistent Java system construction with particular reference to how it might be applied to PJama$_0$ [Atkinson et al. 1996].

1 Introduction

In Carey and DeWitt's retrospective on the past ten years in the database community [Carey and DeWitt 1996], two items are singled out for special attention as "casualties" of the past decade. These are *persistent programming languages* and *database toolkits*. The work reported in this paper raises the suggestion that Carey and DeWitt may have jumped horses a little early, as we claim that something akin to the database toolkit approach will play an fundamental part in the realization of industrial strength, widely used orthogonally persistent systems.

The work reported in this paper has developed in response to our desire to build efficient, robust, scalable orthogonally persistent programming environments. It has been shaped by our attempts to come to terms with the breadth of the technologies involved and the relative smallness of the persistence research community. The effect of these factors has been articulated by Atkinson and Morrison in their 1995 review of orthogonally persistent object systems [Atkinson and Morrison 1995], where they say:

> However, [existing systems] do not manage to provide full database facilities—that is, few can actually demonstrate a complete repertoire of incrementality, transactions, recovery, concurrency, distribution, and scalability. (It appears that this is more a consequence of teams being unable to muster the effort to tackle all of these issues together rather than of any fundamental limits.)

The focus on *orthogonally persistent* systems (i.e. systems where the persistence of data is orthogonal to all other properties of the data [Atkinson and Morrison 1995]), stems from a desire to build systems that elegantly unify the divergent database and programming language paradigms. While ODBMSs and object-relational systems bring programming languages and databases *closer* together, they do not seek to *unify* the two paradigms. The challenge of building efficient orthogonally persistent systems thus goes beyond the pragmatic appeal of building systems for today and instead focuses on systems for the future.

It seems clear to us that wide-spread uptake of orthogonally persistent systems will depend on the efficient and robust delivery of database facilities such as those outlined by Atkinson and Morrison.

*The authors wish to acknowledge that this work was carried out within the Cooperative Research Center for Advanced Computational Systems established under the Australian Government's Cooperative Research Centers Program. We also wish to acknowledge generous support from Fujitsu Ltd and Object Technology International.

The realization of this objective will depend on either a substantial increase in the size of the persistence community's research effort or a change of implementation approach.

This paper outlines a new implementation approach which can be characterized in terms of the following attributes: separation of concerns and concentration of expertise; maximal capitalization on the work of the database research community; and the portability and amenability to collaboration of implementations.

The remainder of this paper is structured as follows: First we will introduce and motivate an architectural framework for store implementation. Next we will present an interface based on an abstraction of that architecture. Finally we will discuss the practical application of PSI to persistent Java.

2 The Transactional Object Cache

The choice of the transactional object cache as the basis for a new design approach is rooted in the authors' experience with scalable store construction in the design of MC-Texas [Blackburn and Stanton 1996] and MC-DataSafe [Blackburn et al. 1997] and in the observation that the persistence community has not been able to build systems with the level of efficiency, robustness and functionality found in database products.

From the perspective of the future development of scalable persistent systems, perhaps the three most important lessons of the MC-Texas and MC-DataSafe experiments are these [Blackburn 1997]:

- The importance and appropriateness of a transactional model of concurrent computation when working in a distributed persistent space.
- The extent to which concurrency control and recovery fall within the mainstream of database research—research which the most (orthogonally) persistent architectures are not suited to extensively exploiting.
- The impact of temporal and morphological grain on performance.

These lessons can be interpreted as indicating a need for an architecture for scalable (orthogonally) persistent systems which:

1. Embodies a transactional model of concurrency control.
2. Separates database and programming language concerns in a way that facilitates the capitalization on research by the database community, particularly in the areas of concurrency control and recovery.
3. Offers an object-grained interface to clients which minimizes the need for copying of data.

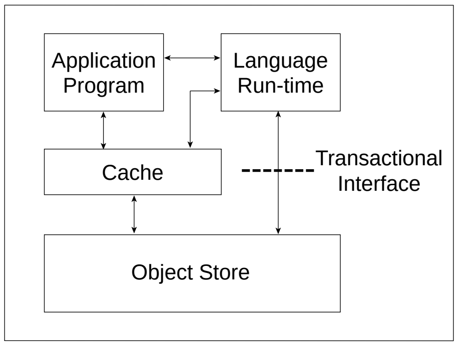


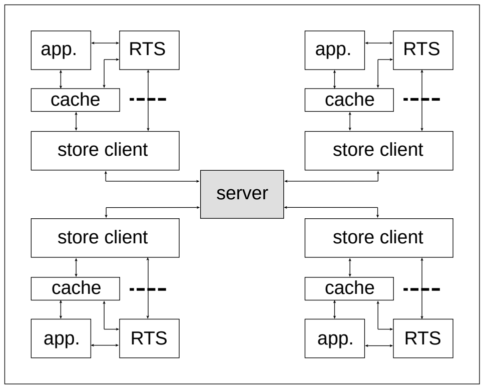
Figure 1: The transactional object cache architecture. The architecture of the object store is transparent to the application.

The transactional object cache architecture (figure 1) is one which satisfies all of these criteria. The basic architecture consists of five key components: an application program; an (optional) language runtime system (RTS); a cache; an object store; and a transactional interface. The basic model is that of the application program operating (via direct memory access) over a cached image of the store. The validity (in transactional terms) of the cached image seen by the application is ensured by appropriate use of the transactional interface.

The first of the above criteria is addressed by virtue of the transactional framework in which all cache consistency actions occur—the architecture is intrinsically transactional. Criterion two is met through the existence of the transactional interface, which separates the store and RTS. The extent to which the second criterion is met will be largely a function of the the interface definition. The dominance of the transactional object cache paradigm in the ODBMS literature [Carey and DeWitt 1986; Carey et al. 1994; Franklin 1996] has lead to mainstream database technology often being targeted at or sympathetic to that approach. Orthogonally persistent programming systems that adopt the transactional object cache architecture therefore stand to profit from the database community's research outcomes in a very direct way. The final criterion is met by virtue of the direct cache access given to the application and the object grained nature of the interface.

