Recovery and Page Coherency for a Scalable Multicomputer Object Store

Stephen M. Blackburn, Robin B. Stanton, Christopher W. Johnson
Department of Computer Science
Australian National University
Canberra ACT 0200 Australia
{*steveb,rbs,cwj*}*@cs.anu.edu.au*

Stephan J. G. Scheuerl
Division of Computer Science
University of St Andrews
Fife UK KY16 9SS
stephan@dcs.st-and.ac.uk

Abstract

This paper presents scalable algorithms for recovery and page coherency in multicomputer object stores. Recovery and coherency are central to object store engineering and distributed memory multicomputers are fundamental to scalable computation. Efficient recovery is implemented through a combination of local logging and a localisation of the transactional workspace model. A vector of update counts is used to efficiently represent global time.

The algorithms have been successfully implemented and tested on a 128 node Fujitsu AP1000 distributed memory multicomputer. The paper presents performance results which indicate good performance and scalability for these algorithms under a range of situations. The work is seen as a step in the continuing development of high performance multicomputer object stores.

1. Introduction

This paper reports on scalable recovery and page coherency algorithms for distributed memory multicomputer object stores (MOS). Recovery and coherency are central to ensuring the durability and consistency of data held in object stores, while scalability is of primary importance to the design of multicomputer software infrastructure.

Section 1.1 introduces multicomputer object stores, discusses their role in the high performance object store world and explains the importance of scalability in MOS design. Because this paper is limited to addressing the issues of recovery and page coherency, the context in which the work is presented is significant. The assumptions and limitations imposed by that context are spelt out in section 1.2.

Section 2 outlines the store architecture designed as part of this project, and explains the role the recovery and coherency algorithms play in making that architecture scalable. Section 3 discusses the recovery algorithm and section 4 outlines the page coherency algorithm. Results and the approach to validation and performance analysis are outlined in section 5.

1.1. Multicomputer Object Stores

This research was conducted in the context of a broader objective of creating a software environment which facilitates the integration of high performance computers into networked computer systems. Distributed memory multicomputers are central to the project because of their key role in the quest for scalable high performance computation. This factor together with the increasing importance of object orientation in computation and data management gives rise to our focus on Multicomputer Object Stores.

Many of the important factors in MOS design are common to the design of conventional object stores, however their relative importance is changed because of the unique architectural properties of multicomputers. Key concerns are problems associated with distribution, concurrency and issues of scalable performance. Multicomputers are distinguished from more general distributed computing environments by their low latency, high bandwidth, high reliability interconnections and their provision for computational isolation (typically through such mechanisms as gang scheduling or physical partitioning).

Scalable memory bandwidth, scalable processing power and scalable I/O are fundamental to scalable computation. Distributed memory multicomputers seek to deliver these characteristics and so play an important role in achieving scalable performance. The development of scalable software for such machines depends on that software not inhibiting scalability through contention and non-locality of data. The objective of scalability underpins many of the design decisions reported in this paper, and is central to the challenge of developing efficient multicomputer object stores.

1.2. Project Context and Assumptions

The goal of this project was to further our understanding of multicomputer object stores, by building on lessons learnt from the Multicomputer Texas [1] experiment and by exploring elements of the DataSafe [2] recovery mech-

anism that might be relevant to multicomputer object store design. To this end, an efficient multicomputer analogue of the DataSafe recovery mechanism was designed and implemented. The target machine, a Fujitsu AP1000, is typical of distributed memory multicomputers.

In taking DataSafe as a starting point for the design of the recovery mechanism, a number of DataSafe characteristics are adopted. The following sections describe the impact various aspects of the DataSafe design had on this project. Relevant characteristics of the exemplar multicomputer, the AP1000, are also discussed.

1.2.1. DataSafe Architecture and Recovery Mechanism The DataSafe recovery mechanism has been implemented in the framework of the Flask [3] object store architecture. The Flask architecture promotes modular implementation of various store components through a layered software abstraction with 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 and recovery mechanisms are more tightly integrated.

There are many approaches to recovery documented in the literature. The DataSafe architecture is based on the DB cache [4], which uses contiguous writes to a circular log file called the "safe" to reduce write time, while using a random access file for the main store. All pages updated in the course of a transaction are written contiguously to the safe. The updated pages are migrated to the store opportunistically, freeing up the safe and updating the store. If insufficient space is available in the safe, the migration of pages to the store may be forced. Although the safe may be thought of as a fixed size circular log file, with minor sophistication to the design, it may be implemented as an adaptable size log. On re-start the safe is checked for data not propagated to the store. A safe map is used to record the location and status of pages in the safe, and by implication, the status of pages in the store. By ensuring atomicity of safe map operations, all store and safe operations are atomic.

The lower half of Figure 1 illustrates the key components of the recovery mechanism. The state depicted in the illustration reflects the following sequence of events: Pages C, F, B, G, H, D, and J were updated by earlier committed transactions. Subsequently, pages B, C and D were opportunistically written back to the store. These events resulted in pages F, G, H, and J residing in the safe, with all other pages residing in the store.

The DataSafe recovery mechanism is not suitable for a scalable implementation because access to a single safe and safe map would become a bottleneck for committing transactions. Section 2.6 explains how local safes are used to provide a scalable recovery mechanism.

