# Distributed Shared Memory with Versioned Objects

**Michael J. Feeley and Henry M. Levy**

*Department of Computer Science and Engineering, FR-35*
*University of Washington*
*Seattle, WA 98195*

*Technical Report 92-03-01*

*To appear in the Proceedings of the Conference on*
*Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*
*October 1992*

## Abstract

*Distributed Object Memory* (DOM) is an abstraction that represents a distributed-memory system as a single shared container of language-level objects. The goal of DOM is to simplify programming of parallel applications for such systems. All accesses to shared memory are made relative to objects that reside in one or more node-local memories. For example, in Amber, a DOM system for a network of workstations, *remote* references are transparent at the language level and are implemented using either remote procedure call or object replication and migration. While DOM can greatly simplify distribution for many application classes, it is not well suited for all domains (parallel-scientific codes, in particular).

To address the shortcomings of DOM for such domains, we introduce *Versioned* DOM (VDOM). In VDOM a version number is associated with each object. Multiple versions of objects can coexist and may be cached in local memories as needed to increase concurrency. Object coherence is driven by *synchronization methods* implicitly associated with each object. Explicit versioning is used as the basis for a memory consistency model that facilitates efficient fine-grain sharing of objects. The units of VDOM coherence are *fragment objects*, from which language-level objects are constructed; this fragmentation solves the false-sharing problem for language-level objects. The performance of VDOM primitives compared to standard DOM coherence is presented along with speedup results for two parallel applications implemented using VDOM.

This work was supported in part by the National Science Foundation under Grants No. CCR-8619663 and CCR-8907666, by the Washington Technology Center, and by the Digital Equipment Corporation Systems Research Center and External Research Program.

## 1 Introduction

For a multiprocessor to scale to even a moderate number of processors, the ideal of a hardware-implemented uniform-access memory must typically be abandoned. The reasons for this have been well documented and are related in part to the scalability of a shared bus in the face of modern processor performance. A multiprocessor based upon *distributed memory* is one attractive solution to this problem. In such a system (e.g., the Intel Hypercube, the Thinking Machines CM-5, or a network of workstations), processors are grouped into *nodes*; each node has locally accessible memory and is connected to other *remote* nodes by

a network. Distributed-memory systems are distinguished from other scalable architectures, such as non-uniform memory access systems (e.g., the BB&N Butterfly), in that they lack hardware support for direct access to remote memory. Instead, remote data is examined or manipulated by explicit message passing from one node to another.

The scalability of distributed-memory multiprocessors built from commodity parts makes them an attractive platform for large-scale parallel processing. However, programming a distributed system is difficult because the programmer must deal explicitly with communication. To represent data that is *remotely* shared among a number of nodes, an application stores a local copy of the data in the memory of each node, and uses message-passing operations such as *send* and *receive* to keep those copies up to date. By explicitly managing the *coherence* of shared data in this way, the application can be carefully coded for good performance. The cost for this performance is (1) added complexity and (2) inherent asymmetry between local and remote sharing. Symmetry is an issue of both conceptual clarity and performance. At the language level, sharing in node-local memory should be expressed in the same way as sharing across the network, while the implementation should provide efficiency for both local and remote sharing.

One solution to these problems is to provide distributed applications with a software-based, shared-memory abstraction. In *Distributed Shared Memory* (DSM) systems such as Ivy [Li & Hudak 89], the application sees a single shared virtual address space, as if shared memory were provided by the hardware. DSM systems rely on standard memory management hardware to control coherence at the granularity of a page. In Ivy, pages that are *read-shared* among nodes are replicated on-demand in node-local memories, while a *write* access to a page causes a synchronous invalidation of all but the writing processor's copy.

DSM greatly simplifies the task of writing distributed applications but it has problems of its own related to performance. Ideally, we want to simplify distributed programming without substantially sacrificing the performance of a carefully coded message-passing version of the program. There are two fundamental problems that keep DSM systems from achieving these goals.

- DSM coherence, and therefore distribution, is at the granularity of a page, which may not match the granularity of data distribution and sharing. The problems of granularity and placement of shared data must be solved by the application in order to avoid increased coherence overhead and unnecessary network messages associated with *false sharing.*
- The DSM coherence mechanism that is replicating and invalidating pages operates blindly without information from the application about data access patterns. Since coherence is driven strictly in an on-demand fashion, optimizations such as prefetching and relaxed memory consistency are less effective.

Thus, the central problem in DSM is the gap between the coherence protocol that manages distribution and sharing, and the application that knows best how to manage its data. Fundamentally, DSM lacks a *programming model* to aid the programmer in expressing the program's needs with respect to data, computation and distribution.

In contrast, *Distributed Object Memory* (DOM) provides a shared-memory abstraction without the problems caused by page-level granularity in DSM. In DOM systems [Jul et al. 88, Liskov 88, Chase et al. 89, Bal & Tanenbaum 88], coherence is at the granularity of an object — an instance of a language type that encapsulates both data and operations. A parallel computation is represented by a set of objects that are *placed* by the application in node-local memories to balance computational load and to maximize reference locality. Access to objects is uniform: at the language level, local and remote objects are accessed in the same way.

Remote references are handled by one of the coherence mechanisms supported by DOM, either by RPC-like [Birrell & Nelson 84] *function shipping*, or by restricted forms of *data shipping*. With function shipping, an operation on a remote object is completed by moving the invoking *thread* of control to the object's local node, where it performs

the operation. Data shipping, on the other hand, maintains a local copy of the object using either a migration or a replication strategy. When a replicated object is modified, consistency is maintained using a synchronous operation either to invalidate or to update all other copies of the object.

DOM closes the gap between the coherence protocol and the application; objects provide the programmer with a natural way to convey information to the runtime system about the units of coherence (i.e., objects) and the coherence policy for their use [Carter et al. 91]. This is a key advantage of DOM over page-based systems. Since the object is an encapsulation of data and all references to it, coherence can be implicitly associated with each object's external interface, while DSM systems must rely on run-time detection of remote data references.

The advantages of the DOM model derive from the fact that it defines a particular model for distributed programming. This in turn places restrictions on the class of applications that are suitable for DOM.

