Overview of distributed shared memory

A Judge P A Nixon V J Cahill B Tangney S Weber
Distributed Systems Group,
Dept. of Computer Science,
Trinity College, Dublin,
Ireland,
Tel: +353-1-608-2666,
Fax: +353-1-677-2204,
{pnixon,vjcahill,tangney,sweber}@cs.tcd.ie

October 29, 1998

So much has already been written about everything that you can't find out anything about it.

— *James Thurber*, Lanterns and Lances (1961)

Loosely-coupled distributed systems have evolved using message passing as the main paradigm for sharing information. Other paradigms used in loosely-coupled distributed systems, such as RPC, are usually implemented on top of an underlying message-passing system. On the other hand, in tightly-coupled architectures, such as multi-processor machines, the paradigm is usually based on shared memory with its attractively simple programming model. The shared-memory paradigm has recently been extended for use in more loosely-coupled architectures and is known as *distributed* shared memory (DSM [153, 178, 58]) in this context. This chapter discusses some of the issues involved in the design and implementation of such a DSM in loosely-coupled distributed systems and briefly discusses related work in other fields.

In DSM systems, processes share data transparently across node boundaries; data faulting, location, and movement are handled by the DSM system. Among other things, this allows parallel programs designed to use the shared-memory abstraction to execute without modification on a loosely-coupled distributed system. The advantages of the shared-memory programming model are well known: shared memory gives transparent process-to-process communication, shared-memory programming is a well-understood problem (compared with message passing), shared-memory programs are usually shorter and easier to understand than equivalent message-passing programs, and large or complex data structures may easily be communicated without marshalling. This transparency and ease of use is associated with a cost and DSM applications are sometimes slower than hand-coded message-passing implementations. In fact, it has traditionally been believed that a well-built, hand-tuned and optimised, message-passing implementation of a parallel program will out-perform a shared-memory implementation. However, more recent work shows that this is not necessarily the case [36, 131] and that issues such as the relative costs of communication and computation are important factors. Moreover, in some cases, applications using DSM can execute faster than equivalent message-passing ones [177], even when the DSM system is implemented on top of a message-passing system.

Lauer and Needham [129] argued that shared-memory (or "procedure-oriented") and message-passing operating systems are, in some sense, equivalent. They show how one model may be mapped into the other and how performance is also equivalent. However, they don't directly address issues such as the model presented to the application programmer and its ease of use. Tam, et al., [178] use an analogy to claim that shared memory is a more powerful paradigm. If message-passing communication is viewed as similar to sequential I/O, then shared memory may be viewed as similar to random access I/O, which is seen as more powerful.

DSM is not without its own complexities and the simple model presented to the programmer is supported by a complex software or hardware structure. When a single piece of data is replicated in more than one place at which it can be accessed or modified, consistency maintenance becomes a problem. In tightly-coupled architectures, the consistency problem is usually addressed in hardware; in loosely-coupled architectures, consistency maintenance is usually addressed with a combination of software and hardware.

Chapter road-map

The bulk of this chapter is divided as follows: in §1 we discuss consistency issues, concentrating particularly on the relationship with data location and synchronisation issues; in §2 we discuss network latency and bandwidth as they relate to DSM systems; in §3 we discuss the other issues involved in implementing and using a DSM system; in §4 we discuss related work in other fields, covering the relationship between software DSM systems and hardware shared-memory multiprocessors in §4.1 and the relationship between DSM and database caching in §4.3; in §5 we look at some of the uses claimed for DSM; and finally, in §6, we attempt to draw some conclusions.

1 Data consistency

Despite recent advances in both local and wide-area networking technologies, network latency is still a major factor in distributed systems and is likely to remain so (see §2). All the DSM systems studied in preparing this survey provide some sort of caching in an attempt to improve performance beyond that provided by doing a network access on every reference to a non-local data item. Each DSM system must decide whether or not to attempt to keep the data consistent, and, if so, what consistency strategy to use. Due to the network-latency problem, the choice of caching and consistency algorithms is of prime importance. This section discusses the issues involved in maintaining consistency.

The first choice that needs to be made is the type of semantics to be provided to the application programmer. There are several options that form part of a range of possible semantic models:

- provide *strict* consistency, where a read always returns the value written by the most recent write—this is effectively *sequential consistency* as defined by Lamport [126];
- provide a *loosely* consistent system where the system enforces some form of weaker consistency guarantees and the application (or compiler or user) can indicate synchronisation points where consistency must be enforced; or
- enforce no fixed consistency mechanism, but provide the application programmer with the facilities necessary to implement user-level synchronisation and consistency.

Many older DSM systems provide strict consistency using the single writer, multiple reader model, where at any one time only one of the following will be true:

- the data is located at exactly one node, and that node has read and, possibly, write access to the data; or
- the data is replicated on two or more nodes, each node only has read access to its copy of the data, and none of the nodes have write access.

DSM systems that do not enforce this model are becoming increasingly popular, due to the performance benefits of relaxing the consistency model, and will be discussed in more detail later (§1.4). However, many of the issues discussed in this chapter apply to both strictly and loosely-consistent DSM systems.

Given the above readers/writer model, there are two problems that must be addressed. Firstly, we must locate a copy of the data when it is non-resident. Secondly, we must synchronise accesses to the data, so that no two sites are writing to separate copies at the same time. We discuss the data location problem in the next section and return to the access synchronisation issue in §1.2.

1.1 Data location

The most common solution to the data location problem is to assign an *owner* node to each item of data: the owner either has *the* writable copy of the data, or one of the read-only copies and a list of other nodes with read-only copies. The owner of the data is the *only* one allowed to write to the data. If other nodes have read-only copies at a time that the owner wishes to write, then we have a write synchronisation problem, which we will return to discuss in §1.2. Li and Hudak (authors of one of the first DSM systems, IVY [134, 135]) provide a good survey of ownership algorithms, and classify them as follows:

Fixed ownership Each piece of data is assigned a fixed owner; the owner can usually be calculated in an easy manner from the address of the data, for example, by hashing. Other processors are never given direct write access to the data, and must negotiate with the owner every time they wish to write to the data. The disadvantages of this scheme are obvious: there is a large overhead on every write access, and the data owner becomes a bottleneck in the DSM system. This is effectively a completely centralised algorithm.

Dynamic ownership The ownership of a data item moves around the DSM system, introducing the problem of locating the owner. In order to address this problem, we introduce the concept of a *manager* for an item of data. The manager need not *own* the data, but is responsible for tracking the current owner of the data. These algorithms can further be sub-divided into two classes:

  Centralised A node is selected as the *manager* of all the shared data. This manager keeps track of the current owner of each item of data. Other processors negotiate with the manager for access to the data. The manager can also keep track of those who currently have read-access to the data, and can queue requests for read or write access. Since read access can be shared, access is usually granted to any readers waiting in the queue when a read request reaches the head of the queue.

  Distributed Since a bottle-neck can easily form at the manager, the next obvious step is to split the ownership information among the nodes. Here we have a similar location problem as for the owner above, i.e., how do we locate the manager for a given piece of data? As before, we can have a *fixed* or a *dynamic* scheme for locating the manager of a data item. The fixed scheme operates as before, with some simple scheme for finding the manager of a piece of data from its address. Li and Hudak propose several dynamic schemes:

Broadcast Each processor manages those data items that it currently owns and management information migrates around with the data. Faulting processors send a broadcast message to find the current owner of each data item. Although this algorithm is simple and appealing, performance is poor due to the number of network messages involved, especially when the underlying network does not support hardware broadcasts.

Dynamic Each processor keeps track of the *probable* owner of each data item. This information is just a *hint* [181], and may be incorrect. If incorrect the hint at least provides the beginning of a sequence of processors through which the true owner may be located. Requests are forwarded until they reach the true owner. The probable-owner field is updated whenever more accurate information about the owner becomes available, such as when a data update is received (since such updates are always sent by the current owner of a data item).

Li and Hudak go on to show that, in the worst case, the number of messages to locate a data item K times, if p processors are using the data item, is $O(p+K \log p)$, if all the contending processors are in the same set of p processors. This result is important as it shows that the DSM system does not degrade as processors are added, but only as more processors contend for the *same* data item.

Li and Hudak also investigated the possibility of further improving the algorithm by sending a broadcast message intermittently to announce the true owner of a data item. They concluded that it was a useful improvement, and show that, immediately after a broadcast, the number of messages required to find a data item for K faults on different processors is $2K - 1$.

Almost all of the DSM systems studied use either the centralised-manager scheme or a variant of the dynamic-distributed manager scheme.

1.1.1 Data-location algorithm extensions

Among the modifications and extensions to the IVY algorithms that have been proposed are the following:

- In Mirage [64], a writer or a set of readers gain access to the data for at least a minimum time interval Δ. The authors claim that this modification reduces thrashing and increases overall throughput. They added a `yield` system call so that a processor can release the data before Δ has expired. This call is used to avoid stalling waiting processors when the holder of the data is no longer using it. The `yield` call improved the worst-case performance considerably. Of course, the need for such a call reduces the transparency and simplicity of the system as a whole.
- Another modification implemented by the authors of Mirage was to allow upgrades and downgrades in the access mode of data items throughout the network. For example, when

a site that has read access to a piece of data desires write access for the same data item, it just upgrades its read access to write access and then uses its local copy. Since Mirage implements multiple readers/single writer, the local readable copy is guaranteed to be up to date. Similarly, when a site loses write access to a piece of data, it keeps a read-only copy. These modifications save the cost of transferring a copy of the shared data when the local copy is known to be up to date.

This extension does not work in conjunction with distributed management, as a page may be invalidated locally and updated remotely while a request for a read-to-write upgrade is outstanding and, as a result, the local copy of the data may become out of date while the request is outstanding. In this case, a page transfer must take place on every write fault, even if a read-only copy is held [117].

- Some of the DSM systems discussed allow read-only portions of the shared address space. This is a useful extension as it reduces, or eliminates, the book-keeping overhead, and can be used for code and read-only data. Read-only sections of the shared address space can be replicated on demand and no write-access synchronisation or management is needed.

- Time-stamps can be attached to the probable-owner fields, so that the age and reliability of hints can be assessed. Also, processors can piggy-back some of their own hint information on other messages that they exchange. This is similar to the object identifiers and location time-stamps used in Emerald [109].

1.2 Write synchronisation

The other main issue in maintaining a strictly-consistent data set is synchronisation of write accesses to the data. This section discusses the issues involved in controlling access to shared data. When there are multiple copies of the data spread around the DSM system, it must be ensured that no processor reads stale data once a write has been completed on some processor. There are two main approaches: a processor can broadcast writes, or it can invalidate all other copies before doing an update. These two approaches are variously known as *write-broadcast* or *write-update*, and *write-invalidate* [176]. We discuss these two approaches in the next two sections.

1.2.1 Write broadcast

One possible solution is to broadcast the effects of all writes to all nodes that have copies of the data, effectively replicating data. If a low-level broadcast primitive is not available, then there will be a problem locating all the sites with copies of the data set. This problem will be examined in more detail later when discussing invalidation.

Write broadcast is usually considered too expensive to be used as a general solution: unless integrated with the language and the synchronisation system, a broadcast may be needed on every write. As a result, only one of the DSM systems discussed uses it. The Orca shared data-object programming language [15, 16, 180, 14] uses a reliable ordered-broadcast protocol to update all replicated copies of the data together. The use of write broadcast is more suitable in this case, as all updates to shared data are encapsulated in operations on objects, and can therefore be easily packaged up by the compiler and run-time system.

Some work has been done comparing algorithms that use broadcast-based replication with those using data migration [177]. Due to the expense of write-broadcast in unintegrated DSM systems, replication is not used by most general-purpose systems. Ordering of broadcasts is a problem, and replication algorithms are usually implemented using a single *write sequencer* site that broadcasts writes to all involved sites. Stumm and Zhou [177] outline a number of methods for increasing the performance of replication systems, but conclude that performance is bad in large systems and in systems where there are many updates.

1.2.2 Invalidation

The other major solution to the problem of synchronising write accesses is to *invalidate* all the other copies of the data item. The basic concept is to send a message to all processors holding a read copy of the data. Obviously, this can easily be done with broadcasts, but the expense of broadcasting can grow rapidly in large DSM systems and hosts may spend a lot of time handling broadcast messages.

If the DSM system is to use individual messages or multicasts instead, then we again have a problem of location: we need to locate the processors that currently have a copy of the data. The most common solution is for the manager site to maintain a *copy-set*: a list of processors with a copy of the data. Li and Hudak propose several methods for managing copy-sets, including schemes for compressing and representing very large copy-sets in large DSM systems.

They also propose a scheme for distributing the copy-set data: the copy-set data is represented as a tree of processors rooted at the owner node. Invalidations are passed down the tree. This modification also has the advantage that on a read fault a processor need only find *any* node with a read-only copy of the data. That node can add the new node to its copy-set data. In practice, the tree is bi-directional, with the links back towards the root formed by the probable-owner back reference.

If the write is allowed to proceed before the invalidation is complete, it is possible for non-current read-only copies of a page to exist for a short period of time. This factor can require invalidations to be acknowledged if strict consistency is required.

1.2.3 Other possibilities for write synchronisation

Nearly all of the systems examined use variants of either write broadcast or write invalidation. This section examines some of the other alternatives used.

In the Clouds system [21, 159, 160, 158], neither invalidation nor write broadcast are used; instead, a lock must be obtained when acquiring access to the data, so conflicting writes are not possible. When releasing the lock, the associated data segment must also be discarded, i.e., access to the data must be relinquished. In an argument similar to the traditional end-to-end argument in operating systems design [167], the designers of the Clouds system claim that all shared-memory access will be controlled by a higher level synchronisation system in any case. (See §3.1 for more details.)

