The Transactional Object Cache: A foundation for high performance persistent system construction.

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

Abstract

This paper argues that caching, atomicity and layering are fundamental to persistent systems, and that the transactional object cache architecture, as an embodiment of these concerns, provides a foundation for high performance persistent system construction. Central to the paper is a description of the semantics of an abstract transactional object cache architecture that supports a wide range of transactional models and is open to a broad spectrum of transactional cache coherency algorithms. The centrality of the abstraction is a consequence of its role in facilitating the definition of a transactional interface, which is the key to the practical application of the transactional object cache architecture. The utility of the architectural framework in general, and the interface in particular, is argued for in the context of existing systems and systems currently under construction within that framework.

1 Introduction

If orthogonal persistence is to have a future beyond academe, its merits will have to be demonstrated to the commercial world in a performance context. It is our contention that the transactional object cache provides implementers of orthogonally persistent systems with an architectural framework which will expedite the process of developing high performance orthogonally persistent systems. Our argument rests on the role of *caching* and *atomicity* in addressing the interrelated engineering challenges of concurrency, replication, coherency, latency, and stability, and on the role of *layering* in aiding implementation through abstraction and the promotion of reuse (and with it, collaboration).

While there exist well-known examples of stores that, to varying degrees, exhibit the transactional object cache architecture (e.g. EXODUS [Carey and DeWitt 1986], Mneme [Moss 1990]), the novelty of this paper lies in the identification of a generalized architecture as a basis for the development of high performance persistent systems, and in the pinning down of its semantics.

The paper begins with an identification of the roles of atomicity, caching, and layering, and their coming together in the form of the transactional object cache (TOC) architecture. This is followed by the core of the paper—a detailed definition of the architecture through a semantic framework. The utility of the transactional object cache architecture as an approach to store construction is then discussed in the context of PSI, a transactional interface based on the architecture and defined in terms of the semantic framework [Blackburn 1997]. The discussion of PSI includes references to PSI-based systems in existence and under development and results which lend support to our contention that the TOC architecture will expedite the process of developing high performance orthogonally persistent systems.

2 The Transactional Object Cache

2.1 Fundamentals of High Performance Persistent Systems

Among the challenges of engineering high performance persistence systems, the concerns of concurrency, replication, coherency, latency, and stability appear to be dominant. While there are many approaches to addressing each of these, we claim that the devices of *atomicity* and *caching* are basic, and further, that *layering* is fundamental in aiding implementation through abstraction and the promotion of reuse and collaboration.

*The authors wish to acknowledge that this work was carried out within the Cooperative Research Centre for Advanced Computational Systems established under the Australian Government's Cooperative Research Centres Program.

Atomicity Atomicity has a central place in computing as the basic property of operations which move machines between well defined states in the presence of concurrent operations and shared data. The powerful "all or nothing" behavior of an atomic operation provides a guarantee that all changes to variables made in the associated execution processes will have occurred at the termination of the operation, or that none of them will. When combined with visibility restrictions between concurrent operations, atomicity provides isolation and well defined points for determining coherency conditions.

Transaction systems address coherency in the face of concurrency and replication by using atomicity to give isolation and serializability to concurrent computations. In addition, they implement failure resilience mechanisms by adding coherency to stability through well defined recovery points that arise from atomicity. A side effect of atomicity is the shifting of temporal grain which, by inducing temporal clustering of communications and so amortizing latency costs, can combat latency in distributed systems.

Caching Caching is fundamental to modern computing systems because of its role as a *latency hider* and *data replicator*. The role of the cache as a hider of latencies such as those induced by networks and disks is well understood and the challenge of maintaining coherence in the face of data replication in such contexts has been widely studied. Caching also has a role as a means of facilitating controlled incoherence through data replication in optimistic systems including some transactional systems and many AI languages.

The cache is a central part of most persistent architectures, either explicitly—such as the buffer cache in most database systems, or implicitly—as in 'workspaces', which are prominent in optimistic systems such as Flask/DataSafe [Scheuerl et al. 1996]. The cache is also central to distributed persistent systems such as client server databases, where the role of the cache extends to hiding network latencies [Franklin et al. 1997].

Layered Software In a layered view of software systems, bottom layers are specific to platform architectures while the top layers are platform independent problem solutions. The layered model then represents steps in the abstraction away from architectural features such as those associated with potential speed of computation. Abstractions over concerns such as distribution, recoverability, and coherence can greatly simplify design and implementation. Such abstraction also serves as a means of separating concerns and so allows a focusing of research and design activities.

The use of layering is widespread in architectures for persistent systems, both with an end to making the implementation of complex systems tractable, and also as means of promoting software reuse [Munro et al. 1994; Carey et al. 1994]. In the context of expediting the construction of high performance persistent systems, layering will aid system design and implementation through abstraction, and will help bridge the existing programming language/storage technology divide by providing an interface through which the separate concerns may be brought together.

2.2 The Transactional Object Cache Architecture

Having identified the roles of atomicity, caching, and layered software in persistent systems construction, we now describe the transactional object cache architecture, which explicitly embodies these concerns. Central to the architecture is the *cache*, to which the underlying store, the language run-time and the application may have direct access. A *transactional interface* mediates the movement of data in and out of the cache, giving the language run-time (and through it the application), transactional guarantees of atomicity, isolation, coherency and durability with respect to its accesses to the cache. The *layering* of storage and programming language concerns is explicit in the architecture.