- In DOM, language-level objects are the unit of coherence. Thus, it is assumed that there is a natural decomposition of the application into objects that can be distributed and shared atomically.

- DOM function-shipping and data-shipping coherence mechanisms involve synchronous network communication. As a result, remote sharing is assumed to be coarse grained.

A number of applications fit well into this model, conforming to these restrictions and benefiting from DOM. However, there are many parallel applications for which these restrictions are too strong. In this paper, we present an extension to DOM with the idea of offering the benefits of object-based shared memory to a larger class of applications. We are particularly interested in scientific applications that suffer performance problems on standard DOM systems.

### 1.1 Versioned DOM

*Versioned Distributed Object Memory* (VDOM) is a new model for parallel-distributed programming that extends the application domain of DOM to include certain parallel-scientific codes, such as grid-based data-parallel applications. These applications are characterized by their decomposition for parallel (and distributed) computation; a uniform global data structure is sub-divided into an *overlapping* decomposition of subproblems. The fact that subproblems overlap contradicts the two DOM restrictions listed above.

To illustrate this problem, consider the example shown in Figure 1. This application performs some computation over a two dimensional grid of points. For parallel execution, the grid is divided into rectangular subregions that are distributed among the processors. In each step of the computation, new values are computed for all points based upon the values of neighboring points. This means that when updating the value on the boundary of a subgrid, it will be necessary to read values from one or more *neighboring* sub-grids, which may be stored on other nodes.

A *natural* object-oriented decomposition of this problem is to represent a sub-grid as an object. However, a sub-grid object is not suitable as the atomic unit of coherence, a violation of DOM assumptions, because parts of it — the edges — are shared with other, possibly remote subproblems. This re-introduces the *false-sharing* problem associated with DSM pages, this time at the object level. Figure 1 shows this by dividing the subregion into the *actual* units of coherence: the interior, which is private to the sub-grid, and the edges and corners, which are shared with neighbors. The edges of neighboring sub-grids are shown shaded.

A second problem with representing these shared edges in DOM is the *way* that they are shared. Elements from a remote neighbor's edge (shaded rectangle) are accessed point-by-point along with elements from local parts of the sub-grid. This precludes the use of *function-shipping* coherence to resolve these remote references, because "ping-ponging" would result, with a separate remote procedure call needed to access each *element*. The

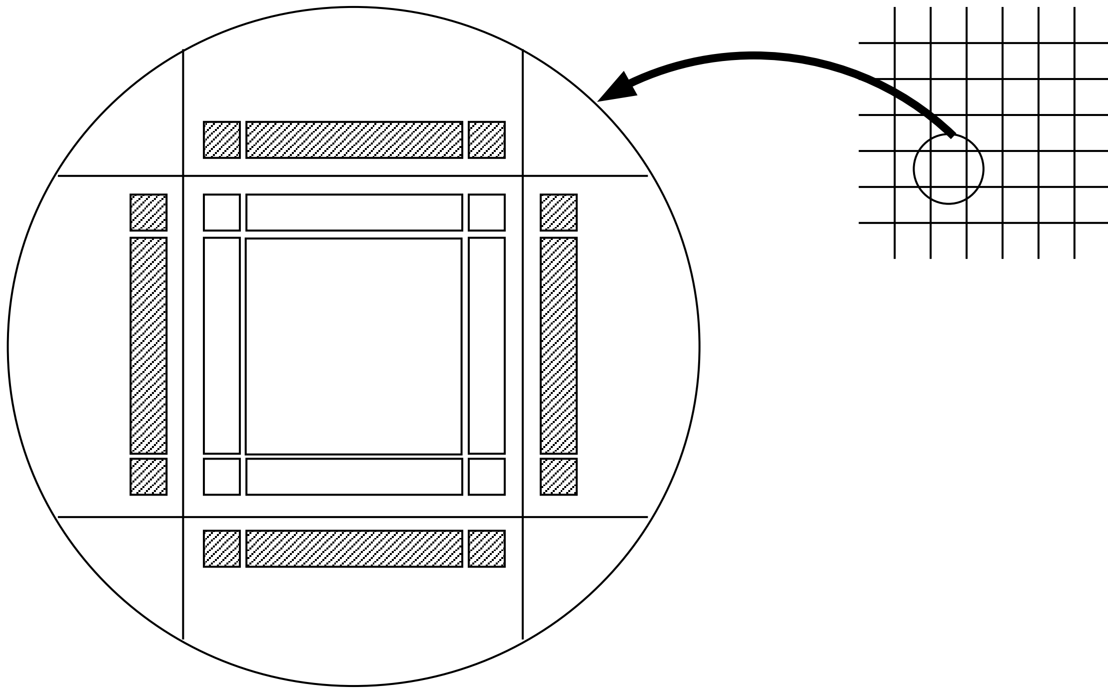

**Figure 1: Fragment Objects for Grid-Based Application**

only way to provide this sharing efficiently is to use *data-shipping* coherence to obtain a copy of the entire edge, so that its elements can be accessed through local memory. However, DOM provides only limited data-shipping coherence in the form of either object migration or replication. Using either of these mechanisms to shared grid edges requires run-time synchronization among sharing nodes and synchronous network communication to move the objects and to invalidate replicas. The network latency associated with these operations makes them unsuitable for the type of fine-grain sharing found in data-parallel applications such as this grid-based example.

VDOM addresses these limitations by extending standard DOM in two fundamental ways.

- *Fragment objects* are introduced to solve the problem of *false sharing* of objects. In VDOM, a language-level object can be composed from any number of fragment objects. Fragment objects are the fundamental units of VDOM coherence. The individual rectangles in Figure 1 are fragment objects.
- We introduce a new model of sharing called *version consistency* (VC) that supports efficient fine-grain sharing of objects. VC is based upon multi-version immutable objects; an update to an object logically creates a *new version* of the object. Specifying which version of an object is to be accessed coordinates sharing without the need for costly remote synchronization; multiple versions provide increased concurrency and eager data shipping to reduce the impact of network latency. These points are discussed in the following section.