Research done on the theory of *snoopy caching* [111] would indicate that a combination of write broadcast and invalidation might be successful, but no DSM system has so far examined this option. (Snoopy caching is a multi-processor caching system where hardware caches watch transactions on the system bus and update their internal information accordingly [176].)

In Mether [145], there is only ever one consistent copy of a page that is moved around the DSM system. Read-only copies are not maintained consistently, and so must be updated explicitly. There are three ways to update a page: request the consistent copy; the holder of the consistent copy may cause a broadcast of the current contents; or, the local copy may be purged, causing the page to be fetched when next accessed. Mether also adds the concept of a *data-driven page fault*. A process may purge its local copy of a page, and block awaiting an update to be broadcast from the holder of the consistent copy.

1.3 Double faulting

Double faulting describes a form of behaviour common in many DSM algorithms. If a process first reads and then writes a page (such as to read and update a counter), many DSM algorithms will incur two network transactions: one to obtain a read-only copy of the page and a second to obtain write access to the page. This will usually result in the page itself being transmitted twice over the network.

The double-faulting problem is considered separately to the data-location problem by Kessler and Livny [117], who discuss several algorithms that attempt to solve this problem:

Hot Potato All faults are treated as write faults, and the ownership and data are transferred on every fault, resulting in one and only one copy of each page at any time. The performance disadvantages are obvious.

Li Li's most complex algorithm from IVY circumvents this problem by always transferring ownership on a read fault. A subsequent write fault may then be handled locally.

Shrewd This algorithm eliminates all unnecessary page transfers with the help of a sequence number per copy of a page, but does not eliminate the extra network transaction. The sequence numbers (or version numbers) are used to track whether a page has been updated since the read-only copy was obtained and therefore needs to be transferred when the request for write access is made. However, network transactions are still required to obtain access and check the sequence numbers.

Kessler and Livny go on to analyse the algorithms with the help of a simulation, and conclude that neither the Li nor the Shrewd algorithm can be said to be superior in all cases. The crucial factor appears to be the read-to-write ratio of the application. A combination of the two algorithms is possible, but hasn't been examined.

1.4 Relaxing consistency

Permitting temporary inconsistencies is a common method of increasing performance in a distributed system [41]. This section examines the various approaches that have been taken to allowing inconsistencies in DSM systems.

The designers of the Munin DSM system [20, 33] examined the issue of strict consistency (as defined in §1) very carefully and came to the following conclusion:

> A shared-memory parallel program specifies only a partial order on the events within the program, both through explicit synchronization between threads of the program, and through implicit knowledge within one thread of the order of events in another thread.

Based on this, they developed a more relaxed definition of memory consistency:

> Memory is **loosely consistent** if the value returned by a read operation is the value written by an update operation to the same object that *could* have immediately preceded the read operation in some legal schedule of the threads in execution.

This definition of loose consistency is somewhat related to the concept of *virtual synchrony* in the ISIS programming environment [25]. In ISIS, message operations appear to be synchronous, i.e., to happen everywhere at the same time. Messaging is implemented using standard asynchronous primitives but complex protocols [24] are used to prevent the application from seeing asynchronous behaviour. Loose consistency is also related to the concept of *serialisability* in transaction systems [81].

By enforcing only the definition of loose consistency given above, Munin is able to delay sending updates until remote threads might see incorrectly ordered updates to shared data. This mechanism of *delayed updates* allows Munin to batch updates in a queue and send them out only when necessary. When the system being considered is not entirely based on DSM, detecting when objects need to be updated is not easy: there is no guarantee that programs will not exchange information outside the DSM system. This is a standard distributed systems problem when the system is not *complete* and is not considered further by most DSM implementors.

When combined with the type-specific semantics described in §1.5, the loose-consistency mechanism is quite powerful, as it can make use of some semantic knowledge about the objects involved. This semantic knowledge can be used to avoid performing updates to data that will not be accessed and to delay performing updates until actually needed. Of course, these approaches compromise the shared-memory concept somewhat and increase the complexity for the programmer.

Stumm and Zhou [177] describe a full-replication (write-broadcast) DSM system that uses delayed broadcasts to improve efficiency. They also describe an *optimistic* replication system where threads are permitted access to shared data before any checking is done. The system assigns version numbers to shared data updates and logs all accesses to shared data to a transactional system. If a thread has seen inconsistent data, it will be rolled back to a previous consistent state.

1.4.1 Release and entry consistency

Gharachorloo, et al., [72, 71] present another memory model for multi-processor cache consistency, known as *release consistency.* In a release-consistent DSM, memory accesses are divided into ordinary and synchronisation-related accesses. Synchronisation accesses are further divided into *acquire* and *release* operations. The *acquire* operation signals that shared data is needed; a processor's updates after an *acquire* are not guaranteed to be performed at other nodes until a *release* is performed. The primary advantage of release consistency is that it allows updates to be tied to synchronisation events and therefore delayed until actually needed by applications. The primary disadvantage is that most release-consistent DSMs require the programmer to make explicit use of *acquire* and *release* operations and incorrect use of the operations can lead to nodes reading stale data or updates being lost. However, these types of operations, or similar synchronisation operations, are required in any concurrent program in order to ensure correctness (see §3.1).

Release consistency is particularly well suited to software DSM systems, and variants of this approach are used in several recent DSM systems. More recent versions of Munin [32, 33] use release consistency combined with the type-specific support described in §1.5. A number of people from the Munin team at Rice have also been working on a variety of *lazy* release-consistent algorithms [115, 55], where the transmission of updates are delayed as long as possible, and have compared these with *eager* algorithm that propagate updates rapidly. They have developed a hybrid algorithm that has good performance for a range of applications. They also examined available networking hardware and concluded that, while relaxed consistency, lazy algorithms, and multiple-writer protocols helped reduce the effects of *false sharing* [29], latency (and specifically *synchronisation latency*) was still a serious problem. Keleher, et al., [114] have implemented a lazily release-consistent DSM system called TreadMarks and have obtained good speedup results for a variety of applications running over ATM network hardware.

A number of other variants of release consistency have been suggested, each making the shared-memory abstraction more complex and less transparent in return for better performance. Feeley and Levy [61] present a modified release-consistent DSM system that supports object-based *version consistency* where applications are permitted to *acquire* specific versions of objects. Versions are immutable and an *acquire* for a specific version will block until a corresponding *release* has been performed. This approach permits multiple versions of data to co-exist in the DSM system and helps solve false sharing problems that occur in some types of parallel applications.

Bershad, et al., [23] have implemented a DSM system called Midway that introduces another new consistency model, *entry consistency.* Entry consistency is weaker than many of the other models suggested, including release consistency; entry consistency requires explicit annotations to associate synchronisation objects with data. On an *acquire*, only the data associated with the synchronisation object is guaranteed to be consistent. This extra weakness permits higher performance implementations of the underlying consistency protocols to be written. Midway also supports stronger consistency models, so that the application programmer can trade-off performance against the extra effort required to write entry-consistent programs. (The DiSOM DSM system [78, 34] is also based on entry consistency.)

Kazimierczak and Andersen [113] propose a new model for DSM consistency maintenance. Their *object-consistent* system combines attributes of both entry consistency and version consistency as described above. Their system has yet to be implemented, but they hope to support the best attributes of both systems.

Iftode, et al. [99] introduce scope consistency. Scope consistency associates synchronisation mechanisms and data similar to entry consistency. The association however is not explicitly made by the developer. The developer declares the beginning and the end of a scope. The system dynamicly detects the asssociation between data and scopes. The consistency of data is ensured in corresponding scopes.

Ananthanarayanan, et al., [9] claim that neither release consistency nor type-specific consistency (as implemented by Munin) are sufficient to solve the consistency problem for some applications. For example, with algorithms like the bounded buffer, both invalidation and write broadcast are inefficient since each new data item is only read by a single node before another update is done, and the synchronisation system will know which node will be accessing and removing the object from the buffer. They suggest that the update mechanism needs to be linked to synchronisation of access to shared variables and propose an extended set of primitives that support the implementation of an *Augmented Shared Memory* (ASM) system in the Clouds environment. The approach they propose is very similar to entry consistency and links synchronisation variables to specific data items.

1.4.2 Slow memory

Hutto and Ahamad [88, 87] present a theoretical examination of *slow memories* (defined below), and other forms of weakly-consistent memory. They present several reasons in favour of weakly-consistent memory. Firstly, they claim that strictly-consistent memories must be "slow", quoting Lipton and Sandberg's Latency Theorem [137]:

> No matter how clever or complex a protocol is, if it implements a consistent shared memory, it must be "slow". A consistent shared memory cannot be both fast (memory access time independent of the message latency, τ) and scalable.

In a similar argument to that presented by Clouds, they also claim that access will always be controlled by a higher-level synchronisation system, and therefore that strictly-consistent memory implicitly enforces unnecessary consistency.

In the cited papers, Hutto and Ahamad discuss various levels of consistency, categorised by various properties. They present two example memory types:

Slow memory: Reads must return *some* value that has previously been written to the location being read. Once a value has been read, no earlier writes to that location (by the processor that wrote the value read) can be returned. Local writes must be immediately visible.

Slow memory provides *very weak* consistency, as even writes by the same processor to different locations are not ordered. As a result, programming with slow memory is rather unusual. For example, traditional mutual exclusion algorithms do not work, though mutual exclusion can be achieved using modified algorithms and a working variant is presented.

Causal memory: All processors must agree on the order of causally-related effects. Events not related by potential causality (concurrent events) may be observed in different orders. In a later paper, Ahamad, et al., [7] examine the issues involved in actual programming of causal memories. They conclude that most programs written for consistent memory work correctly in causal-memory systems as long as synchronisation is done through shared-memory variables. They also note that for some algorithms, minor changes can be made to implement elegant solutions that make use of the properties of causal memory, particularly if the programmer is allowed to specify the procedure for merging concurrent writes.

Causal memory is closely related to the relaxed-consistency mechanisms discussed earlier, especially virtual synchrony in ISIS. In fact, Schmuck has proved that a large class of problems for which a *linearization function* can be found all have causal (ISIS cbcast) implementations [172] and, therefore, could also be implemented using causal memory.

In the next section, we move on to survey consistency systems that make use of some application-specific information in order to achieve higher performance.

1.5 Application/type-specific consistency

In the database arena, a fair amount of effort has been expended studying the effect of type and type-specific support on transactional systems. Research has shown that by adding some semantic-level knowledge of the application-level types in use and their inter-operation, performance gains can be realised. Schwarz [174] and Weihl [199] both cover the theoretical aspects of performing transactions on typed objects. Harty [81] concentrates on the issues involved in synchronisation and recovery when using type-specific concurrency control. Resende, et al., [166] propose a semantic-locking protocol for transaction management in an object-oriented database; by making use of semantic information, their protocol can reorder transactions, cope with complex objects, nested transactions, and referentially-shared objects.

A similar application-specific approach has been proposed for use in DSM systems. The issue was first raised under the title "Problem-oriented shared memory" by Cheriton [44]. Cheriton claims that there are three basic paradigms for handling shared state:

- Communication paradigm.

- (consistent) Shared-memory paradigm.
- Problem-oriented shared-memory paradigm.

To quote [44]:

> *Problem-oriented shared memory* is a shared memory that implements fetch and store operations specialized to the particular problem or application it is supporting. In particular, a problem-oriented shared memory commonly provides a specialized form of consistency and consistency maintenance that exploits application-specific semantics.

Cheriton goes on to propose that consistency constraints be relaxed and more use be made of problem semantics. He suggests that, in some cases, stale data may be detected on use by the client, and the client may then recover. A example would be hint caching. In some applications, stale data may actually be sufficiently accurate, provided that the client can obtain up to date information when necessary. In other applications, some data may be optional in the sense that the client can continue without it. Other applications may tolerate having the results of store operations being lost or undone, for example, an application that regularly replaces the entire data set.

The first DSM system to implement application-specific support was Munin [20]. Munin[1] supports multiple-consistency protocols and the runtime system accepts hints from the compiler or user to determine the consistency protocol to be used for each object. The default, in the absence of hints, is to use a general multiple-readers/single-writer consistency mechanism, much like that employed by IVY. Munin supports several different object types that are based on the results of a survey of shared-memory access characteristics. The main types of objects they identified are (from most specific to least specific):

Synchronisation: Lock objects, which may be managed using the distributed lock system described in §3.1.

Private: Objects identified as private are not managed by the runtime at all. However, if the object is remotely referenced, the object is brought back under the control of the DSM system.

Write-once: Objects that are initialised once and then only read, and so may be efficiently replicated.

Result: Objects that are not read until fully updated. The DSM system knows that updates to different parts of the object will not conflict, so it can use the delayed-update mechanism to maximum benefit.

Producer consumer: Objects that are written by one thread, and read by a fixed set of threads. Munin performs *eager object movement* for this type of objects, where updates are transmitted to the readers before they request them.

[1] Note that there are several different versions of Munin documented [20, 32, 33]. This section discusses the original version initially. Interesting modifications are discussed elsewhere.

Migratory: Objects that are accessed by a single thread at a time. These can be efficiently handled by integrating the movement of the object with the movement of the lock associated with the object, giving some of the benefits of the Clouds scheme.

Write-many: These are objects that are written by many different nodes at the same time, sometimes without being read in between. Write-many objects may be efficiently handled using delayed updates (see §1.4).