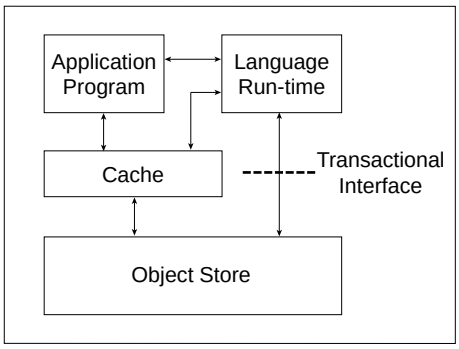

Figure 1: The transactional object cache architecture. The architecture of the object store is transparent to the application.

The explicit acknowledgement of the centrality of the cache and the provision for mediated direct access to it contrasts with other architectures for persistent systems [Matthes et al. 1996; Munro et al. 1994] which provide the abstraction of a 'persistent heap', implemented with a relatively expensive procedure call interface for data access. This aspect of the transactional object cache architecture makes it far more conducive to high performance implementations than such alternatives.

By providing an abstraction over the object store through layering, the architecture is transparent to distribution of the underlying store, and coupled with the concurrency control delivered by the transactional interface, facilitates the implementation of multi-user client-server or highly scalable multiprocessor implementations independently of the language run time or application [Blackburn and Stanton 1997]. This potential for transparently scalable store implementations reinforces the applicability of the architecture to a high performance context.

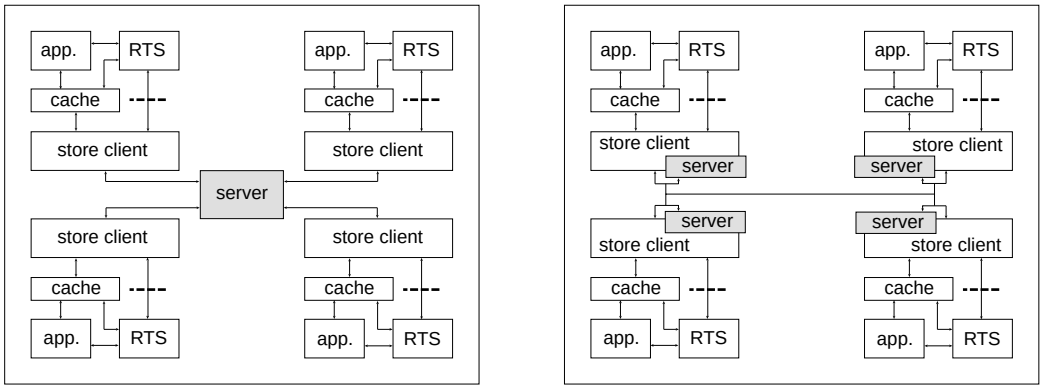

Figure 2: Client server (left) and client peer (right) realizations of the underlying object store. The abstraction over store architecture provided by the transactional object cache architecture gives distribution transparency to the client application and run time system.

3 An Abstraction of the Transactional Object Cache Architecture

The cornerstone of the transactional object cache architecture is the interface which separates storage and programming language concerns, and through which all cache operations are mediated. The effective realization of the architecture therefore depends on the existence of a well defined interface. The interface definition must capture the separation of storage and programming language concerns in a manner that does not rob the persistent programming language implementer of efficiency or functionality, nor deny the store implementer of flexibility in approach.

Central to the design of such an interface is a clear understanding of the architecture. The definition of a clear abstraction of the transactional object cache architecture thus forms the core of this paper. The abstraction is based on, and is a refinement of, the abstraction presented in [Blackburn 1997].

3.1 Architectural Elements

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 identification of the precise semantics of these operations with respect to the various areas of store management 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.

The transactional object cache can be viewed in terms of transactions operating over an abstract store consisting of a collection of 'deltas', where each delta has associated with it an object image that corresponds to some update to the store. A transaction's (potentially invalid) 'view' of the store can then be defined as some projection of the deltas. From the perspective of a transaction operating over such a store, the state of a delta (or set of deltas) can be modeled in terms of three concerns:

- stability,
- visibility,
- and availability.

The concept of stability allows reasoning about the *permanency* of any deltas that might be contributed by a transaction to the store. Visibility provides a means of determining the *validity* of a particular projection

of deltas (according to some notion of correctness). Availability models the *presence* in a transaction's cache of object images associated with the deltas forming a transaction's projection of the store.

By developing abstract interfaces with respect to each of these concerns, a transaction's interaction with the store can then be precisely modeled in terms of the three concerns. The remainder of this section will focus on the development of those abstract interfaces. Having developed the models and abstract interfaces to them, there will be a discussion on how the three concerns come together to give a complete understanding of a transaction's interaction with the store.

3.1.1 Core and Extended Functionality

In the following sections, each of the above three areas of concern are described and abstract interfaces with respect to them are identified. The approach taken is fundamentally influenced by the objective of this task of abstraction—providing a basis for the subsequent development of a rich and yet flexible (concrete) transactional interface. In order to maintain a simple kernel while at the same time offering richness and generality in any subsequently developed concrete interface, the following areas of transactional functionality have been identified and will guide the development of each of the abstract interfaces:

Core This functionality is fundamental and should be supported by all transactional storage implementations.