### 1.2 Organization of this Paper

This paper describes VDOM, a new model for distributed programming. VDOM is an extension of DOM that is motivated by the desire to support certain parallel-scientific applications that are poorly suited to standard DOM. In Section 2, we provide an overview of VDOM, how it is used and how it is implemented. Our memory consistency model, *version consistency*, is presented in Section 3. Performance results of our implementation

are discussed in Section 4 along with information about two applications written using VDOM. Related work and alternative approaches to the simplification of efficient distributed programming are briefly discussed in Section 5. Section 6 summarizes our contributions.

## 2 Overview of VDOM

VDOM provides a multi-version abstraction of shared object memory based on *fragment objects* and *version consistency*. Objects are shared by caching them in local memories with coherence driven by synchronization operations associated with each object. The fundamental atomic units of coherence in VDOM are fragment objects, from which language-level objects are assembled. Objects are immutable; when an object is changed, a new version of it is created. Increased concurrency is achieved by allowing multiple versions of an object to coexist; old versions of an object can be read while a new version is being created. In a distributed system, versioning also provides a means of latency hiding for eager updating of cached objects. The idea of using immutable objects to increase concurrency is exploited by both the Cedar distributed file system [Schroeder et al. 85] and the Cosmos distributed software development environment [Walpole et al. 89].

### 2.1 Fragment Objects

VDOM uses *fragmented objects* to solve the object-level false-sharing problem. The term "fragmented object" was introduce in another context as a way of representing a language-level object that is subdivided into parts that may be distributed in different ways [Shapiro et al. 90]. In VDOM, fragment objects are the basic atomic units of coherence. Language-level objects may be composed from any number of these fragment objects. For example, recall the sub-grid decomposition shown in Figure 1. The entire sub-grid, including the shaded edges of its neighbors, is represented by a single language-level object. This object is physically allocated as a number of fragment objects: the interior matrix, the sub-grid's edge vectors (North, South, West and East) and corners, and the neighboring subregion's edges and corners. Since the edges of subregions overlap each other, the shaded fragment objects are actually allocated only once as part of their own sub-grid; the language-level object contains pointers to these neighboring-edge fragment objects.

The application never references a fragment object directly. When defining a language-level class, the application specifies distribution and sharing information for objects of that class. This information is used by the class's constructor method to allocate constituent fragment objects. The application-defined class inherits a generic superclass that handles fragment object allocation and reference, hiding fragmentation from the application. For example, SOR and WaTor, the two sample applications discussed in Section 4.2 and Section 4.3, are implemented using a generic *2-d sub-grid* class. A library of generic classes could be defined for various geometries and sharing paradigms.

The external interface of a language-level object is unaffected by the fragmentation of the object; references to elements of a composite object are translated by superclass methods to access the appropriate fragment object. For example, the sub-grid superclass might have a method `valueAt` that returns the value of a point given its $x$ and $y$ coordinates. To the application, the sub-grid looks like a two dimensional matrix. However, `valueAt` translates the coordinates given to it by the application into a reference to one of the sub-grid's fragment objects. In SOR and WaTor a method called `index` is used that returns a pointer to an element given its coordinates. We expect that compiler optimization could be used to make this translation of coordinates as efficient as direct access to a contiguous array, however, these optimizations were done by hand in our implementation.

### 2.2 Programming in VDOM

Our implementation of VDOM is built as an extension of the Amber DOM system [Chase et al. 89, Feeley et al. 91], providing programmers with an additional set of predefined C++ classes. Shared objects in an application inherit a superclass that

| *Operation* | *Version Number* |
| --- | --- |
| AcquireRead | optional |
| AcquireWrite | required |
| AcquireWriteOnly | required |
| ReleaseRead | no |
| ReleaseWrite | no |

**Table 1: VDOM Operations**

supports the VDOM synchronization primitives shown in Table 1. These operations are used to drive the coherence protocol that keeps local memories up to date as new versions of objects are released. Before an object can be accessed, it must be acquired by making a call to either AcquireRead, AcquireWrite, or AcquireWriteOnly.

The first time that a remote object is acquired, a call is made to the object's *owner* node to request that the object be cached in the caller's local memory. The object's owner is determined from the virtual address that names the object (possibly following forwarding addresses left by the object if it moves). Once it is cached, all new versions of the object will be eagerly sent to that node as soon as they are produced. Thus, subsequent attempts to acquire the object on that node will complete locally.

To acquire a particular version of an object, the application specifies the object's *version number* as a parameter to the acquire. For AcquireRead, the version number is optional; not specifying it will acquire an arbitrary version of the object (in fact, the latest version stored on that node). The acquire returns a pointer to the requested version of the object, blocking when necessary to wait for that version to be produced. Once a version is in a node's local memory, attempts to acquire it will complete immediately, without any other synchronization.

When acquiring an object for writing, the application specifies the version number that it intends to create. The AcquireWrite completes as soon as the preceding version of the object is present in node-local memory. Mutual exclusion for the update is guaranteed implicitly, without inter-node synchronization, as part of the *version consistency* memory model discussed in Section 3. The AcquireWriteOnly operation is optimized so that it does not copy the current value of an object when creating a new version; it is useful when the update writes the *entire* object.

After an object is modified, ReleaseWrite is used to signal the coherence protocol to update the local memory of all nodes that are sharing the object. The new version of the object is sent to these nodes asynchronously; the writer does not wait for it to propagate across the network.

## 2.3 Immutable Objects

An important benefit of VDOM is that coherence is maintained asynchronously. This is true not only of the ReleaseWrite operation as discussed above, but also on the receiving end of an update. A new version of an object can be copied directly into the local memory of any node (making it available for the next Acquire there) without synchronizing with any thread on that node. If VDOM allowed threads to update objects "in-place", this concurrency would not be possible, because sending an update might conflict with other accesses to that object on the receiving node. With immutable objects, any thread accessing the object on the receiving node must be accessing an *older* version, thus there can be no such conflict. In addition to eliminating costly remote synchronization, multi-versioned memory also supports concurrent reading and writing. Versioning of objects is useful in another way; version numbers are used as the basis for VDOM's memory consistency model, *version consistency*.

