# Scalable Multicomputer Object Spaces: a Foundation for High Performance Systems

Stephen M. Blackburn and Robin B. Stanton*

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

**Abstract**

*The development of scalable architectures at store levels of a layered model has concentrated on processor parallelism balanced against scalable memory bandwidth, primarily through distributed memory structures of one kind or another. A great deal of attention has been paid to hiding the distribution of memory to produce a single store image across the memory structure. It is unlikely that the distribution and concurrency aspects of scalable computing can be completely hidden at that level.*

*The paper argues for a store layer which respects the need for caching and replication, and to do so at an "object" level granularity of memory use. These facits are interrelated through atomic processes, leading to an interface for the store which is strongly transactional in character. The paper describes the experimental performance of such a layer on a scalable multi-computer architecture. The behaviour of the store supports the view that a scalable cached "transactional" store architecture is a practical objective for high performance based on parallel computation across distributed memories.*

## 1 Introduction

The high performance computing enterprise has been characterised largely by the development of highly specialised parallel computing platforms and the search for effective solutions to problems in computational science and large scale simulation. In the process, the associated operating systems, file systems, data-management support, programming languages and program models became differentiated from what is now seen as "commodity" computing technologies and software engineering practices.

Although high performance computing goals continue to be pursued through specialised hardware platforms, considerable progress has been made to position the enterprise within an integrated "scalable computing" framework. In the idealised version of the framework, high performance computing ceases to exist as a differentiated area. Of course from an engineering perspective, it seems that there will always be differentiation due to the tradeoffs between costs and performance for special purpose computing. However with general purpose computing applications in mind, the focus on scalability reflects the importance of understanding how to design systems which, wherever possible, do not have to be refashioned when they are migrated to higher performance platforms.

For our purposes, the problem of scalability is derived simply from a layered view of systems in which the 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 associated with potential speed of computation.

The development of scalable architectures at the lower levels has concentrated on processor parallelism combined with scalable memory bandwidth, primarily through distributed memory structures of one kind or another. To this end a great deal of attention has been paid to abstracting away the distribution of memory through schemes which support a programming model having a single address space across the memory structure. The cache coherent non-uniform memory access architecture (CC-NUMA), is perhaps the best known example of this

*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.

work.

Notwithstanding the impressive progress that is being made with multi-threaded processing and memory structures of the CCNUMA kind, for quite fundamental reasons it appears to be unlikely that the distribution and concurrency aspects of scalable computing can be hidden at that level. This observation lies at the heart of the work reported in this paper.

Apart from the merits of providing for a single image address space, the paper argues for a store layer which respects the need for caching and replication, and to do so at an "object" level granularity of memory use. Caching and replication appear to be irreplaceable for moderating the impact of latency on performance in the presence of distribution and concurrency. The ways in which they interact within a store layer depends on coherency and visibility requirements. In the approach presented in this paper, caching and replication are interrelated through a model of computation based on atomic processes. In the model, coherency and visibility requirements are derived from the atomicity property, leading to an interface for the store which is strongly transactional in character. The stability of a store, and the recovery of defined states when abnormal events occur are critical for an important class of large scale, high performance systems. The treatment of these aspects in the paper is based on strategies developed for database technologies, where they have been intensively researched over the past two decades.

The paper is in two sections; the first places the control of caching and replication in the context of a store layer while the second describes the experimental performance of such a layer on a scalable multicomputer architecture. The behaviour of the store supports the thesis that a scalable cached "transactional" store architecture is a practical vision for parallel computation across distributed memories.

## 2 Scalability, Atomicity and Parallel Program Models

### 2.1 Scalability

The value of designing computing platforms which can be scaled to deliver performance at any point on the wide spectrum of practical computer systems design has always been recognised. The high end of the spectrum presents the biggest challenges for this otherwise deceptively simple idea. In general terms, increasing the speed of computation requires increasing processor speed, system communications and memory bandwidth so as to maintain an overall balance allowing performance for particular applications to scale, potentially at least, with the platform. When, as with current high performance platform technologies, the maintenance of that balance at increased performance levels depends on significant architectural changes, the problem of scalability becomes one of finding a higher level architecture which caters for the changes in architecture without forcing a change in the programming model used for applications.

At the high end of computation, processing speeds are typically increased by adding additional processors and providing the means by which their activities can be coordinated to increase the degree of parallelism in the system. Memory bandwidth can be also be increased by adding memory modules. However, maintaining the balance between memory bandwidth and processor speeds under increasing parallelism leads to increases in latency due to the distribution of memories and processors. This situation is intrinsic to scalability at the high end since it reflects the physics of computation.