Logging Facilities such as intra-transaction optimistic computation and intra-transaction stability are desirable, but are usually only supported by logging stores.

Extended Transactions Through extended transactions some of the restrictions of simple ACID transactions are broken, leading to a wide range of possible transactional semantics including chaining and nesting.

3.2 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 (an ordered set), h of atomic events, e_i:

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

where '.' is used to denote the append operation with respect to both histories and events thus: $e.e' = \{e, e'\}$; $h.e = \{e_1, \ldots, e_n\}.e = \{e_1, \ldots, e_n, e\}$; and $h.h' = \{e_1, \ldots, e_n\}.\{e'_1, \ldots, e'_m\} = \{e_1, \ldots, e_n, e'_1, \ldots, e'_m\}$.

By identifying a durable global stability history, φ, and a set of non-durable local stability histories, $\Phi = \{\langle 0, \phi_0\rangle, \langle 1, \phi_1\rangle, \ldots, \langle n, \phi_n\rangle\}$, 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 ϕ_t (where $\langle t, \phi_t\rangle \in \Phi$). In this model, a transaction is made durable via an operation which appends ϕ_t to φ (i.e. $\varphi' = \varphi.\phi_t$, where φ and φ' denote before and after values of φ respectively). Three types of events, e, are defined: $e \in \{\delta, s, m\}$, where δ denotes state changing events, s denotes stability events, and m denotes marker events. These event types allow changes in store state, intra-transactional stability semantics, and checkpoint/rollback semantics to be modeled 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. In keeping with the notion of core and extended functionality, the most restrictive set of primitives (those needed to support basic ACID transactions) is described first, with subsequent primitives adding generality. The complete set of well-defined primitives are sufficient to fully describe the stability semantics of a transactional object cache.

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

$$\phi'_t = \phi_t.e$$

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

Table 1: Stability primitives.

should be read as shorthand for:

$$\Phi' = \{\langle t_i, \phi_{t_i}\rangle \in \Phi | t_i \neq t\} \cup \{\langle t, \phi_t.e\rangle | \langle t, \phi_t\rangle \in \Phi\}$$

3.2.1 Stability and Core Functionality

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) $(\phi_t = \{\}) \ \wedge \ (\Phi' = \Phi \cup \{\langle t, \phi_t\rangle\})$

NotifyUpdate(t,o) $\phi'_t = \phi_t.\delta^o$, where δ^o is an event describing a change of state to some object, o.

AbortUpdates(t) $\Phi' = \{\langle t_i, \phi_{t_i}\rangle \in \Phi | t_i \neq t\}$

MakeDurable(t) $(\varphi' = \varphi.\phi_t) \ \wedge \ (\Phi' = \{\langle t_i, \phi_{t_i}\rangle \in \Phi | t_i \neq t\})$

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 $\Phi' = \{\langle t, \phi_{t_S}.s\rangle | (\exists \langle t, \phi_{t_S}.s.\phi_{t_E}\rangle \in \Phi | s \notin \phi_{t_E})\}$

In the absence of a primitive for adding stability events, s, to local histories, *all* local histories will be deleted by the EvictVolatile primitive, leaving only the global durable history, φ, intact.

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

3.2.2 Stability and Logging

Logging adds the capacity to roll back to an identifiable point in the transaction, undoing the effect of all NotifyUpdates that occurred after that point in the transaction's history. The addition of intra-transactional stability allows transaction histories to be made stable without making them atomically durable. Three primitives are needed to capture logging as part of stabilization. They are described here again with respect to a given transaction, t:

CheckpointUpdates(t) $\phi'_t = \phi_t.m_i$, where m_i denotes a uniquely labeled marker event.

RollbackUpdates(t,i) $\phi'_t.m_i.\phi_R = \phi_t$. In other words, the history ϕ_R comprising the events after m_i is rolled back.

StabilizeUpdates(t) $\phi'_t = \phi_t.s$, where s denotes a stabilize event.

StabilizeUpdates will give an otherwise volatile transaction history stability with respect to EvictVolatile events (see definition of EvictVolatile above).

3.2.3 Stability and Extended Transaction Models

Chrysanthis and Ramamritham [1994] have shown *delegation* to be a powerful facility that can be used as basis for extended transaction models. Delegation refers to the delegation of responsibility for the stability of some operation/s from one transaction to another. Delegation can be thought of as the moving of some set of events from one transaction's history to that of another.

The concept of delegation is fundamental to a number of families of extended transaction models including nested transactions [Moss 1981], joint transactions [Pu et al. 1988], and split transactions [Pu et al. 1988]. For this reason the following call with respect to two transactions t_i and t_j and some object o is added:

DelegateUpdates(t_i,t_j,o) $(\phi'_{t_i} = \phi_{t_i} \setminus \phi^o_{t_i}) \ \wedge \ (\phi'_{t_j} = \phi_{t_j} . \phi^o_{t_i})$, where $\phi^o_{t_i}$ is a sub-history of ϕ_{t_i} consisting of all events δ^o relating to a change in state of o.

