Message-Driven Relaxed Consistency in a Software Distributed Shared Memory

Povl T. Koch   Robert J. Fowler   Eric Jul

Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, 2100 Copenhagen, Denmark
{koch,fowler,eric}@diku.dk

Abstract

Message-passing and distributed shared memory have their respective advantages and disadvantages in distributed parallel programming. We approach the problem of integrating both mechanisms into a single system by proposing a new *message-driven coherency* mechanism. Messages carrying explicit causality annotations are exchanged to trigger memory coherency actions. By adding annotations to standard message-based protocols, it is easy to construct efficient implementations of common synchronization and communication mechanisms. Because these are user-level messages, the set of available primitives is extended easily with language- or application-specific mechanisms. CarlOS, an experimental prototype for evaluating this approach, is derived from the lazy release consistent memory of TreadMarks. We describe the message-driven coherency memory model used in CarlOS, and we examine the performance of several applications.

1 Shared Memory and Messages

We present an approach to distributed parallel computation that combines user-level message-passing with a software distributed shared memory (DSM) based on a relaxed model of memory consistency. Specifically, messages carrying *causality annotations* are used explicitly to drive and control a set of memory-consistency mechanisms derived from the *lazy release consistent* (LRC) protocol [12]. Some annotations allow messages to be transmitted without incurring memory-consistency overhead, while others force the receiver to a state consistent with the sender.

This approach is a simple and sound architectural basis for the construction of hybrid systems that provide both shared memory and message-passing. In particular, this strategy permits and encourages the use of efficient message-based protocols to build a user-extensible set of interprocess coordination (synchronization, scheduling, and communication) mechanisms for controlling an underlying abstraction of shared memory. We present the message-driven memory model, the problems that it addresses, and examples of its use. The model is compatible with hardware- as well as software implementations. We describe a software prototype called CarlOS that was built by applying extensive modifications to the TreadMarks DSM system [11].

There are advantages and disadvantages to message-passing as well as to shared memory. Message-passing gives programmers and compilers explicit control over the choice of data communicated and over the time of transmission. With appropriate interfaces and protocols, it is relatively easy to overlap computation with communication. The explicit nature of message-passing is perceived also as its main weakness; programmers and compilers need to plan and to program explicitly every communication action. Such planning is especially difficult for applications that use complex, pointer-based data structures. When exact access patterns are not known, performance is affected by compromises among the volume of data communicated, the number of messages sent, and the amount of time that processes must wait for messages to be delivered.

In contrast, shared memory is often viewed as a simpler and more intuitive abstraction. In hardware implementations of distributed shared memory, communication occurs without the overhead of processor interrupts, context switching, and software network protocols, even when the system is based on a low-level message-passing network. Mechanisms to reduce or hide memory latencies [9], including update-based coherence protocols [20], are intended specifically to allow memory operations to be overlapped with computation. Related techniques can be used in software implementations to control the overhead of using shared memory by delaying and coalescing communication overhead [3, 12].

1.1 Shared Memory: Problems and Fixes

There are many common operations whose shared-memory implementations perform worse than do their message-based equivalents. Mismatches in alignment and size between the system's memory blocks and user data structures reduce the control that memory-based protocols have over *which* data are communicated. Multiple memory operations are needed to transfer a data structure when it spans two or more blocks. On the other hand, transferring a large block is inefficient when only a small part of it is needed. There is also the false sharing problem: even when the values of individual variables only need be communicated infrequently, co-locating several variables on a large block increases the chance that there will be fine-grain sharing of the block.

There are additional problems specific to the "lazy, demand-driven communication" model implicit in invalidation-based protocols. To propagate a newly computed value from one processor to another, a shared-memory approach needs at least two (cache-miss, data) messages, and the requesting process will be blocked until the reply arrives. If invalidations and acknowledgements are counted, it takes three or more messages. In contrast, message-passing offers an "eager, producer-driven communication" model that allows data to be delivered as soon as it is ready and that overlaps communication with computation. The memory-based protocol requires a separate synchronization protocol, whereas message delivery combines data transfer and synchronization.

Interprocess coordination operations (synchronization and scheduling) are especially important, so many systems, whether hardware- or software-based, supplement their shared-memory implementations with additional mechanisms for synchronization. Additional incentives to give synchronization special treatment are provided by models of memory consistency [1, 6, 8, 12] that require that synchronization operations be identified to the memory coherence mechanism. On message-based hardware, this often means that the implementation of shared memory relies on a set of message-based synchronization mechanisms. In our message-driven consistency strategy, we make those mechanisms visible at the user-level.