There are two generic strategies for hiding the effects of latency on individual processors. The first is to increase the degree of concurrency using "independent" threads so that the effect of latency on any one thread can be masked by activities on other threads. The second is to cache information so as to reduce the distance from any particular processor to memory. The first, concurrency masked latency, leads to scalability at the application level if the threads being swapped are making progress towards the solution of a single application problem. This thread based contribution to masking latency at a single processor holds equally for multiple processor configurations. In a similar vein, the effectiveness of replicating potentially shared data in multiple caches depends on maintaining coherence across the cached values without causing significant additional latency costs through the coherence mechanisms.

It is clear then that concurrency and replication are key elements of any scalable architecture that extends to the high end of computation. The extent to which these two elements enable higher system performance for particular applications depends on being able to devise effective coherency maintenance and cache management strategies. Effectiveness in this context can be measured by the extent to which they reduce the impact of distribution on speed-up for particular applications.

There is a very rich research literature concerned with managing the impact of distribution on parallelism. At the risk of oversimplification, perhaps

the strongest application independent themes in this literature are on the one hand concerned with languages and program models for parallel computing and on the other, with creating single image spaces across distributed memories. A central idea in this paper is that atomicity is a particularly effective concurrency control primitive for managing the impact of distribution, and from that point of view it is under represented as a contributor to parallelism in both program model and single image space areas.

### 2.2 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" behaviour 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.

While atomicity is fundamental for operations at hardware levels, few programming languages have provided composition constructs for higher level atomic operations. Largely this is due to the cost of the "all or nothing" semantics of atomic operations weighed against their perceived value, rather than to the concurrency control and coherency maintenance aspects of atomicity per se. The cost of the "all or nothing" semantics derives from the need to make the initial state recoverable until the operation has terminated either by transiting to a new state or by aborting to the initial state. Sophisticated state saving mechanisms, mostly involving taking copies and/or logging incremental changes, have evolved over the past decade however the cost is still significant. This cost is born in systems which which support failure within non-deterministic "choice points" as do many AI languages, and of course it is born in transaction systems where atomicity primarily serves the paramount need to keep a database in well defined states determined by individual transactions. In the database world, the cost of state saving is also set against the need for stable stores and recoverability in the face of exceptional events. These concerns are increasingly being felt in the non-database world which is responding with technologies for constructing persistent systems [Atkinson and Morrison 1995].

Returning to the role that atomicity can play in managing the impact of distribution in parallel systems, it is worthwhile recalling that the use of replication to mask or reduce latency comes at the cost of coherence maintenance which in general has the effect of increasing latency. This coherence induced latency can be moderated by optimistic strategies in which an assumption of coherence is tested at some later point in time thereby masking the latency to an extent which depends on the truth of the assumption. The cost of exploiting optimism to enable a higher degree of parallelism is a combination of the state saving costs mentioned above and the opportunity costs, if any, of any "wasted" computing resources. From this point of view, optimistic strategies and failure enabled non-determinism have very similar operational semantics.

A further observation concerning coherence induced latency is that some applications are tolerant to data which is incoherent in the sense being discussed here, and that in such cases, coherency tests can be discarded. This visibility issue is picked up in a later section of the paper.

The assertion made in the introduction, that atomicity as a concurrency control primitive appears to be particularly well suited to scalability with its replication requirement, rests on the above arguments and on the modularity provided for locality of coherence. This is a safe assertion for some important classes of application, for example database applications and related information servers. It is not yet clear how broad the application area for such atomicity based stores will prove to be however the fundamentals of scalability argue that they are likely to have a central role in the design of high end systems.

### 2.3 Programming Models

The interplay between replication and concurrency in scaling computation to the high end through their role in weakening latency constraints points to the value of atomic process modules being visible at the store layer. With a focus on a broad range of applications this raises questions about whether current parallel programming models are suited to deriving such modules from application programs.

The major programming models for parallelism are based on data parallel, message passing and shared variable paradigms. For our purposes, data parallelism is seen as a highly specialised form of message passing.

Message passing paradigms have arguably been the mainstay of applied research into concurrent computing, due in large part to the CSP framework [Hoare 1978; Milner 1980]. With the emergence of MPP technology in the last decade, MPI [Message

Passing Interface Forum 1994] and PVM [Geist and Sunderam 1992] emerged as practical portable layers. Both are now standards in the parallel computing community.

From the perspective of scalability, beyond the point where replication becomes the key element in latency moderation, the message passing model appears to have serious restrictions. The strength of the model is isolation; there are no shared variables. Processes are regarded as localised modules with distribution managed by copying values through message transmission between processes. The notion of replicated variables, central to the cached architecture argued for above, is not easily accommodated by extensions to the paradigm and in any case to do so would (re)introduce shared variables between otherwise isolated processes.