The efficient implementation of delegation in the context of a log-based recovery system is addressed in [Martin and Ramamritham 1997].

3.3 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 exposed to 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. 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. Transactional 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 distinct from 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 in section 3.5.

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

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

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

$$(e \in \psi) \Rightarrow (\exists t_i \in T | (e \in \psi_{t_i} | (\forall t_j \in T | (i \neq j)) | e \notin \psi_{t_j}))$$

The notion of irrevocability, which is central to modeling transactions, is introduced by defining ψ_I, a sub-history of ψ whose members are *irrevocably* part of ψ. The property of immutability can be used to capture the notion of transaction commit—the commit of a transaction corresponds to the movement of events from ψ_t to ψ_I. 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. Three key ideas are introduced to this end: *temporal breadth* of activities, the notion of object *versions*, and the concept of *workspaces*.

By giving *temporal breadth* to read and write events we reflect the reality of cached access to data—an object is made available to a transaction for reading or writing over a period of time. A visibility model must therefore be constructed in terms of read and write events with well defined temporal breadth. To this end we introduce read and write intention and completion events: r, $\overline{r}$, w, and $\overline{w}$.

The notion of *object versions* is used to capture the reality that every valid read to an object must be with respect to some prior write to that object. In a coherent store and in the absence of replication, one might expect a read to always be with respect to the object state as determined by the most recent write to the object. However, in the context of revocability, replication and the possibility of incoherent views of the store, the constraint may be weaker. To accommodate the notion of versions, we adopt the convention of associating a unique version with each write completion, and associating with each read intention a version that reflects the write seen by that read: (with respect to an object, o) $\overline{w}_{o_v}$, r_{o_v}, where r_{o_v} is a read that saw an image of o as defined by the write completion event $\overline{w}_{o_v}$.

The concept of *workspaces* is introduced to reflect the fact that in a cached store, overlapping reads and writes with respect to a common object may be with respect to either distinct or common instances of that object. In the case where they are with respect to distinct instances, concurrent writes will not be

visible to the reader. However, in the case where they are with respect to the same instance, write activity is implicitly visible to the reader. A trivial example of that latter is the case where overlapping reads and writes occur within a single transaction. More complex examples arise when advanced transaction models are used or when transactions share caches for space efficiency reasons. In order to reflect this possibility in the visibility model, we associate a workspace identifier, w, with read and write events when the distinction is necessary: $r_{o_w}, \bar{r}_{o_w}, w_{o_w}, \bar{w}_{o_w}$.

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 $T(\psi_i)$ which tests termination on all reads and writes within a sub-history ψ_i (the notation $a \prec b$ is used to denote a preceding b in ψ):

$$T(\psi_i) \quad = \quad (\forall r_o \in \psi_i | (\exists \bar{r}_o \in \psi_i | (r_o \prec \bar{r}_o))) \ \wedge \ (\forall w_o \in \psi_i | (\exists \bar{w}_o \in \psi_i | (w_o \prec \bar{w}_o)))$$

In addition, a workspace isolation function, $W(\psi_i, \psi_j)$, is defined to be true only if read events in ψ_i do not conflict with any write events in ψ_j with respect to the same object, o, when sharing a common workspace, w:

$$W(\psi_i, \psi_j) \quad = \quad \forall r_{o_w}, \bar{r}_{o_w} \in \psi_i, | (\nexists w_{o_w}, \bar{w}_{o_w} \in \psi_j | (r_{o_w} \prec \bar{w}_{o_w}) \wedge (w_{o_w} \prec \bar{r}_{o_w}))$$

Finally, a serializability function $S(\psi_i, \psi_j, \psi_k)$ is defined such that $S(\psi_i, \psi_j, \psi_k)$ is true only if the store image as seen by ψ_i is consistent (serializable) with respect to ψ_j, where ψ_k denotes a sub-history of all events with which conflicts are ignored[1]:

$$S(\psi_i, \psi_j, \psi_k) \quad = \quad \forall r_{o_v} \in \psi_i | (\exists \bar{w}_{o_v} \in (\psi_i \cup \psi_j \cup \psi_k) | (\nexists \bar{w}_{o_{v'}} \in \psi_j | (\bar{w}_{o_v} \prec \bar{w}_{o_{v'}})))$$

In other words, all reads in transaction t_i were with respect to writes within either t_i (ψ_i) or t_j (ψ_j), or were writes with respect to which t_i was ignoring conflicts (ψ_k). Furthermore, all reads were with respect to writes that did not occur before the most recent write in ψ_j to the object concerned (o). Typically S is used such that $\psi_j = \psi_I$ (the immutable set).

With a visibility model and 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.

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

Table 2: Visibility primitives.

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

- Appending an event to a sub-history implies appending the event to ψ: $(\psi_t{}' = \psi_t.e_i) \Rightarrow (\psi' = \psi.e_i)$.
- Truncating a sub-history implies removal of events from ψ: $(\psi_t{}'.e_i = \psi_t) \Rightarrow (\psi' = \psi \setminus e_i)$, where $\setminus$ denotes history difference.
- The operation $\psi_i \cup \psi_j$ denotes the order-preserving merging (union) of two sub-histories.