## 2.4 Summary

The key features of VDOM that distinguish it from other DOM systems are summarized as follows:

- *Fragment objects* provide a finer grain of coherence than language-level objects.
- Object coherence is driven by *synchronization accesses* implicitly associated with each object.
- Objects are immutable and multiple versions of an object can coexist.

- *Version consistency* is a new memory model that is used to express several forms of sharing, maintaining coherence without costly inter-node synchronization.

# 3 Version Consistency

A *memory consistency model* is a contract between an application and the mechanism that implements shared memory (for VDOM this is the runtime system). The model defines certain restrictions on the use of shared objects; applications that adhere to these restrictions are given guarantees about the coherence of those objects. In a *strict* consistency model, every read of a data item returns the value last written; this requires a system-wide synchronization on every write. Strict consistency can be unnecessarily costly in performance, because for predictable results, reads and writes to shared data must still be managed through critical sections. In a *weak* consistency model [Gharachorloo et al. 90, Adve & Hill 90], the application makes synchronization requests (acquire and release) visible to the hardware or runtime system. The runtime system can then use this information to minimize the number and frequency of messages between sharing nodes by providing the minimum coherence required to meet the actual needs of the program. In return, the application agrees to abide by the contract and examine the shared data only at appropriate times.

*Version consistency* (VC) is a weak model based upon the following two restrictions on access to shared objects.

1. Objects are immutable; modifying an object creates a new version.
2. Each access selects a specific version of an object on which to operate.

With version consistency, the application provides information about the relative *order* of all accesses to a shared object by specifying a version number with each Acquire.[1] The runtime system uses this information to optimize coherence and synchronization for the object. In DOM, where ordering information is not available, coherence can only be maintained at runtime using costly network operations to synchronize sharing nodes and to invalidate (or update) local caches. Version consistency provides this synchronization implicitly. For a VC program to be correct, the sequence of all accesses to a shared object must contain only one AcquireWrite for each version of the object. Thus, permission to update version $i$ of a object can be passed with the release of version $i$ to each sharing node. The distinguished thread that issues the acquire to produce version $i+1$ proceeds, without further synchronization, when it finds version $i$ in its local memory.

This model is useful for applications that have an explicit or implicit *ordering* on all accesses to a shared object. Much of the sharing found in data-parallel applications is of this form. For example, both *producer consumer* and *write sharing* have this characteristic, as we will discuss in the next section. These applications often have an iteration or step number that can be used naturally as the basis for the version numbers it specifies.

## 3.1 Sharing Found in Data-Parallel Applications

Data-parallel applications exhibit two types of fine-grain sharing that are not easy to express efficiently in DOM systems, as described below. Support for both classes of sharing can easily be built using version consistency.

- *Producer consumer* is a style of sharing where one thread (the producer) updates an object to be read by one or more other threads (the consumers). In this style of sharing it is often important that each update (or new version) is consumed exactly once, or once by each consumer.
- *Write sharing* is another common style of sharing where there are multiple threads that alternate updating an object. In this case, it is important to ensure mutually-exclusive access to the object and to give threads a way

[1] As stated previously, an AcquireRead can omit the version number if it is unimportant which version of the object is to be accessed.

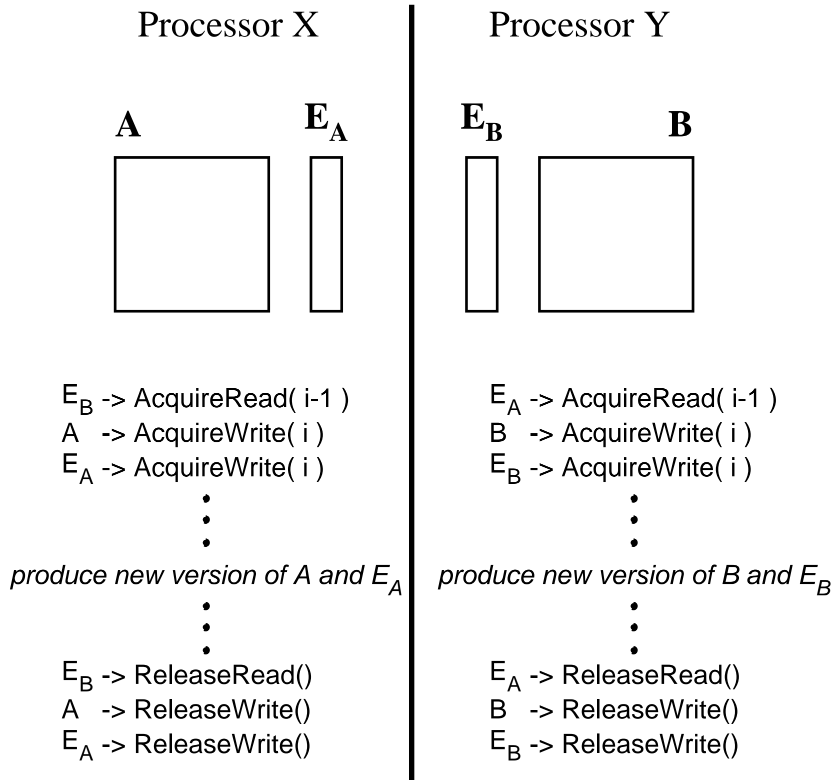


**Figure 2: Example of Producer Consumer with VC**

of determining an ordering on the updates to the objects. In data-parallel applications, it is usually important that threads update an object in a specific order, or sometimes that all threads have updated once before any thread updates it again.

## 3.2 Producer Consumer with VC

Figure 2 shows an example, of how *producer-consumer* sharing is supported in VDOM. In this example, threads on two processors (either with shared or distributed memory) are sharing two *edge* objects $E_A$ and $E_B$ and accessing private *interior* objects $A$ and $B$. This is a simplified version of the type of sharing found in grid-based applications similar to SOR (Section 4.2).