Many of the problems with shared memory can be mitigated by using improved implementation techniques. Relaxed models of memory consistency have been used effectively to overlap communication and computation, thus reducing and hiding the latencies of *writes* and of coherence operations. The read-latency problem has been addressed by using update-based protocols [20], by augmenting invalidation-based protocols with prefetching[17], and by multithreading. See [9] for a comparative evaluation of these alternatives.

Software DSMs have fewer opportunities to use buffering and pipelining to overlap computation with communication. Even so, the LRC protocol delays or eliminates many memory-related communication events, update-based protocols can decrease read latencies, and multiple-writer protocols [3] greatly reduce the impact of false sharing.

Although sophisticated implementation techniques improve some aspects of the performance of shared memory, there still are many cases in which message-passing retains its advantages. If one adds message-passing to the user interface of a shared-memory system, however, it has to be done so that message-based protocols and the state of memory are consistent with one another. A conceptually simple approach is to deliver user messages only after all preceding memory operations have been performed.[1] Several groups [13, 14, 15] are working on systems that combine some form of message-passing with shared memory at the hardware level. By using the same pipelines to carry user messages and memory-system messages in these designs, in-order delivery can be ensured without adversely affecting the performance of the memory subsystem.

In some systems, allowing messages to bypass pending memory operations may offer substantial performance advantages. In particular, the strategy underlying the LRC protocol delays performing memory-related communication as long as possible. It also buffers and coalesces such operations. Since many uses of message-passing are intended to avoid the overhead of memory coherence, implicitly forcing memory consistency every time a message is delivered would be counterproductive. The message-driven memory consistency model is intended to provide a means of guaranteeing memory consistency when it is required, while permitting messages to bypass the consistency mechanisms.

2 Message-Driven Memory Consistency

The message-driven consistency model is intended for systems in which there is an underlying message-passing layer on top of which all synchronization and communication mechanisms are built, including the shared-memory abstraction and a user-level message-passing interface. Since everything is built on top of messages anyway, we give up the pretence that synchronization can or should be based on access to variables in shared memory. *All* reasoning, whether formal or informal, about the ordering of memory events is expressed in terms of an ordering relation induced by passing user-level, synchronizing messages. Specifically, if processor A sends a synchronizing message m to processor B, any modifications to shared memory visible on A before m was sent become visible to B when B receives m. If all messages are synchronizing messages, the ordering of memory events is consistent with "happened before" as defined by Lamport [16] for message-passing systems. Since all reasoning is in terms of synchronizing messages, they are the only mechanism needed to drive the memory system into consistent states. A consequence of

[1] In-order point-to-point delivery of messages is not a sufficient condition for correctness. A user-level message might be delivered via the shortest route, whereas a memory coherence operation might be delivered first to another node, e.g. the node that holds the relevant directory entry, before being delivered to the final destination(s).

this strategy is that any message-based protocol for coordinating processes will ensure memory consistency automatically, so long as the protocol can be proven correct using techniques consistent with Lamport's model.

Forcing memory coherence on every message has several side effects that can affect performance adversely. Consider Figure 1. Processors $P1$ and $P2$ share variable x. Processor $P1$ has written x while holding a lock and $P2$ wants to read x. $P2$ sends a "get lock" message to $P1$ and will get a "release lock" message in reply. Since memory is kept consistent with the happened-before (henceforth denoted $\rightarrow$) relation on messages, the "release lock" message correctly ensures that the value of x written by $P1$ is visible at $P2$. If all messages induce $\rightarrow$ on memory, this protocol induces an *unintended symmetry*, because the request message from $P2$ to $P1$ also causes $P1$ to become consistent with $P2$. Furthermore, the transitivity of the $\rightarrow$ means that this unintended and unnecessary consistency will involve other processors. Thus, although there is no intention to induce an ordering between $e3$ and $e4$, that ordering is required by the model because of the (dashed) message from $P3$ to $P2$. This transitivity will extend to every processor with which P1 communicates in the future.

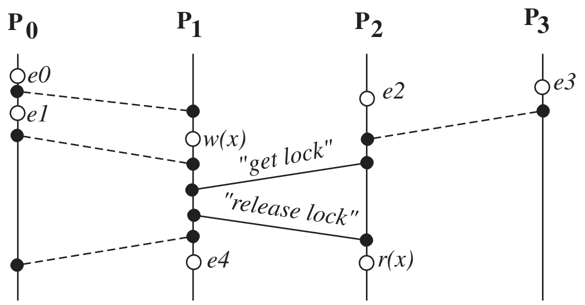

Figure 1: Memory consistency induced by messages. The desired outcome is that a lock protocol (solid lines) make the value of x written by P_1 visible when x is read at P_2. If all messages force "happened before" constraints, then the system becomes overly consistent.