Furthermore, by definition any manipulation of the immutable sub-history, ψ_I is not permitted.

[1]Conflict ignorance is important for some advanced transaction models. See section 3.3.3.

3.3.1 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) $(\psi_t = \{\}) \ \wedge \ (T' = T \cup \{t\})$

ReadIntention(t,o) ${\psi_t}' = \psi_t.r_{o_v}$

ReadComplete(t,o) ${\psi_t}' = \psi_t.\overline{r}_o$

WriteIntention(t,o) ${\psi_t}' = \psi_t.w_o$

WriteComplete(t,o) ${\psi_t}' = \psi_t.\overline{w}_{o_v}$

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

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

Finalize(t) $(T(\psi_t) \wedge S(\psi_t, \psi_I, \psi_{ic_t}) \wedge W(\psi_t, \psi_w))$, where ψ_{ic_t} is the sub-history of ψ consisting of all events with which t is ignoring conflicts, and $\psi_w = \psi \setminus (\psi_I \cup \psi_t \cup \psi_{ic_t})$ (i.e. all events in other unfinalized transactions except those events with which conflicts are being ignored).

Expose(t) $({\psi_I}' = \psi_I \cup \{\overline{w} \in \psi_t\}) \wedge (\psi' = \psi \setminus \{e \in \psi_t | e \neq \overline{w}\})$. Thus the write completion events of t become irrevocable and all other events in ψ_t are discarded.

3.3.2 Visibility and Logging

Rollback must be handled by the visibility mechanism. After a rollback, no evidence of any updates or reads that were rolled back should be visible. The following primitives are required to support rollback with respect to visibility:

CheckpointVisibility(t) ${\psi_t}' = \psi_t.m_i$

RollbackVisibility(t,i) ${\psi_t}'.m_i.\psi_R = \psi_t$

3.3.3 Visibility and Extended Transaction Models

Delegation of transactional responsibility for an operation impacts on visibility just as it does on stability (section 3.2.3). When operations are delegated, visibility of those operations is transferred to the target transaction (i.e. visibility events are removed from the sub-history of one transaction and become part of the sub-history of another transaction—their place in the history ψ is unchanged). In addition, some advanced transaction models allow controlled relaxation of isolation. The following primitives define advanced visibility semantics with respect to visibility operations on o between t_i and t_j:

DelegateVisibility(t_i,t_j,o) $(({\psi_{t_i}}' = \psi_{t_i} \setminus \psi_{t_{i_o}}) \wedge ({\psi_{t_j}}' = \psi_{t_j} \cup \psi_{t_{i_o}}))|(T(\psi_{t_{i_o}}) \wedge S(\psi_{t_{i_o}}, \psi_{t_j}, \psi_k)$, where $\psi_{t_{i_o}}$ is a sub-history of ψ_{t_i} consisting of all events e_o relating to o, and $\psi_k = (\psi \setminus (\psi_{t_{i_o}} \cup \psi_{t_j}))$.

IgnoreConflict(t_i,t_j,o) $({\psi_{ic_{t_i}}}' = \psi_{ic_{t_i}} \cup \psi_{t_{j_o}}) \ \wedge \ ({\psi_{ic_{t_j}}}' = \psi_{ic_{t_j}} \cup \psi_{t_{i_o}})$. Thus for all events relating to object o, t_i and t_j are added to each other's 'ignore conflict' sub-histories (ψ_{ic_t}). A subsequent call to Finalize will thus ignore conflicts between h_{t_i} and h_{t_j} for those events.

A meaningful implementation of IgnoreConflict would allow participating transactions to use the same workspace for accesses to o. This will present implementation challenges in the context of a distributed cache as some form of transparent, coherent distributed shared memory (DSM) would need to exist with respect to those transactions and the set of objects.

3.4 Availability

A third dimension of the transactional cache architecture is availability. A cached store design is motivated by the desire to hide IO latency and introduce replication through caching. While stability is concerned with what changes to store state are made durable, and visibility is concerned with the validity of the image 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.

Core
Fix
Unfix

Table 3: Availability primitives.

Availability can be modeled in terms of each active transaction, t, operating over a logically distinct cache c_t within which is 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.

Only two primitives are necessary for the description of an availability model:

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 access to a given object. The stability and validity of the available object images is a function of the stability and visibility control mechanisms respectively.

3.5 Integrating the models

The preceding sections have developed rich abstractions for each of the three previously identified aspects of a transaction's perspective of a delta or set of deltas composing some part of the store. From the perspective of a running transaction, the operational interactions of these three abstractions is fairly straight-forward.

The stability abstraction can be used to model the permanency of some set of deltas contributed by the running transaction. Assuming a conventional transactional framework, the correctness of the deltas must be assured (through the visibility abstraction) prior to those deltas being made durable (irrevocably stable). Under some transaction models, deltas may be made stable (but not irrevocable) independent of their 'correctness'. The availability abstraction relates to the stability abstraction as the means by which a transaction may materialize and operate on new deltas. The visibility and availability abstractions come together to materialize for the transaction a projection of the store which it can operate over, and determine the validity of. Unless coupled with the visibility model, the availability model can not materialize any meaningful view of the store.