The algorithm iterates through a number of steps. On each step, a thread reads the most recently generated version of its neighbor's edge object, and then generates a *new* version of its own edge, also updating its own interior object. On step $i$, the thread on processor $X$ will issue the three VDOM acquire primitives shown in the figure; the thread thus gains read access to the *previous* version of $E_B$ (version $i-1$), and write access to new versions of $A$ and $E_A$ (version $i$). The thread implicitly acquires read access to version $i-1$ of $A$ and $E_A$. When the thread on processor $X$ issues the call ReleaseWrite on $E_A$, version $i$ of $E_A$ is immediately sent to the local memory of processor $Y$, so that it will be there on the next step when the thread on processor $Y$ tries to acquire it. So long as there is sufficient work in computing the new values of the interior of $B$ before $Y$ acquires $E_A$, the network latency of the update of $E_A$ will be completely hidden from the application.[2] Recall that since multiple versions of a shared object can coexist, version $i$ of $E_A$ can arrive in processors $Y$'s local memory while a thread on that node is in the process of reading version $i-1$.

[2] In the real implementation, Y's AcquireRead of $E_A$ occurs after it has computed the interior region, $B$.

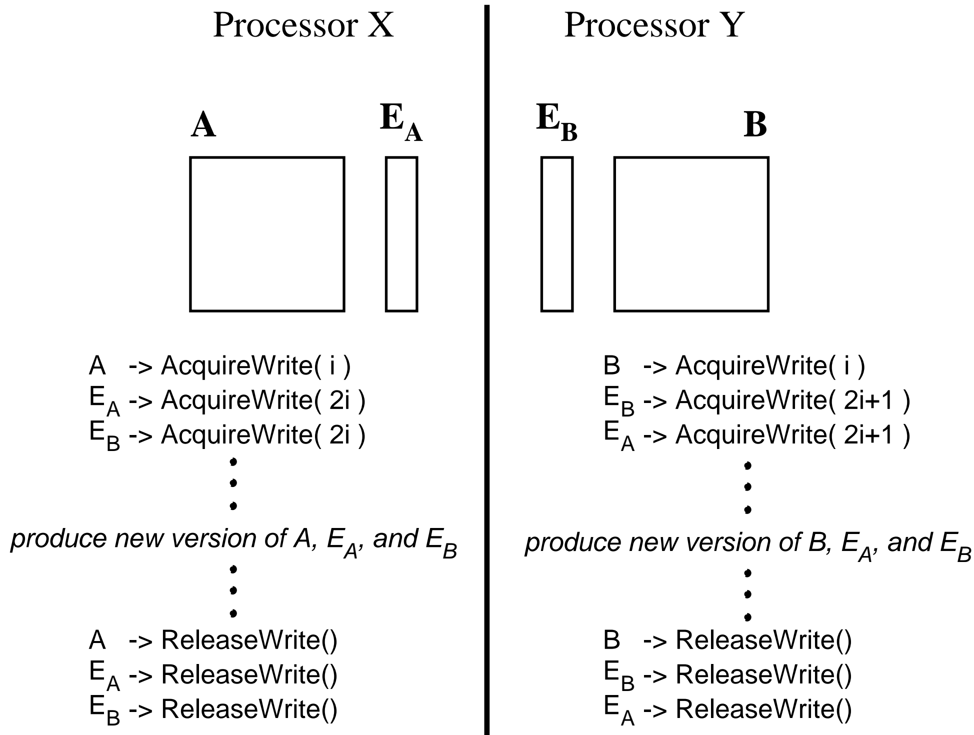


**Figure 3: Example of Write Sharing with VC**

### 3.3 Write Sharing with VC

Figure 3 shows an example of *write sharing*. As in the previous example, this figure also shows threads on two processors sharing two objects $E_A$ and $E_B$. The difference is that now, the threads must gain *write* access to both objects on each step. The WaTor application discussed in Section 4.3 utilizes this type of sharing.

On each step of the algorithm, threads alternate updating the shared edge objects $E_A$ and $E_B$; this alternation is explicit in the algorithm. Work on the computation of interior values in $A$ and $B$ is used to overlap with the communication associated with this sharing. The code fragment shown in the figure is all that is needed to coordinate the write-shared behavior of the two threads.

On step $i$, the thread on processor $X$ is the first to acquire $E_A$ and $E_B$, creating version $2i$. When that thread issues the ReleaseWrite of the edges, version $2i$ is sent to processor $Y$. As soon as it arrives, the thread on processor $Y$ is allowed to acquire the edges to create version $2i+1$. When the thread on processor $Y$ releases the edges, version $2i+1$ is sent back to processor $X$ so that on iteration $2i+1$, the thread there can create version $2(i+1)$.

### 3.4 Advantages of Version Consistency

Version consistency supplies a memory model that is well suited to the fine-grain sharing exhibited by many data-parallel applications. This model has a number of benefits, related both to expressiveness and to efficiency. These benefits are summarized below.

- *Synchronization* between threads that are sharing an object is achieved using VC, without the need for any other synchronization mechanisms. A thread wishing to access version $i$ of an object will automatically wait, blocking if necessary, until that version is made available by the thread that updates version $i-1$. This eliminates the need for *barriers* or other forms of global synchronization that

can be particularly costly in a distributed system.

- *Writing* to a shared object is controlled implicitly without any synchronization and without the need to invalidate old copies of the object that may be cached in the local memories of other processors. Having version $i-1$ of an object is necessary and sufficient for any thread on that processor to acquire it for writing, producing version $i$.

- *Mutual Exclusion* is guaranteed by the ordering of write accesses. This is part of the contract the application enters into in order to use the memory model. Since all accesses are explicitly ordered, the application guarantees that an AcquireWrite to a particular version of an object will occur at most once during the execution of the program.

- *Increased Concurrency* is achieved, because threads may be reading version $i$ of an object while another thread is updating it, producing version $i+1$. This also allows a new version to be disseminated to other nodes without needing to synchronize with any thread that might be reading an older version of the object.

- *Latency Hiding* is automatic. Following a Release of an update to an object, the new version of the object is eagerly sent to all processors that currently have that object cached, *anticipating* the next Acquire of that object. In this way, Acquire operations on remotely shared objects will complete locally whenever possible.