2.1 The Transactional Object Cache as a Platform for Distribution

While caching has a natural role in persistent system design as a means of hiding disk latency, it is also important to distributed systems where it is used to hide network latencies. There exists a well established literature on the distributed cache coherency problem. Approaches to this problem fall into two broad camps: transactional cache coherency [Franklin et al. 1997], based on the transactional notion of *isolation*; and distributed shared memory [Adve and Gharachorloo 1995], based on the programming language community's *cooperation-oriented* view of concurrency. Given the transactional nature of most orthogonally persistent systems, the first approach is of most interest here.



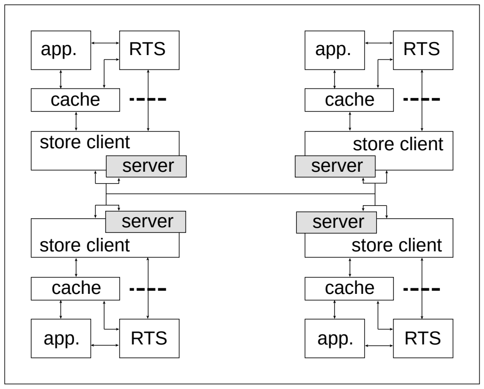


Figure 2: Client-server (left) and client-peer (right) transactional object cache architectures. In both cases the distributed nature of the underlying architecture is transparent to the run-time systems and applications.

Although not explicitly object-based, all of the the wide range of approaches to transactional cache coherency surveyed by Franklin et al. in [Franklin et al. 1997] are directly applicable to the transactional object cache architecture. These approaches span almost all dimensions of the concurrency control design space and include classic optimistic and locking architectures.

A powerful property of the architecture is that its semantics are defined in strict transactional terms and as a consequence embody the distribution-independent concurrency that flows from the isolation-oriented transactional view of concurrency, thus opening the door to transparency of distribution in implementations[1]. Approaches to transparent distribution in this context include client-server (distributed clients, single store), and client-peer [Carey et al. 1994; Blackburn and Stanton 1996; Blackburn 1997] (distributed clients, distributed store) architectures (figure 2).

3 PSI: A Transactional Storage Interface

Having identified the transactional object cache architecture and the value of a common storage interface, we will now briefly describe PSI, which we have developed to fulfill this role.

An important element of the interface is the identification of core and extended functionality. The core interface provides the client with the minimal functionality needed for basic ACID transactional

[1]The isolation guarantee of ACID transactions gives us this property. In the case where isolation is weakened, such as in some advanced transaction models, concurrency and distribution become more opaque.

object caching, whereas the PSI extensions place greater demands on the PSI implementer but give the client features such as advanced storage structures (indexes and collections) and intra-transactional checkpointing and rollback. To keep the description of PSI brief, we will only define the core PSI functionality here. A complete description of PSI appears in [Blackburn 1997].

3.1 A Semantic Framework

In order to effectively articulate the semantics of the interface, we first identify a clear abstraction of the transactional object cache architecture.

Although the key building blocks for a transactional interface may be fairly clear (begin, commit, abort etc.), the goal of flexibility both above and below the interface makes the identification of the precise semantics of these operations with respect to the various areas of store management more difficult. For example, a number of questions are raised by a simple write to the cache. When is that write made stable? When may the buffer associated with that data be freed? When will that change be made visible to other transactions? Formulating answers to these questions is made all the more difficult by a tendency for the various concerns to be blurred in the literature. In order to help meet our objective of flexibility in our interface design, three key concerns of a transactional object cache are separated and identified:

- stability,
- visibility,
- and cache management.

The first two are unambiguously central to a transactional storage interface; the third is included in a pragmatic response to the demands of efficient store construction. By establishing abstract interfaces to each of these concerns, a store interface can be built up and described precisely in terms of its semantics with respect to each of the separate concerns. The remainder of this section will focus on the development of those abstract interfaces. The approach taken will be to treat the three concerns as *orthogonal*—separate models of stability, visibility and cache management will be developed. Having developed the models and abstract interfaces to them, there will be a discussion on how the three concerns come together to form a rich abstraction of the transactional object cache architecture.

3.1.1 Stability

Stability is fundamental to most transaction models. The commit of an ACID transaction requires that all changes made by that transaction be made *durable*. Durability combines stability with irrevocability. By contrast, changes made stable (but not durable) may be subsequently rolled back.

In order to properly describe the stability semantics of a transactional object cache, it is helpful to first develop an abstract model of stability. Stability in this context concerns the maintenance of a stable image of a system state that corresponds in a meaningful way with the state of a dynamic system that is otherwise volatile.

The state of such a system can be represented as a history, h of (state changing) atomic events, e_i:

$$h = e_0.e_1.e_2 \dots e_n$$

By identifying a durable global stability history, hs_g, and a set of volatile local stability histories, $HS_l = \{\langle 0, hs_0\rangle, \langle 1, hs_1\rangle, \dots, \langle n, hs_n\rangle\}$, the atomicity and durability of a simple transactional system can be defined by describing the changes of state associated with a given transaction t in hs_t (where $\langle t, hs_t\rangle \in HS_l$). In this model, a transaction is made durable via an operation which appends hs_t to hs_g (i.e. $hs'_g = hs_g.hs_t$, where hs_g and hs'_g denote before and after values of hs_g respectively). With the addition of marker events, m, and stability events, s, the semantics of checkpoint/rollback and intra-transactional stability can be described respectively.

Having set in place a simple abstract model of stability capable of representing a wide range of stability scenarios, the remainder of this section will identify a series of stability primitives in terms of that model.