Although the abstraction covers a wide range of transaction models, it seems likely that support for layered transactions and other open transaction models[Weikum and Schek 1992] will require some meta-level extensions to the abstraction. 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 may be trivially adapted to account for such.

4 Applying the Transactional Object Cache to Systems Construction

The utility of the transactional object cache architectural framework as a tool for expediting the construction of high performance persistent systems hinges on the existence and utilization of a well defined transactional interface. The abstraction of the architecture presented above paves the way for the definition of such an interface. PSI (persistent store interface) [Blackburn 1997] is a first attempt to develop a well-defined transactional interface for the architecture. The following sections report on the development and utilization of the PSI interface.

4.1 PSI—an Interface Definition

PSI is a software interface definition for the transactional object cache architecture. In designing the PSI interface, we sought to balance a number of objectives: to flexibly support the needs of persistent programming languages (PPLs); to strongly separate storage and programming language concerns; and to admit small and fast PPL implementations. This combination of objectives presented a constrained trade-off space for the interface design, within which PSI represents just one point. It is not the purpose of this paper to describe the PSI interface in detail nor to recount all of the various design decisions made. The interested reader is instead referred to [Blackburn 1997].

PSI was shaped by the abstraction presented in the preceding section and the semantics of each call were pinned down in terms of its semantic primitives. As with the abstraction, the interface is defined in terms of core primitives which give support for basic ACID transactions, and extensions for logging, extended transactions, and indexing. An appropriate separation of concerns between the object store and the PPL was central to the interface design. Key design choices include the following:

- PSI requires the store to provide a persistent graph of objects reachable from a single root (for efficiency reasons the store should be garbage collected);
- Residency checks and write detection are undertaken by the PPL;
- PSI includes a simple classing mechanism allowing store-side GC, but leaves swizzling to the PPL;
- PSI provides the PPL with primitives that will support a wide range of transactional models and leaves concurrency control implementation to the store;
- PSI does not prescribe an approach to recovery, however implementation of checkpoint and rollback (PSI's logging extensions) requires some form of logging.

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

Table 4: Key calls in the PSI interface.

4.2 Implementing PSI

The openness of the transactional object cache architecture to a range of transactional concurrency control and recovery schemes is an important attribute of PSI. With the exception of the logging and extended transaction primitives, which place strong demands on recovery and concurrency control, the PSI interface should be implementable in terms of most of the wide range of transactional recovery [Härder and Reuter 1983] and concurrency control [Franklin et al. 1997] schemes.

At the time of writing there exist two major PSI store implementations and two more implementations are underway. One implementation is relatively light-weight (does not support concurrent transactions on a single processor), uses a log-based recovery scheme and the AOCC cache coherency algorithm [Adya et al. 1995] in its multi-processor form. This implementation has been tested extensively and demonstrated to perform well and to be highly scalable [Blackburn 1997]. The second implementation is built as a SHORE 'value-added server' (VAS) [Carey et al. 1994], and as such it inherits the heavy-weight capabilities (and commensurate performance tradeoffs) of the SHORE system. The first of two implementations currently in progress uses the PS-AA [Franklin et al. 1997] cache coherency algorithm in place of AOCC used in our existing light-weight implementation. The second implementation builds PSI directly on top of the SHORE Storage Manager (SSM), with minor changes to some aspects of the SSM. We expect this tighter coupling and fine tuning to lead to performance gains over our SHORE-VAS implementation.

4.3 Utilizing PSI

The identification and formalization of the transactional object cache architecture and the subsequent definition of the PSI interface was motivated by a desire to promote the implementation of high performance persistent systems. Our particular objectives include the implementation of a high performance, scalable, orthogonally persistent Java. Unfortunately our early plans of adapting PJama [Atkinson et al. 1996] to utilize PSI have been beset by source code licensing difficulties, which, at the time of writing, appear to have just been resolved.

In order to gauge the performance of PSI implementations in relation to other systems, we have implemented the OO7 benchmark [Carey et al. 1993] directly on top of the PSI interface. Despite the

limitations of such benchmarks, OO7 can provide a helpful guide to performance and in addition, the size of the OO7 suite makes it a non-trivial 'proof of implementation' test [Carey et al. 1994]. Space constraints preclude the inclusion of any detailed analysis of the OO7 results in this paper, however we make the claim that given the breadth of the various OO7 operations measured, the overall results provide a fair indication of the relative performance characteristics of the various systems measured. The reader is refered to [Carey et al. 1993] for a detailed discussion of the benchmark. Figure 3 presents results comparing our light-weight PSI implementation with PJama[2], Texas [Singhal et al. 1992] and SHORE.