Partly due to the absence of support for replication in the message passing model, atomicity is not as relevant a concept for the model as it is for cached architectures. The strength of isolation between processes in the message passing model provides one aspect of atomicity, however other aspects of coherency and concurrency control are left to the application program. It may be of course that the message passing interface evolves support for cached variables however that would be to change the programming model that is currently supported by the paradigm.

### 2.4 Shared Variables

The other major paradigm is the shared variable model for parallel computation. In this model, processes interact through shared variables. The model is derived from the widespread practices which date back to the early days of computing, the invention of semaphores, monitors and many other ideas for solving generic mutual exclusion problems in single processors supporting concurrent processing.

Unlike message passing, the shared variable model does not reflect distribution. In the context of distributed of parallel memories, implementation of the model rests on an interface which provides a single image of a store. In practical terms the single images are provided through distributed shared memory (DSM) technology [Carter 1995; Keleher 1994], or more directly through hardware which creates single address spaces (CCNUMA [Lenoski et al. 1992] architectures for example ). In either case it is not straightforward to incorporate the replication and concurrency issues in the shared variable model without making significant changes. For example, caches are not part of the architecture so any replication that may be part of the single image mechanism is not available to layers above the store. An immediate consequence of hiding replication is that coherency is also hidden.

Since the model is not "distribution aware", atomicity is not as important as it is in a cache based architectures. However it should be noted that creating atomicity or transactional package on top of the shared variable model would not deliver the latency moderation role being sought for scalability since that role relies visibility of particular caches.

## 3 Scalable Multicomputer Object Spaces

The previous sections describe the problem of scalably providing a single image store in a distributed memory context, with *atomicity*, *caching*, and *layered software* identified as properties of fundamental importance to any such architecture.

This section explores the practicalities of implementing such a system by reviewing two experiments in scalable single image object space design and using the lessons of these experiments to motivation a new design approach which is the subject of section 4.

### 3.1 Experiences with Multicomputer Object Space Design

The two experiments in multicomputer object space design described below were conducted in the context of developing *persistent* single image multicomputer object spaces with a strong focus on scalability. The first of these, MC-Texas, is non-transactional, implementing the single image object space through a crude transparent DSM mechanism. The second, MC-DataSafe, is based on a transactional object storage architecture.

#### 3.1.1 MC-Texas

MC-Texas, a multicomputer variant of the Texas persistent store [Singhal et al. 1992] is one of the early steps taken on the road to designing and implementing a highly scalable persistent system [Blackburn and Stanton 1996]. Despite its limitations, MC-Texas remains a significant example of a DSM-based object store.

**Texas** Texas groups persistent objects on pages and uses standard hardware memory protection mechanisms to detect references to objects that are not currently in memory. When a page of objects is faulted into memory via this mechanism, persistent address pointers in the page are detected and replaced by pointers into virtual memory according to a map of

persistent to virtual addresses. If a persistent address is not in the map, virtual address space is allocated for the page containing that address and the new entry inserted into the <persistent, virtual> address map. The newly allocated page in the virtual address space is access protected to trap the first attempt to access the data, at which point the persistent data is paged in.

**Atomicity** The absence of any atomicity mechanism in the Texas architecture lead to MC-Texas adopting a DSM-like non-transactional approach to distributed cache coherency. The consistency semantics of this system are relatively weak.

The task of committing to disk a consistant image of the store is made difficult in MC-Texas by the absence of atomicity. The necessary synchronisation of distributed process state is left to the user.

**Caching** The use of page-grained hardware mechanisms to detect user demands strongly colours the behavior of both Texas and MC-Texas. On one hand, this leaves clients with direct memory access to the cached data—reducing the access overhead for 'hot' data to zero. On the other hand, the grain of *access detection* is the page, which opens considerable opportunity for 'false contention'—accesses on unrelated objects that happen to be co-located on a single page may result in needless contention for that page.

**Scalable software layer** As a software layer, MC-Texas presents its clients with an abstraction of a single shared persistent memory, and as such sits at a level above dominant programming models such as basic message passing, which only layer weakly above the communications level. Although the scalability of the single image abstraction presented by MC-Texas was limited, the MC-Texas experiment proved to be a valuable first step in the exploration of the single image concept.

**Key Outcomes** The MC-Texas experiment highlighted: difficulties associated with a distributed programming model that does not support atomicity; the performance advantages of giving users direct access to cached data; and the feasability of the single store image concept.

### 3.1.2 MC-DataSafe

The goal of the MC-DataSafe experiment was to investigate scalability of page coherency and recovery mechanisms. The experiment was conducted in the context of the DataSafe [Scheuerl et al. 1996] instance of the Flask object store architecture [Munro et al. 1994].