To avoid overhead associated with forcing memory consistency on every message, we propose relaxing the event ordering model so that only designated messages induce $\rightarrow$ on memory. This is accomplished by marking each message with a memory-consistency annotation. In the example of Figure 1, we can eliminate the unintended symmetry and its consequences by marking the "get lock" message such that it does not induce memory consistency.

While the transitivity of $\rightarrow$ is required to capture correctly the notion of potential causality, the actual chains of causality may be more restricted. In Figure 1, for example, events $e0$ and $e1$ on processor $P0$ both happened before the "release lock" message was sent, so the model requires that their effects should be visible on node $P2$ when it reads x. On the other hand, only $e0$ happened before the $w(x)$ on $P(1)$, so if the only real communication between $P1$ and $P2$ is through the value of x, it is not necessary to propagate the effects of $e1$. Furthermore, if the value of x does not depend upon $e0$, its effects also need not be propagated.

We leave for further research a general solution to the problem of limiting the transitivity in message-driven memory consistency models. Our interim approach is to use synchronization protocols that avoid inducing unnecessary memory consistency. We also define an annotation that explicitly limits transitivity by propagating only those changes to shared memory that were performed by the sender. This mechanism is used in the implementation of the global barrier primitive in TreadMarks. A node arriving at a barrier sends consistency information in a system-level message to a barrier manager. When all nodes have arrived, the manager merges the consistency information and sends it back out in the message that signals that the barrier has fallen. Because the union of the individual contributions produces a globally consistent view, it is sufficient for each node to send only a description of its own actions. We include the limited transitivity message in the model specifically for use in implementing barriers, because one of our goals is to always do at least as well as TreadMarks.

2.1 Message Annotations in CarlOS

To implement the memory consistency model described above, each user-level message in CarlOS carries one of these annotations:

RELEASE messages are synchronizing messages. In terms of release consistency memory models, the operation of sending a RELEASE message is a *release* event and the operation of accepting it at the receiver is a matching *acquire* event. Any program that mixes message-passing and shared memory will be consistent if all messages are marked RELEASE, but performance may suffer because the system will be kept overly consistent.

REQUEST messages are non-synchronizing. While it does not hurt to include this annotation in any message, it is intended to be used when a RELEASE message will be returned. CarlOS appends a description of the sending node's knowledge of the state of shared memory so that a precisely-tailored RELEASE message can be sent in response.

NONE messages are non-synchronizing. They are intended to be used when the expected response is not a RELEASE message. They do not interact with the memory consistency mechanisms in any way.

RELEASE_NT messages are the non-transitive form of the release messages.

NONE and REQUEST messages are semantically equivalent, but they have different costs. Using a REQUEST message entails the overhead of including and processing status information. Using a NONE message risks the possibility that a subsequent RELEASE message will contain more consistency information than necessary.

2.2 Forwarding of Release Messages

In many protocols, messages are sent to intermediaries that relay the message contents to their ultimate destination. For example, requests for a mobile resource often are sent to a fixed "home node" or "manager". The manager then forwards each request to the current location of the resource. In CarlOS, protocols that forward NONE or REQUEST messages are relatively easy to manage, because there is no problem with unintended side-effects on memory consistency.

There also are cases in which a RELEASE message is relayed through an intermediary. Consider the problem of implementing a shared task queue. In a typical shared-memory style implementation, the queue is protected by a lock. In addition to the overhead of locking and of migrating some part of the queue's representation to the node currently accessing the queue, the locking mechanism guarantees that the node will become consistent with a past state of every processor that has touched the queue.

To avoid the cost of locking and migration, Carter experimented with an RPC-based queue implementation using a fixed manager [2]. Because the items in the queue could contain references to shared memory, each node explicitly forced memory into a consistent state by performing a `flush` operation before accessing the queue. A similar implementation in CarlOS uses a RELEASE message in each of the *enqueue* and *dequeue* operations to ensure that the eventual recipient of a queued item becomes memory-consistent with the node that created the item. In a straightforward implementation, the queue manager node becomes consistent with every node that enqueues items. Whenever a node dequeues an item, it will be made consistent with the manager and, transitively, with *all* of the nodes that enqueued the items currently in the queue.