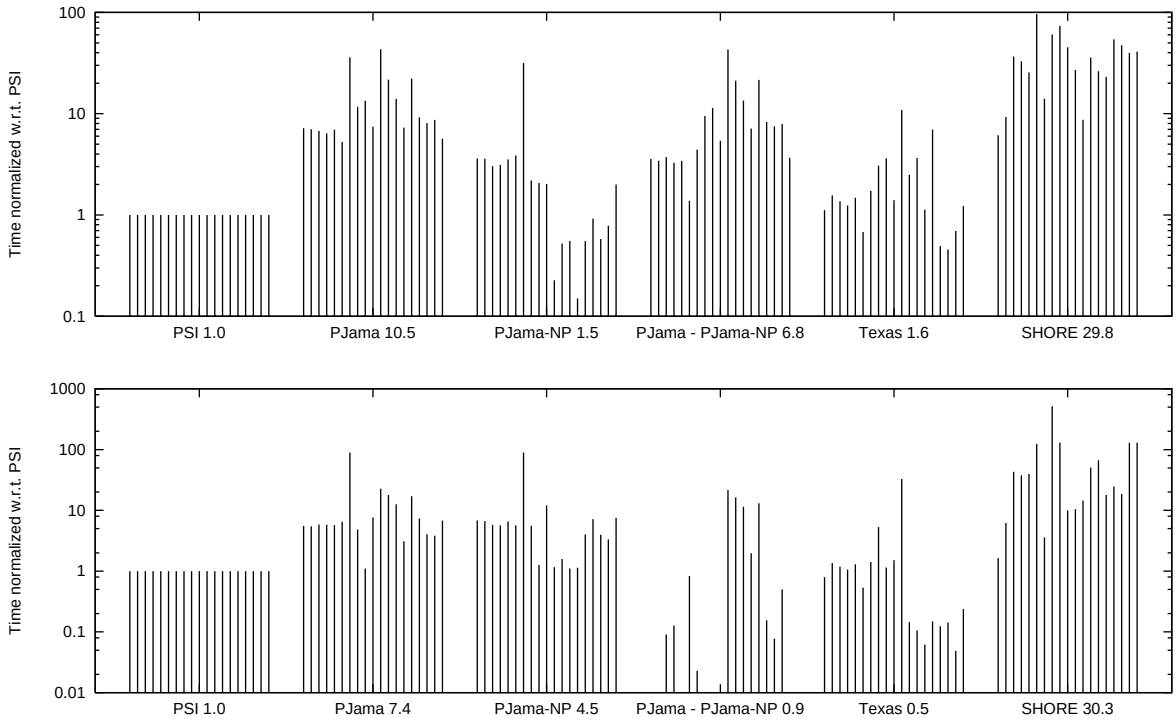

Figure 3: Cold (top) and hot (bottom) times for light-weight PSI, PJama, Texas, and SHORE for 17 OO7 benchmark operations (t1...q8) running over the 'small' database. Each bar represents the time for one benchmark operation normalized with respect to PSI and plotted on a log scale. The store costs for PJama are approximated by the difference between persistent ('PJama') and non-persistent ('PJama-NP') times ('PJama - PJama-NP'). The geometric means of the normalized times are printed to the right of each label. All runs were performed on an unloaded 166MHz Ultra1 CPU with 128MB RAM and separate store and log disks.

These performance results reflect the fact that in most respects SHORE is a different class of store to the others, which by comparison are very light-weight. For example, SHORE is a client-server system capable of supporting multiple simultaneous clients. Consequently OO7 runs in a separate process to the store, and uses RPC and shared memory primitives to communicate. The slower performance of Texas compared with PSI can probably be attributed to Texas's relatively expensive commits, and the inappropriateness of the page faulting mechanism in the context of pages full of fine-grained frequently changing meta-data (such as indexes). By contrast, Texas's page faulting mechanism helped it to perform about twice as fast as PSI on the hot runs.

The costs associated with the Java VM are fairly clear in the non-persistent PJama results (PJama-NP). On shorter operations such as queries (rightmost in each cluster) PJama-NP looks good relative PSI because IO costs dominate for PSI. However, this is offset by poor performances on the traversals (leftmost) where IO costs are amortized over many accesses. Overall PJama-NP runs about 1.5 times slower than PSI.

PJama's storage costs as approximated in the rightmost results above are significant compared with total costs (storage plus run-time) for PSI. This, coupled with information from detailed discussions with the PJama team which suggest that adapting PJama to utilize a PSI store should be relatively straight-

[2]Our Java OO7 implementation is publicly available at `http://cs.anu.edu.au/~Steve.Blackburn/upside/dist/`

forward, gives us confidence that a PSI-based persistent Java could be readily implemented and would perform well. Furthermore, the transparency provided by PSI to the underlying architecture will allow any such persistent Java to fully utilize scalable multicomputer PSI implementations.

In light of the aforementioned licensing difficulties we are proceeding with a persistent Java implementation based on GNU Kaffe. This implementation effort has resulted in close scrutiny of PSI as a platform for orthogonally persistent programming languages and has lead to minor refinements to the interface.

5 Future Work

The pinning down of transactional object cache architecture and the definition of the PSI transactional interface opens a range of possibilities for new work and creates a new context for existing work. As outlined in the preceding section, a wide range of work is in progress, both on the store and programming language sides of the interface. Among the milestones we see ahead are the following:

- Evolution of the model to account for the needs of layered transaction models (and more generally, open transaction models) [Weikum and Schek 1992], which will most likely require a meta-layer.
- Completion of a high performance, 'industrial strength', garbage collected PSI store. We are using the SHORE storage manager as the kernel for this effort.
- A highly scalable 'single image store' version of the above. This will depend on the success of our current scalability experiments with garbage collection and the PS-AA cache coherency algorithm.
- An orthogonally persistent Java bound to the PSI interface, and so able to utilize various PSI stores. We have a project underway based on GNU Kaffe.
- The use of PSI by other persistent programming language efforts, and the construction of other PSI compliant stores.