**Flask** Flask has a layered architecture in which various store components are integrated through well defined interfaces. A property of the Flask architecture is that conflict detection is undertaken by the upper layers. This frees, to a large extent, the lower layers from interference management and increases flexibility in the choice of concurrency model in the upper layers. This is in contrast to systems where concurrency control mechanisms are more tightly integrated with the lower layers.

**Atomicity** Flask supports a transactional coherency model and supports atomicity as part of the user interface.

The MC-DataSafe page coherency algorithm operates at a transactional grain—consistency checks are done at commit time. In addition to tightening the bound on coherency-related traffic, transactional grain coherency has the advantage of overlapping coherency related messaging with commit-related messaging and IO latencies.

**Caching** Central to the Flask architecture is the abstraction of a resilient persistent heap. The lower levels of the architecture present the upper layers with this abstraction via a word-grained procedural interface which performs an address translation for each access—the cache is thus not directly exposed. The combination of fine temporal and morphological grain with the overhead of procedure calls and hash-based address translation results in a continuous computational overhead, placing the Flask architecture at the other end of the granularity spectrum to Texas, both temporally and morphologically.

The Flask architecture goes beyond the simple notion of cache offered by Texas, and supports the notion of 'workspaces'—potentially inconsistent replicated data—thereby facilitiating optimistic computation.

**Scalable software layer** MC-DataSafe takes the concept of building scalable layered abstractions a few steps further than MC-Texas by presenting the clients with an even richer, if less general, environment that exhibits a high degree of scalability [Blackburn et al. 1997]. MC-DataSafe builds a transactional

storage layer from the scalable resources of the parallel file system, message passing and distributed memory available on the Fujitsu AP1000

**Key Outcomes** The MC-DataSafe experiment highlighted among other things: the natural affinity between the transactional model of computation and scalable single image persistent systems; and the value of controlled inconsistency in replication as means of facilitiating optimism.

## 4 A Reference Architecture: The Transactional Object Cache

The previous sections have identified the challenge of developing scalable architectures and have highlighted some of the lessons of the MC-Texas and MC-DataSafe experiments in scalable multicomputer object store design. In response to this, a reference architecture for scalable multicomputer object space construction—the transactional object cache—is now presented.

From the perspective of the future development of scalable object spaces, perhaps the three most important lessons of the MC-Texas and MC-DataSafe experiments are these:

- The importance and appropriateness of a transactional model of concurrent computation when working in a distributed object space.
- The impact of temporal and morphological grain on performance.

These lessons can be interpreted as indicating a need for an architecture for scalable object spaces which:

1. Embodies a transactional model of concurrency control.
2. Offers an object-grained interface to clients which minimizes the need for copying of data.

The transactional object cache architecture (figure 1) is one which satisfies these criteria. The basic architecture consists of five key components: an application program; an (optional) language run-time 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. The second criterion two is met by virtue of the direct cache access given to the application and the object grained nature of the interface.

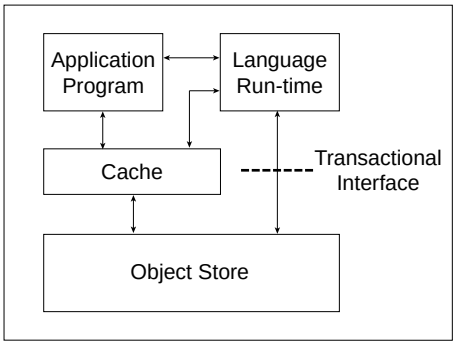


Figure 1: The transactional object cache architecture. The architecture of the object store is transparent to the application.

The remainder of this section describes key characteristics of the transactional object cache architecture.

**Morphological and Temporal Granularity of Interaction** Interface operations are object-grained[1] rather than page or word grained, allowing the problem of false sharing to be avoided while overheads associated with procedure calls to the interface are minimized. The extent to which problems of false sharing are avoided in a given implementation will depend on the algorithms employed in that implementation. The choice of appropriate morphological grain for data transfer, concurrency control, and replica management has been the subject of extensive study and experimentation [Bernstein et al. 1987; Zaharioudakis and Carey 1997], the results of which are directly applicable to transactional object cache implementations.

By presenting its clients with a buffer to which they have direct memory access, the object cache treads the middle ground between Flask and Texas. Explicit client interaction with the cache interface is limited to notification of demand for read or write access to an object—fine-grained accesses are done via direct memory loads and stores. This approach requires minimal user / store interaction while avoiding the problem of false contention due to temporal exaggeration of user needs.

[1] Finer-grained operations are sometimes desirable, for example in the case of operations on indexes [Blackburn 1997].

**A Transactional Interface** There are a number of important consequences of working with a cache architecture which is intrinsically transactional, particularly with respect to concurrency control.