We provide a forwarding mechanism to avoid unnecessarily propagating consistency information through intermediaries like the queue manager. When a message marked RELEASE arrives, low-level code that first receives the message is allowed to inspect the message before any memory consistency actions are performed. This code must perform one of three actions on the message. If it *accept*s the message, the required memory consistency operations are performed. It also may *forward* the message and its encapsulated consistency information to another node, or it may use a *store* operation to indicate that the final disposition of this message has been deferred. For the purposes of the memory consistency model, a message is not considered to have been *delivered* to user-level until the *accept* operation is performed.

3 Use of Annotated Messages

Because CarlOS does not include any built-in synchronization beyond messages, the first order of business was to allow the system to function as a conventional DSM. We therefore used simple message-based protocols to provide several different types of lock, two kinds of barrier, and a centralized manager for shared work queues.

The standard CarlOS lock uses a simple distributed queue protocol. To acquire a lock, a node sends a REQUEST message to the lock's manager node, which in turn forwards the message to the node that last requested the lock, *i.e.* the node at the tail of the queue. If the lock is not held, then the previous holder sends a RELEASE message immediately. Otherwise, the requesting node joins the request queue. When the lock is released, the node at the head of the queue is notified using a RELEASE message. Semaphores and condition variables have similar implementations.

Each TreadMarks-style barrier is assigned a manager node. Clients arriving at a barrier send RELEASE messages to the manager. If this is a global barrier, RELEASE_NT messages can be used. The manager node accepts the arrival messages to make itself consistent with all of the client nodes. To signal the fall of the barrier, the manager sends departure messages marked RELEASE to the client nodes. When each client accepts the departure message, it becomes consistent with the manager and, hence, with all of the other clients.

High-level coordination mechanisms also have straightforward implementations. Stacks and queues for shared work are built using the fixed manager strategy described in the previous section. Enqueue requests and dequeue replies are marked RELEASE, while the dequeue request messages are marked REQUEST. The manager code acts as a forwarding agent for the messages in the queue; it never accepts any RELEASE messages.

Many numerical applications have communication patterns amenable to message-passing. Prominent examples include hydrodynamics and engineering codes that iteratively solve partial differential equations using finite difference or finite element techniques. Small amounts of data are easy to include in a message. If the data are numerous or difficult to marshall, it is easier to use a shared-memory style of communication combined with a notification message marked RELEASE sent to the node that will use the data. If the underlying memory coherence mechanism uses update rather than invalidation, the actual data transmission occurs eagerly and asynchronously when the notification message is sent.

4 Design and Implementation

CarlOS ("Commodity Architecture Research Laboratory Operating System") is being developed as a platform for research in distributed parallel computing on workstation networks. The current hardware configuration consists of four Digital AXP workstations running DEC OSF/1 and connected via Ethernet. Future expansion will include the addition of a second Ethernet to support experiments with very low-cost network upgrades, a high-performance network (ATM), and more nodes.

A major goal of the CarlOS project is to explore hybrid distributed computing environments that integrate aspects of distributed shared memory based on “flat” memory spaces with aspects of distributed object-based languages [10]. This objective affects the design in several ways. In particular, the development of our message-driven memory consistency model was motivated by the need to combine explicit user-controlled object migration, function shipping in the form of remote invocation, and a flat, coherent address space.

Currently, CarlOS runs entirely in Unix user mode and uses the standard Unix system interface. The implementation strategy is evolutionary and will entail some kernel-level support, particularly in the network drivers. We began with the TreadMarks [11] code. While the basic mechanisms of lazy release consistency are intact, data structures and internal protocols have been restructured extensively to support the internal concurrency and asynchrony implicit in the message-driven consistency model. Other modifications include support for fine-grain user-level multithreading.

4.1 Address Space

Applications see an address space containing three disjoint regions. First, each node has a *private memory* region. This is used internally by CarlOS. Applications can use it for node-local private data structures that are not shared. Second, we provide a *non-coherent shared memory* region in which address mappings are kept consistent across all nodes. Although CarlOS does not maintain the consistency of this region, the single address map provides a consistent interpretation for pointers to objects in this region. The intention is that memory consistency for this region will be provided by run-time libraries using message-passing protocols such as those used in Amber [4]. Third, the *coherent shared memory* region contains user-allocated data kept consistent with the message-driven coherency mechanism.

4.2 Memory Consistency Mechanisms

In both TreadMarks[2] and CarlOS, the execution history of each node is divided into an indexed sequence of *intervals* whose endpoints occur at the acquire and release events executed on that node. In TreadMarks, these occur within the execution of built-in synchronization primitives. In CarlOS, they occur when RELEASE messages are sent and accepted, respectively (see Section 4.3). The memory-consistency state of each node is summarized by a *vector timestamp*, each element of which is the index of the most recently seen interval from the corresponding node. Each interval is summarized by a list of *write notices*, one for each page that was modified in the interval.