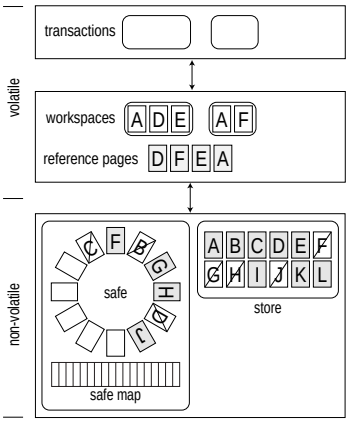

Figure 1. **The DataSafe architecture.**

1.2.2. Workspaces for Optimistic Computation The DataSafe employs workspaces as a means of isolating uncommitted transaction updates. This mechanism helps to ensure the transactional properties of atomicity, coherency, isolation and durability (ACID) properties [5].

Workspaces are separate address space contexts through which each transaction sees the store. Until a transaction modifies a page, it has read-only access to a shared *reference page* (cached store page). When a transaction attempts to write to a page for the first time, a copy is made of the *reference page*. The transaction's address map is modified so that further references by the transaction to that page will see the workspace copy rather than the reference page. By comparing workspace pages with the corresponding reference pages, updates to the workspace pages can be determined. When a transaction commits, all updates made by that transaction are propagated from the workspace to the reference pages which are atomically written to the safe and then opportunistically written to the store. Updates are discarded on abort. For a more complete discussion of the workspace model see [6], and [7].

The upper half of Figure 1 illustrates the workspace concept. The two transactions underway are accessing the pages D, F, E, and A, so reference copies of these pages are in memory. The pages being updated by each of the transactions have been copied into their respective workspaces.

The concurrency control layer assumed by DataSafe assures object isolation (section 1.2.3), a property that is exploited in the implementation of the following mechanism, which ensures that updates generated from workspace pages are coherent with respect to other updates to the same store page. In the absence of update logging or any other such means of identifying which part of a page has been modified, delta (update) pages are generated by bitwise XORing each committing workspace page with the corresponding reference page.

Each committed workspace page becomes the new reference page and the delta is applied to each workspace copy of the original reference page – maintaining the invariant that the only difference between each workspace page and its corresponding reference page is the updates made by the transaction running in that workspace.

The application of deltas to all other workspace pages requires the *synchronous* cooperation of the corresponding transactions. This is acceptable in a uniprocessor context as only one transaction can be executing at a time. However, it is inappropriate in a multiprocessing context where the cost of synchronisation would grow with the number of concurrently executing transactions, making the system fundamentally unscalable.

Section 2.4 describes a scalable approach to distributed workspaces, and section 4 describes algorithms that maintain page level coherency in that environment.

1.2.3. Object Isolation The approach to page coherency described in section 1.2.2 assumes a concurrency control layer that enforces object isolation. Object isolation is also assumed in our distributed architecture. For this reason, object isolation is defined here:

Given a history, H, representing the ordering of events in the concurrent execution of a set of transactions, a transaction $t_i \in H$ will only commit if for all overlapping committed transactions $t_j \in H$ the read set of t_i does not intersect the write set of t_j.

This definition of object isolation is predicated on a total ordering on transaction initiation and termination events, with only a partial ordering of other events. In this context the concept of "overlapping transactions" is intuitive. A less strong definition of object isolation may be made in the context of a total ordering on all events.

1.2.4. AP1000 Distributed Memory Multicomputer The platform for this work was the Fujitsu AP1000 multicomputer [8]. The AP1000 is a distributed memory machine consisting of a number of SPARC 1 nodes connected via a high speed, low-latency network, with memory and possibly other resources such as disk attached to each of the nodes. The machine used in this project was configured with 128 nodes, each with 16MB of RAM. 32 of these nodes had an option board with an I/O processor, 2MB of RAM, and a 2MB/sec peak throughput disk of 512MB – an aggregate 16GB with 64MB/sec theoretical peak throughput. The HiDIOS parallel file system[9] on the AP1000 provides all nodes with a single file system image, the distribution of the disks being transparent to the user. Files are striped across the 32 disks in 128KB stripes. Reads or writes that span multiple stripes can exploit the parallelism of the file system, with multiple disks serving the request in parallel. While the HiDIOS file system readily delivers more than 50MB/sec for large writes, under the test conditions reported in section 5 (80KB write per commit), only about 20MB/sec was seen.

The communications network of the AP1000 supports inter-node message passing as the main internal communications mechanism. MPI-1, a widely accepted portable message passing interface [10], is implemented on the AP1000 [11] and was used in this project. Under our test conditions MPI delivered a latency of 125μsec (3125 SPARC IU cycles) and 2.69MB/sec bandwidth for a one-way trip — an $n_{1/2}$ value of 336 bytes.

Handling messages asynchronously would enable the efficient implementation of responsive servers by avoiding the need to poll. Such a mechanism is not available under MPI-1. However the handled-receive (hrecv()) operation, an enhancement proposed in the draft MPI-2, is locally implemented. The operation allows a handler to be asynchronously invoked on arrival of a matching message (much like an interrupt handler). This powerful mechanism provides a means of emulating preemptively scheduled server threads in processes that are otherwise single-threaded clients.