The global stability history is initially empty and there are no local stability histories ($hs_g^{initial} = empty \wedge HS_l^{initial} = \{\}$). In the following shorthand will be used to refer to simple modifications of local history, the effects on HS_l being implicit. For example, the notation

$$hs'_t = hs_t.e$$

Core	Logging	Extended Trans.
BeginUpdates	CheckpointUpdates	DelegateUpdates
NotifyUpdate	RollbackUpdates	
AbortUpdates	StabilizeUpdates	
MakeDurable		
EvictVolatile		

Table 1: Stability primitives. Only the semantics of core primitives are described in this paper. See [Blackburn 1997] for a complete description of all primitives.

should be read as shorthand for:

$$HS'_l = \{\langle t_i, hs_{t_i}\rangle \in HS_l | t_i \neq t\} \cup \{\langle t, hs_t.e\rangle | \langle t, hs_t\rangle \in HS_l\}$$

The following primitives, described in terms of the above stability model, are sufficient to describe the stability semantics of a basic flat ACID transaction, t:

BeginUpdates(t) $hs_t = empty \ \wedge \ HS'_l = HS_l \cup \{\langle t, hs_t\rangle\}$

NotifyUpdate(t,o) $hs'_t = hs_t.e_o$, where e_o is an event describing a change of state to some object, o.

AbortUpdates(t) $HS'_l = \{\langle t_i, hs_{t_i}\rangle \in HS_l | t_i \neq t\}$

MakeDurable(t) $hs'_g = hs_g.hs_t \ \wedge \ HS'_l = \{\langle t_i, hs_{t_i}\rangle \in HS_l | t_i \neq t\}$

In addition to these, there needs to be a primitive with global scope that describes the eviction of all volatile data (allowing system crash and program termination to be modeled):

EvictVolatile $\forall \langle t, hs_t\rangle \in HS_l \ \ hs'_t.s.E_c = hs_t$, where $E_c = e_{c_0}.e_{c_1}\ldots e_{c_n} \ \wedge \ s \notin E_c$

A simple ACID transaction would thus consist of BeginUpdates followed by zero or more NotifyUpdates and then one of MakeDurable, EvictVolatile or AbortUpdates.

3.1.2 Visibility

Visibility is another issue of fundamental importance to transaction models. ACID transactions ensure *isolation* by restricting visibility of changes made by uncommitted transactions. Extended transaction models often allow the controlled relaxation of isolation. There are a wide range of approaches to implementing visibility control, the design space for which spans many dimensions [Franklin et al. 1997].

Central to an understanding of visibility is the notion of transactions operating over potentially invalid images of the state of a store. The responsibility of the visibility control mechanism is to ensure that no transaction that saw an invalid image of the store be allowed to commit. As outlined by Franklin et al. [1997], there are two broad implementation alternatives: *avoidance* based schemes, where transactions are prevented from ever being exposed to invalid images of the store; and *detection* based schemes, where exposure to an invalid image of the store is detected and the transaction prevented from committing[2]. In either case, the visibility control mechanism must be able to determine the validity of the image of a store seen by a given transaction. Validity is usually defined in terms of serializability—a transaction is valid only if it can be serialized with respect to all previously validated transactions.

In order to describe visibility semantics concisely, a reference model for visibility will first be described. Note that this model is *orthogonal* to the model for stability presented in the previous section. The integration of stability, visibility and cache management semantics to fully capture the semantics of the transactional object cache is addressed at the end of this chapter.

The visibility semantics of a transactional system can be described in terms a single history, hv, of visibility events, e_i:

$$hv = e_0.e_1.e_2\ldots e_n$$

[2]Franklin et al. [1997] argue for a taxonomy of concurrency control approaches based on a separation into avoidance and detection based schemes. The taxonomy specifically avoids the ambiguity of the related pessimistic/optimistic distinction.

A transaction, t, is then modeled as a sub-history of hv, hv^t, and the store image seen by t is defined by the visibility events composing hv^t. T denotes the set of all transactions in hv, where all transactions are disjoint with respect to hv and T completely covers hv:

$$(e \in hv) \Rightarrow ((\exists t_i \in T | (e \in t_i)) \wedge (\forall t_j \in T, t_i \neq t_j \;\; e \notin t_j)))$$

The notion of irrevocability, which is central to modeling transactions, is introduced by defining *irrevocable*(e) to denote that e is irrevocably part of hv. More generally, *immutable*(t) is defined such that hv^t is a fixed sub-history of hv (i.e. membership of hv^t is static) and *immutable*$(t) \Rightarrow ((e \in hv^t) \Rightarrow$ *irrevocable*$(e))$. The property of immutabilty can be used to capture the notion of transaction commit—all committed transactions are immutable while uncommitted transactions are mutable (both revocable and appendable).

The visibility events which compose the histories must capture sufficient semantic detail such that the validity of the store image as projected by a given sub-history can be determined. Furthermore, the events must capture the range of visibility scenarios possible in a cached store, most notably: shared access to an image of an object and the possibility of multiple 'versions' of objects existing as a result of replication. These facets of visibility are covered by the definition of read and write begin and end events with respect to versions, v, of objects, o, in particular workspaces, w: $r_{o_{v,w}}$, $\bar{r}_{o_w}$, w_{o_w}, and $\bar{w}_{o_{v,w}}$.

The concept of workspace is used here to refer to a single, potentially shared, image of an object. Interactions and potential conflicts between transactions sharing a single image of an object (for space efficiency reasons, for example) can thus be modeled. Object version numbers, v, monotonically increase and are incremented as part of each $\bar{w}_{o_v}$ event (which corresponds to the new version of o becoming visible in some scope). Read events, $r_{o_{v,w}}$, may be with respect to any existing version, v, of o and any workspace w.