Modifications to pages in shared memory are detected using the Unix virtual memory protection facility (`mprotect`) and a SIGSEGV handler. All clean shared pages are marked read-only. On a write-access fault to a protected page, a copy (a *twin*) is created and the page is marked read-write. When the node receives a write-notice for a modified page, or when another node requests the page or its modifications, the page is compared with its twin and the modifications are recorded in a run-length encoded *diff* structure. Then, the twin is removed, and the page is marked read-only. Invalid pages that have not been discarded can be brought up to date by applying an appropriate sequence of diffs, perhaps from multiple writers. Modifications can be requested from another node using a *diff request* message. A node also can request an entire page if it does not hold a copy.

4.3 Annotated Messages

Sending a CarlOS message is an asynchronous operation. CarlOS messages currently are implemented using UDP/IP datagrams supplemented with a sliding window protocol to assure reliable, in-order delivery. The CarlOS message interface defines a form of *active messages* [19]. Each message contains a pointer to a handler function that is invoked when the message is received and to which the body of the message is passed as an argument. Although using active messages has a negligible performance advantage in the current implementation on top of OSF/1 and UDP/IP, it is still a useful structuring technique for building multithreaded systems. Future versions of CarlOS built closer to the underlying hardware and using faster, lower-latency networks should be able to realize a substantial performance advantage.

A memory consistency annotation is a user-visible component of the message. Any consistency information appended by CarlOS is invisible at the user level. A NONE message does not interact with the memory consistency mechanisms. Currently, CarlOS piggybacks the vector timestamp of the sending node onto a REQUEST message. In the future, it also may include additional information to guide a hybrid update/invalidate protocol.

When sending a RELEASE message, CarlOS creates a new interval on the sending node. It then appends to the message a vector timestamp that represents the minimum timestamp required for a recipient to become con-

[2] In discussing the low-level memory coherence mechanisms in CarlOS, we often use descriptions that also apply to TreadMarks. Our presentation focuses on the differences between the systems, and space limitations prevent us from including too much detail that has appeared elsewhere. A detailed description of the TreadMarks implementation can be found in [11].

sistent on the basis of other information in the message. This timestamp is necessary to handle forwarding correctly. The consistency information appended to the message comprises interval descriptions that the sender has determined are needed to make the immediate receiver consistent. If an invalidation-based consistency strategy is used, the interval descriptions contain only write notices. If an update or hybrid strategy [7] is used, the message also will contain a set of diffs. Thus far, we have used only the invalidation strategy in CarlOS.

When a node accepts a RELEASE message, it performs the actions required by an acquire event in LRC. This entails creating a new interval, applying all of the write notices in the message, and updating its vector timestamp. Using the invalidation strategy, applying the write notices means invalidating the named pages. Using the update or hybrid strategies, pages to which a "complete" set of diffs can be applied remain valid. Occasionally, a forwarded message does not contain enough consistency information. This is checked by comparing the accepting node's current vector timestamp with the required vector timestamp in the message. If the required vector timestamp is "later" in any dimension, the consistency information in the message is inadequate. In such cases, the accepting node requests additional information from the original sender of the message.

Compared with TreadMarks, additional care is required in CarlOS for the creation, propagation, and application of consistency information. This is due to the concurrency and asynchrony implicit in the existence of unsolicited RELEASE messages and of multiple outstanding requests from multithreaded nodes.

The RELEASE and RELEASE_NT annotations differ in that messages marked with the latter contain only consistency information about intervals created at the sending node. The correct vector timestamps are sent, however, so the receiving node is able to determine whether it has been left with an inconsistent view of memory. The inconsistency can be resolved by requesting additional consistency information.

When a user-level message is delivered by the sliding window protocol, control is passed to the handler function specified in the message. As in most implementations of active messages, message handlers are intended to be low-level code that perform a limited amount of processing as an extension to an interrupt-handling function. They are not allowed to block, nor should they touch coherent shared memory. Critical sections between the message handlers and higher-level code are handled by blocking the delivery of incoming messages. The handler is allowed to inspect the message. Before the handler terminates, it must either inform CarlOS that the message has been stored for later disposition or it must dispose of the message, either by accepting it or by forwarding it to another node. Forwarding or accepting the message frees the message buffer.