2. A Distributed Stable Store Architecture

A distributed store architecture based loosely on the DataSafe is the framework within which distributed recovery and page coherency experiments were conducted. Under this distributed architecture, transactions and their associated workspaces are distributed across nodes, with access to a shared store. The following sections outline key aspects of this design.

2.1. Client-Server Architecture

Following the Multicomputer Texas work [1], we have adopted a client-server architecture where servers run as threads within client processes. A product of this approach is *dynamic* maintenence of client/server resource usage ratios. By contrast, in a conventional client-server architectures, resources are normally *statically* distributed between client and server nodes, precluding dynamic client/server resource usage. Another advantage of this approach over other client server architectures is that the work load is physically more distributed as the number of nodes that are serving is increased. As a result, the communications network is less likely to suffer from bottlenecks produced by congestion at individual nodes. By having the client and server share the same address space, greater caching opportunities exist and the cost of a request to the local server is very low. We use the MPI hrecv() facility (see section 1.2.4) to emulate preemptive daemon server threads.

2.2. Single Store Image

Multicomputers are distinguished from more general distributed computing systems by their relatively low latency, high bandwidth, high reliability interconnect. For this reason we choose to introduce a high degree of distribution transparency and present a single store image to all nodes. In this environment, all nodes see the same PID (persistent identifier) address space and access objects in the same way, regardless of any notion of the object's location.

2.3. Global Concurrency Control

Following the Flask architecture, we assume a layered architecture including the *assumed existence* of a concurrency control layer. By adopting such an architecture we are free to independently develop the lower layers of the store, maintaining a focus on distributed page coherency and recovery mechanisms. As with the DataSafe implementation, we assume the concurrency control layer enforces object isolation (section 1.2.3). In this context a transaction runs on *only one node*. Each node may concurrently execute a number of transactions.

2.4. Distributed Workspace Model

Section 1.2.2 describes the workspace concept and the DataSafe implementation of workspaces. The key implementation issue is maintaining coherency between multiple workspaces sharing a common reference page so that at commit time the changes made by the transaction owning the workspace can be readily detected and applied to the store.

The DataSafe's workspace model is modified here by *localising* the management of coherency between workspace pages to each node. Rather than using workspaces to provide *up to date versions of pages* for writing at commit time, workspaces are used to derive *deltas* (updates) associated with the committing transaction. It is sufficient to record deltas at commit time to ensure recoverability.

To derive a delta, all that is needed are two versions of the page which differ only in terms of those changes made by the executing transaction. This is achieved in our distributed store by having local reference pages and *only maintaining inter-workspace coherency between workspaces sharing that reference page*, thereby avoiding inter node synchronisation (section 1.2.2).

Figure 2 illustrates the distributed DataSafe store. For clarity, only the activity associated with transactions on the first node is illustrated. The sequence of events that led to the depicted state are the same as those for Figure 1 (see section 1.2.1). Although the reference pages for the other nodes are not shown, those for the first node (pages D, F, E, and A) are shown as being local to that node. If any of the other nodes were accessing any of these pages, it would have its own reference page.

2.5. Distributed Page Management

Although inter-workspace coherency is relaxed, coherency must still be maintained. The approach taken in our distributed store architecture is to delegate page management responsibilities to nodes on a per-page basis. The delegation can be implemented with a simple hash function (as illustrated in Figure 2).

Each node manages a set of pages (according to the delegation algorithm), maintaining a master copy for each page in that set that is accessed by any node in the system. All committed updates to a page are propagated to the corresponding master page by the committing node. The owner (manager) of a master page is responsible for coherently applying updates to the page and opportunistically writing pages back to the store. The algorithms involved are discussed in detail in sections 3 and 4.

Whenever a coherent version of a page is required by any node, a request for that page is sent to the owner of the master copy and the owner returns a coherent version of the page, reading it in from disk if necessary. This approach to page management has the side effect of introducing a layer of page caching, reducing the number of disk reads. Performance results (section 5) show that cache effects are significant, contributing to super-linear speedup.

2.6. Local Logging

While the argument for a single store image suggests a need for coordinated access to the store, there is no need for globally coordinated access to a single safe. On the contrary, by localising safe use, the need for coordination is significantly reduced, thereby enhancing the scalability of the system. Our architecture therefore uses local logging. At commit time, the committing node writes deltas to its *local* safe and propagates the updates to the relevant master nodes.

The combination of local logging and the use of deltas rather than updated pages significantly reduces the coordination overhead at commit time, thereby greatly improving the scalability of the system. The side effects of this approach are two-fold. By committing deltas rather than updated pages, store pages are always necessary for recovery (deltas are applied to the original to produce the updated pages). This means that a technique such as shadowing must be used to atomically update store pages. The second consequence is that the recovery process is more complex. The details of the recovery process are elaborated in section 3.4. Figure 2 illustrates the local safes and safe maps necessary for local logging. Note that in contrast to Figure 1, the safes contain *deltas* (denoted by a prime), and the corresponding