Having constructed such a model of visibility, a number of functions are defined that will enable a user to reason about the validity of an image of the store as seen by a particular transaction t. The first of these is a termination function $T(hv^i)$ which tests termination on all reads and writes within a sub-history hv^i (the notation $a \rightarrow b$ is used to denote a preceding b in hv):

$$T(hv^i) \quad = \quad (\forall r_o \in hv^i \, (\exists \bar{r}_o \in hv^i \;\; (r_o \rightarrow \bar{r}_o))) \;\; \wedge \;\; (\forall w_o \in hv^i \, (\exists \bar{w}_o \in hv^i \;\; (w_o \rightarrow \bar{w}_o)))$$

In addition, a workspace isolation function, $W(hv^i, hv^j)$, is defined such that it is true only if no read events composing a given sub-history hv^i overlap with any write events in sub-history hv^j *and* are with respect to a common workspace image of an object:

$$\begin{aligned} W(hv^i, hv^j) \quad = \quad & (\forall r_{o_w}, \bar{r}_{o_w} \in hv^i \;\; (\nexists w_{o_w} \in hv^j \;\; (r_{o_w} \rightarrow w_{o_w} \rightarrow \bar{r}_{o_w}))) \;\; \wedge \\ & (\forall w_{o_w}, \bar{w}_{o_w} \in hv^j \;\; (\nexists r_{o_w} \in hv^i \;\; (w_{o_w} \rightarrow r_{o_w} \rightarrow \bar{w}_{o_w}))) \end{aligned}$$

Finally, a serializability function $S(hv^i, hv^j, hv^k)$ is defined such that $S(hv^i, hv^j, hv^k)$ is true only if the store image as seen by hv^i is consistent (serializable) with respect to hv^j, where hv^k denotes a sub-history of all events with which conflicts are ignored:

$$\begin{aligned} S(hv^i, hv^j, hv^k) \quad = \quad & \forall r_{o_v} \in hv^i \;\; (((\exists \bar{w}_{o_v} \in hv^j) \; \vee \; (\exists \bar{w}_{o_v} \in (hv^i \cup hv^k))) \;\; \wedge \\ & (\nexists \bar{w}_{o_{v'}} \in hv^j \;\; (\bar{w}_{o_v} \rightarrow \bar{w}_{o_{v'}}))) \end{aligned}$$

The inclusion of hv^k is necessary because given a decision to ignore conflicts between events in hv^i and hv^k, update events in hv^k form part of the valid store image seen by hv^i.

With the visibility model and the three validity functions defined, an abstract interface with respect to visibility in a transactional object cache can now be defined. The model is sufficiently rich to allow the user of the abstract interface to assess the transactional validity of a very wide range of visibility scenarios. The abstract interface will be introduced in terms of core and extended functionality (as with the stability interface) and consequently begins with the particular (i.e. basic ACID) and extends to the general.

In the following description, a number of conventions will be used:

- Appending an event to a sub-history implies appending the event to hv: $(hv^{t'} = hv^t.e) \Rightarrow hv.e$.
- Truncating a sub-history implies removal of events from hv: $(hv^{t'}.e_i = hv^t) \Rightarrow (hv' = hv \setminus e_i)$, where $\setminus$ denotes history difference.
- The operation $hv^i \cup hv^j$ denotes the order-preserving merging (union) of two sub-histories.

Furthermore, by definition any manipulation of a sub-history corresponding to an immutable transaction is not permitted.

Core	Logging	Extended Trans.
BeginVisibility	CheckpointVisibilty	DelegateVisibility
ReadIntention	RollbackVisibility	IgnoreConflict
ReadComplete		
WriteIntention		
WriteComplete		
AbortVisibility		
Terminated		
Finalize		
Expose		

Table 2: Visibility primitives. Only the semantics of core primitives are described in this paper. See [Blackburn 1997] for a complete description of all primitives.

3.2 Visibility and Core Functionality

Using the above model of visibility, the following primitives are sufficient to describe the visibility semantics of a simple flat ACID transaction, t:

BeginVisibility(t) $hv^t = empty \ \wedge \ T' = T \cup \{t\}$

ReadIntention(t,o) $hv^{t\prime} = hv^t.r_o$

ReadComplete(t,o) $hv^{t\prime} = hv^t.\bar{r_o}$

WriteIntention(t,o) $hv^{t\prime} = hv^t.w_o$

WriteComplete(t,o) $hv^{t\prime} = hv^t.\bar{w}_{o_v}$

AbortVisibility(t) $(hv' = hv \setminus hv^t) \ \wedge \ (T' = T \setminus \{t\})$, where the symbol $\setminus$ denotes history difference and set difference respectively (i.e. the events composing sub-history hv^t are removed from hv).

Terminated(t,o) $T(hv^{t_o})$, where hv^{t_o} refers to a sub-history of hv consisting of all events in transaction t relating to object o.

Finalize(t) $(T(hv^t) \wedge S(hv^t, hv^i, hv^{ic_t}) \wedge W(hv^t, hv^w))$, where hv^i is the sub-history of hv consisting of all irrevocable events, hv^{ic_t} is the sub-history of hv consisting of all events with which t is ignoring conflicts, and $hv^w = hv \setminus (hv^t \cup hv^{ic_t})$.

Expose(t) $immutable(t) = true$

3.3 Cache Management

A third dimension of the transactional cache architecture is cache management. A cached store design is motivated by the desire to hide IO latency and introduce replication through caching. While visibility is concerned with the state of the store as it might be seen by a given transaction, cache management is concerned with the *availability* of that image to the transaction.

Cache management can be modeled in terms of each active transaction, t, operating over a logically distinct cache c_t within which are present some set of objects: $c_t = \{o_0, o_1, \ldots, o_n\}$. An object is only available to a transaction if present in that transaction's (logically distinct) cache.

Core
Fix
Unfix

Table 3: Caching primitives.

Only two primitives are necessary for the implementation of a cache management scheme:

Fix(t,o) $c_t' = c_t \cup \{o\}$.

Unfix(t,o) $c_t' = c_t \setminus \{o\}$.

With these the client can notify the store of when it requires availability to a given object. The state of the available objects is a function of the visibility control mechanism.

3.3.1 Generality and Completeness

Each of the three orthogonal abstractions outlined above are general—in the sense that they are premised only by intrinsics of scalable persistent systems, namely *caching*, *atomicity* by way of transactions, and *layered software abstractions*—and complete in so far as they support the wide range of scenarios derivable from a combination of ACID transactions, delegation, isolation relaxation, intra-transactional stability, and checkpoint / rollback.[3] When brought together, the orthogonal abstractions yield a full abstraction of the transactional object cache with the same generality and completeness. The remainder of this section gives a brief overview of the full abstraction.

The relationships between each of the abstractions are not symmetric. Visibility can be thought of as dominant because it is visibility that defines the image of the store seen by each transaction. By contrast, stability and cache management have ancillary roles of defining the stability and availability of the store image as determined by the visibility model.