Read-mostly: These objects are updated rarely, so a broadcast scheme may be used for updating them.

General read-write: The default case, used when multiple threads are reading and writing the object, or when no hint of object type is given. The scheme used to handle this type of object is similar to the ownership scheme used in IVY that is itself a modification of a multi-processor caching algorithm called the Berkeley algorithm (see §4.1).

The results of the survey done by the authors showed that a very small percentage of all accesses to shared data fall under the general read-write type. They also note that a program moves through various stages of execution, and the types associated with objects change as time progresses. The authors suggest extending their runtime system to attempt to determine the type of an object dynamically, possibly using profiling information gathered by the DSM system.

In a subsequent paper [19], Bennett, et al., study a number of parallel applications using a simulation system. At small granularities (which they call *element* access), it would appear that many accesses do fall under the various categories outlined above. However, at larger granularities (*object* access), almost all accesses seem to fall under the *WriteMany* and *WriteOnce* categories; these are the classes of access most likely to be used in a page-based DSM system. The conclusion is that page-based access is at too large a granularity to allow fine type-specific tagging to be used.

In more recent versions of Munin [33, 32], the initial list of eight different access patterns is reduced to five: conventional, read-only, migratory, write-shared, and synchronisation. The original set of access patterns contains some redundancy. As noted in §1.4.1, more recent versions of Munin also support release consistency, and in conjunction with this were extended with a new write-shared protocol based on page duplication and differencing. Carter, et al., [33] study the performance benefit gained from supporting multiple consistency protocols. For the application that they study, a 30% performance improvement is obtained from using multiple protocols when compared with using their most flexible fixed protocol (write-shared with page differencing).

Another approach to application-specific support is to allow the programmer to control, at some level, the operation or implementation of the underlying DSM system. This approach draws on the open implementation (or meta-protocol) arguments presented by Kiczales, et al., [118, 119] proposing that the underlying implementations of an operating system service must be accessible and tunable by the application in order to maximise performance.

Kulkarni, et al., describe the construction of a DSM system using the 'Π' architecture [124] that allows the DSM system to be tailored to suit the varying needs of applications and hardware configurations using a meta-protocol approach. Taking a more direct approach to supporting DSM specifically, Pérez-Cortés, et al., [157] describe the Arias DSM service. Arias divides the shared address space into multiple zones; each zone may have its own synchronisation and consistency protocol, built using generic primitives provided by the system. Similar systems include: Lee, et al., [132] who present a user-level virtual-memory caching system providing a similar level of application-level control; Engler, et al., [56] who present a completely user-level implementation of a virtual-memory system constructed using the Aegis exokernel [57]; Feng [86] who presents an OS architecture that supports typed objects and user-level consistency control; and Larus, et al., [128] who present a compiler-controlled memory system allowing the compiler to control the replication and merge policies of cache-consistent shared memory.

Griffioen, et al., [77, 76] propose an novel application-specific alternative called *spatial consistency* for use in the Unify system. In a spatially-consistent system, the consistency algorithms can be linked to the application-level geometric meaning of the shared data. In this way, data that is *further away* from the node's workspace can be less consistently maintained than data that is nearby. This approach has application in many geometrically-based parallel applications.

The file caching mechanism [187] in the Lucas system [112] supports a protocol customisation system that introduces a high-level model for cache-consistent protocol descriptions, allowing the application programmer to briefly describe protocols that are adapted for a particular application. A similar approach could be taken in a DSM environment.

In the next section, we go on to discuss the effect that network performance has on the performance of a DSM system.

2 Network performance issues

Communications latency is generally much larger in a DSM system than it would be in a multi-processor, and bandwidth generally much lower. This leads to different design decisions in areas such as consistency and cache management where the increased latency increases the cost of operations, such as data invalidation, which are relatively cheap in a multi-processor. Latency also limits the rate at which threads on different nodes can communicate and synchronise.

It is interesting to note that recent advances in both local and wide-area networking technologies (such as ATM [143], SONET, switching technology [11]) have had a significant effect on bandwidth and latency. In particular, bandwidths have climbed steeply and speeds of over 1Gbps are available even over wide-area networks. Meanwhile, end-to-end latencies have dropped, but not as steeply. This has narrowed the performance gap between hardware and software shared-memory systems somewhat, resulting in wider applicability of DSM technology and increased interest in research in the area.

However, latency is still an important issue and limits the speed that a DSM system can reach. In both local- and wide-area networks, latencies will remain much higher than in tightly-coupled architectures. This will remain the case even in the presence of increasing bandwidth [185] and will continue to limit the degree of convergence of shared-memory systems for tightly- and loosely-coupled architectures. Work is being done to improve the latency of local-area systems in particular, since wide-area systems are limited by the speed of light in any case, and, using ATM networks in particular, local-area network latencies have dropped by several orders of magnitude [197, 154].

As processor speeds climb, the relative importance of network latencies has also climbed. Where 10 years ago a processor might have only been able to complete a few thousand instructions during a network round-trip, newer processors can complete from one to several orders of magnitude more instructions in the round-trip time of newer networks. For applications running in parallel on such systems (including parallel applications running on DSM systems), a key factor is the computation-to-communication ratio [50]. Applications with high computation-to-communication ratios generally perform well on loosely-coupled architectures, whereas applications with low ratios generally require the low latency of tightly-coupled architectures to achieve good speed-up ratios.

This is analogous to the *memory wall* that researchers such as Wulf and McKee [202] and Mogul [146] are worried about in traditional hardware design. As in the case of network latencies (and to a lesser extent bandwidths), processor speeds are climbing faster than memory speeds. Wulf and McKee speculate that the limit where memory speeds will dominate, and processor speed increases will no longer increase overall performance, is less than a decade away. A radical architectural rethink is needed.

In the next section, we go on to discuss a number of other issues that effect the design of DSM systems.

3 Other design issues

Aside from the primary issue of consistency, there are a number of other design issues that must be discussed.

3.1 Synchronisation

Most parallel applications will use some sort of synchronisation mechanism to order and control accesses to shared data. The most important thing to note in DSM systems is that just blindly using standard test and set operations on bytes in shared pages will produce a high fault rate; faults are usually expensive, making this approach unacceptable. The effect of any synchronisation primitives on the fault behaviour must be carefully considered.

As previously mentioned, the designers of the Clouds DSM system [159] maintain that the synchronisation primitives, such as semaphores, should be integrated with the data consistency protocol. Clouds merges locking with the cache consistency protocol, so that the user may obtain both a lock and the data in one network transaction. This approach has the advantage that no invalidation messages are required, since the granting of the lock guarantees that there are no conflicting copies; it has the disadvantage that an explicit unlock/discard operation is required to release access to the data. This is acceptable in Clouds, as the DSM system was designed specifically to support object invocation and the data can be discarded and unlocked when the invocation returns. In other DSM systems, the run-time, compiler, or application would need to be modified to discard data items at appropriate times.

IVY [134, 135] provides *eventcounts* [163] as the basic synchronisation primitives. These are implemented outside of the DSM system, using RPC.

Mermaid [210, 209] implements a separate distributed synchronisation facility providing P and V operations and events.

Munin [20, 32, 33] provides a distributed lock mechanism using *proxy objects* to reduce network load. Proxy objects are maintained by a lock server on each node; when a thread wants to obtain a lock on an object, it attempts to lock the proxy instead. The server obtains the global lock if it is not already held locally. Global locking is done by negotiating with all the other lock servers in the system. Each lock may be migrated from server to server, and part of the Munin system allows objects to be migrated along with their locks: see §1.5 for more details.

Anderson, et al., [10] suggest locking using an Ethernet-style back-off algorithm instead of spin or wait locks in a shared-memory environment. This approach reduces the amount of communication and contention in the shared memory area where the locks are held. Goodman, et al., [74] propose a set of efficient primitives for software synchronisation on a multi-processor that may be extendible to the DSM case. Their system uses busy waiting only on local memory (which is efficient) and uses broadcast and queuing operations to achieve global synchronisation. Lee and Ramachandran [133] suggest building a lock-based protocol into the cache-consistency system in a multi-processor; by making use of broadcasting and bus snooping, they avoid the cache contention that traditional busy waiting would cause. Graunke and Thakkar [75] survey and compare a wide range of synchronisation protocols applied to multi-processors. Johnson [108] compares a number of fast distributed mutual-exclusion algorithms.

Another important issue, especially when relaxed consistency is used between synchronisation points, is the granularity of synchronisation used. A simulation study by Chen, et al., [42] would indicate that, for general parallel systems, synchronisation at the loop-iteration level gives the best performance. Unrau, et al., [188] discuss a new multi-processor operating system called Hurricane that uses a hybrid system combining coarse and fine-grained locking to achieve good results. How these results apply to DSM systems, where the synchronisation overhead is generally higher, has yet to be seen.

As an interesting alternative approach, Bershad [22] suggests implementing concurrently-accessible objects without using locking at all. Instead, update protocols are constructed using an atomic compare-and-swap operation to guarantee consistency.

In the next section, we discuss data-granularity issues.

3.2 Granularity

When caching objects in local memory, it is necessary to decide what level of granularity to use. Many of the DSM systems discussed use a fixed block size in the cache, rather than varying the granularity based on the size of the data item. Usually this is due to constraints imposed by the system hardware and memory management, as many DSM systems use virtual-memory hardware to trap accesses to shared data.

The choice of the block size in the cache depends on several issues. It depends on the cost of communication: for example, on many local area networks there is little difference between the time required to send a 1 byte message and that required to send a 1024 byte message. Transmitting bulk changes rather than single-byte modifications would therefore seem desirable. The choice of granularity also depends on the locality of reference in the application, as thrashing may occur when two machines are both accessing the same block (this is also known as *false sharing* or the *ping-pong effect*). This would seem to argue for a smaller block size. In practice, a compromise must be achieved, as with conventional virtual-memory systems.

There are good reasons for choosing a block size that is the same as that of the virtual-memory management unit on the system, or a multiple thereof. Among other things, it allows the hardware to be used to help in the maintenance of consistency. The choice is complicated somewhat when heterogeneous machines are being used, but in these cases, the lowest common multiple of hardware supported page sizes can usually be used, e.g., in CMU's shared-memory server [65, 66] built on top of Mach [1, 182, 205, 162]. Most of the DSM systems discussed use a page-based granularity with 1K byte to 8K byte pages. There is evidence that larger page sizes than this cause locality of reference problems [134, 135].

The only major DSM system that uses a fixed small granularity is Memnet [53, 178] where a hardware based DSM system was implemented on a high speed token ring. Here they used a 32 byte block (or chunk, as they call it). The choice of a small block size is appropriate as Memnet is much closer to a shared-memory multi-processor than it is to a software DSM system, because the entire processor is blocked on a cache miss and a block size similar to cache line sizes in multi-processors is used; the processor is not actually aware of the distributed nature of its address space. Also, the ratio between remote and local memory-access times is much lower than in the software based DSM systems due to the dedicated token ring (200Mbps) and hardware assistance. In contrast, Mether [144, 145], a software implementation of Memnet on top of SunOS, uses an 8K page size. However, the concept of a *short page* was later added to Mether, allowing access to the first 32 bytes of a full page.

A number of DSM systems avoid using hardware assistance to trap accesses to shared data and therefore are not constrained to use hardware page sizes to monitor access to shared data. Orca [180] and object consistency [113] (among others) use language-level objects as their fundamental granularity and therefore use a variable granularity. Zekauskas, et al., [208] study the use of software write-detection at a lower level using compiler and run-time extensions, citing the cost of hardware faults as a significant issue in many systems. They conclude that software write-detection is faster, sometimes by a considerable margin, and that less data is transferred over the network, due to the finer granularity of access tracking. However, they concede that the handling of hardware faults is quite slow in the system that they study and that software write-detection is not always faster in systems with lower hardware overheads. Hosking and Moss [85] also study the issue and conclude that software traps have lower overhead than hardware traps.

In the next section, we discuss address-space structuring issues in designing DSM systems.

3.3 Address-space structure

Another important issue is the structure of the address space supported by the DSM system. The single most important factor here is whether or not the user is presented with a single shared address space. A secondary issue is how the shared region itself is organised.

3.3.1 Single shared address-space systems

In a single shared address-space system (or single-level store), the system appears as a set of threads executing in a single shared distributed address space. Data items always appear at the same addresses on all nodes. This is the Amber model of computation [39]. In IVY [134, 135], each address space is split into a private and a shared region; there is a single shared region for all processes. The Arias system [157] supports a single global address space made up of shared zones, each of which may be persistent.

Single address-space systems have had a resurgence in popularity with the arrival of 64-bit processors. A number of researchers believe that a 64-bit address space is large enough to act as a single global address space for all the memory (both primary and secondary) in a distributed system. Examples of such systems include Psyche [70], Opal [40, 123, 38], Lucas [103], and Angel [150]. Single address space systems are particularly popular in the object-oriented community as they allow secondary storage to be unified with primary storage and remove the need for pointer swizzling.

Security and protection are a major problem with single address-space systems, and current approaches either rely on hardware assistance (Opal and Mungi [196]), software capabilities (Arias) or probabilistic algorithms [204].

3.3.2 Separate shared address spaces

Another approach is to divide each process's address space into different fixed regions, some of which are private and not shared, and some of which are shared with some other processes. Ra, the Clouds kernel [21], takes this approach using O, P, and K address regions, with the O region shared between all processes executing in a given object; the P and K regions are local to a process and kernel respectively. Here objects always appear at the same address but may not be visible from every address space.