4.4 Support for multithreading

The current implementation of TreadMarks allocates only one thread of control per physical node. This means that a node blocks on a page or diff fault until it receives the needed reply. This blocking leads to high idle times. Multiprogramming is the classic technique for hiding the latencies of blocking operations, so CarlOS is designed to support multiple user threads per node. We take the position that each language implementor should be able to build a customized thread package, so we have designed support for building thread packages on top of CarlOS. We provide a hook to make an upcall to a user-level scheduler to prevent user code from blocking on remote coherent shared memory operations (page requests, diff requests, etc.).

The message-driven memory consistency mechanism is available to implement inter-node synchronization and scheduling. The upcall interface is available to schedule or to reschedule threads when they block or unblock, thereby masking remote latencies. Upcalls out of handlers for active messages provide a mechanism for building remote invocation and other interfaces to thread schedulers on remote nodes. The non-coherent shared memory region is ideal for the allocation of thread control blocks and stacks, because it allows pointers to be used to name these structures while allowing message handlers to access them safely.

To support user-level multithreading, we reimplemented all of the internal message-passing in TreadMarks to use the CarlOS message interface and its reliable message delivery facility. This eliminates blocking of Unix receive calls. We also modified the internal data structures so that multiple page and diff requests can be left outstanding.

5 Performance Measurements

In this section, we report our early experiences with the performance of CarlOS. The performance measurements were made on a cluster of DEC 3000/300 workstations with 150 MHz Alpha AXP processors running DEC OSF/1 v1.3 on an isolated Ethernet segment. Timings were obtained using a fine-grain low-overhead timing facility based on AXP architecture's processor cycle counter [5].

We examine at least two versions of each of three applications running on CarlOS: TSP, Quicksort, and Water. One version of each application is a "strictly shared memory" program that runs on TreadMarks and is synchronized using locks and barriers. The other versions are hybrids that still keep data in coherent shared memory, but that use message-passing for high-level process coordination. We start out by providing an overview of the results. Each of the applications is described and analyzed in more detail in a later section.

We first compared the unmodified applications running on TreadMarks and running on CarlOS to examine the impact of replacing the built-in synchronization mechanisms with similar implementations using annotated messages.

For the TSP and Quicksort application, the performance penalty is a 5-6% increase in total execution time. This is mainly due to the generality of CarlOS message handling. Both of these programs have high contention for locks and suffer from a large increase in acquisition latency. Water performed as well on CarlOS as on TreadMarks.

Figure 2 summarizes the average execution times on CarlOS of the two versions of each of the three applications. The original lock and barrier versions of the applications are marked '/lock' and the hybrid versions are marked '/hybrid'. The execution times have been broken down into application computations (marked User), the cost of OSF/1 calls (Unix), all CarlOS message-passing and shared memory overhead (CarlOS), and time spent waiting for remote operations to complete (Idle).

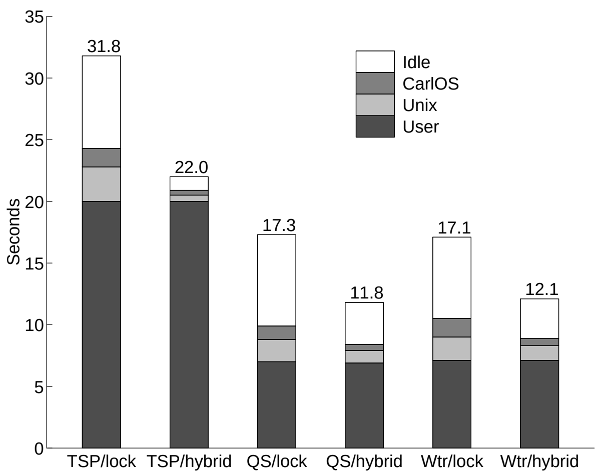

Figure 2: Execution breakdown for the TSP, Quicksort, and Water applications on four nodes.

Since the algorithms are unchanged in each variation, the user times in each pair of applications are almost identical. The main benefit from using the hybrid strategies is a reduction in communication costs. The hybrid programs all use far fewer messages than the original versions. Although message size increases, the total volume of data communicated decreases. All three components of overhead are reduced. The largest benefits are in the reduction of the idle-time components of overhead. Even using the hybrid strategies, the idle-time components remain large compared to the CarlOS and Unix components.

5.1 The TSP Application

The Traveling Salesman Problem application (TSP) uses a branch-and-bound algorithm to find the shortest path through 19 cities. In the original version of TSP, a lock is used to provide mutual exclusion among writers trying to update the current bound ("best tour") and another lock is used to protect a shared work queue. Since the bound is updated with a single instruction, it is not necessary to provide mutual exclusion between readers and writers.