Firstly, transactional semantics are distribution-independent—in the sense that ACID[2] semantics encapsulate a notion of isolation that will hold independent of whether the underlying store is distributed. This means that the architecture may be distributed transparently with respect a client interface that guarantees ACID semantics. The transactional object cache architecture is thus sympathetic to our objective of delivering users a *single image address space*.

Secondly, the substantial body of work on transactional cache coherency algorithms [Franklin 1996] is directly applicable to the architecture. The implementer is thus able to experiment with a wide range of algorithms including recent developments such as the avoidance-based PS-AA algorithm [Zaharioudakis and Carey 1997] and detection-based AOCC [Adya et al. 1995].

**Client-Server Architecture** The transactional object cache architecture is naturally client-server, client RTSs caching data provided by server stores, which are responsible for stability and coherency. As with all client-server systems, the transactional object cache can be implemented as client-peer simply by co-locating clients and servers (figure 2).

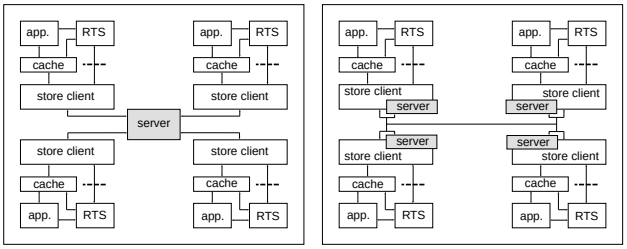


Figure 2: Client-server (left) and client-peer (right) transactional object cache architectures. In both cases the distributed nature of the underlying architecture is transparent to the run-time systems and applications.

[2]The term ACID refers to the fundamental properties of transaction models: Atomicity, Consistency, Isolation, and Durability [Gray and Reuter 1993].

# 5 An Abstraction of the Transactional Object Cache Architecture

The role of the transactional object cache architecture as the basis for the design of multicomputer object spaces has been made clear in the previous section. Its importance comes as a consequence of its embodiment of three fundamentals of scalable object space design, namely *caching*, *atomicity*, and a *layered architecture*. The last of these is seen in the strong separation of concerns it embodies. Such a separation of concerns opens opportunities for a focusing of expertise and increased collaboration between research groups.

Effectively realizing these opportunities will depend on the existence of a well defined interface which captures the separation of concerns in a manner that does not rob the RTS implementer of efficiency or functionality, nor deny the store implementer of flexibility in approach. The first step in the design of such an interface is to identify a clear abstraction of the transactional object cache architecture. The value of such abstractions as a means of meaningfully capturing the behavior of such systems is exemplified by schemes such as ACTA [Chrysanthis and Ramamritham 1994] and CACS [Stemple and Morrison 1992]. In the absence of a pre-existing abstraction of the transactional object cache, one must be developed—which is the purpose of this section.

We begin with an identification of key architectural concerns before examining transactional object cache semantics in terms of these concerns.

## 5.1 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 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 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 the objective of flexibility in the interface design, two key concerns of a transactional object cache are separated and identified[3]:

- visibility,

[3]In the case where the store is persistent, 'stability' becomes an additional concern. See section 7 and [Blackburn 1997].

- and cache management,

The first is unambiguously central to a transactional storage interface, the second is included in a pragmatic response to the demands of efficient object space construction.

By establishing abstract interfaces to both of these concerns, a store interface can be built up and described precisely in terms of its semantics with respect to each of the concerns. The remainder of this section will focus on the development of abstract interfaces with respect to visibility and cache management. Having developed the models and abstract interfaces to them, there will be a discussion on how the concerns come together to form a rich abstraction of the transactional object cache architecture.

## 5.2 Visibility

Visibility is an 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 1996].

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 [1996], 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.

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

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

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 $hv$:

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

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 immutability 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 $hv$):

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

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:

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

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:

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

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 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 visibility semantics of a transactional object cache.

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

Table 1: 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 $hv$: $(hv^{t'} = hv^t.e_i) \Rightarrow hv.e_i$.
- 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.

### 5.2.1 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)** $hv^t = empty \ \wedge \ T' = T \cup \{t\}$

**ReadIntention(t,o)** $hv^{t'} = hv^t.r_o$

**ReadComplete(t,o)** $hv^{t'} = hv^t.\bar{r}_o$

**WriteIntention(t,o)** $hv^{t'} = hv^t.w_o$

**WriteComplete(t,o)** $hv^{t'} = hv^t.\bar{w}_{o_v}$

**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)** $immutable(t) = true$

### 5.2.2 Visibility and Logging

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

**CheckpointVisibility(t)** $hv^{t'} = hv^t.m_i$

**RollbackVisibility(t,i)** $hv^{t'}.m_i.E_r = hv^t$, where $E_r = e_{r_0}.e_{r_1} \ldots e_{r_n}$

#### 5.2.3 Visibility 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 visibility of some operation/s from one transaction to another.