By contrast, some DSM systems (including Mirage [64] and CMU's Mach server [65, 66]) allow shared data to exist at differing addresses in different processes address spaces. However, neither system does transparent pointer translation, so the address changes are not entirely transparent to the application.

3.3.3 Shared-region organisation

The other important issue is the structuring of the shared region itself. Some of the DSM systems discussed—for example, IVY and Mether—use a single flat region: one continuous range of virtual addresses represent the shared address space and are managed by the DSM system. This single address space is usually sub-divided into pages.

Most of the other DSM systems (Choices [106], Clouds, Mach, Mirage, and Munin) use paged segmentation: the shared region consists of disjoint pieces that are usually managed separately and are not all mapped in any one process. Frequently, the segments (sometimes called memory objects, or windows) are related to the backing store. For example, in Clouds, the object address space consists of windows onto larger segments; these segments are usually maintained on secondary storage.

In the next section, we discuss network protocol issues in designing DSM systems.

3.4 Network protocol

The choice of network protocol to use for communication between different parts of the DSM system is important since it will effect the operation of the DSM protocol itself. For example, if an unreliable message layer is used then the DSM layer must handle errors itself.

The authors of Choices [106] (repeating Saltzer *et al.* [167]) argue that by using an unreliable datagram-based message layer in combination with timeouts and a problem-specific error-handling system, a faster DSM system may be built. They make use of application-specific information in a state-based, event-driven protocol specifically designed for the DSM system. This problem-specific protocol has advantages over a heavy-weight generalised reliable protocol. Similarly, Mether uses UDP (an unreliable datagram protocol supported by most UNIX systems) with timeouts to handle lost packets.

In contrast, some of the other DSM systems discussed use a reliable request-response message layer and move error handling into the message layer. For example, Clouds uses TAL/RaTP, which is a somewhat like V's VMTP [45] but simpler. Munin and Mirage also use reliable messaging.

Another approach is suggested by Comer and Griffioen [47] who use a dedicated communications protocol (RMCP) to provide remote memory-based swap and paging space to clients on a local-area network.

Orca [15, 16, 180] stands alone as the only DSM system that consistently uses broadcasts. They use a reliable ordered-broadcast protocol to distribute information about updates.

IVY is the only one of the DSM systems to use RPC as its basic communication mechanism, an approach that has disadvantages when requests are forwarded: the reply must be returned via the forwarding node, rather than directly to the requester. Also, the number of messages sent is sometimes increased by acknowledging messages that do not need to be acknowledged, since most RPCs have a return message for every call. Of course, by using a known and stable message-handling layer, IVY doesn't need to worry about message loss, message buffering, acknowledgments and other message-layer issues.

In the next section, we discuss how DSM systems interact with virtual-memory management and secondary-storage systems.

3.5 Replacement policy and secondary storage

Since the memory of each machine in the DSM system is effectively a cache of part of the shared data set, there must be a policy to deal with cache overflow. The usual least-recently used (LRU) policy used in traditional virtual-memory systems does not work well without modification, as some pages that may have been referenced very recently should be replaced; for example, a page that has been invalidated since its last use should be replaced before any page that contains valid data. Also, pages that are read-only and replicated elsewhere should have a high replacement priority as they do not need to be saved to disk before being replaced. Pages that are writable, or do not have replicas on other machines, must be saved.

A related issue is the location at which to save pages that are being replaced. Traditional virtual-memory systems always save replaced pages on local disk, but, in a DSM system, we also have the option of saving pages into the main memory of other nodes. With the trend to have machines with larger and larger memory sizes and prospect of even faster networks, this option may become more attractive.

Markatos and Dramintinos [140] study the use of main memory in other nodes to store pages. They conclude that this is faster than disk storage and they also discuss an algorithm that allows this remote paging to be fault-tolerant in the presence of a single-node failure.

Malkawi, et al., [139] examine the choice of pages to replace and present a number of algorithms that they evaluate using a trace-based simulator for both cube and mesh-based multi-processors. Their results suggest that when the cost of inter-node faults is low, it is best to pick replacement pages from the working set of the current process; when inter-node faults are costly, it is better to pick pages that belong to other local or remote processes. When memory contention is high, the best algorithm for selecting a node to send the page to is the least-active neighbour; round-robin and least-loaded neighbour algorithms have lower performance unless local memories are very large.

Memnet [53, 178] is unique among the DSM systems discussed in that it actually reserves part of the main memory of each node as storage for chunks, so that each chunk is guaranteed a storage location on some node when it is pushed out of another machine's memory. The authors of IVY investigated the possibility of replacement to other nodes memories and proposed piggy-backing memory-state information (such as the amount of free memory) on other network messages.

In the database field, Franklin [68, 67, 31] discusses similar issues that arise in client-server database-cache management. He discusses a wide range of algorithms, including algorithms that cache in other workstation's memory or cache on local disk. Some algorithms depend on transaction support and may not be suitable in every DSM system. When evaluated using a simulator and a wide range of workloads, no clear winner emerges, though local-disk caching proved to be worthwhile for scalability reasons. Franklin does not consider systems with multiple *server*s, so it is not clear how his results would scale to such systems.

Dahlin, et al., [51] simulate the use of remote client memory for caching files in a distributed file system. They present and analyse four different algorithms and conclude that remote client-memory caching is very useful. In the best cases, the number of disk accesses is halved and read response time drops by 73%. Their algorithms could be easily adapted for use in a DSM system that uses secondary storage as backing store for shared data.

Feeley, et al., [62] study the use of *global memory* (i.e., all the main memory in a workstation cluster) in the more general case, where global memory is used for everything in the OS: file buffer caching, VM for applications, and transactions. Their system runs on a cluster of machines connected via an ATM LAN. For memory intensive applications, performance improvements of a factor of 1.5 to 3.5 have been measured.

Of course, the underlying operating system may not allow control over the precise replacement policy used. This is an example of a general issue that must be considered in designing a DSM system: the relationship between the DSM system and the underlying operating system, if any. Most DSM systems have either been built into specialised operating systems or been built on top of operating systems such as Mach that allow considerable control over the memory management system. However, there can still be problems. For example, Mach only allows control over what is done with a page once the OS decides to page it out. In this case, however, a solution is available: Mach has been extended by McNamee and Armstrong [141] to allow user-level page-replacement policies.

In the next section, we discuss those DSM systems that support more than one hardware platform and those that support the automatic translation of data between hardware types.

3.6 Heterogeneity support

A few of the DSM systems discussed make attempts to provide support for heterogeneous systems, including: the Mermaid system [210, 209], CMU's Mach server [65, 66], and DiSOM [78, 34]. Mermaid is implemented between Sun workstations running SunOS and C, and DEC Firefly shared-memory multi-processor workstations running Topaz and Modula-2+. Mermaid is truly heterogeneous, while CMU's system assumes a homogeneous operating-system base. The Mermaid and CMU systems are page-based. DiSOM is an object-oriented DSM system that supports a number of different host systems. The first part of this section discusses heterogeneity issues for page-based systems; we return to language-level systems at the end of the section.

3.6.1 Page-based heterogeneity

The first heterogeneity issue is to do with support for different page sizes without considering the data contents of the pages; this is not strictly support for heterogeneous machines, as Mach permits the page size to be selected on a per-machine basis. The Mach page-size algorithm was as follows:

- for requests *smaller* than the scheduler[2] page size, the request is rounded up to the scheduler page size; and,
- for requests *larger* than the scheduler page size, the request is fulfilled by multiple scheduler pages (all shipped at once).

This problem is not as simple as it sounds, as difficulties arise when trying to avoid false contention between competing multi-page requests, and when trying to satisfy requests rapidly while maintaining fairness.

In Mermaid, Zhou, et al., discuss two algorithms for handling the page size problem: the *smallest page-size* algorithm, and the *largest page-size* algorithm, i.e., use either the smallest or the largest page size that occurs in the DSM system. Using a large page size can increase thrashing and *false sharing* [29], where non-overlapping accesses appear to overlap due to large page granularity. On the other hand, using a small page size can increase the overheads. The performance results they present show that the *larger* algorithm has better performance, but that the *smaller* algorithm has better stability in the presence of data interference problems.

The second issue is that of real heterogeneous-data support and goes beyond the byte-order and word-size problems faced by many network applications: within pages of virtual memory, machines are free to assign any meaning they like to a sequence of bits. The authors of the CMU DSM system divide the problem into two sub-problems: hardware data-types, such as integers and floating-point numbers; and software types, such as C structures. Their current system takes care of hardware-type translation and should be extensible to take care of software types. They do not address the issue of variable-size data and pointers.

The CMU DSM system works by attaching a type tag to each segment of the shared-memory object, and provides translated segments as required. At the application level, a typed memory allocator is used to place data in the appropriate regions. While this approach cannot deal with structures directly, it can handle them if an extra level of indirection is added: each structure element is replaced by a pointer to storage allocated to store that data type. Another suggestion that they make is to pretranslate pages on (multi)processors with spare memory and CPU cycles.

[2] In CMU's scheme, the manager node is known as the *fault scheduler*, as it is responsible for scheduling a series of read and write requests.

A similar scheme is used in Mermaid, with each page containing only one data type. However, user level conversion routines are supported, so the types involved may be arbitrarily complex. The main limitations of page-translation systems are:

- the size of the data type must be the same on all involved DSM systems; and
- there is usually a loss of precision when converting floating-point data types.

3.6.2 Language-assisted heterogeneity

In DiSOM, the entire system is object-oriented and operates at the language level. Heterogeneity is addressed by requiring the application programmer to provide customised packing and unpacking routines to marshal the data contained within a given object into host-independent form. (Though, for some uses, these routines could be generated by the compiler.)

Another approach is described by Gokhale and Minnich [73]. In their DSM system, they use a specialised compiler that supports IEEE 1596.5 machine-independent types. This allows most of the work to be done at compile time instead of run time. The main problem is with performance: for example, load/store is 2–6 times slower. Their system is implemented using C++ operator overloading. Pointers are not handled.

In the next section, we discuss fault tolerance as it impinges on DSM system design.

3.7 Fault tolerance

Most of the DSM systems discussed so far ignore the fault-tolerance issue or maintain that it is an issue that should be handled by the underlying operating system. However, it would appear that in practice a DSM system would strongly effect the fault tolerance of a system. For example, in a DSM system where n systems are sharing access to a set of data, the failure of any one of them could lead to the failure of all the connected sites (or, at least, some of the processes on each site). We are also presented with an unusual failure-handling problem. It is fairly easy to see how to handle a failed message or RPC, but how do you handle a failed page-fault?

Interest in fault tolerance in DSM systems has increased in recent years as DSM has received more widespread use. Morin and Puaut [148] survey a number of fault-tolerant DSM systems in greater detail than this section presents.

The original Clouds DSM system [158] provided recoverability using shadowing of segments and a transactional system using commits. The recovery system was not really integrated with the DSM system and is merely implemented at the segment storage site. More recent work [43] describes the implementation of a transactional system called "Invocation-Based Concurrency Control" (IBCC) above Clouds. They intended to implement IBCC as a separate layer, but found this to be impossible. In order to maintain a consistent view of data when one transaction is active at multiple nodes, they were forced to integrate the transaction system with the DSM support system. Feeley, et al., [60] present another fault-tolerant transactional DSM system, which is based on extending the work done by Satyanarayanan, et al., [169, 170] on Lightweight Recoverable Virtual Memory (RVM).

Wu and Fuchs [201] describe a recoverable DSM system based on the IVY DSM model. Their DSM system consists of diskless workstations connected to disk servers over a LAN. A copy of a consistent shared-memory state is maintained on disk. Using an incremental checkpointing system based on a twin-page disk storage-management system, they provide rapid-rollback recovery. Two physically-contiguous disk pages are allocated for each writable page in the shared virtual-address space. When a page is fetched from disk, both are read and only one is retained, based on a simple time-stamp and state-based selection algorithm. Dirty memory pages can be written to disk at any time. They also describe a mechanism for reconstructing the *owner* and *copy-set*[3] data needed by the DSM system when there is a failure.

Ouyang and Heiser [156] discuss the addition of fault-tolerance using checkpointing and study in particular the effects of message loss on such systems. They present a variant of the two-phase commit protocol that has lower overhead and may be used in such systems. Kermarrec, et al., [116] present an integrated DSM system for a tightly-coupled architecture (an Intel Paragon) where the management of the recovery information is integrated into the DSM consistency protocol. The DiSOM system [78] incorporates a checkpoint-based failure recovery system which has the interesting property that no extra messages are required during failure-free operations. Richard and Singal [102] present yet another fault-tolerant DSM system based on logging and asynchronous checkpointing. Their system has the interesting property that process recovery is independent and no operational processes will be rolled back during recovery.

Tam and Hsu [179] take a different approach and treat the DSM location and ownership (*token*) information as a distributed database, and view page migrations as transactions on a global *token database*. Transactions are committed using a unilateral-commit protocol that allows fast recovery of the token state with minimal run-time overhead. Tam and Hsu describe an approach that is based on one of Li's DSM algorithms. A checkpointing and logging scheme is used to store the actual pages; the token database just helps maintain a consistent view after a failure.

Vaughan, et al., [195] describe a fault-tolerant persistent object system called Casper that provides a DSM service. Casper is implemented using shadow paging and concentrates primarily on maintaining a self-consistent view of the system that may be pushed to disk.

Markatos and Dramintinos [140] study the use of main memory in other nodes to store pages. They discuss an algorithm, based on duplicate copies of pages, that allows this remote paging to be fault-tolerant in the presence of a single-node failure.