We replaced the above lock-based implementation of the queue with a centralized implementation using messages. While the changes to the source code were small, the modifications did change the locations at which some parts of the computation are performed. The manager node on which the queue is located is responsible for generating the queued tours, and it also participates in the search for the best tour. To request a tour index from the queue, a client node sends a REQUEST message to the manager, which then sends its reply in a RELEASE message. The tour descriptors are kept in coherent shared memory. In the CarlOS version, the client nodes never touch the queue directly, so the representation of the queue is never propagated to the clients. As in the lock version, the first step in updating the bound in the CarlOS version is to test the value on the client node. If a better bound has been found, it is posted to the master in a REQUEST message. The master writes the value to shared memory and replies with a RELEASE message. The new value becomes visible to each client when the latter next accepts a RELEASE message from the manager. Message-passing is used only to implement the shared work queue. Shared memory is used to distribute the current bound and the tour descriptions. While better performance could have been obtained easily by transferring the data in messages, the point of this example is to explore the impact of alternative synchronization mechanisms.

Execution times and communication statistics for TSP are presented in Table 1. The columns display the number of nodes (N), the execution time speedups,[3] the total number of messages sent, their average size in bytes, and an estimate of network utilization.[4] The network utilization figure is included to illustrate the degree to which the Ethernet is a bottleneck.

Version	N	Time (s)	Speed-up	Messages #	Messages Size	Net Util.
Lock	2	52.3	1.64	5,838	133	1%
	3	39.7	2.16	8,626	168	3%
	4	31.8	2.69	10,403	219	6%
Hybrid	2	44.9	1.91	1,204	356	1%
	3	31.0	2.76	1,916	426	2%
	4	22.0	3.89	2,198	498	4%

Table 1: The TSP application on CarlOS using coherent shared memory and either locks or message-passing.

The hybrid version of TSP obtains much better speedup than does the lock version (3.89 versus 2.69 on four nodes). The performance problem with the lock implementation of TSP is that the shared work queue migrates among all the

[3] Speedups are calculated relative to the single-node execution of the same application.

[4] Calculated assuming an ideal 10 Mbit/sec. Ethernet. (Ethernet and UDP headers are **not** included, so the network utilization estimate is conservative.)

nodes that manipulate it. Although the network load is light (1-5%), relatively slow message delivery contributes to the high idle times. See Figure 2. The hybrid version sends less than one-fourth as many messages. The average message size increases in the hybrid version because many small messages are eliminated, and some of the consistency information that they would have carried appears in the remaining messages.

5.2 The Quicksort Application

The Quicksort application sorts an array of 256K integers in shared memory. When subarrays get smaller than 1K elements, the nodes use a local Bubblesort. Initially, a shared stack contains a descriptor for the whole array. In each step of the algorithm, a node pops a descriptor off the queue and sorts the corresponding subarray. If the subarray is bigger than the threshold of 1K elements, the node partitions the array, pushes a descriptor for the smaller subarray on the queue, and recursively Quicksorts the larger subarray. When the whole array has been sorted, a barrier is used to collect all of the sorted subarrays, thereby making all nodes consistent.

In the lock version of this program, the shared work stack is protected by a lock, so each operation on it requires that a lock be acquired and released. The shared work stack thus migrates among all of the nodes that manipulate it, and each node that manipulates the stack becomes consistent with all of the previous manipulators of the stack. The use of diffs ensures that only data near the top of the stack is actually communicated.

In a hybrid version of Quicksort (named Hybrid-1), we used a non-migrating shared work queue. The node that hosts the queue manager code also participates in the sorting. The manager node represents the queue as a list of pointers to "enqueued" messages that have been stored. When a remote node issues a dequeue request, the stored message at the head of the queue is forwarded. If the request for work comes from the manager node, the message is accepted. Note that a client node waits for the reply message in a dequeue operation, but that enqueue operations are completely asynchronous.

To measure the impact of using correct annotations and the use of the CarlOS forwarding mechanism, we created two minor variations of the hybrid version of Quicksort. In one variation, all messages for accessing the shared work queue are RELEASE messages. This is the Hybrid-2 program. In another variation, the forwarding mechanism is not used. The performance of this program is nearly identical to that of Hybrid-2.

The performance of the lock and hybrid versions of Quicksort is summarized in Table 2. This Quicksort program is communication-intensive; a four-node execution consumes half of the gross bandwidth of the Ethernet for user data. The request-reply pattern of communication in the lock version limits the possible overlap of computation and communication. This is reflected in a speedup of 1.54 on four nodes. In contrast, the Hybrid-1 program gains a speedup of 2.27. This is partly due to a 58 percent decrease in the number of messages sent and a 31 percent decrease in data communicated. It requires substantial overlap of communication and computation, however, to get a speedup of over two with a network utilization of at least 50 percent. The asynchrony of the "enqueue" operation in the hybrid program provides this.