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 $hv$ is unchanged). The following primitives define visibility semantics with respect to two transactions $t_i$ and $t_j$ and some object, $o$:

**DelegateVisibility($t_i$,$t_j$,o)** $hv^{t_i\prime} = hv^{t_i} \setminus hv^{t_{io}} \;\wedge\; hv^{t_j\prime} = hv^{t_j} \cup hv^{t_{io}}$, where $hv^{t_{io}}$ is a sub-history of $hv^{t_i}$ consisting of all events $e_o$ relating to a change in state of $o$. These semantics are complex unless $T(hv^{t_{io}})$ holds.

**IgnoreConflict($t_i$,$t_j$,o)** $(hv^{ic_{t_i}\prime} = hv^{ic_{t_i}} \cup hv^{t_{jo}}) \;\wedge\; (hv^{ic_{t_j}\prime} = hv^{ic_{t_j}} \cup hv^{t_{io}})$. Thus for all events relating to object $o$, $t_i$ and $t_j$ are added to each other's 'ignore conflict' sub-histories ($hv^{ic_t}$). 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.

### 5.3 Cache Management

The second key dimension[4] of the transactional cache architecture is cache management. A cached object space design is motivated by the desire to hide 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 2: 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 access to a given object. The state of the available objects is a function of the visibility control mechanism.

### 5.4 Generality and Completeness

Both of the orthogonal abstractions outlined above are general—in the sense that they are premised only by the intrinsics of scalable object space design, 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, and checkpoint/rollback. When brought together, the abstractions yield a full abstraction of the transactional object cache with the same generality and completeness.

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 may be trivially adapted to account for such.

## 6 Scalability of the Transactional Object Cache

Having identified the transactional object cache as a basis for scalable multicomputer object space de-

[4] See footnote 3.

sign, results that point to the scalability of the architecture are now presented. The reader is referred to [Blackburn 1997] for a more detailed analysis of the experiments.

## 6.1 PSI/AOCC Scalability Experiments

The basis for the following experiments is an implementation of PSI, a transactional object cache interface whose semantics have been defined in terms of the abstractions presented in section 5 and [Blackburn 1997]. The generality of the abstraction upon which PSI is based allows it to be implemented using a wide range of approaches to concurrency control. The experiments reported below were with respect to PSI/AOCC, an implementation of the AOCC (Adaptive Optimistic Concurrency Control) 'detection-based' transactional cache coherency algorithm [Adya et al. 1995] within the PSI framework. AOCC is the most recently published and apparently the best performing 'detection-based' algorithm. Our plans include the implementation of PS-AA [Franklin 1996; Zaharioudakis and Carey 1997], the leading 'avoidance-based' algorithm, within the PSI framework and conduct a comparitive study.

### 6.1.1 AOCC

The Adaptive Optimistic Concurrency Control (AOCC) algorithm [Adya et al. 1995] falls within a class of detection-based client-server cache consistency protocols which defer consistency actions until commit [Franklin 1996]. Clients operate over locally cached objects and check the consistency of their cache with the server only at commit time. In addition to commit-time checks, the server may piggy-back invalidations on top of other communications with the client (for example, a response to a client object fault). Object invalidation and faulting is done at an adaptive grain—pages are used for initial faults but object-grain updates are sent in response to subsequent invalidations.

AOCC is designed for a client-server setting, and in the context of multiple cooperating servers it adopts a two-phase commit protocol. The use of the two-phase commit protocol is significant because it has important ramifications for the scalability of AOCC [Blackburn 1997]. For this reason, the two phase commit protocol is described here.

Each client preparing to commit first sends *prepare* notices to each server 'owning' an object in the committing transaction's read or write set. The prepare notice includes a time stamp and read and write sets. Each participating server replies with either a tentative commit or an abort depending whether the proposed commit conflicts with any transaction that is already committed or is in receipt of a tentative commit at that server. The preparation phase happens *concurrently*—the client sends all requests before receiving any replies. If a client receives tentative commits from all participating servers, it goes ahead with the commit, sending *acks* and updated objects to all servers 'owning' objects in the transaction's write set. Otherwise, the client aborts and sends *nacks* to all servers that offered a tentative commit.

### 6.1.2 Experimental Framework

In order to establish the scalability characteristics of PSI/AOCC, the store was subjected to a range of workloads under various levels of contention.