Juul and Fleisch [110] discuss weakening of robustness in a DSM system. They argue that weakened robustness can be interesting in the same way that weakened consistency can. It is not always necessary to support complete and consistent recovery after a failure. Juul and Fleisch

[3] The set of nodes holding a read-only copy of the data.

propose a model which distinguishes a recoverable system, that must be able to survive any single-site failure, from a reliable system that also ensures consistency after recovery.

Upfal and Wigderson [189, 190] describe a formal model for distributing data in a distributed system that has very strong fault-tolerant properties. Where n is the number of processors in the system and $\epsilon \ll 1$, their scheme can sustain up to $O(\log n)$ *maliciously-chosen* faults and up to $(1 - \epsilon)n$ random ones, without any information or efficiency loss. The algorithm they present is effectively a majority-based replication system using time-stamps. Friedman [69] presents a theoretical overview of the effects of virtual synchrony (such as that provided by ISIS or Horus) on fault tolerance in a DSM system. He discusses how to define consistency conditions for various DSM consistency models.

In the next section, we discuss how memory allocation effects the design of DSM systems.

3.8 Memory allocation

In any shared-memory system where dynamic memory allocation is allowed, careful consideration must be given to the memory-allocation approach to be used: the same piece of shared memory must not be allocated more than once. A central memory-allocating node can be used, but the performance can be quite bad, due to networking overheads. Amber [39] allocates large chunks of the shared address space to each site at startup, and has an address-space server that hands out more memory on demand. This allows memory allocation to be done locally most of the time.

Another approach, taken by CMU in their Mach server [65, 66], is to use a parallel memory-allocation algorithm. They implemented a parallel version of Knuth's FreeList algorithm and claim improved performance over the FirstFit scheme used in IVY. Other measurements used by CMU indicate that a two-level allocation scheme, such as that used by Amber, is effective in reducing shared-memory contention. They also suggest allocating and deallocating memory in batches to reduce overheads.

3.9 Data persistence

Many of the DSM systems surveyed support some form of secondary-storage persistence for data held in shared memory. It is in this area that the overlaps between DSM systems and database systems are most apparent (see §4.3). There are a number of design issues that a DSM providing persistency must address.

Due to the inherently distributed nature of the data in a DSM system, it is not easy to obtain a globally-consistent snapshot of the data on secondary storage. All of the fault-tolerant DSMs (§3.7) address this issue in one way or another, and Casper [195] illustrates techniques that may be used to obtain a self-consistent view of the shared data to push to secondary storage. Kato, et al., [112] describe a different approach where the application programmer is given explicit control over the migration of shared-data segments between memory and secondary store. The Arias system [157] (based on work on Guide [80]) divides the shared global address space into "zones"; each zone may be marked as persistent and Arias supplies stable storage and distributed logging to provide atomic updates on persistent zones.

Garbage collection can be a major problem in persistent DSM systems, as it is difficult to track references to shared data that may be simultaneously accessible on multiple nodes. Ferreira and Shapiro [63] describe a mechanism to achieve global garbage collection in a DSM-based distributed-object system.

Naming and location are also significant issues once data becomes persistent. If the entire system is persistent, based on shared memory, and all access is transparent, then naming may not be an issue as all data can be directly accessed through direct pointers. This is true for some single-level address-space systems such as Opal (§3.3.1). In other systems, naming can be a significant issue; Little, et al., [138] discuss the model issues involved.

In the next section, we discuss other fields of research that are related to DSM systems and their use.

4 Related work

There are a number of research areas that have significant overlaps with work in the DSM field. This section discusses the areas of most importance: multi-processor caching, message passing, and database cache management.

4.1 Shared-memory multi-processor caching

There are several differences between shared-memory multi-processors and DSM systems. Shared-memory machines (usually) have a small, fast, local cache and use a fast bus to communicate with the main shared memory and other local caches. This leads to sophisticated consistency protocols being implemented in hardware so that conflicting writes may be handled with the minimum delay. In DSM systems, there is usually no physically shared memory, communication costs are high, and a cache miss is usually handled more like a page fault. Shared-memory machines usually use a much smaller cache granularity, due to the lower latency in handling misses.

There is a strong relationship between data consistency in a DSM system and hardware cache consistency in a shared-memory multi-processor [5, 136, 184, 71]. This is most directly illustrated by the consistency-maintenance scheme used in IVY, which is a modified version of the Berkeley multi-processor cache-consistency algorithm [203]. Both systems allow pages to be in four states. In the IVY case they have *nil*, *read*, *owned read*, and *write*. In the Berkeley algorithm they have *invalid*, *valid*, *shared dirty*, and *dirty*. The main modifications in the distributed version were to allow for the extra cost involved in broadcasting invalidations to all DSM nodes; this is effectively free in a shared-memory multi-processor. In the comparison done by Yang *et al.* [203], the Berkeley algorithm has the best performance of any of the algorithms that rely on invalidation instead of write broadcast. It is interesting to note that the same algorithm is suitable in a loosely-coupled architecture.

The most relevant type of shared-memory multi-processor—from the point of view of DSM systems—is the so called *non-uniform memory access* (NUMA) machine. In this type of machine there are usually two types of memory: fast local memory and slower global memory. The problems faced in implementing cache consistency in such machines [127] are very similar to those faced in implementing a DSM system. A common form of cache consistency used in these machines (and most interesting from the DSM point of view) is *directory-based cache coherence* [35]. Directory-based coherence describes any protocol that doesn't use broadcast, and must store the locations of all the cached copies of each block of shared data. The Stanford Distributed-Directory Protocol [183] is also worth examining from the DSM point of view, as it seems well suited to a loosely-coupled and distributed environment.

An example NUMA system is the PLATINUM [49], a software implementation of consistency on the BBN Butterfly multi-processor. PLATINUM uses a directory-based cache consistency algorithm based on invalidation. The algorithm is very similar to that used in IVY. The most important difference is the concept of *freezing* a piece of data in main memory. This extension makes use of the ability of NUMA machines to read global memory directly; so, when a write hot-spot is detected, data may be fixed in global memory rather than moving it back and forth between local memories.

PLATINUM deliberately does not hide page boundaries from the programmer, enabling the programmer to reduce interprocessor interference by allocating independently accessed data to distinct pages. This helps reduce the phenomenon of *false sharing* [28].

Holliday [84] discusses the problems of page migration in NUMA machines (and on a BBN Butterfly Plus in particular). He concludes that, while page migration does not degrade performance or impact robustness, page migration is at best marginally effective for the type of numerical applications he considers.

Gharachorloo, et al., [72] discuss memory consistency and event ordering from the multi-processor point of view and present the first work on *release consistency* (§1.4.1). Adve and Hill [6] examine a new definition of weak ordering as a contract between the hardware and the software, under which the hardware agrees to appear sequentially consistent provided the software obeys certain rules. Another shared-memory system of interest to DSM users is PLUS [26], a multi-processor architecture tailored to the fast execution of a single multi-threaded process. PLUS supports shared memory, efficient synchronisation, and pre-demand replication of pages.

Due to the increasing numbers of components in tightly-coupled multi-processors, the probability of failure in such machines is increasing. Morin, et al., [147] show that cache-only memory architectures (COMA) are good candidates for fault-tolerance in such systems. They demonstrate how to build a fault-tolerant shared memory system and simulate the results; for their simulated architecture (based on a KSR1), the performance degradation varies from 5% to 35% while retaining scalability. Chapin, et al., [37] present another approach used in the FLASH system [82].

They divide a large system into multiple cells and confine the effects of faults within a single cell. They also present a mechanism to allow the recoverable sharing of memory between cells using FLASH firewalls.

4.2 Tightly-coupled non-shared-memory multi-processors

Recent work indicates that the boundaries between hardware and software shared-memory systems are becoming increasingly blurred, as are the differences between a shared-memory and a non-shared-memory multi-processor.

Large-scale multi-computer machines, such as the Intel Paragon and Thinking Machines CM-5, do not have hardware-supported shared memory and use hardware-assisted message passing for communication. These machines can be treated as fast distributed systems with DSM techniques used to support shared memory [151, 206]. A number of more recent shared-memory systems for tightly-coupled machines are all software based and make use of no hardware assistance in maintaining shared-memory consistency. The most interesting of these is the C Region Library implemented by Johnson, et al., [107]; the CRL is all software based and needs no compiler or operating system support. The programmer, however, is responsible for annotating the application code in a similar manner to many release-consistent DSMs. The performance of the CRL is competitive with hardware DSMs and within 15% of the performance of the native shared-memory support on MIT's Alewife shared-memory multi-processor. Some all-software systems, such as that described by Scales and Lam [171], run unmodified both on tightly-coupled multi-processors such as the CM-5 and on loosely-coupled networks of workstations running PVM.

Other systems propose hardware and software combinations in order to increase performance [83, 168, 200]. Schoinas, et al., [173, 59] study both hardware and software-based access control in the Blizzard system (a fine-grained DSM system running on a tightly-coupled architecture over the Tempest system [164]); they conclude that the software system varies from slightly faster to twice as slow as the hardware-assisted version. Choi and Yew [46] propose a cache-consistency scheme that uses compiler directions to control hardware support; they simulate the results for large-scale multi-processors like the Cray T3D and conclude that their system has comparable performance and lower cost than fully hardware-supported shared memory. Joe and Hennessy [105] blur the lines between shared and non-shared memory architectures by proposing an alternative to NUMA systems where node memory is only used to cache data from global memory; their architecture is called cache-only memory (COMA) or attraction memory.

On the loosely-coupled front, all-software DSM systems such as Midway [23] and TreadMarks [114] are designed for use in fast switched local networks (such as ATM-based networks). These types of networks bear many of the characteristics of mesh-based multi-computers. Midway is designed to run both on ATM clusters and on multi-computers such as the CM-5 and Paragon. Dwarkadas, et al., [55] discuss the implementation of release-consistent DSM on emerging network technologies and conclude that faster network technologies (ATM, in particular) allow software-based DSM systems to be used for medium-grained parallel applications, where Ethernet was only really suitable for very coarse-grained applications. Cox, et al., [48] compare the performance of TreadMarks running on eight nodes over 100Mbps ATM with the performance of a shared-memory multi-processor with similar processing power. They conclude that TreadMarks is slower when large amounts of synchronisation are required; however for more moderate synchronisation needs and where memory bandwidth needs are high, TreadMarks can be better. They simulate larger networks and conclude that the shared-memory system scales better, but that a hybrid solution (shared-memory multi-processors in a cluster connected via ATM) can scale well.

Finally, more recent systems have started to use hardware assistance in loosely-coupled architectures. Blumrich, et al., [27] describe a virtual-memory mapped network interface, part of the SHRIMP multi-computer; data transfers can be achieved with the involvement of the receiving CPU and the system can be used to automatically detect, propagate, and merge updates in a DSM system [100]. Shubhendu, et al., [149] study consistent network interfaces that reside on cachable consistent busses and improve latency and bandwidth for fine-grained communication. Banerji, et al., [18] describe a workstation cluster using high-speed fibre to allow a portion of each machine's address space to be shared. Reinhardt, et al., [165] describe decoupled hardware to support DSM in networks of workstations; they integrate DSM access control with the network interface and protocol processor using a mix of off-the-shelf components and custom devices. In simulation, their fastest system is only twice as slow as completely-customised hardware DSM systems such as the Stanford FLASH system [82].

There is little clear understanding of the performance issues in shared memory and DSM systems. Iftode, et al., [101] simulate the performance of a number of applications on three different platforms: an all-software lazy release-consistent system, a release-consistent system with hardware assistance, and an all-hardware NUMA architecture. They conclude that the software DSM systems perform quite well for a variety of applications up to systems of at least 32 processors. However, performance is much more variable than for the all-hardware system and the problem sizes needed to obtain good parallel performance are substantially larger. The hardware-assisted software system [100] tends to perform significantly better than the all-software system.

4.3 Database caching

Despite using dissimilar terminology, many page- and object-buffering schemes used in distributed databases are effectively DSM systems of one sort or another. The algorithms used for cache management are similar in many cases, and thus, as mentioned in §3.5 and §1.5, we can learn a lot from the work done in this field. The difference of prime importance is that many database systems rely on transactions as an integral part of their cache management. For example, cache managers frequently use optimistic algorithms and rely on the ability to abort transactions to enforce consistency rules. Non-transactional DSM systems do not have this option and must use algorithms that operate in a pessimistic fashion and do not require abort and rollback facilities.

Both Franklin [31] and Wang [198] discuss cache-management algorithms that could have application in a DSM system. Object-oriented database (OODB) systems are also interesting, as many of them have sophisticated page and object buffer cache management. OODB systems such as O_2 [17, 54], Orion-2 [120], and ObjectStore [125] also bear similarities to DSM systems. Their cache management systems are all similar in function to consistency-management systems in DSM systems, but, as mentioned above, all rely on transactional support in the database system and are therefore difficult to apply directly to DSM system implementation.

Day [52] discusses object groups versus pages in the context of database server pre-fetching. O'Toole and Shrira [155] discuss adaptive switching between paged-based and object-based access in a OODB environment.

4.4 Function shipping versus data shipping

All DSM systems are based inherently on data shipping where data is brought to where it is needed. However, as discussed in §**??**, function shipping also has a place in a DSM system. This section covers some of the background information available comparing and contrasting function and data-shipping approaches.