In keeping with our view that reuse and collaboration are important to the success of high performance orthogonal persistence, we aim to make our software developments publicly available where possible. Finally, we do not see PSI as an immutable interface definition—it is our hope that through wider use, the interface will undergo a process of refinement.

6 Conclusions

The transactional object cache architecture is the embodiment of three fundamental architectural devices: *atomicity*, *caching*, and *layering*. We argue that the importance of these devices in addressing key challenges of high performance persistent systems design makes the transactional object cache architecture an ideal foundation for the construction of such systems.

An abstraction of a generalization of the architecture has been presented along with a formalizing of its semantics. The generality of the abstraction opens it to a wide range of transaction models, cache coherency algorithms, and recovery systems. The identification of the semantics of the model has paved the way for the definition of a concrete interface and so a practical realization of the architectural approach.

The PSI transactional interface and examples of its implementation and application form early evidence of the utility of the transactional object cache approach. The performance and scalability of initial PSI implementations gives further support to the view that the approach may expedite the development of high performance orthogonally persistent systems.

7 Acknowledgements

The authors wish to thank Laurent Daynès, John Zigman, Luke Kirby, David Walsh, and David Sitsky for many helpful suggestions and comments on this work.

Bibliography

ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. 1995. Efficient optimistic concurrency control for distributed transactions. In M. J. CAREY AND D. A. SCHNEIDER Eds., *Proceedings on the*

1995 ACM-SIGMOD International Conference on the Management of Data, Volume 24 of *SIGMOD Record* (San Jose, CA, U.S.A., May 22–25 1995), pp. 23–34. ACM.

ATKINSON, M. P., DAYNÈS, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent java. *SIGMOD Record 25*, 4 (Dec.), 86–75.

BLACKBURN, S. M. 1997. *Persistent Store Interface: 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. AND STANTON, R. B. 1997. Scalable multicomputer object spaces: a foundation for high performance systems. In J. DARLINGTON Ed., *Third International Working Conference on Massively Parallel Programming Models* (London, Nov. 12–14 1997).

CAREY, M. J., DE WITT, D. J., AND NAUGHTON, J. F. 1993. The OO7 benchmark. In P. BUNEMAN AND S. JAJODIA Eds., *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, Volume 22 of *SIGMOD Record* (Washington D.C., U.S.A., May 26–28 1993), pp. 12–21. ACM.

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., 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.

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., CAREY, M. J., AND LIVNY, M. 1997. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems 22*, 3 (Sept.), 315–363.

HÄRDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15*, 4 (Dec.), 287–317.

MARTIN, C. P. AND RAMAMRITHAM, K. 1997. Delegation: Efficiently rewriting history. In A. GRAY AND P.-Å. LARSON Eds., *Proceedings of the Thirteenth International Conference on Data Engineering* (Birmingham U.K., April 7–11 1997). IEEE Computer Society Press.

MATTHES, F., MÜLLER, R., AND SCHMIDT, J. W. 1996. Towards a unified model of untyped object stores: Experiences with the Tycoon Store Protocol. In *Advances in Databases and Information Systems (ADBIS'96), Proceedings of the Third International Workshop of the Moscow ACM SIGMOD Chapter* (1996).

MOSS, J. E. B. 1981. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, U.S.A.

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., CONNOR, R. C., MORRISON, R., SCHEUERL, S., AND STEMPLE, D. W. 1994. Concurrent shadow paging in the Flask architecture. In M. ATKINSON, V. BENZAKEN, AND D. MAIER Eds., *Sixth International Workshop on Persistent Object Systems*, Workshops in Computing Series (WICS) (Tarascon, France, Sept. 5–9 1994), pp. 16–37. Springer-Verlag.

PU, C., KAISER, G. E., AND HUTCHINSON, N. C. 1988. Split-transactions for open-ended activities. In F. BANCILHON AND D. J. DEWITT Eds., *VLDB'88, Proceedings of the 14th International Conference on Very Large Data Bases* (Los Angeles, CA, U.S.A., Aug. 29–Sept. 1 1988), pp. 26–37. Morgan Kaufmann.

SCHEUERL, S. J. G., CONNOR, R. C. H., MORRISON, R., AND MUNRO, D. S. 1996. The DataSafe failure recovery mechanism in the Flask architecture. In *Proceedings of the Australian Computer Science Conference* (Melbourne, Australia, Jan. 1996), pp. 573–581.

SINGHAL, V., KAKKAD, S. V., AND WILSON, P. R. 1992. Texas: An efficient, portable persistent store. In A. ALBANO AND R. MORRISON Eds., *Fifth International Workshop on Persistent Object Systems*, Workshops in Computing Series (WICS) (San Miniato, Italy, Sept. 1–4 1992), pp. 11–33. Springer.

WEIKUM, G. AND SCHEK, H.-J. 1992. Concepts and applications of multilevel transactions and open nested transactions. In A. K. ELMAGARMID Ed., *Database Transaction Models for Advanced Applications*, Chapter 13, pp. 515–553. San Mateo, CA, U.S.A.: Morgan Kaufmann.