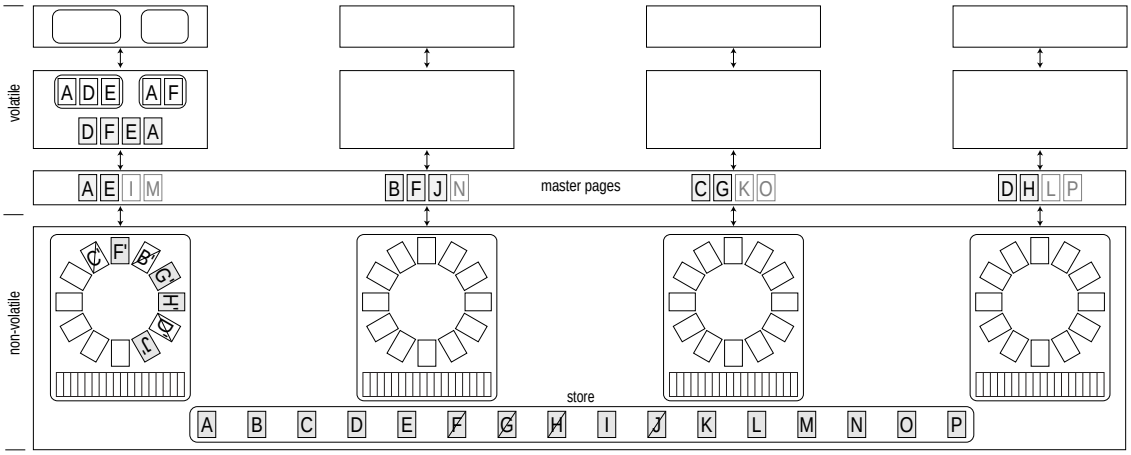

Figure 2. **A distributed DataSafe-like store.**

store pages are required for recovery (even though they are not valid for reading).

Further work needs to be done in determining whether it is better to localise safes on a per-disk basis (maximising write locality) or to distribute safes across multiple disks (maximising opportunities for write parallelism).

2.7. Transaction Grain Page Coherency

The assumed concurrency control layer's enforcement of object isolation, as defined in section 1.2.3, does not enforce page isolation. The replication of reference pages across different nodes (section 2.4) requires page coherency among reference pages to be enforced by the lower layers of the system. This is achieved by maintaining coherency of master pages (section 2.5), and ensuring that cached reference pages are invalidated appropriately.

The object isolation property is transaction grained, in the sense that conflict is defined in terms of transaction initiation and termination rather than in terms of conflict event and transaction termination (see section 1.2.3). The coherence of an object is therefore guaranteed by the concurrency control layer from the point when a transaction begins until that transaction commits. Therefore coherency need only be established at a transaction grain rather than at the activity grain. As a consequence, the level of communication associated with maintaining page coherency is greatly reduced. Section 4 describes the page coherency algorithm.

2.8. Centralised Serialisation Server

Maintenance of coherency demands some degree of synchronisation between nodes. As far as possible the responsibility for coherency is left to individual nodes, in order to minimise communications. Centralisation of services is usually avoided in distributed systems because of the potential bottlenecks as the system scales up. Centralisation can, however, have the advantage of significantly reducing communication costs. For example an N to N communication may be reduced to an N to 1 communication followed by a 1 to N communication by centralising the procedure. A resolution to the question as to whether it is best to centralise or not may be helped by contrasting the likelihood of the target machine's overall message passing bandwidth being swamped with the likelihood of the centralised server overloading a node to the point where that service becomes a bottleneck.

A centralised server is adopted here to provide a global serialisation service. The requirement of only providing transaction grain coherency, coupled with the use of an update count vector as a global clock (section 2.9), enables the development of very efficient coherency algorithms (sections 3 and 4). Our performance results support the argument that the server is efficient and does not become a bottleneck to scalability (section 5.4).

2.9. Update Count Vector Time

Cheaply obtaining a global time against which (partial) ordering of distributed events can be made is a long standing problem in distributed computing. The role of global time in our system is to place a total ordering on transaction *start* and *commit* events.

Our solution to the problem is to use a vector of *update counts* to represent global time. The key temporal reference events are transaction commits. Each commit corresponds with the transmission of updates to one or more master nodes. Global commit orderings can therefore be described in terms of the count of updates at each node. The commit that sent node 0 its first update and node 2 its fourth would be identified by the vector [1,0,4,2], where nodes 1 and 3 had received none and two updates respectively prior to the commit.

A transaction that started immediately after that commit

would be issued the same update vector to identify its start. When requesting data from node 3 it would quote an update count of two, allowing node 3 to ensure that it had received its second update before responding to the request.

The centralised serialisation server is employed to maintain the update vector. It updates the relevant entries at each commit time (uniquely identifying each commit) and provides transactions with the vector, allowing them to identify their start time. The algorithms using update count vector time are described below.

3. Distributed Commit and Recovery

The previous sections outlined the context in which transactions operate and argued for the design choices made. The following sections describe the algorithms associated with commit and recovery, using an adaptation of conventional set notation. The algorithms, together with the page coherency algorithms (section 4) guarantee the ACID properties of transactions in the distributed store architecture.