Klaiber and Levy [121] compare message-passing and shared-memory architectures when used to implement a number of data-parallel applications. There is no clear winner in their comparison and they conclude that the link between data transfer and synchronisation is crucial and that shared-memory cache-consistency algorithms suffer when a fixed protocol is used. Chandra, et al., [36] document a similar study comparing the performance of well-written and equivalent message passing and shared-memory implementations of a number of parallel applications running on a detailed simulation of the CM-5 architecture. They conclude that the supposed advantage of message-passing is not clear cut; three of four applications ran at the same speed even though the communications patterns were different.

Thekkath, et al., [186] argue that function shipping is inefficient in current local-area networks, due largely to the latency issues discussed in §2. They claim that data and control transfers must be separated and propose a new communications structure based on remote network access to protected memory segments (effectively a partial DSM system).

Gunaseelan and LeBlanc [79] study the issues that arise in systems that support both function and data-shipping simultaneously. In particular, they study event ordering issues and the interaction with debugging when shared objects interact via RPC. They derive some clock conditions that may be used to timestamp and order operations on a shared object.

The Munin system has recently been experimentally extended with function-shipping support [33]. Initial results indicate that the ad-hoc addition of function shipping can produce significant speedups for certain applications.

Koch, et al., [122] propose integrating message passing and DSM in a new *message-driven coherency* mechanism implemented in their CarlOS system. Messages carry explicit causality annotations and trigger memory-consistency actions. By adding annotations to standard message-based protocols, it is possible to construct efficient implementations of common synchronisation and communications mechanisms. Since the mechanism is based on user-level messages, the set of available primitives can be easily extended with language or application-specific mechanisms. Their system is derived from TreadMarks. However, initial tests show the new mechanism to be slower than TreadMarks.

5 Example uses for DSM systems

This section gives some examples of uses to which DSM has been put.

Process migration This is an example of a problem that may be handled very simply in a DSM environment (assuming that the process's entire address space is being shared). The only requirement is to suspend the process and transfer its state over to the new machine; DSM can then be used to fault in the address space on demand. Using DSM to implement process migration has much the same advantages as the copy-on-reference migration system used in Accent [207]: there is no wasted effort transferring data that processes do not use again, and good handling of the case where a process ends shortly after being transferred. As in Accent, process migration using DSM should show a practically-constant transfer time independent of program size. Accent's policy of pre-fetching nearby pages on a fault could be applicable in the DSM case also.

A sample process migration implementation using DSM was implemented in the Clouds system [161]. Milojicic, et al., [142] implemented a similar system for Mach.

Object migration DSM can be put to good use in an object-oriented system to implement object migration. For example, using DSM an object can be migrated to a new node while leaving it accessible at the old node. DSM will keep the two copies of the object consistent. Once there are no longer any active references to the object at the old node, it can be removed. COOL-2 [8, 130] uses a DSM based object-migration system.

It is interesting to compare this system of object migration with that implemented in Emerald [109], where objects are physically removed from one node and placed on another and never exist on two nodes at once. A lot of the implementation effort in Emerald goes into dealing with objects that are no longer accessible, having been moved. Also, there is a strong similarity between the object-location scheme in Emerald, which uses forwarding addresses and hints, and the page-location scheme in IVY, which also uses forwarding addresses and hints. Emerald uses broadcasts to locate objects for which no information is available, a case that does not arise in IVY as all pages have some hint information associated with them at initialisation time.

Object invocation In the Clouds system, the DSM is primarily used as a mechanism to implement object invocation. Rather than moving a process to a remote object, the object is mapped locally. By instrumenting the DSM system to gather statistics, it might be possible to provide useful information to an object and process-migration system.

Memory use Using a DSM system, it is possible to make use of more memory than is available on a single node. For some applications with large memory requirements this can make the difference between thrashing and running with a low paging rate. Li and Hudak [135] give an example of solving partial differential equations (PDEs) where the distributed version achieves *super-linear* speedup, i.e., the n processor implementation goes more that n times faster than the single-processor version. This is because the full data set will not fit in the memory on one processor and the program causes heavy paging. This super-linear speedup effect can happen in any parallel application which is spread over multiple nodes.

Two-phase commit The paper on programming in Clouds [161] gives this as another example of a standard distributed algorithm that may be implemented using DSM. Information is passed in shared memory and using semaphores.

Shared-memory parallel programs As mentioned before, one of the advantages of distributed shared memory is that programs written with shared-memory multi-processors in mind can be easily ported to DSM systems. Many display good performance despite the different performance trade-offs present in a DSM environment. Examples of applications that have been successfully run on DSM systems include ray tracing [104, 30], parallel sorting, PDE solving [135], and matrix multiplication. Crovella *et al.* [50] discuss the issues involved in predicting the performance of parallel programs in shared-memory multi-processors, based on their communication to computation ratio; this ratio is even more relevant in the DSM field, where communication costs are generally higher and programs with a high communication to computation ratio may not cross over well. Nishikawa and Steenkiste [152] discuss load balancing in distributed-memory environments and in their Nectar system in particular. They highlight the importance of data dependencies on load balancing in a distributed-memory environment and discuss the effects of both data dependencies and heterogeneous tasks on load-balancing algorithms and communication requirements.

6 Conclusions

Overall, it would appear that DSM is a useful extra service for a distributed system to provide. DSM provides a simple programming model that is suitable for a wide range of parallel-programming problems. With careful application of the DSM approach, good speedups may be achieved for many parallel programs [135]. It would also appear that DSM might be useful for distributed, as well as parallel, problems. DSM should not be viewed as a replacement for RPC or for a message-passing style of communications. In the DSM model of communication, data is moved to the user; in the RPC model, operations are moved to the data. Both models are useful, and have their advantages and disadvantages. An advantage of DSM is that no explicit communication is necessary. Use may also be made of locality of reference. On the other hand, the handling of security and protection issues in DSM systems is unclear, and fault tolerance may be more difficult to achieve. Choosing between DSM and message passing for a given application is quite difficult, and the trade-offs must be considered carefully. Forin *et al.* [65] believe

> ...that the amount of data exchanged between synchronization points is the main indicator to consider when deciding between the use of distributed shared memory and message passing in a parallel application.

LeBlanc and Markatos [131] discuss the issue for shared-memory multi-processors, but come to no clear conclusion.

To achieve efficiency, synchronisation issues must be explicitly handled by the DSM system itself, or by some integrated environment. The main reason for handling synchronisation explicitly is the large performance problems introduced by using the ordinary synchronisation primitives on distributed-memory locations.

Most of the recent DSM systems discussed no longer implement strict consistency in the way that IVY did. Most now use some form of consistency relaxation to increase performance. There doesn't appear to be consensus yet on the correct approach here, but the most promising avenue of research would appear to be in the area of release [55] and entry-consistent DSM systems such as TreadMarks [114] and Midway [23]. There is much work to be done in the area of supporting the programming of release-consistent DSM systems so that the programmer is not required to explicitly annotate parallel programs. The CMM described in the next chapter provides a mechanism where release consistency (and many other DSM protocols) may be implemented with few annotations in the functional code of the application.

Most of the DSM systems discussed use a paged segmented-memory model, where the address space is split into different sections, some of which are managed by the DSM system. However, the arrival of 64-bit processors has increased the popularity of single shared address-space systems (§3.3.1).

As network latency dominates the performance timings of DSM systems [66], the choice of low-level network protocol is of prime importance. While there is no general consensus here, the most common choice is to use a reliable message-passing system rather than RPC or unreliable datagrams; however, end-to-end arguments [167] would weigh against this choice. The ability to forward messages without having to reply to the original sender is important in several of the distributed algorithms.

Increased network speeds and enhanced switch-based network technology are becoming increasingly important in DSM systems. Newer technologies solve many of the bandwidth problems in DSM, but latency, and synchronisation latency in particular [185, 55], is still a major problem. Hybrid hardware and software DSM systems are also becoming common, particularly in the high-performance multi-computer arena (§4.2).

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Summer Conference*, pages 93–112, 1986.

[2] ACM. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, USA, October 1994. ACM Press. Also available as Operating Systems Review 28(5), December, 1994.

[3] ACM SIGOPS. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, USA, December 1995. ACM Press. Also published as ACM Operating Systems Review 29(5).

[4] ACM SIGPLAN. *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, October 1994. ACM Press. Available as ACM SIGPLAN Notices 29(10).

[5] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. WRL Research Report 95/7, DIGITAL Western Research Laboratory, Palo Alto, California, USA, September 1995.

[6] Sarita V. Adve and Mark D. Hill. Weak Ordering – A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* [96], pages 2–14. Also published in ACM SIGARCH News, 18(2), June 1990.

[7] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and Programming Causal Distributed Shared Memory. In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, May 1991.

[8] Paulo Amaral, Christian Jacquemot, Peter Jensen, Rodger Lea, and Adam Mirowski. Transparent object migration in COOL-2: Position Paper. Technical Report CS/TR–92–30, Chorus Systèmes, 1992. ECOOP'92, Utrecht, The Netherlands, (June 29, 1992).

[9] R. Ananthanarayanan, Mustaque Ahamad, and Richard J. LeBlanc. Application Specific Coherence Control for High Performance Distributed Shared Memory. In *Symposium on Experiences with Distributed and Multiprocessor Systems* [191], pages 109–128.

[10] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. Technical Report 88–09–04, Department of Computer Science, University of Washington, September 1988.

[11] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-Speed Switch Scheduling for Local-Area Networks. *ACM Transactions on Computer Systems*, 11(4):319–352, November 1993.

[12] Association for Computing Machinery. *Proceeding of the* 12^{th} *ACM Symposium on Operating Systems Principles*, December 1989. ACM Operating Systems Review, 23(5), Special Issue.

[13] Association for Computing Machinery. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, NC, USA, December 1993. ACM Operating Systems Review 27(5).

[14] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and T. Ruhl. Performance evaluation of the orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, February 1998.

[15] Henri Bal. *Programming Distributed Systems*. Silicon Press, Prentice Hall, 1990.

[16] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. A Distributed Implementation of the Shared Data-Object Model. In *Distributed and Multiprocessor Systems Workshop* [192], pages 1–19.

[17] François Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System: The Story of* O_2. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Mateo, California, USA, 1992.

[18] Arindam Banerji, Dinesh Kulkarni, John Tracey, Paul Greenawalt, and David Cohn. High-Performance Distributed Shared Memory Substrate for Workstation Clusters. In *Proceedings of the Second International Symposium on High Performance Distributed Computing* [90], pages 344–351.

[19] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* [96], pages 125–134. Also published in ACM SIGARCH News, 18(2), June 1990.

[20] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. *SIGPLAN Notices*, 25(3):168–176, March 1990. Proceedings of the 2^{nd} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

[21] José M. Bernabéu-Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahamad, Willian F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran. *Clouds*—A Distributed, Object-Based Operating System Architecture and Kernel Implementation. In *Proceedings of the EUUG Autumn Conference*, pages 25–37, October 1988.

[22] Brian N. Bershad. Practical Considerations for Non-Blocking Concurrent Objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems* [93], pages 264–273.

[23] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. Technical Report CMU–CS–93–119, School of Computer Science, Carnegie Mellon University, 1993. A version of this paper appeared in COMPCON 1993.

[24] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. Technical Report TR91–1192, Department of Computer Science, Cornell University, February 1991.

[25] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[26] Roberto Bisiani and Mosur Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* [96], pages 115–124. Also published in ACM SIGARCH News, 18(2), June 1990.

[27] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, and Edward W. Felten. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* [95], pages 142–153. ACM Computer Architecture News 22(2).

[28] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the 12^{th} ACM Symposium on Operating Systems Principles* [12], pages 19–31. ACM Operating Systems Review, 23(5), Special Issue.

[29] William J. Bolosky and Michael L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Experiences with Distributed and Multiprocessor Systems Symposium* [193], pages 57–71.

[30] Roy H. Campbell and Daniel A. Reed. TAPESTRY: Unifying Shared and Distributed Memory Parallel Systems. Technical Report UIUCDCS–R–88–1449, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1988. TAPESTRY Technical Report No. TTR88-1.

[31] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architectures. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 357–366, May 1991. Also appears as SIGMOD Record 20(2), June, 1991.

[32] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, Pacific Grove, California, USA, October 1991. Association for Computing Machinery. ACM Operating Systems Review, Special issue 25(5).

[33] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[34] Miguel Castro, Paulo Guedes, Manuel Sequeira, and Manuel Costa. Efficient and Flexible Object Sharing. Technical Report INESC RT 18-95, IST – INESC, R. Alves Redol 9, 1000 Lisboa, Portugal, July 1995.

[35] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990. Special issue on Cache Architectures in Tightly Coupled Multiprocessors.

[36] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* [2], pages 61–73. Also available as Operating Systems Review 28(5), December, 1994.

[37] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* [3], pages 12–25. Also published as ACM Operating Systems Review 29(5).

[38] Jeff Chase, Valérie Issarny, and Hank Levy. Distribution in a Single Address Space Operating System. In *Proceedings of the Fifth ACM SIGOPS European Workshop*, Le Mont Saint-Michel, France, September 1992. Association for Computing Machinery.

[39] Jeffery S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12^{th} ACM Symposium on Operating Systems Principles* [12], pages 147–158. ACM Operating Systems Review, 23(5), Special Issue.

[40] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[41] Arbee L.P. Chen, Kuo-Fang Chieng, Tony C.T. Kuo, Eddy J.D. Lee, and Shu-Chin Su. Improving the Performance of a Distributed Computing System through Inconsistent Caches. In *The Third Workshop on Future Trends of Distributed Computing Systems*, pages 18–24, Taipei, Taiwan, April 1992. IEEE Computer Society.