By and large the integrated semantics of the full abstraction are straight-forward. However, it should be emphasized that the cache is merely a means of accessing the store image as defined by the visibility model. Any access to the cache outside the context of a fix(t,o), unfix(t,o) pair is not meaningful and any access within the context of a fix(t,o), unfix(t,o) pair is only meaningful insofar the visibility model indicates the validity of such an access.

Finally it should be noted that although the abstraction is presented in terms of object-grained semantics, it is applicable to data movement and coherency at any granularity and so may be trivially adapted to account for such.

3.4 Separation of Concerns

Having presented the semantics of a transactional object cache, we now look at how various elements of persistent programming system design impact on the PSI interface design. We start by identifying key design choices for a persistent programming language (PPL) implementer. Having identified these, we determine the extent to which PSI will take a role in that aspect of PPL construction on the basis of its relative proximity to storage and language issues. In the remainder of this section, we identify six key issues and for each argue the case for PSI's role with respect to that issue. We refer to the PPL as the interface's "client" throughout this section.

Persistence Identification PSI presents its clients with a store that supports persistence by reachability from a single root and that guarantees the referential integrity of object identifiers (OIDs) *within the scope of a transaction*. An OID is undefined as soon as it leaves its transactional scope. Support for extended transaction models allows clients to use delegation of transactional scope to avoid re-traversing from the root of persistence at the start of each transaction. This approach does not inhibit the client PPL from presenting applications with a more elaborate space of named entry points, it is left to the client to ensure that all advertised entry points remain reachable. In order to efficiently implement persistence by reachability, PSI introduces the concept of *descriptors*. PSI associates each object with a (hidden) descriptor field. The descriptor field points to an object that encodes the location of references (pointers) within the object.

PSI's approach to object typing contrasts strongly with ODMG's ODL [Cattell and Barry 1997] and SHORE's SDL [Carey et al. 1994], which are object definition languages that attempt to provide a means for storing objects and their types in a language independent manner. PSI is language independent, but does not attempt to address the issue of *poly-lingual access* to objects. The need for PSI to be aware of types is limited to the requirement that it be able to efficiently and correctly garbage collect the store—a need that is adequately met by the identification of references and is-one-of relationships as provided by the descriptor model.

[3]It is hard to *prove* completeness, however the literature *suggests* the completeness of the range of scenarios covered by the abstractions.

Residency Checks and Write Detection Object faulting is usually achieved through some sort of *residency check* at each object access. Similarly, some form of *write detection* is typically used to identify updated objects for writing back to stable store. The spectrum of approaches to residency checking and write detection include explicit checks by an interpreter at the time of word access or update (Napier88 [Munro 1993]) and use of page-grained hardware memory protection (the Texas persistent store [Wilson and Kakkad 1992]). The choice of the most appropriate mechanism involves tradeoffs which, for a particular client, are likely to be heavily context-dependent [Hosking 1995]. For this reason PSI relies on the client making *explicit* read and write requests, leaving the choice of detection mechanism to the client's implementer.

Swizzling The choice of an appropriate swizzling strategy is complex and influenced by application characteristics and PPL implementation approach (interpreted versus compiled PPL, for example). For this reason PSI does not *impose* a particular approach to swizzling on the client PPL, but instead provides hooks that facilitate swizzling—by allowing the client to identify references through descriptors, for example.

Concurrency Control and Cache Management The question as to which level the management of concurrency control should live within a persistent programming system is an important and difficult one. One approach is to give the upper layers levers in the form of transactional primitives within a flexible, extended transaction framework such as that formalized in ACTA [Chrysanthis and Ramamritham 1994]. The PSI interface is based on a rich abstraction of a transactional object cache which gives the client the levers necessary for the execution of a wide range of transaction models while leaving the store designer considerable implementation scope [Blackburn 1997].

Recovery There exist a wide variety of approaches to recovery. Most are compatible with the semantics of durability and failure in the context of basic ACID transactions and some support more sophisticated stability semantics such as intra-transaction stabilization and roll-back. The ACID notion of durability is included in the core PSI interface while intra-transaction stabilization and rollback form the basis for PSI's logging extension.

Advanced Storage Structures Some prospective PSI clients are likely to make use of extended storage structures such as index and collection types [Albano et al. 1995]. While these can be readily constructed on top of an object store (including PSI), the specialized nature of such data types has lead to the publication within the database community of considerable implementation optimizations with respect to concurrency control and storage management. In response to this, the implementation of index and collection types is the basis for one of PSI's extensions.

3.5 The interface

The PSI core interface is now defined in terms of the semantic framework outlined in section 3.1. In addition to the extra functionality of extensions for logging, extended transactions and advanced storage structures, the interface includes a number of housekeeping functions for opening and closing the store (including store recovery), setting the cache size etc. For the sake of brevity, only the core interface is described here (the interested reader is referred to [Blackburn 1997]). The transactional elements of the PSI interface are listed in table 4.

The modules that are not transactional in nature are illustrated in table 5. In addition there is a function, PSI_LIO, which gives asynchronous and list semantics to most of the transactional operations in much the same way as the POSIX lio (list-directed IO) interface [ISO/IEC and IEEE 1990] does for Unix file operations.

PSI_Read A copy of a specified object is forced into the cache. While PSI will not guarantee the object to be fresh, it will ensure that no transaction exposed to a stale object be allowed to commit (see section 3.1.2). The object will either be left in-place—in which case the client is returned a pointer to the object—or copied to a client-specified buffer, depending on the value of a boolean, copy, passed by the client. In terms of the semantics outlined in section 3.1, the read has no impact on stability but implements ReadIntention(t,o) with respect to visibility and if the read is in-place, Fix(t,o) with respect to cache management.

Core	Logging	Extended Trans.	Indexing
PSI_Read	PSI_Checkpoint	PSI_Delegate	PSI_Insert
PSI_Write	PSI_Rollback	PSI_IgnoreConflict	PSI_Fetch
PSI_New	PSI_ThisCheckpoint		PSI_Delete
PSI_NewTransaction	PSI_Stabilize		
PSI_Commit			
PSI_Abort			
PSI_Unfix			
PSI_Fix			

Table 4: The PSI transactional interface.