3.1. Commit

This section describes the commit algorithm. Important characteristics of this algorithm are:

- By using a safe, the log data is written contiguously to disk, minimising seek time overhead, thereby reducing disk latency.
- Local safes are used, avoiding the need for any synchronisation of safe writes.
- By logging deltas rather than updated pages, the coherency of reference pages can be relaxed and the need for propagating changes to *all* workspace and reference copies of the page at commit time is removed.
- Updates may be sent lazily to nodes holding masters.
- The update vector supplied by the serialisation server is used to ensure update coherency by guaranteeing update ordering.

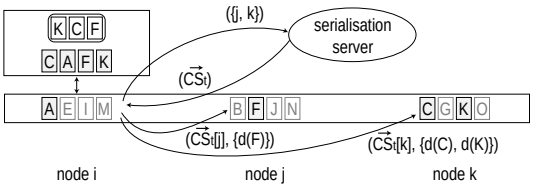

Figure 3. **Commit protocol.**

The algorithm involves the following six steps. We assume a transaction, t is committing the set of pages it has updated, Pw_t.

Create log A log L_t associated with transaction t is created. The log is a set of deltas defined as follows: $L_t = \{d(p)|p \in Pw_t\}$, where $d(p)$, a delta associated with page p, is derived by XORing p with the corresponding reference page $r(p)$, and Pw_t is the set of pages written by t.

Start log write The log, L_t, is appended to a safe, a circular on-disk log file. If necessary the safe is flushed to ensure sufficient space, by requesting the write-back to store of pages in the safe that have not already been opportunistically written back. By using a non-blocking disk write, the write to safe may be overlapped with other steps of the commit algorithm.

Send timestamp request A message containing MN_t is sent to the serialisation server. MN_t is the set of nodes that hold masters of pages in Pw_t: $MN_t = \{mn(p)|p \in Pw_t\}$, where $mn(p)$ is the node where the (unique) master of the page p resides.

Receive timestamp A vector of communication sequence numbers $\vec{CS}_t$ is received from the serialisation server. $\vec{CS}_t$ reflects the place of transaction t in a global ordering of updates by capturing for each node to which updates are sent, the number of updates sent or about to be sent to that node, including those to be sent by t.

Complete safe write Once the asynchronous log write has completed, the safe map is atomically updated to reflect the state of the safe. For each delta $d(p)$ written to the safe, the page number of p and the communication sequence number at the corresponding master node ($\vec{CS}_t[mn(p)]$) are written into the safe map slot corresponding with the safe page used for the delta $d(p)$. The atomic update of the safe map atomically completes the commit.

Distribute deltas An update U_{t_n} is sent to each node $n \in MN_t$. U_{t_n} is the set of all of t's deltas associated with the node n: $U_{t_n} = \{d(p)|p \in P_t, mn(p) = n\}$. Each update message is tagged with $\vec{CS}_t[n]$, where n is the receiving node.

3.2. Receipt of Deltas.

The page coherency algorithm (section 4), is dependant on the coherent application of deltas to master pages. The following algorithm for the receipt of deltas ensures that deltas are received in the same order that the corresponding commits are made (according to global time, as defined by the update vector issued by the serialisation server). The delta receipt algorithm for any node n is:

Establish receive A receive is established for an update message tagged with CS_n, the receiving node's communication sequence number. Upon receipt of the appropriately tagged message, the next step is commenced.

Receive update. For each delta, $d(p)$ in the update message, the corresponding master page $m(p)$ is updated ($m(p) := m(p)$ XOR $d(p)$), the source node is inserted in the update set U_p for the page ($U_p := n \cup U_p$), and the update sequence for that page is set to CS_n ($US_p := CS_n$). CS_n is incremented and a new update receive is established.

This algorithm was implemented efficiently by using the MPI-2 hrecv() function (section 1.2.4) to emulate a pre-emptive server thread which handled the receipt of deltas.

3.3. Opportunistic Write-back.

The opportunistic write-back algorithm must update the store and allow for the freeing of redundant safe pages. In the DataSafe, this operation is straight forward: the out of date store page is overwritten with the new version and then the safe map is atomically updated so that the store page is valid and the safe page is freed. This situation is complicated in our distributed store because the safes contain *deltas* rather than *updated pages*, so the store page is always necessary for recovery (the updates held in the safe must be applied to the store page to reestablish the page). A solution is to use shadowing to ensure that store pages are updated atomically.

Each master node has a shadowing file. The shadowing file consists of a number of pages that may be used for shadows and an update array ($UA_n[]$) which is used to record the progress of a write-back. There is a one-word entry in the update array for each master page managed by the node. Update array entries are initially set to NULL.

The following algorithm is dependant on its implementation being robust with respect to the arrival of update messages during the write-back operation.

Make shadow A shadow, $s(p')$ is made of the page to be overwritten, p'.

Write update The store page, p', is overwritten by the master $m(p)$ and the relevant update array entry is atomically set to US_p ($UA_n[p] := US_p$). The shadow $s(p')$ is then removed.

Notify clients Each node in the update set for the updated page, $n \in U_p$ is notified of the update. The notification message consists of p and US_p, allowing the recipient to free the safe page or pages relating to updates of p up to time US_p.

Complete write-back The update set for the page is set to null ($U_p := \{\}$).

The write-back operation may be batched to reduce disk I/O costs and also to allow client notifications to be batched, reducing communications traffic.