[42] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The Impact of Synchronization and Granularity on Parallel Systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* [96], pages 239–248. Also published in ACM SIGARCH News, 18(2), June 1990.

[43] Raymond C. Chen and Partha Dasgupta. Integrating Consistency Control and Distributed Shared Memory: The Travails of an Implementation. In *Symposium on Experiences with Distributed and Multiprocessor Systems* [191], pages 279–295.

[44] David R. Cheriton. Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design. In *Proceedings of the 6^{th} International Conference on Distributed Computing Systems*, pages 190–197, May 1986.

[45] David R. Cheriton. Request-Response and Multicast Interprocess Communication in the V Kernel, 1986?

[46] Lynn Choi and Pen-Chung Yew. Compiler and Hardware Support for Cache Coherence in Large-Scale Multiprocessors: Design Considerations and Performance Study. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* [175], pages 283–294. Also available in Computer Architecture News 24(2), May 1996.

[47] Douglas Comer and James Griffioen. Efficient Order-Dependent Communication in a Distributed Virtual Memory Environment. In *Symposium on Experiences with Distributed and Multiprocessor Systems* [191], pages 249–262.

[48] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* [95], pages 106–117. ACM Computer Architecture News 22(2).

[49] Alan L. Cox and Robert J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12^{th} ACM Symposium on Operating Systems Principles* [12], pages 32–44. ACM Operating Systems Review, 23(5), Special Issue.

[50] Mark Crovella, Ricardo Bianchini, Thomas LeBlanc, Evangelos Markatos, and Robert Wisniewski. Using Communication-to-Computation Ratio in Parallel Program Design and Performance Prediction. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing* [92], pages 238–245.

[51] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* [194], pages 267–280.

[52] Mark Day. Object Groups may be better than Pages. In *Proceedings of the Fourth Workshop on Workstation Operating Systems* [89], pages 119–122.

[53] Gary S. Delp and David J. Farber. Memnet—a Different Approach to a Network. Department of Electrical Engineering, University of Delaware.

[54] O. Deux et al. The O_2 System. *Communications of the ACM*, 34(10):34–48, October 1991.

[55] Sandhya Dwarkadas, Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, San Diego, California, USA, May 1993. Computer Architecture News, 21(2).

[56] Dawson R. Engler, Sandeep K. Gupta, and M. Frans Kaashoek. AVM: Application-Level Virtual Memory. In *Fifth Workshop on Hot Topics in Operating Systems* [97], pages 72–77.

[57] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* [3], pages 251–266. Also published as ACM Operating Systems Review 29(5).

[58] M. Rasit Eskicioglu. A Comprehensive Bibliography of Distributed Shared Memory. *ACM Operating Systems Review*, 30(1):71–96, January 1996.

[59] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ionnis Schoinas, Mark D. Hill, James R. Larus, Anne Rogers, and David A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Supercomputing 1994*. IEEE, IEEE Computer Society Press, November 1994.

[60] Michael J. Feeley, Jeffrey S. Chase, Vivek R. Narasayya, and Henry M. Levy. Integrating Coherency and Recoverability in Distributed Systems. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* [194], pages 215–227.

[61] Michael J. Feeley and Henry M. Levy. Distributed Shared Memory with Versioned Objects. Technical Report UW–CSE–92–03–01, Department of Computer Science and Engineering, University of Washington, March 1992. To appear in OOPSLA 1992.

[62] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* [3], pages 201–212. Also published as ACM Operating Systems Review 29(5).

[63] Paulo Ferreira and Marc Shapiro. Garbage Collection and DSM Consistency. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* [194], pages 229–241.

[64] Brett D. Fleisch and Gerald J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proceedings of the* 12^{th} *ACM Symposium on Operating Systems Principles* [12], pages 211–223. ACM Operating Systems Review, 23(5), Special Issue.

[65] Alessandro Forin, Joseph Barrera, and Richard Sanzi. The Shared Memory Server. In *USENIX Winter Conference*, pages 229–243, 1989.

[66] Alessandro Forin, Joseph Barrera, Michael Young, and Richard Rashid. Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach. Technical Report CMU–CS–88–165, School of Computer Science, Carnegie Mellon University, August 1988.

[67] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global Memory Management in Client-Server DBMS Architectures. Computer Sciences Technical Report 1094, Computer Sciences Department, University of Wisconsin-Madison, June 1992. An abridged version of this paper appears in: The Proceedings of the 18th International Conference on Very Large Data Bases, Vancouver, BC, Canada, August 1992.

[68] Michael Jay Franklin. *Caching and Memory Management in Client-Server Database Systems*. PhD thesis, University of Wisconsin-Madison, 1993.

[69] Roy Friedman. Using Virtual Synchrony to Develop Efficient Fault Tolerant Distributed Shared Memories. Technical Report TR95-1506, Department of Computer Science, Cornell University, Ithaca, NY 14853, USA, March 1995.

[70] William E. Garrett, Ricardo Bianchini, Leonidas Kontothanassis, R. Andrew McCallum, Jeffery Thomas, Robert Wisniewski, and Michael L. Scott. Dynamic Sharing and Backward Compatibility on 64-Bit Machines. Technical Report TR 418, University of Rochester, Computer Science Department, April 1992.

[71] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, Santa Clara, California, USA, April 1991. Association for Computing Machinery. SIGARCH Special issue, 19(2).

[72] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* [96], pages 15–26. Also published in ACM SIGARCH News, 18(2), June 1990.

[73] Maya B. Gokhale and Ronald G. Minnich. An Implementation of the Shared Data Formats Standard for Distributed Shared Memories. In *Experiences with Distributed and Multiprocessor Systems Symposium* [193], pages 109–121.

[74] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75. ACM, April 1989.

[75] Gary Graunke and Shreekant Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990. Special issue on Cache Architectures in Tightly Coupled Multiprocessors.

[76] James Griffioen, Rajendra Yavatkar, and Raphael Finkel. Unify: A scalable, loosely-coupled, distributed shared memory multicomputer. Technical Report 226-93, Department of Computer Science, University of Kentucky, Lexington, KY 40506-0027, January 1993.

[77] James Griffioen, Rajendra Yavatkar, and Raphael Finkel. Extending the Dimensions of Consistency: Spatial Consistency and Sequential Segments. Technical Report CS248-94, Department of Computer Science, University of Kentucky, Lexington, KY 40506-0027, April 1994.

[78] Paulo Guedes, Miguel Castro, and Nuno Neves. The DiSOM Distributed Shared Object Memory. In *Sixth SIGOPS European Workshop*, Wadern, Germany, September 1994. ACM SIGOPS, ACM Press.

[79] L. Gunaseelan and Richard J. LeBlanc, Jr. Event Ordering in a Shared Memory Distributed System. In *Proceedings of the 13th International Conference on Distributed Computing Systems* [93], pages 256–263.

[80] Daniel Hagimont, P.-Y. Chevalier, A. Freyssinet, S. Krakowiak, S. Lacourte, J. Mossière, and X. Rousset de Pina. Persistent Shared Object Support in the Guide System: Evaluation & Related Work. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [4], pages 129–144. Available as ACM SIGPLAN Notices 29(10).

[81] Kieran J. Harty. Transactions in a Distributed System. Master's thesis, Department of Computer Science, Trinity College, Dublin, October 1986.

[82] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* [2], pages 38–50. Also available as Operating Systems Review 28(5), December, 1994.

[83] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993.

[84] Mark A. Holliday. Reference History, Page Size, and Migration Daemons in Local/Remote Architectures. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 104–112. ACM, April 1989.

[85] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* [13], pages 106–119. ACM Operating Systems Review 27(5).

[86] Feng Huang. Operating System Support for Flexible Coherence in Distributed Object Systems. In *Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS'95)* [98], pages 171–174.

[87] Phillip W. Hutto and Mustaque Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. Technical Report GIT–ICS–89/39, School of Information and Computer Science, Georgia Institute of Technology, October 1989.

[88] Phillip W. Hutto and Mustaque Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In *Proceedings of the* 10^{th} *International Conference on Distributed Computing Systems* [91], pages 302–309.

[89] IEEE. *Proceedings of the Fourth Workshop on Workstation Operating Systems*, Napa, California, USA, October 1993. IEEE Computer Society Press.

[90] IEEE. *Proceedings of the Second International Symposium on High Performance Distributed Computing*, Spokane, Washington, USA, July 1993. IEEE Computer Society Press.

[91] IEEE Computer Society. *Proceedings of the* 10^{th} *International Conference on Distributed Computing Systems*, May 1990.

[92] IEEE Computer Society. *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, Arlington, Texas, December 1992.

[93] IEEE Computer Society. *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, Pennsylvania, USA, May 1993. IEEE Computer Society Press.

[94] IEEE Computer Society. *Proceedings of the 3rd International Workshop on Object-Orientation in Operating Systems*, Asheville, North Carolina, USA, December 1993.

[95] IEEE Computer Society. *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Illinois, April 1994. IEEE Computer Society Press. ACM Computer Architecture News 22(2).

[96] IEEE Computer Society and ACM SIGARCH. *Proceedings of the 17th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, May 1990. Also published in ACM SIGARCH News, 18(2), June 1990.

[97] IEEE Computer Society Technical Committee on Operating Systems and Application Environments. *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, USA, May 1995. IEEE Computer Society Press.

[98] IEEE Computer Society Technical Committee on Operating Systems and Application Environments. *Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS'95)*, Lund, Sweden, August 1995. IEEE Computer Society Press.

[99] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, June 1996.

[100] Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[101] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Understanding Application Performance on Shared Virtual Memory Systems. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* [175], pages 122–133. Also available in Computer Architecture News 24(2), May 1996.

[102] Golden G. Richard III and Mukesh Singhal. Using Logging and Asynchronous Checkpointing to Implement Recoverable Distributed Shared Memory. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 58–67, Princeton, New Jersey, USA, October 1993. IEEE Computer Society Technical Committee on Distributed Processing, IEEE Computer Society Press.

[103] Shigekazu Inohara, Keitaro Uehara, Hajime Miyazawa, and Takashi Masuda. Sharing Persistent Data Structures on Wide Address Spaces in the Lucas Operating System. Technical report, Department of Information Science, Faculty of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan, 1993.

[104] David W. Jensen and Daniel A. Reed. Ray Tracing on Distributed Memory Parallel Systems. Technical Report UIUCDCS–R–89–1551, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1989. TAPESTRY Technical Report No. TTR19.

[105] Truman Joe and John L. Hennessy. Evaluating the Memory Overhead Required for COMA Architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* [95], pages 82–93. ACM Computer Architecture News 22(2).

[106] Gary M. Johnson and Roy H. Campbell. An Object-Oriented Implementation of Distributed Shared Virtual Memory. In *Distributed and Multiprocessor Systems Workshop* [192], pages 39–58.

[107] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* [3], pages 213–228. Also published as ACM Operating Systems Review 29(5).

[108] Theodore Johnson. A Performance Comparison of Fast Distributed Mutual Exclusion Algorithms. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 258–264, Santa Barbara, California, USA, April 1995. IEEE Computer Society Press.

[109] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[110] Niels Christian Juul and Brett D. Fleisch. A Memory Approach to Consistent, Reliable Distributed Shared Memory. In *Fifth Workshop on Hot Topics in Operating Systems* [97], pages 108–112.

[111] Anna Rochelle Karlin. Sharing Memory in Distributed Systems – Methods and Applications. Technical Report STAN–CS–87–1164, Dept. of Computer Science, Stanford University, January 1987.

[112] Kazuhiko Kato, Atsunobu Narita, Shigekazu Inohara, and Takashi Masuda. Distributed Shared Repository: A Unified Approach to Distribution and Persistency. In *Proceedings of the 13th International Conference on Distributed Computing Systems* [93], pages 20–29.

[113] Czeslaw K. Kazimierczak and Birger Andersen. Object Consistency – A New Model for Distributed Memory Systems. In *Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS'95)* [98], pages 33–36.

[114] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter Conference*, pages 115–132, San Francisco, January 1994. USENIX Association.

[115] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[116] Anne-Marie Kermarrec, Gilbert Cabillic, Alain Gefflaut, Christine Morin, and Isabelle Puaut. A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability. Publication Interne 897, Institut de Recherche en Infomatique et Systèmes Aléatoires (IRISA), Campus de Beaulieu, 35042 Rennes Cedex, France, January 1995.

[117] R. E. Kessler and Miron Livny. An Analysis of Distributed Shared Memory Algorithms. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 498–505. IEEE Computer Society, June 1989.

[118] Gregor Kiczales and John Lamping. Operating systems: Why object-oriented? In *Proceedings of the 3rd International Workshop on Object-Orientation in Operating Systems* [94], pages 25–30.

[119] Gregor Kiczales, Marvin Theimer, and Brent Welch. A new model of abstraction for operating system design. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 346–349, Dourdan, France, September 1992. IEEE Computer Society.

[120] Won Kim, Nat Ballou, Jorge F. Garza, and Darrell Woelk. A Distributed Object-Oriented Database System Supporting Shared and Private Databases. *ACM Transactions on Information Systems*, 9(1):31–51, January 1991.

[121] Alexander C. Klaiber and Henry M. Levy. A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* [95], pages 94–105. ACM Computer Architecture News 22(2).

[122] Povl T. Koch, Robert J. Fowler, and Eric Jul. Message-Driven Relaxed Consistency in a Software Distributed Shared Memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* [194], pages 75–85.

[123] Eric J. Koldinger, Jeffery S. Chase, and Susan J. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–186, Boston, Massachusetts, October 1992. Association for Computing Machinery. SIGARCH Special issue, Volume 20.