### 3.5 Version Consistency and Message Passing

Version consistency shares some of the characteristic benefits of the message-passing programming paradigm [Tomlinson & Scheevel 89]. By using version numbers, the program achieves the same implicit synchronization found with message-passing; the consumer of a message can synchronize with its producer by issuing a blocking Receive that waits for the producer to Send the message. However, VC provides this synchronization directly with shared data objects, instead of *message ports* (the typical end-points of communication in a message-passing system). It also knows which processors should received updates, sending them without waiting for the next Acquire. Doing this in a message-passing system would require that the application have knowledge of all processors that are sharing the object. Finally, version consistency works in local shared-memory just as well and in the same way as remotely, over the network. This provides the symmetry between local and distributed parallelism discussed above.

## 4 Performance

We have implemented VDOM as an extension of Amber. Amber is a DOM programming model that extends C++ with a set of classes providing objects with mobility primitives, light-weight threads, and synchronization primitives such as spinlocks, monitors, condition variables, and barriers. Amber runs on a network of Firefly multiprocessor workstations [Thacker et al. 88] running the Topaz operating system. All measurements were made on a network Fireflies connected by a 10-megabit/second Ethernet. Each workstation consists of four (3 MIP) C-VAX processors.

In this section we present the results of performance measurements. We first compare the overhead and latency of the primitive coherence operations for three systems: Amber DOM, our implementation of VDOM, and a page-based DSM integrated into a version of Amber [Habert 90]. Following this, we present performance results for two applications implemented in VDOM.

### 4.1 Micro Performance

This section presents performance data for the basic operations of our implementation of VDOM. Table 2 shows the performance of the coherence mechanisms of three systems: (1) standard Amber DOM, (2) our implementation of VDOM, and (3) a page-based DSM integrated into a version of Amber.

| System | Operation | Latency (ms) |
|---|---|---|
| *Amber* | local invoke/return | 0.012 |
| | remote invoke/return | 8.320 |
| | object move | 12.430 |
| | ReadRun (local) | 0.250 |
| | ReadRun (remote) | 21.050 |
| | WriteRun (local, no invalidate) | 0.210 |
| | WriteRun (remote, no invalidate) | 19.120 |
| | WriteRun (remote, one invalidate) | 28.540 |
| *VDOM* | AcquireRead (local) | 0.017 |
| | AcquireRead (blocking) | 0.233 |
| | AcquireWrite | 0.023 |
| | ReleaseWrite (no update) | 0.044 |
| | ReleaseWrite (signal one waiter) | 0.300 |
| | ReleaseWrite (one remote update) | 0.390 |
| | remote update | 4.180 |
| *DSM* | read fault (remote) | 11.690 |
| | write fault (local, one invalidate) | 8.254 |
| | write fault (remote, no invalidate) | 7.680 |
| | write fault (remote, one invalidate) | 14.770 |
| | invalidation | 6.570 |

**Table 2: Performance of Coherence Operations (Amber, VDOM, DSM)**

The performance for each system is discussed below. None of these systems are highly optimized, and the performance of each of them could be improved; thus, these figures should not be taken as lower bounds. However, it should be clear that some operations are qualitatively more complex than others, because they involve one or more global synchronizations and message exchanges.

**Amber:** The first set of data is for standard Amber DOM system. Remote invocation is Amber's *function-shipping* (i.e., RPC-oriented) coherence mechanism. The time of 8.320 msec shown is the time for a synchronous call/return from a thread on one node to a object on another node, and involves round-trip network latency and scheduling overhead. The overhead associated with Amber's object migration *data-shipping* coherence is the 12.430 msec show for an object move. The performance of Amber's invalidation-based object-replication coherence mechanism [Faust 90] follows this in the figure. Accesses to a replicated object are from within either a ReadRun or a WriteRun. A ReadRun completes *locally* if the object is already cached in local memory (0.250 msec). If the object is not local, then 21.050 msec are required to enter a ReadRun. Before a thread enters a WriteRun on an object, all replicas of the object are synchronously invalidated. If there are no replicas this takes 0.210 msec, while if there is a single replica it takes 28.540 msec. To enter a WriteRun on a remote object requires 19.120 msec.

**VDOM:** The performance of VDOM's *data-shipping* coherence is a significant improvement over the standard DOM mechanisms. A key observation to make when comparing VDOM performance to the other systems listed is that unlike the others, VDOM operations hide network latency. To Acquire a local object for reading takes only 0.017 msec. If an Acquire must block until the requested version arrives in local memory, the

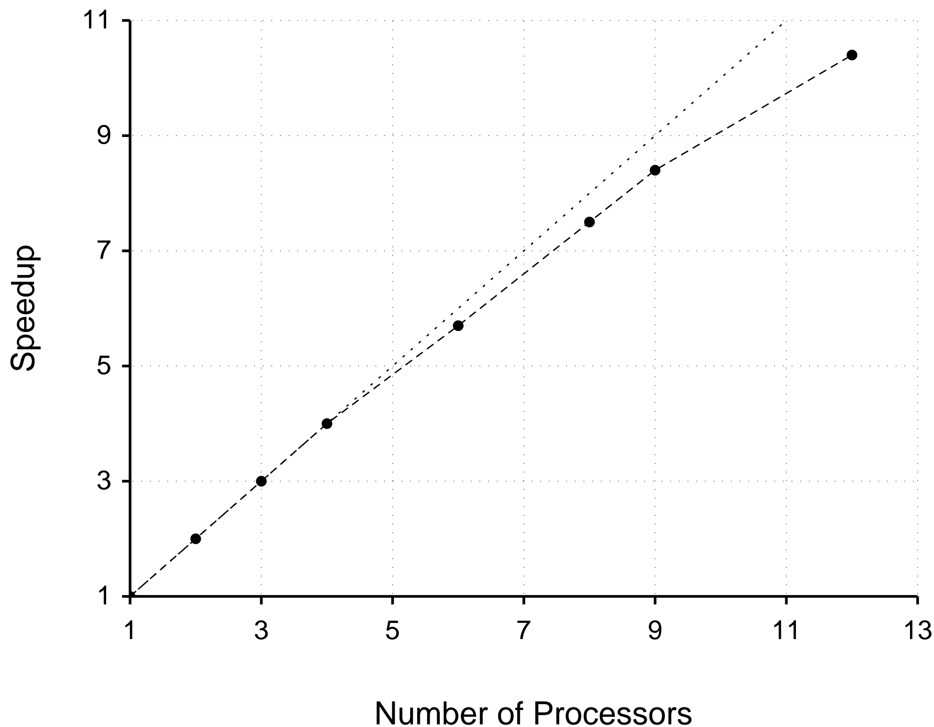


**Figure 4: VDOM SOR Performance (288×288 Grid)**

overhead increases to 0.233 msec (plus the receive time). To Acquire a local object for writing is only 0.023 msec, a three order of magnitude improvement over DOM synchronous invalidation. The Release following an update of an object will take 0.300 msec if there is a local thread waiting to Acquire the new version, 0.390 msec if there is a single remote replica of the object, or 0.044 if neither is true. A total of 4.180 msec are needed to send a new version from the writer node to the local memory of another node. This is about half the time of remote invocation, and a third the time of object migration. The reason for this improvement is that the Release is asynchronous to threads on both the sending node and on the receiving node.