The HOTCOLD workload [Carey et al. 1994] was adopted as the basis for the PSI/AOCC scalability experiments. This workload was chosen both because of its suitability for modeling low to moderate levels of contention in a page serving object cache architecture and because of its familiarity within the database community. The HOTCOLD workload models contention by assigning each client a disjoint range of pages within the object space which that client will *tend* to access most frequently, called a 'hot range'. The remainder of the database is referred to as that client's 'cold range'. By varying the probability of hot and cold accesses, various levels of contention can be modeled.

In a multi-server context, the overall level of communication and the number of servers involved in a given commit are a function of both locality in the hot/cold sense, and the locality of the hot ranges with respect to servers. The HOTCOLD workload was thus extended to include a notion of hot range/server locality.

**Localized HOTCOLD workload** *Localized* HOTCOLD can be thought of as a best-case scenario for hot range/server locality. The hot range for each client is chosen so that it is 'owned' by just one server (skewing was used in the experiments to ensure that the 'owner' was not the local server). In this situation, the overall level of communication and number of servers involved in the two-phase commit will only grow as the number of accesses to objects in the cold range grows and so will be relatively low.

**Scattered HOTCOLD workload** *Scattered* HOTCOLD can be thought of as worst-case. Here the hot

range for each client is chosen so that the pages are distributed across *all* servers. The level of communication and the number of servers involved in each commit is therefore generally high and is largely independent of the level of contention.

In both cases, per-processor throughput is measured for different levels of contention and a constant per-processor workload (i.e. the workload scales with the number of nodes). In addition, the transaction length is varied. As the transaction size increases the read and write sets for each transaction grow. Consequently the probability of conflict rises and a larger number of servers are involved in each commit.

## 6.2 PSI/AOCC Scalability Results

PSI/AOCC was tested with the localized and scattered HOTCOLD workloads under a wide range of values across various dimensions of the HOTCOLD parameter space. The results of these tests (amounting to more than 2000 execution runs on the AP1000) are summarized in figure 3. The precise details of the workload parameter settings are presented in [Blackburn 1997] along with more detailed results.

The graphs demonstrate that PSI/AOCC scales very well under most conditions and that scalability is sensitive to the probability of cold accesses, transaction length, and hot range/server locality.

The scalability results in figure 3 are presented in terms of the *effective* number of processors seen when using 128 processors under a range of conditions. The *effective* number of processors is a measure of the throughput seen in the 128 processor case relative to the throughput for the base case[5]. If the system were to scale perfectly the *effective* processor measure would be 128. Figure 3 is therefore only a comparison of 128 processor throughput against base case for a range of conditions. The results in [Blackburn 1997], from which these were drawn, present per-processor throughput as a function of $N$.

Although the results suggest that PSI/AOCC scales well, closer analysis of the workloads reveals that biases built into the workloads result in the graphs in figure 3 *understating* the scalability of PSI/AOCC [Blackburn 1997].

The key finding of the PSI/AOCC scalability experiments—that the PSI/AOCC architecture exhibits a high degree of scalability under a wide range of conditions—is qualified by the following:

- Client-server locality is very important in the context of long, read-write transactions. In this

[5]The client-server nature of the AOCC algorithm dictates a base case of $N = 2$.