[124] Dinesh C. Kulkarni, Arindam Banerji, Michael R. Casey, and David L. Cohn. Structuring Distributed Shared Memory with the Π Architecture. In *Proceedings of the 13th International Conference on Distributed Computing Systems* [93], pages 93–100.

[125] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[126] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[127] Richard P. LaRowe, Jr. and Carla Schlatter Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[128] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* [2], pages 208–218. Also available as Operating Systems Review 28(5), December, 1994.

[129] Hugh C. Lauer and Roger M. Needham. On the Duality of Operating System Structures. In *Proceedings of the Second International Symposium on Operating Systems*, IRIA, October 1978. Reprinted in Operating Systems Review 13(2), April 1979, pp. 3–19.

[130] Rodger Lea, Paulo Amaral, and Christian Jacquemot. COOL-2: an object oriented support platform built above the CHORUS Micro-kernel. Technical Report CS/TR–91–103, Chorus Systèmes, 1991. In *Proceedings of 1991 IWOOOS*, Palo Alto, CA, USA, (October 1991).

[131] Thomas J. LeBlanc and Evangelos P. Markatos. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing* [92], pages 254–263.

[132] Chao-Hsien Lee, Meng Chang Chen, and Ruei-Chuan Chang. HiPEC: High Performance External Virtual Memory Caching. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* [194], pages 153–164.

[133] Joonwon Lee and Umakishore Ramachandran. Synchronization with Multiprocessor Caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* [96], pages 27–36. Also published in ACM SIGARCH News, 18(2), June 1990.

[134] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, August 1986.

[135] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[136] David J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[137] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS–TR–180–88, Princeton University, Department of Computer Science, September 1988.

[138] M.C. Little, D.L. McCue, and S.K. Shrivastava. Maintaining Information about Persistent Replicated Objects in a Distributed System. In *Proceedings of the 13th International Conference on Distributed Computing Systems* [93], pages 491–498.

[139] Mohammad Malkawi, Deborah Knox, and Mahmoud Abaza. Page Replacement in Distributed Virtual Memory Systems. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing* [92], pages 394–401.

[140] Evangelos P. Markatos and George Dramintinos. Implementation of a Reliable Remote Memory Pager. In *USENIX Technical Conference*, San Diego, CA, USA, January 1996. USENIX Association.

[141] Dylan McNamee and Katherine Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Mach Workshop*, pages 17–29. USENIX Association, October 1990.

[142] Dejan S. Milojicic, Peter Giese, and Wolfgang Zint. Experiences with Load Distribution on Top of the Mach Microkernel. In *USENIX Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diega, California, USA, September 1993. USENIX Association.

[143] Gary J. Minden, Joseph B. Evans, David W. Petr, and Victor S. Frost. An ATM WAN/LAN Gateway Architecture. In *Proceedings of the Second International Symposium on High Performance Distributed Computing* [90], pages 136–143.

[144] Ronald G. Minnich and David J. Farber. The Mether System: Distributed Shared Memory for SunOS 4.0. In *USENIX Summer Conference*, pages 51–60, 1989.

[145] Ronald G. Minnich and David J. Farber. Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory. In *Proceedings of the 10^{th} International Conference on Distributed Computing Systems* [91], pages 468–475.

[146] Jeffrey C. Mogul. Big Memories on the Desktop. In *Proceedings of the Fourth Workshop on Workstation Operating Systems* [89], pages 110–115.

[147] Christine Morin, Alain Gefflaut, Michel Banâtre, and Anne-Marie Kermarrec. COMA: an Opportunity for Building Fault-tolerant Scalable Shared Memory Multiprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* [175], pages 56–65. Also available in Computer Architecture News 24(2), May 1996.

[148] Christine Morin and Isabelle Puaut. A Survey of Recoverable Distributed Shared Memory Systems. Publication Interne 975, Institut de Recherche en Infomatique et Systèmes Aléatoires (IRISA), Campus de Beaulieu, 35042 Rennes Cedex, France, December 1995.

[149] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grained Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* [175], pages 247–258. Also available in Computer Architecture News 24(2), May 1996.

[150] Kevin Murray, Tim Wilkinson, Peter Osmon, Ashley Saulsbury, Tom Stiemerling, and Paul Kelly. Design and Implementation of an Object-Orientated 64-bit Single Address Space Microkernel. In *Microkernels and Other Kernel Architectures Symposium*, pages 31–43. USENIX Association, September 1993.

[151] S.S. Nemawarkar, R. Govindarajan, G.R. Gao, and V.K. Agarwal. Analysis of Multithreaded Multiprocessors with Distributed Shared Memory. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 114–121, Dallas, Texas, USA, December 1993. IEEE Computer Society Press.

[152] Hiroshi Nishikawa and Peter Steenkiste. A General Architecture for Load Balancing in a Distributed-Memory Environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems* [93], pages 47–54.

[153] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, pages 52–60, August 1991.

[154] Andreas G. Nowatzyk, Michael C. Browne, Edmund J. Kelly, and Michael Parkin. S-Connect: from Networks of Workstations to Supercomputer Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 71–82, Santa Margherita Ligure, Italy, June 1995. ACM, ACM Press. Also available as Computer Architecture News 23(2), May, 1995.

[155] James O'Toole and Liuba Shrira. Shared Data Management Needs Adaptive Methods. In *Fifth Workshop on Hot Topics in Operating Systems* [97], pages 129–135.

[156] Jinsong Ouyang and Gernot Heiser. Checkpointing and Recovery for Distributed Shared Memory Applications. In *Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS'95)* [98], pages 191–199.

[157] Elizabeth Pérez-Cortés, Pascal Dechamboux, and Jay Han. Generic Support for Synchronization and Consistency in Arias. In *Fifth Workshop on Hot Topics in Operating Systems* [97], pages 113–118.

[158] David V. Pitts and Partha Dasgupta. Object Memory and Storage Management in the *Clouds* Kernel. In *Proceedings of the 8^{th} International Conference on Distributed Computing Systems*, pages 10–17. IEEE Computer Society, June 1988.

[159] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying Synchronization and Data Transfer in Maintaining Coherence of Distributed Shared Memory. Technical Report GIT–ICS–88/23, School of Information and Computer Science, Georgia Institute of Technology, June 1988. Published in *Proceedings of the 1989 Conference on Parallel Processing.*

[160] Umakishore Ramachandran and M. Yousef A. Khalidi. An Implementation of Distributed Shared Memory. In *Distributed and Multiprocessor Systems Workshop* [192], pages 21–38.

[161] Umakishore Ramachandran and M. Yousef Amin Khalidi. Programming with Distributed Shared Memory. Technical Report GIT–ICS–88/38, School of Information and Computer Science, Georgia Institute of Technology, October 1988.

[162] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. Technical Report CMU–CS–87–140, School of Computer Science, Carnegie Mellon University, July 1987.

[163] David P. Reed and Rajendra K. Kanodia. Synchronisation with Eventcounts and Sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

[164] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* [95], pages 325–336. ACM Computer Architecture News 22(2).

[165] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* [175], pages 34–43. Also available in Computer Architecture News 24(2), May 1996.

[166] Rodolfo F. Resende, Divyakant Agrawal, and Amr El Abbadi. Semantic Locking in Object-Oriented Database Systems. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* [4], pages 388–402. Available as ACM SIGPLAN Notices 29(10).

[167] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[168] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. Region-Oriented Memory Management in Shared-Memory Multiprocessors. Technical Report CSRI–269, Computer Systems Research Institute, University of Toronto, April 1992.

[169] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* [13], pages 146–160. ACM Operating Systems Review 27(5).

[170] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1):33–57, February 1994.

[171] Daniel J. Scales and Monica S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* [194], pages 101–114.

[172] Frank Bernhard Schmuck. *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems.* PhD thesis, Cornell University, August 1988. Available as TR 88-928, Dept. of Computer Science, Cornell University, Ithaca, NY 14853-7501.

[173] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* [2], pages 297–306. Also available as Operating Systems Review 28(5), December, 1994.

[174] Peter M. Schwarz. *Transactions on Typed Objects.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1984. Also available as TR CMU-CS-84-166.

[175] SIGARCH. *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, Pennsylvania, USA, May 1996. ACM Press. Also available in Computer Architecture News 24(2), May 1996.

[176] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990. Special issue on Cache Architectures in Tightly Coupled Multiprocessors.

[177] Michael Stumm and Songnian Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[178] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber. A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems. *ACM Operating Systems Review*, 24(3):40–67, July 1990.

[179] Va-On Tam and Meichun Hsu. Fast Recovery in Distributed Shared Virtual Memory Systems. In *Proceedings of the* 10^{th} *International Conference on Distributed Computing Systems* [91], pages 38–45.

[180] Andrew S. Tanenbaum and Henri E. Bal. Programming a Distibuted System Using Shared Objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing* [90], pages 5–12.

[181] Douglas B. Terry. Caching Hints in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):48–54, January 1987.

[182] Avadis Tevanian, Jr. and Richard F. Rashid. MACH: A Basis for Future UNIX Development. Technical Report CMU–CS–87–139, School of Computer Science, Carnegie Mellon University, June 1987.

[183] Manu Thapar and Bruce Delagi. Stanford Distributed-Directory Protocol. *IEEE Computer*, 23(6):78–80, June 1990. Special issue on Cache Architectures in Tightly Coupled Multiprocessors.

[184] Manu Thapar and Bruce Delagi. Cache Coherence for Large Scale Shared Memory Multiprocessors. *Computer Architecture News*, 19(1):114–119, March 1991.

[185] Chandramohan A. Thekkath and Henry M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[186] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* [2], pages 2–11. Also available as Operating Systems Review 28(5), December, 1994.

[187] Keitaro Uehara, Hajime Miyazawa, Kouhei Yamamoto, Shigekazu Inohara, , and Takashi Masuda. A Framework for Customizing Coherence Protocols of Distributed File Caches in Lucas File System. Technical Report 94-14, Department of Information Science, Faculty of Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan, December 1994.

[188] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with Locking in a NUMA Multiprocessor Operating System Kernel. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* [194], pages 139–152.

[189] Eli Upfal and Avi Wigderson. How to Share Memory in a Distributed System. Technical Report STAN–CS–84–1024, Dept. of Computer Science, Stanford University, October 1984.

[190] Eli Upfal and Avi Wigderson. How to Share Memory in a Distributed System. *Journal of the ACM*, 34(1):116–127, January 1987.

[191] USENIX. *Symposium on Experiences with Distributed and Multiprocessor Systems*, Newport Beach, CA, USA, March 1992.

[192] USENIX Association. *Distributed and Multiprocessor Systems Workshop*, October 1989.

[193] USENIX Association. *Experiences with Distributed and Multiprocessor Systems Symposium*, San Diego, California, USA, September 1993.

[194] USENIX Association. *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, Monterey, California, USA, November 1994. ACM Press.

[195] Francis Vaughan, Tracy Lo Basso, Alan Dearle, Chris Marlin, and Chris Barter. Casper: a Cached Architecture Supporting Persistence. *Computing Systems*, 5(3):337–364, Summer 1992. USENIX Association / University of California Press.

[196] Jerry Vochteloo, Stephen Russell, and Gernot Heiser. Capability-Based Protection in the Mungi Operating System. In *Proceedings of the 3rd International Workshop on Object-Orientation in Operating Systems* [94], pages 108–115.

[197] Thorsten von Eicken, Veena Avula, Anindya Basu, and Vineet Buch. Low-Latency Communication over ATM Networks using Active Messages. In *Proceedings of Hot Interconnects II*, Palo Alto, California, USA, August 1994. Abridged version of paper is available in IEEE Micro Magazine, 15(1), pp. 46–64, Feb 1995.

[198] Yongdong Wang and Lawrence A. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 367–376, May 1991. Also appears as SIGMOD Record 20(2), June, 1991.

[199] William Edward Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, March 1984.

[200] Andrew W. Wilson, Jr., Richard P. LaRowe, Jr., and Marc J. Teller. Hardware Assist for Distributed Shared Memory. In *Proceedings of the 13th International Conference on Distributed Computing Systems* [93], pages 246–255.

[201] Kun-Lung Wu and W. Kent Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[202] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[203] Qing Yang, Laxmi N. Bhuyan, and Bao-Chyn Liu. Analysis and Comparison of Cache Coherence Protocols for a Packet-Switched Multiprocessor. *IEEE Transactions on Computers*, 38(8):1143–1153, August 1989.

[204] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space. In *Proceedings of the USENIX Summer Technical Conference*, pages 175–186, Cincinnati, Ohio, June 1993. USENIX Association.

[205] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffery Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. *ACM Operating Systems Review*, 21(5):63–76, November 1987. Special Issue, Proceedings of the Eleventh ACM Symposium on Operating Systems Principles.

[206] Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, and Durriya Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the USENIX Winter Technical Conference*, pages 449–468, San Diego, January 1993. USENIX Association.

[207] Edward R. Zayas. Attacking the Process Migration Bottleneck. *ACM Operating Systems Review*, 21(5):13–24, November 1987. Special Issue, Proceedings of the Eleventh ACM Symposium on Operating Systems Principles.

[208] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation* [194], pages 87–100.

[209] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous Distributed Shared Memory. Technical Report CSRI–244, Computer Systems Research Institute, University of Toronto, September 1990.

[210] Songnian Zhou, Michael Stumm, and Tim McInerney. Extending Distributed Shared Memory to Heterogenous Environments. In *Proceedings of the* 10^{th} *International Conference on Distributed Computing Systems* [91], pages 30–37.