Extended OID	Housekeeping
PSI_GetSA	PSI_Init
PSI_GetOID	PSI_Open
	PSI_Close
	PSI_Recover

Table 5: PSI non-transactional calls.

PSI_Write The client's intention to update an (existing) object is asserted. The reference passed by the client may be to the in-place version of the object or to a private copy. The call implements WriteIntention(t,o). In the case where the object reference is in-place it also asserts Fix(t,o) semantics. ReadIntention(t,o) semantics are not asserted—a client would therefore typically call PSI_Read before PSI_Write (although this is not necessary if the client is not concerned with the object's prior state).

PSI_New Space is allocated space for a new object or a new array object, the call returning a cache pointer and an OID (see [Blackburn 1997] for a description of the PSI storage model). The nearOID parameter allows the caller to nominate an object as a placement hint (predefined constants allow users to nominate other sorts of hints, for example NEAR_ANY). PSI_New takes a descriptorOID argument which allows the user to define *is-one-of* relationships and to identify the structure of the new object. In addition to the allocation of space, PSI_New implements WriteIntention(t,o) and Fix(t,o) semantics.

PSI_NewTransaction The data structures associated with a new transaction are created and a transaction handle is initialized with respect to that transaction. The caller may provide a callback for handling pre-emptive aborts (see section 3.1.2). BeginUpdates(t) and BeginVisibility(t) semantics are asserted.

PSI_Commit The commit process is two phase. The first phase terminates the transaction by first asserting DelegateUpdates and DelegateVisibility with respect to any delegations flagged for commit time, and then asserting Unfix(t,o), ReadComplete(t,o), WriteComplete(t,o) and NotifyUpdate(t,o) with respect to all objects accessed by the transaction. The second phase is conditional on Isolate(t) and Finalize(t) being true. If they are true MakeDurable(t) and then Expose(t) semantics are asserted. Otherwise PSI_Commit returns failure, and the transaction is aborted (AbortUpdates(t) and AbortVisibility(t) are asserted). ACI transactions can be constructed by delegating all updates prior to commit. Although PSI_Commit is described here in terms of a series of steps, its implementation must be atomic.

PSI_Abort All resources associated with the transaction are released. Unfix(t,o) is asserted with respect to all objects accessed by the transaction and then AbortUpdates(t) and AbortVisibility(t) are asserted.

PSI_Unfix The availability of the specified object is removed by asserting Unfix(t,o) (the object remains unavailable until the need for it is re-asserted via PSI_Fix, PSI_Read, or PSI_Write). If the object was being updated in-place (i.e. PSI_Write was asserted with respect to an in-place version of the same object), NotifyUpdate(t,o) semantics are asserted with respect to the object.

PSI_Fix The need for an object to be made available is asserted through Fix(t,o). This call is only valid in the context of read or write intentions for that object already being asserted but not completed (PSI_Unfix must have been called subsequent to the PSI_Read, or PSI_Write in order to make the object unavailable).

4 Applying PSI to Persistent Java Implementations

Having defined the PSI core, we now investigate the application of PSI to persistent Java implementations, using the PJama$_0$ architecture [Atkinson et al. 1996] as a reference point[4].

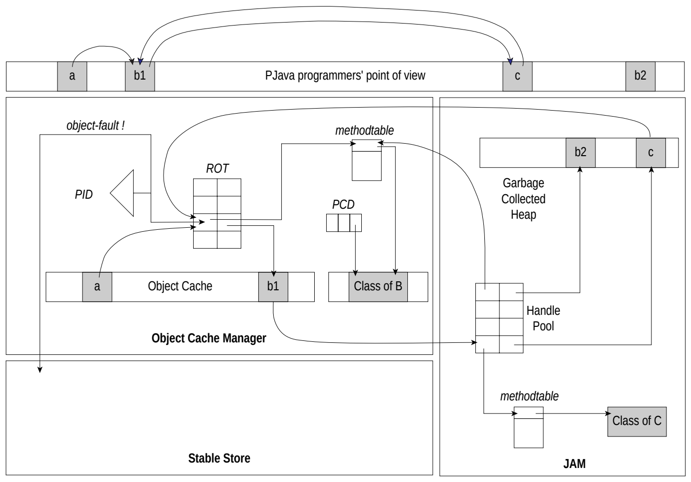


Figure 3: PJama$_0$ architecture [Atkinson et al. 1996], illustrating JAM, Object Cache Manager, and Stable Store modules. The details of the PJama$_0$ Stable Store implementation have been hidden. (Figure adapted from figure 2 in [Atkinson et al. 1996].)

The PJama$_0$ architecture (figure 3) is comprises the three key modules: the Java Abstract Machine (JAM), an Object Cache Manager, and a Stable Store. A key feature of the PJama$_0$ architecture is the minimal extent to which the JAM is disturbed [Atkinson et al. 1996]. The persistence mechanisms in PJama$_0$ are thus concentrated in the the Object Cache Manager and the Stable Store, the Object Cache Manager faulting objects from Stable Store and requesting stabilization of data by the Stable Store when necessary. The Stable Store is implemented on top of RVM [Satyanarayanan et al. 1994], a segment-based transactional storage system.

4.1 Implementation Approaches

By including an object cache manager as a central component, the PJama$_0$ architecture lends itself to a PSI-based implementation. Minimally, PSI could be used in place of the Stable Store. A more tightly coupled approach might see the object cache manager operating over PSI's cache rather than copying objects into its own cache. Given the amenability of the PJama$_0$ architecture to a PSI implementation, the dominant PSI/PJama design issue is therefore likely to be whether a single-level or two-level (as in PJama$_0$), buffering strategy should be employed.