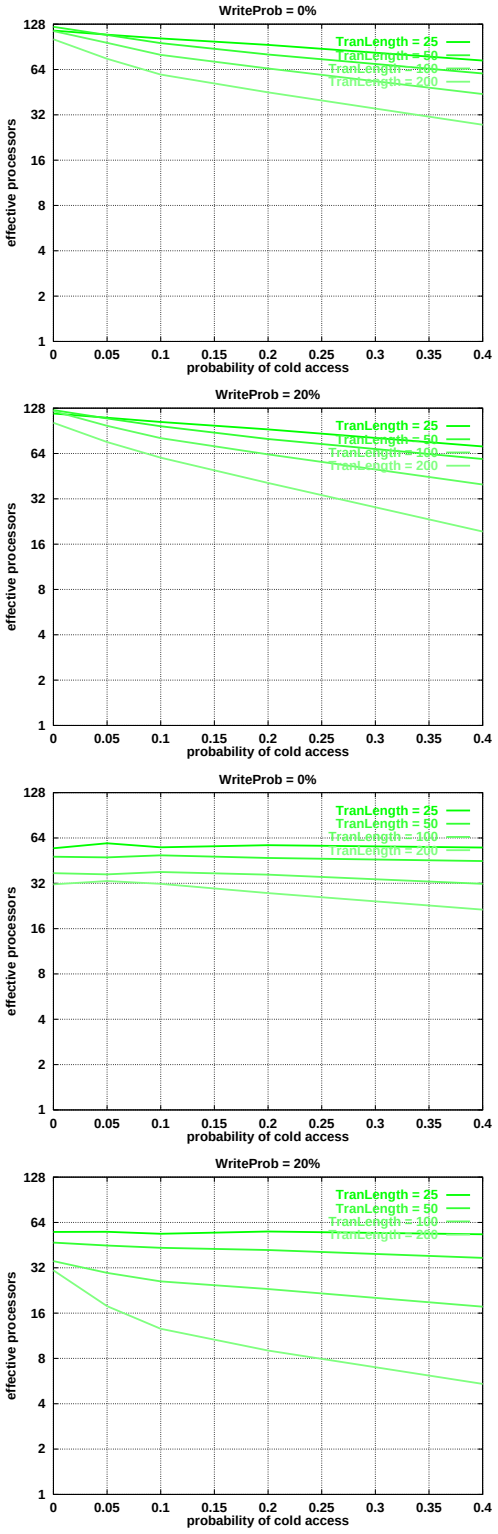


Figure 3: Scalability of PSI/AOCC on a 128 node Fujitsu AP1000. *Top to bottom:* localized read-only; localized read-write; scattered read-only; and scattered read-write.

context poor client-server locality leads to exposure of inscalability in the two phase commit protocol used by PSI/AOCC (compare second top and bottom, figure 3).

- Transaction length and hot range/cold range locality impact on the scalability of PSI/AOCC by increasing in the number of servers involved in commits, which results in scale-dependent false abort problems.

- The results reported here represent only one, fairly general, workload. For a more complete picture of PSI/AOCC scalability to emerge, the system will have to be examined in the context of a wider range of workloads.

It is noteworthy that once biases in the workloads are taken into account [Blackburn 1997], significant scalability problems are evident in only three out of the sixteen cases presented in figure 3. In the thirteen remaining cases, scalability ranges from very good to outstanding. Furthermore, the scalability of practical applications is unlikely to ever fall as low as the worst results reported here, as no practical applications is likely to be so heavily network I/O bound (the experiments set the time spent on computation other than cache coherency, *ThinkTime*, at 0, which is clearly unrealistic for normal applications).

## 7 Persistence

The persistence of objects held in the store, the central component of the transactional object cache architecture, has not been addressed in the previous sections of this paper. In fact, persistence can be considered as an orthogonal property of the architecture, the transactional object cache abstraction (section 4) being straightforwardly extened through a 'stability' abstraction. The semantics of the transactional object cache can thus be described in terms of *three* orthogonal concerns [Blackburn 1997]:

- visibility,
- cache management,
- and stability.

Blackburn [1997] uses this extended abstraction as the basis for the development of PSI (Persistent Store Interface), a pinning down of the transactional interface that is central to the transactional object cache architecture. PSI is the cornerstone of a new approach to scalable persistent object system design.

## 8 Conclusions

Our work on scalable object stores is a contribution to the challenge of finding architectural principles for cost effective scalable computing more generally. High end computing has a special role in scalability research since it provides performance benchmarks against which new ideas can be tested. Although such benchmarks are highly dependent on application layer programs, there are lower levels which ideally are scalable for a very wide range of problems. With this in mind we have shown that practical scalable object stores can be constructed on a parallel computer platform using a transactional cache architecture and that its performance scales to a high end platform. Our results include a reference architecture and interface for such stores. The problem of distribution in high end parallel platforms is met by supporting atomicity at the interface. The atomic property of processes operating in the store is effective for concurrency and coherency control among distributed caches.

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

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency control and recovery in database systems*. Addison-Wesley series in computer science. Addison-Wesley, Reading, MA, U.S.A.

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., 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., 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.

CARTER, J. B. 1995. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*. Special issue on distributed shared memory.

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.

GEIST, G. A. AND SUNDERAM, V. S. 1992. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience 4*, 4 (June).

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo.

HOARE, C. 1978. Communicating sequential processes. *CACM 21*, 8 (Aug.), 666–677.

KELEHER, P. 1994. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Computer Science, Rice University, Houston, TX, U.S.A.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSY, J., HOROWITZ, M., AND LAM, M. S. 1992. The Stanford Dash multiprocessor. *Computer 25*, 3 (March), 63–79.

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.

MILNER, R. 1980. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag.

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.

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.

STEMPLE, D. AND MORRISON, R. 1992. Specifying flexible concurrency control schemes: An abstract operational approach. In *15th Australian Computer Science Conference* (Hobart, Australia, Jan. 1992), pp. 873–891.

ZAHARIOUDAKIS, M. AND CAREY, M. J. 1997. Highly concurrent cache consistency for indices in client-server database systems. In *Proceedings on the 1997 ACM-SIGMOD International Conference on the Management of Data*, Volume 26 of *SIGMOD Record* (Tucson, AZ, U.S.A., May 1997). ACM.