3.4. Recovery

Before closing the store, each node writes back all updated master pages that have not already been written back. By coordinating the close of the store, a successful close will allow all nodes to flush their safes. The recovery process is initiated if, on start up, any node finds a non-empty safe. For this reason, the recovery process is closely linked to the opportunistic write back algorithm 3.3, which is the only means of freeing safes during normal processing. Once the recovery (if any) is complete, each node reinitialises its shadow file before commencing normal store operations.

Because the ordering of opportunistic write-backs need not coincide with update message ordering, deltas to be recovered will not, in general, correspond exactly to a consecutive subset of the updates received prior to the crash. For this reason, during the recovery process updates are received in any order and receipt of all messages is guaranteed by establishing that all messages have been sent and assuming that no messages have been lost or are in transit. Until all messages have arrived, no updates can be guaranteed to be coherent. So ensuring that write-backs only occur after the delta receipt phase maintains the recoverability of the recovery process itself.

The recovery algorithm consists of four steps:

Re-distribute deltas For each delta $d(p)$ in the safe, the delta and the communication sequence number stored in the safe map entry corresponding with $d(p)$ are sent to the corresponding master node $mn(p)$.

Receive updates Before receipt of any update messages, the update sequence for each page managed by the recipient is set to the corresponding value in the update array ($\forall p | (mn(p) = n).US_p := UA_n[p]$). During the recovery process, update messages may be received in any order. For each delta $d(p)$ in an update message, the source node, n is inserted in the update set U_p for the page ($U_p := n \cup U_p$), and if the delta's sequence number $s(d(p))$ is greater than the page's update sequence, US_p, the corresponding master page $m(p)$ is updated ($m(p) := m(p)$ XOR $d(p)$) and the update sequence for that page, US_p, is set to $max(US_p, s(d(p)))$. Write-backs are not permitted during the delta re-distribution and receipt phases.

Write-back Once all updates have been received (a barrier synchronisation can be used to enforce this), all update pages are written back according to the write-back algorithm outlined in section 3.3.

Completion of recovery The write-back operations will result in clients being able to free their safes. Once all nodes have emptied their safes, the recovery process is complete.

4. Coherency

The decision to replicate reference pages (section 2.4) led to the need for coherency to be maintained between each master page and the corresponding reference pages. As argued in section 2.7, the assumption of object isolation means that coherency need only be maintained at a transaction grain (rather than operation grain) for a given transaction. The coherency mechanism must therefore guarantee that all pages seen by a transaction are consistent with all transactions committed prior to the start of the transaction (according to their ordering in global time). The object isolation property ensures that we need not worry about whether a transaction sees the effects of transactions committed after that transaction began.

This scenario leads to a relatively simple coherency algorithm.

Send invalidation request At the start of each transaction, send an invalidation request with a list of all locally cached pages to the centralised serialisation server.

Process invalidation request The centralised serialisation server consults a table of update timestamps and by comparing the time of the last invalidation request for the client node with the update timestamp for each of the pages in the request message, establishes a list of invalidated pages. The server responds to the client with a list of invalidated pages and a timestamp corresponding to the time of the invalidation request.

Invalidate cache On receiving the invalidation message, the client marks each of the invalid pages as invalid.

Request page When the client faults on a page, it sends a request to the node holding the master, asking for a copy of the page at least as recent as the transaction start time. This is done efficiently by quoting the transaction timestamp vector element corresponding with the node holding the master.

Update page On receiving the new page, a check is done to see if there are any workspace copies of the page in use. If so, the differences between the new page and the stale reference page are applied to each workspace page. The object isolation property guarantees that this operation will not lead to inconsistency. Once any workspace pages have been updated, the stale reference page is overwritten with the new one and the cache page is marked valid.

5. Performance

The objective underlying this project was the development of efficient architectures and algorithms for multicomputer object stores, drawing on the experience of DataSafe and Multicomputer Texas. The following sections outline our approach to performance analysis and its results.

5.1. A Flexible Test Harness

As stated earlier in this paper, the focus of the work was on recovery and coherency. The full development of a multicomputer object store was beyond the scope of the project. For this reason, it was important to develop a test harness that could reasonably emulate the load of a working store and associated applications. It was also important that the test harness stress the behaviour of our system under scalability and speedup conditions, as these are seen as important characteristics of multicomputer object stores.

The test harness developed for this project allows various synthetic transaction loads to be run against the system. The harness issues read, write, begin, commit, abort, and switch instructions to the distributed store layer, simulating an application running multiple concurrent transactions on top of a concurrency control layer, which in turn runs on top of our distributed store layer. The harness ensures that the object isolation property is enforced.

The harness does *not* simulate application-level computation, so the simulations represent an I/O bound application. One could expect decreased disk and network congestion if application-level computation was added. In this sense, the tests are worst-case in nature.

5.2. Scalability

Scalability is a measure of the capacity to efficiently scale resources with increasing problem size. With the above test harness, this can be tested by increasing the test load in proportion to the number of nodes used while keeping all other parameters static. Ideally, the time to complete the problem would remain static or even decrease as the number of nodes increased in proportion to the load. However, distribution usually brings with it overheads, so in practice the goal is to approach the ideal as closely as possible.