Copying Versus Non-copying o maximize data transfer efficiency, caching stores usually move data in and out of the cache at as coarse a grain as possible. In the case of an object cache, this can result in many objects needlessly being brought into the cache at each object fault. Object clustering can help but it cannot *guarantee* good results because of the stochastic nature of the problem. Another approach is to re-pack objects in a second level cache. This can increase the efficiency of memory use significantly, but incurs the overhead of a memory-to-memory copy for each object fault, which, in the face of memory bandwidth bottlenecks becoming a dominant feature of modern processor architectures, is an increasingly unattractive option. The implementer is thus faced with a time-space tradeoff: either optimize for

[4]We chose PJama$_0$ because to our knowledge it is the only orthogonally persistent Java with a detailed published account of its architecture.

time by avoiding copies, or optimize for space by introducing a packed cache. The approach taken in the design of $PJama_0$ is to optimize for space [Atkinson et al. 1996]. A third alternative is to adopt a hybrid approach [Kemper and Kossmann 1994] which involves adaptively switching between the two polices as making each of the respective tradeoffs becomes more essential.

The results presented in [Kemper and Kossmann 1994] suggest that a hybrid approach is likely to be optimal. Such an approach could either be implemented within PSI or on top of PSI. Experimentation is necessary to determine which of these schemes will perform best.

4.2 Stable Store Alternatives

Having pointed to the suitability of PSI as the basis for persistent Java construction, we will now briefly assess alternatives.

4.2.1 RVM

Initial performance results for $PJama_0$ [Jordan 1996] suggest that RVM performs reasonably well. However, the stated design goals for RVM [Satyanarayanan et al. 1994], which include simplicity and portability of the RVM implementation suggest that a more targeted storage system, such as PSI, is likely to perform better and give the PJama implementer more flexibility.

The simplicity of the RVM design comes at the cost of efficiency, of loading the client with implementation responsibilities, and of reduced functionality such as resilience to media failure [Satyanarayanan et al. 1994]. The client implementer is left to take care of important elements of the transactional storage system such as *distribution*, *nesting*, and *serializability*. RVM's authors state that these areas of functionality were excluded in order to provide clients with flexibility with respect to the implementation choices associated with each [Satyanarayanan et al. 1994]. In terms of the semantic framework of section 3.1, RVM's transactional semantics are very weak. It supports only basic *stability* and *cache management* semantics (no support for logging or delegation) and makes no guarantees about *visibility* semantics other than those that are implicit in stability semantics (commit stabilization followed by access by a later transaction).

By contrast, PSI provides a simple yet rich abstraction of a transactional object cache which gives the *client implementer* flexibility through a collection of powerful levers by way of fully implemented transactional mechanisms. Furthermore, *PSI implementors* are given a great deal of scope to explore different implementation strategies, so to the extent that implementations of these are made available to client implementers, client implementers will have a wide choice of storage implementation approaches available to them.

While care was taken in the RVM design to maximize the portability of the RVM *implementation*, the design philosophy for PSI is closer to that of MPI (a widely used message passing interface [Message Passing Interface Forum 1994]), where the emphasis is not on the interface *implementation* being portable, but rather the interface efficiently *providing portability* to its clients by appropriately abstracting complex, vendor specific message-passing mechanisms. In the case of MPI, such efficiency is usually obtained precisely by exploiting non-portable hardware features on each target platform. It seems likely that stable store implementations will need to do the same in order to deliver performance comparable with that offered by commercial database vendors.[5]

4.3 Other Approaches

A range of alternative implementation approaches exist. The two most obvious being an integrated Stable Store / RTS implementation and the use of another storage layer like RVM or PSI.

Integrated Implementation Approaches The "integrated" implementation approach has been used by many, if not most orthogonally persistent programming systems to date. By "integrated", we mean that the design does not attempt to strongly separate database and programming language technologies. The absence of any such implementations that efficiently offer a full range of database features (as outlined in [Atkinson and Morrison 1995]) gives support to the view that the approach is ultimately inappropriate in the context of a small research community such as the persistent systems community.

[5]For example some operating systems offer non-portable control over memory management that can greatly improve caching performance and reduce TLB misses. Also, in the context of distributed stores, optimal communication mechanisms vary from platform to platform (e.g. TCP / IP versus MPI or native message passing).

Alternative Storage Layers Mneme [Moss 1990] is perhaps the most natural candidate as an alternative storage layer. Like PSI, Mneme is based on the transactional object cache architecture and so is well suited to the PJama architecture. However, Mneme is distinguished from PSI in two key respects: First, Mneme does not directly support extended transaction models, which are a feature of the PJama design. Mneme leaves the implementation of extended transaction models to higher levels of abstraction, only directly supporting simple transactions. Secondly, Mneme is a store implementation rather than an interface definition, so it does not offer the same opportunities for collaboration between store implementers and PJama builders as PSI.

There are few other purpose-designed stand-alone transactional storage layers. Carey and DeWitt's review [Carey and DeWitt 1996] lists a number of key examples of database system toolkit projects, some of which come close to the transactional storage layer approach. Carey et al. illustrate problems with the database toolkit approach by reporting a number of the problems they and other users encountered with EXODUS [Carey and DeWitt 1996; Carey et al. 1994]. These include: users wanting to use EXODUS to build an object *server* and being stuck with a client-server architecture, their server thus becoming a server-on-a-client; control over low level details being hidden from 'serious' implementers (*too high* an abstraction); and application programmers finding it a bit too low level (*too low* an abstraction).

All of the database toolkit projects differ from PSI in a number of ways, perhaps most important of those being that PSI is not a *system implementation* but an *interface specification* based on a rich storage abstraction. A system based on PSI is therefore not limited to a single storage implementation approach or architecture. To the contrary, it may be used as the platform for experimentation with a range of approaches to storage management.

5 Conclusions

We have presented PSI, an interface based on an abstraction of the transactional object cache architecture. We argue that the use of such an interface will play an important role in overcoming the problem of mastering the breadth of technology involved in the construction of orthogonally persistent systems by separating concerns and so allowing a concentration of expertise. PSI should also encourage portability of persistent systems and enhance opportunities for collaboration. Finally, we have shown how PSI might be integrated into a persistent Java implementation, using PJama$_0$ as an example.

Bibliography

ADVE, S. V. AND GHARACHORLOO, K. 1995. Shared memory consistency models: A tutorial. Technical Report WRL Research Report 95/7 (Sept.), Digital Equipment Corporation, Palo Alto, CA, U.S.A.