**DSM:** The final set of performance results were obtained from a page-based DSM system built into Amber [Habert 90] and are presented for rough comparison to the object-based systems. It takes a total of 11.690 msec to replicate a remote page following a "read fault". When a "write fault" occurs, it takes 8.254 msec to invalidate a single remote page, 7.680 msec if the page is remote but no invalidation is needed, or 14.770 to do both. A single invalidation of a local page requires 6.580 msec.

### 4.2 Application 1: SOR

Successive Over Relaxation (SOR) computes the steady-state temperature of a rectangular plate given the temperature at its edges. The plate is represented as a two-dimensional array of floating-point temperature. The algorithm proceeds for a number of iterations where on each iteration, Laplace's equation is used to compute the new value of an element of the array from its old value and the average of its North, South, East and West neighbors.

To decompose the problem for parallel computation, the grid is divided into sub-grids, one for each processor. A subregion updates its temperatures by reading the edges of each of its four neighbors.

This application is an example of *producer-consumer* sharing as described in Section 3.2. Figure 4 shows the speedup achieved with a VDOM-

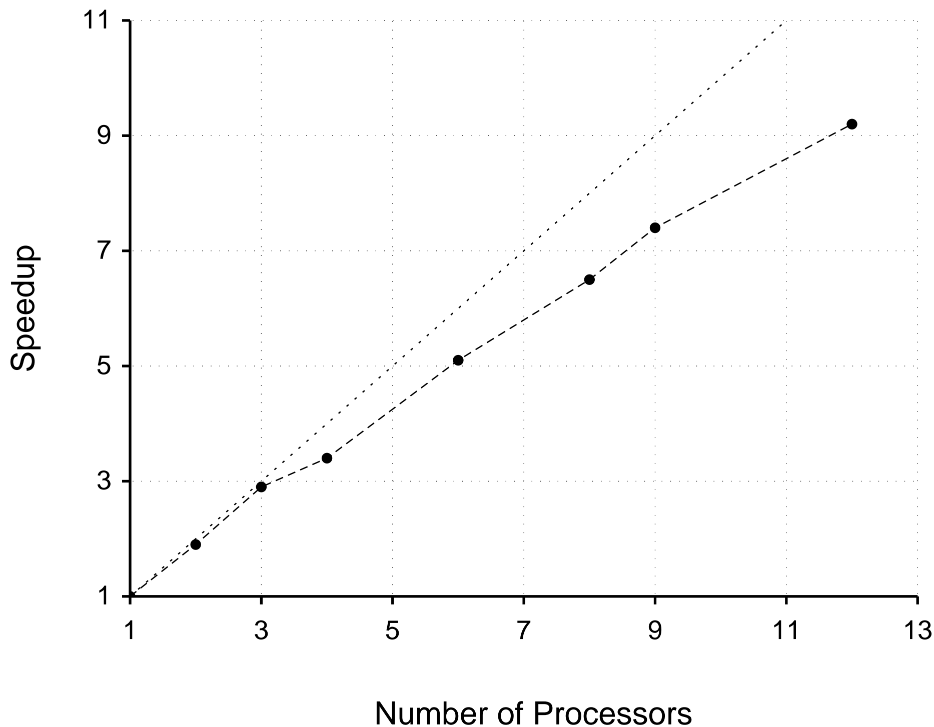


**Figure 5: VDOM WaTor Performance (288×288 Grid)**

based implementation of SOR with a 288×288 grid, on from 1 to four nodes each using from 1 to 3 processors. Speedup is very nearly linear, which means VDOM operations successfully hid network latency and provided sufficient concurrency.

### 4.3 Application 2: WaTor

WaTor [Fox et al. 88] is a simulation of an ocean populated by minnows and sharks. The ocean is represented by a two dimensional grid of points. Each point may be empty or it may contain either a minnow or a shark. The algorithm proceeds in an iterative fashion. On each step, fish move one cell in a randomly chosen direction, and may breed; sharks can eat minnows (or die of starvation).

This problems demonstrates *write-shared* objects, since on an iteration, a fish might move from the edge of one subregion to the edge of a neighboring subregion, requiring that that subregion have write access to its own edges and its neighbor's edges as in Figure 3. Figure 5 shows the speedup achieved with a VDOM-based implementation of WaTor using write-shared edges. The size of the grid was 288×288 points and data was collected for configurations of from 1 to 4 nodes using from 1 to 3 processors per node.

## 5 Comparison to Related Work

A number of systems have been designed and built to provide some form of Distributed Shared Memory in order to make distributed programming easier. Among these are page-based systems [Li & Hudak 89, Carter et al. 91] and object-based systems (DOM) [Liskov 88, Jul et al. 88, Bal & Tanenbaum 88, Chase et al. 89]. Page-based DSM provides the illusions of shared-memory but without providing a programming model that gives the programmer a way to easily express data decomposition and access patterns. As a result, the coherence mechanisms of these systems typically must operate without this information, and as a result can suffer poor performance resulting from false sharing.