We tested the system under a range of loads, with different read and write ratios, different transaction lengths, varying commit-abort ratios, etc. The results indicate good scalability, even under worst-case loadings. The graph in Figure 4 illustrates the scalability of the system under a loading of 10 transactions per node, each involving 20 write operations, where each write results in commiting one 4K page. The transactions operated over an object space of 1024 pages, sufficient for the parallel file system to stripe across all disks. The points on the graph indicate the average completion times per node and the error bars indicate the standard deviation. The results are averaged over several runs.

For 16 and 32 processors, cache effects bring the performance close to ideal (5.44 sec). In the worst case, 128 nodes take an average of just under 11 seconds to complete 128 times the workload of the one node case.

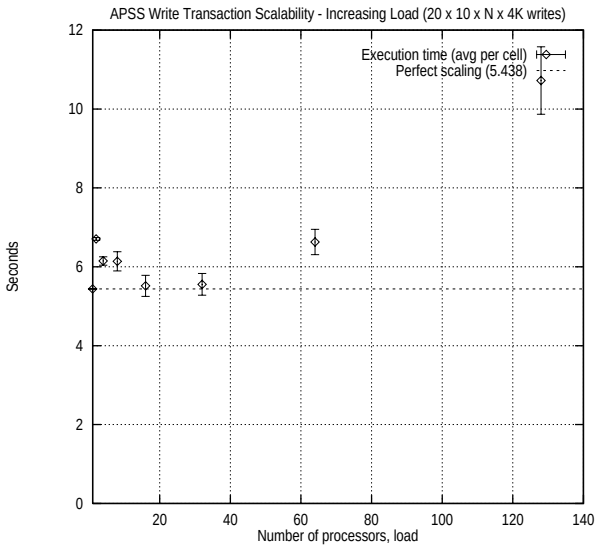

Figure 4. **Scalability results.**

Analysis of various cost components, such as the centralised server and disk performance suggest that the key performance bottleneck is the machine's I/O capacity (communications network and disk).

5.3. Speedup

Speedup is a measure of the extent to which the execution time for a fixed size problem decreases as more resources are made available to solve that problem.

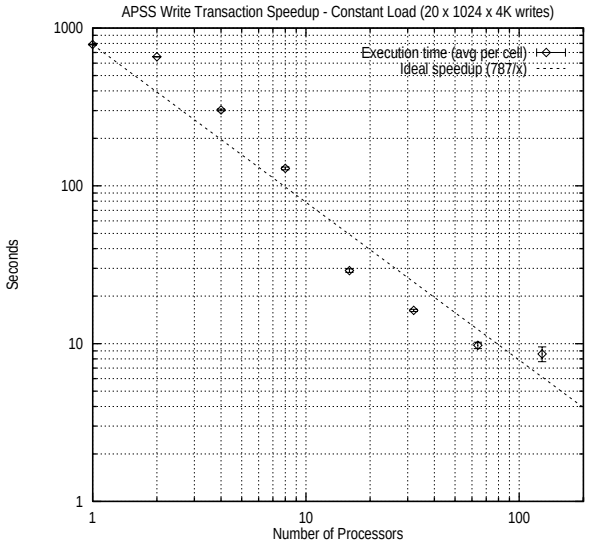

Figure 5. **Speedup results.**

Speedup tests were performed by setting a large workload and measuring the completion time as the number of nodes assigned to the problem increased. The graph in Figure 5 represents the results of a typical experiment involving 1024 transactions, each comprising 20 write operations which each result in committing one 4K page. The line of ideal speedup from the single node case is shown. These results indicate that the system exhibits very good speedup characteristics. The better than ideal results for 16, 32 and 64 node cases can be accounted for by cache effects outweighing any bottlenecks introduced by the increased degree of distribution.

5.4. Cost of Centralised Serialisation

The decision to centralise a key aspect of the architecture, the serialisation service, was based on the assumption that other elements of the system would become bottlenecks before the cost of the centralised server became significant and that by centralising the server, major savings would be made in the recovery and coherency algorithms. The measurement of the cost associated with the centralised server was therefore of particular interest to us.

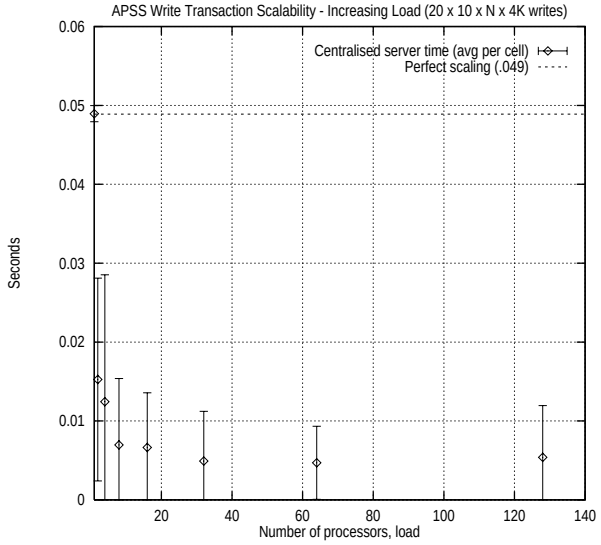

Figure 6. **Average time spent waiting for centralised serialisation server under proportionately increasing loads.**

Figures 6 and 7 indicates the costs associated with the centralised server under the same test loads as those depicted in Figures 4 and 5 described in the above sections. The graphs indicate clearly that the centralised server scaled very well. They also indicate that the cost of the centralised serialisation server was negligible (about 1/1000th of the transaction execution time), and as expected, further testing showed no measurable gain from using an additional node as a dedicated server.