ALBANO, A., BERGAMINI, R., GHELLI, G., AND ORSINI, R. 1995. An introduction to the database programming language Fibonacci. *The VLDB Journal 4*, 3 (July), 403–444.

ATKINSON, M. P., JORDAN, M. J., DAYNÈS, L., AND SPENCE, S. 1996. Design issues for Persistent Java: A type-safe, object-oriented, orthogonally persistent system. In R. CONNOR AND S. NETTLES Eds., *Seventh International Workshop on Persistent Object Systems* (Cape May, NJ, U.S.A., May 1996), pp. 33–47. Morgan Kaufmann.

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent systems. *The VLDB Journal 4*, 3 (July), 319–402.

BLACKBURN, S. M. 1997. *High Level Storage Abstractions: A foundation for scalable persistent system design*. PhD thesis, Australian National University, Canberra, Australia. Available online at `http://cs.anu.edu.au/~Steve.Blackburn/`.

BLACKBURN, S. M., SCHEUERL, S. J. G., STANTON, R. B., AND JOHNSON, C. W. 1997. Recovery and page coherency for a scalable multicomputer object store. In H. EL-REWINI AND Y. N. PATT Eds., *30th Hawaii International Conference on System Sciences* (Hawaii, U.S.A., Jan. 7–10 1997), pp. 523–532.

BLACKBURN, S. M. AND STANTON, R. B. 1996. Multicomputer object stores: The Multicomputer Texas experiment. In R. CONNOR AND S. NETTLES Eds., *Seventh International Workshop on Persistent Object Systems* (Cape May, NJ, U.S.A., May 29–31 1996), pp. 250–262. Morgan Kaufmann.

CAREY, M. J. AND DEWITT, D. J. 1986. The architecture of the EXODUS extensible DBMS. In *Proceedings of the First International Workshop on Object-Oriented Database Systems* (Pacific Grove, CA, U.S.A., Sept. 1986), pp. 52–65. IEEE.

CAREY, M. J. AND DEWITT, D. J. 1996. Of objects and databases: A decade of turmoil. In T. M. VIJAYARAMAN, A. P. BUCHMANN, C. MOHAN, AND N. L. SARDA Eds., *VLDB'96, Proceedings of the 22th International Conference on Very Large Data Bases* (Mumbai (Bombay), India, Sept. 3–6 1996), pp. 3–14. Morgan Kaufmann.

CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M. L., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O. G., AND WHITE, S. J. 1994. Shoring up persistent applications. In R. T. SNODGRASS AND M. WINSLETT Eds., *Proceedings on the 1994 ACM-SIGMOD Conference on the Management of Data*, Volume 23 of *SIGMOD Record* (Minneapolis, MN, U.S.A., May 24–27 1994), pp. 383–394. ACM.

CAREY, M. J., FRANKLIN, M. J., AND ZAHARIOUDAKIS, M. 1994. Fine-grained sharing in a page server OODBMS. In R. T. SNODGRASS AND M. WINSLETT Eds., *Proceedings of the 1994 ACM-SIGMOD International Conference on the Management of Data*, Volume 23 of *SIGMOD Record* (Minneapolis, MN, U.S.A., May 24–27 1994), pp. 359–370. ACM.

CATTELL, R. G. G. AND BARRY, D. K. Eds. 1997. *The Object Database Standard: ODMG 2.0*. Morgan Kaufman.

CHRYSANTHIS, P. AND RAMAMRITHAM, K. 1994. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems 19*, 3 (Sept.), 450–491.

FRANKLIN, M. J. 1996. *Client Data Caching: A Foundation for High Performance Object Database Systems*, Volume 354 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, MA, U.S.A. This book is an updated and extended version of Franklin's PhD thesis.

FRANKLIN, M. J., CAREY, M. J., AND LIVNY, M. 1997. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems 22*, 3 (Sept.). Accepted for publication: see `http://www.acm.org/tods/`.

HOSKING, A. L. 1995. Benchmarking persistent programming languages: Quantifying the language / database interface. In *OOPSLA'95 Workshop on Object Database Behavior, Benchmarks and Performance* (Austin, TX, U.S.A., Oct. 15–19 1995).

ISO / IEC AND IEEE. 1990. *ISO/IEC 9945-1, IEEE Std 1003.1 Information technology — Portable Operating System Interface (POSIX)*.

JORDAN, M. 1996. Early experiences with Persistent Java. In M. JORDAN AND M. ATKINSON Eds., *First International Workshop on Persistence and Java* (Drymen, Scotland, Sept. 16–18 1996). Available online at DB&LP server: `http://www.informatik.uni-trier.de/~ley/db/`.

KEMPER, A. AND KOSSMANN, D. 1994. Dual-buffering strategies in object bases. In J. B. BOCCA, M. JARKE, AND C. ZANIOLO Eds., *VLDB'94, Proceedings of the 20th International Conference on Very Large Data Bases* (Santiago de Chile, Chile, Sept. 12–15 1994), pp. 427–438. Morgan Kaufmann.

MESSAGE PASSING INTERFACE FORUM. 1994. MPI: A message-passing interface standard. In *International Journal of Supercomputing Applications*, Volume 8 (Nov. 1994). Also available as University of Tennessee Technical Report CS-94-230.

MOSS, J. E. B. 1990. Design of the Mneme persistent object store. *ACM Transactions on Information Systems 8*, 2 (April), 103–139.

MUNRO, D. S. 1993. *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, University of St Andrews, St Andrews, Scotland. Available online at: `http://www-ppg.dcs.st-andrews.ac.uk/Publications/`.

SATYANARAYANAN, M., MASHBURN, H. H., KUMAR, P., STEERE, D. C., AND KISTLER, J. J. 1994. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems 12*, 1 (Feb.), 33–57. Important corrigendum appears in *ACM Transactions on Computer Systems 12*, 2.

WILSON, P. R. AND KAKKAD, S. V. 1992. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *1992 International Workshop on Object Orientation in Operating Systems* (Dourdan, France, 1992), pp. 364–377. IEEE: IEEE Computer Society Press.