In Munin [Carter et al. 91], the user annotates data objects with type information that is used to select from one of eight different coherence policies through appropriate settings of seven coherence attributes. As with other DSM systems, distributed programming in Munin is simplified, because the user is not directly involved in data layout. Aside from type annotation, data structures are declared without regard to how they will be distributed or shared. Munin solves the false-sharing problem using a page copy/compare strategy based upon *release consistency* [Gharachorloo et al. 90]. Shared data is replicated on the granularity of a page. The first reference to a remote page causes a copy of the page to be saved. All read and write operations to the page following this access complete locally until a *release* synchronization operation is issued by the thread; then the updated page is compared with the saved copy and the differences are sent synchronously to nodes that are sharing the page. VDOM achieves similar performance to Munin for a wide range of sharing interactions without requiring a page fault, copy and compare, expensive operations whose performance will not scale with improvements in processors [Ousterhout 90]. VDOM, on the other hand, requires a bit more care in declaring shared data, but the application can avoid the need for costly run-time, on-demand coherence overhead.

Object-based models are naturally adapted to distributed systems. Examples of DOM systems include Amber [Chase et al. 89, Feeley et al. 91], Argus [Liskov 88], Emerald [Jul et al. 88] and Orca [Bal & Tanenbaum 88]. In contrast to these other distributed systems, the goal of Amber is to execute a single application that performs a parallel computation. However, all of these systems have provided memory consistency at the level of a language object. Many systems provide only *function shipping* while others also provide *data shipping* in a limited form. VDOM extends this model by providing object-based coherence using *fragment objects* and a data-shipping model of sharing called *version consistency*. The addition of these capabilities makes object-based distributed memory models more appropriate for many types of data-parallel applications.

VDOM provides a multi-versioned abstraction of shared object memory. Doing this increases concurrency and provides a basis for the version consistency sharing model. Using multiple versions to improve the performance of read-only transactions is discussed in [Weihl 87]. Using version numbers as the basis for synchronization is similar to [Reed & Kanodia 79], where it is shown that any high-level synchronization mechanism can be built from primitives based on *eventcounts*. VDOM builds upon this idea to incorporate synchronization directly with the data being shared. This not only provides a natural, object-oriented synchronization mechanism but also allows synchronization operations to be used to direct memory coherence.

# 6 Conclusion

We have described Versioned Distributed Object Memory (VDOM), an extension of traditional Distributed Object Memory systems that is tuned to the needs of a large class of scientific numerical applications. The important aspects of VDOM that distinguish it from other DOM systems include:

- Shared objects are cached in local memories with coherence driven by Acquire and Release synchronization *methods* associated with each object. All accesses to a shared object are bracketed with an Acquire/Release pair. This makes *data shipping* a first-class citizen along side of *function shipping* that is featured in DOM.

- Objects are immutable; a *write* operation logically creates a new version of an object. Multiple versions of an object can coexist in one or more local memories as needed to increase concurrency.

- Efficient fine-grain sharing is achieved using *version consistency*, a weak memory consistency model in which an Acquire can request access to a specific version of an object. VC

supplies a data-oriented means for synchronization and ordering of accesses to shared objects.

- A language-level object can be represented by a composition of *fragment objects*, in order to achieve a finer grain of distribution and sharing. This is in contrast with DOM where language-level objects are the indivisible units of coherence.

Our measurements show the inherent performance advantage of the VDOM mechanisms through their reduction in the frequency of system-wide synchronizations; this is particularly useful in certain producer-consumer and write-shared programming styles. VDOM permits object-oriented programming of grid-based parallel applications, with the concurrency and latency-hiding advantages that typically require carefully- and painfully-coded asynchronous message passing.

## Acknowledgments

We would like to thank Jeff Chase, Kathy Faust, Sabine Habert, Chris Hebert, Ed Lazowska, and Bill Lee for their helpful comments. Special thanks to Chris Hebert for producing the diagrams. We would also like to thank the DEC Systems Research Center for providing the Firefly workstations and the Topaz operating system software.

## References

[Adve & Hill 90] Adve, S. V. and Hill, M. D. Weak ordering - a new definition. In *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News*, pages 2–14. ACM, June 1990.

[Bal & Tanenbaum 88] Bal, H. E. and Tanenbaum, A. S. Distributed programming with shared data. In *Proceedings of the International Conference on Computer Languages*, pages 82–91, October 1988.

[Birrell & Nelson 84] Birrell, A. D. and Nelson, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Carter et al. 91] Carter, J. B., Bennett, J. K., and Zwaenepoel, W. Implementation and performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164. ACM, October 1991.

[Chase et al. 89] Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[Faust 90] Faust, K. An empirical comparison of object mobility mechanisms. Master's thesis, Department of Computer Science and Engineering, University of Washington, April 1990.

[Feeley et al. 91] Feeley, M. J., Bershad, B. N., Chase, J. S., and Levy, H. M. Dynamic node reconfiguration in a parallel-distributed environment. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 114–121, July 1991.

[Fox et al. 88] Fox, G. C., Johnson, M. A., Lyzenga, G. A., Otto, S. W., Salmon, J. K., and Walker, D. W. *Solving Problems on Concurrent Processors*, volume I. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Gharachorloo et al. 90] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News*, pages 15–26. ACM, June 1990.

[Habert 90] Habert, S. Distributed shared memory in Amber. Technical report, University of Washington, December 1990.

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Li & Hudak 89] Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Liskov 88] Liskov, B. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Ousterhout 90] Ousterhout, J. K. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[Reed & Kanodia 79] Reed, D. P. and Kanodia, R. K. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

[Schroeder et al. 85] Schroeder, M. D., Gifford, D. K., and Needham, R. M. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 25–35, December 1985.

[Shapiro et al. 90] Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., and Valot, C. SOS: An object-oriented operating system – assessment and retrospectives. *Computing Systems*, 1990.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Tomlinson & Scheevel 89] Tomlinson, C. and Scheevel, M. Concurrent object-oriented programming languages. In Kim, W. and Lochovsky, F. H., editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 5, pages 79–124. ACM Press, New York, New York, 1989.

[Walpole et al. 89] Walpole, J., Blair, G. S., Malik, J., and Nicol, J. R. A unifing model for consistent distributed software development environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 183–190, February 1989.

[Weihl 87] Weihl, W. E. Distributed version management for read-only actions. *IEEE Transactions on Software Engineering*, SE-13(1):55–64, January 1987.