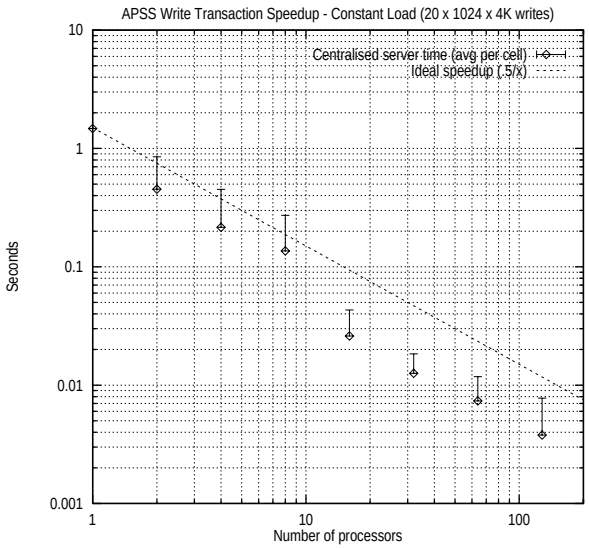

Figure 7. **Average time spent waiting for centralised serialisation server under constant load.**

6. Conclusion

We sought to build from our experiences with DataSafe and Multicomputer Texas and develop an efficient multicomputer analogue of the DataSafe. In doing so, we did not attempt to build a complete multicomputer object store, but concentrated on the recovery and page coherency problems. By restricting our focus, it was necessary to make assumptions about the behaviour of the rest of the store. To the extent that that was necessary, we tried to base our assumptions on the behaviour of the existing DataSafe implementation, and assume such behaviours might be implemented in a multicomputer context.

The architectures and algorithms developed exhibit a number of important characteristics, including:

- A client server architecture utilising new mechanisms to implement daemon server threads which allow responsive servers to run within client processes.
- A single store image, providing a single PID space to all nodes.
- An efficient distributed workspace model based on replicated reference pages.
- Local logging of commits on per-node safes.
- An efficient centralised serialisation server.
- Global time implemented through the centralised serialisation server using a vector of update counts.

Performance analysis of the system has shown that the algorithms exhibit good speedup and scalability characteristics. The cost of the centralised serialisation server was also shown to be negligible.

References

[1] S.M. Blackburn and R.B. Stanton, "Multicomputer object stores: the Multicomputer Texas experiment", in R. Connor and S. Nettles, editors, *Seventh International Workshop on Persistent Object Systems*, Cape May, NJ, May 1996.

[2] S.J.G. Scheuerl, R.C.H. Connor, R. Morrison, and D.S. Munro, "The DataSafe failure recovery mechanism in the Flask architecture", *in Proceedings of the Australian Computer Science Conference*, pp. 573–581, Melbourne, Australia, Jan. 1996.

[3] D.S. Munro, R.C. Connor, R. Morrison, S. Scheuerl, and D.W. Stemple, "Concurrent shadow paging in the Flask architecture", in M. Atkinson, V. Benzaken, and D. Maier, editors, *Sixth International Workshop on Persistent Object Systems*, pp. 16–37, Tatascon, France, Sep. 1994.

[4] K. Elhardt and R. Bayer, "A database cache for high performance and fast restart in database systems", *ACM Transactions on Database Systems*, vol. 9, pp. 503–525, Dec. 1984.

[5] T. Härder and A. Reuter, "Principles of transaction-oriented database recovery", *ACM Computing Surveys*, vol. 15, pp. 287–317, Dec. 1983.

[6] I. Gold and H. Boral, "The power of the private workspace model", *Information Systems*, vol. 11, pp. 1–7, 1985.

[7] H. Kung and J.T. Robinson, "On optimistic methods for concurrency control", *ACM Transactions on Database Systems*, vol. 6, pp. 213–226, June 1981.

[8] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato, "CAP-II architecture", *in Proceedings of the First Fujitsu-ANU CAP Workshop*, Kawasaki, Japan, November 1990. Fujitsu Laboratories Ltd.

[9] A. Tridgell and D. Walsh, "The HiDIOS file system", *in Proceedings of the Fourth Parallel Computing Workshop*, London, September 1995. Fujitsu Laboratories Ltd.

[10] Message Passing Interface Forum, "MPI: A message-passing interface standard", *in International Journal of Supercomputing Applications*, vol. 8, November 1994, Also avialable as University of Tennessee Technical Report CS-94-230.

[11] D. Sitsky, D. Walsh, and C. Johnson, "An efficient implementation of the message passing interface (MPI) on the Fujitsu AP1000", in M. Ishii, editor, *Proceedings of the Third Parallel Computing Workshop*, Kawasaki, Japan, November 1994. Fujitsu Laboratories Ltd.