Version	N	Time (s)	Speed-up	Messages #	Messages Size	Net Util.
Lock	2	19.6	1.36	2,426	1,209	12%
	3	18.6	1.44	5,144	1,446	32%
	4	17.3	1.54	6,866	1,560	50%
Hybrid-1	2	17.5	1.53	1,406	1,704	11%
	3	13.9	1.93	2,282	2,265	30%
	4	11.8	2.27	2,870	2,564	50%
Hybrid-2	4	14.2	1.89	4,361	2,254	55%

Table 2: The Quicksort application on CarlOS using coherent shared memory and either locks or message-passing.

The correct use of consistency annotations and of forwarding reduces the amount of consistency overhead, as measured in the numbers of diffs created and of diff requests sent. This overhead is paid in the form of the immediate costs of creating intervals and diffs as well as in the cost of communicating them in messages. There is also a delayed cost in the form of a global garbage collection of these data structures.[5] Thus, Hybrid-1 consumes resources more slowly than does the original program. On this problem instance, Hybrid-1 avoids a garbage collection that the original program required. For much larger, longer-running problem instances, the difference would not be whether there is a garbage collection, but rather how many garbage collections are necessary.

The messages in Hybrid-2 are all RELEASE messages, so it creates many more intervals and diffs than does Hybrid-1. Thus, while Hybrid-2 still benefits from the message-based queue, it also pays for the added diffs. This increases the volume of data communicated, and it represents a large relative increase in the amount of consistency overhead. The relatively small contribution of consistency overhead to total execution time, however, limits the magnitude of the effect.

5.3 The Water Application

Water is a molecular dynamics simulation application from the SPLASH suite [18]. Each iteration of the program consists of several phases separated by barriers. In a problem instance with P processors and N molecules, each processor is assigned N/P molecules. In the most

[5] CarlOS uses TreadMarks' memory management mechanisms for system data structures (intervals, diffs, etc.). When the free space for system structures falls below a threshold, a global garbage collection is performed, thereby forcing more messages to be sent.

communication-intensive phase of each iteration, the processors compute the intermolecular forces for all pairs of molecules. While pairs are examined in this phase, non-zero forces are computed only if the molecules are close to each other. The resulting force vectors acting on each molecule and on each atom within the molecule are accumulated in a net force vector that is part of the molecule's representation. This computation on molecule pairs is partitioned among the processors by making each processor responsible for computing the interactions between its set of molecules and half of the remaining molecules. Since several processors contribute to the force computation for each molecule, it is necessary to coordinate their updates. Using a fine-grain approach, each processor could update the force vector of each of its own molecules as many as $N(1-P)/2P$ times and that of each of the other molecules N/P times. Since the net force on each molecule is the vector sum of the individual contributions, the SPLASH report suggests reducing the amount of interprocess interaction by having each processor accumulate its own contributions and then perform a single update to the force vector of each molecule. The lock version of Water does this by accumulating all of the forces in a private array in the first part of the phase and by updating the molecules in the second part of the phase. To ensure that the updates are applied correctly, each molecule is protected with a lock.

The hybrid version of the program was derived from the lock version by defining an update function to replace the lock-update-unlock sequence for each molecule. The node that generates the update information sends a NONE message to the node that owns the molecule to invoke the update function. The sequential delivery property of CarlOS messages guarantees that the updates are applied atomically, thus eliminating the need to use locks on individual molecules.

Version	N	Time (s)	Speed-up	Messages #	Messages Size	Net Util.
Lock	2	23.3	1.34	6,920	368	9%
	3	19.4	1.61	11,348	374	17%
	4	17.3	1.81	15,423	379	27%
Hybrid	2	18.4	1.70	2,546	889	10%
	3	14.4	2.20	4,155	876	20%
	4	12.1	2.58	5,634	871	32%

Table 3: The Water application on CarlOS using coherent shared memory and either locks or message-passing.

We ran the lock and hybrid version of Water for 343 molecules for five steps. The results are summarized in Table 3. For four nodes, the hybrid version sends about a third less data than does the lock version. Idle time is reduced from 6.7 to 3.3 seconds per node. This is a 51 percent reduction.

In the hybrid versions of the other two programs, we used function shipping to implement process coordination mechanisms. In this case, messages are used to eliminate explicit synchronization by using user-level function shipping to eliminate both the need to migrate data in shared memory and the need to perform explicit synchronization.

5.4 Choice of Annotations

One important issue is whether or not all of the consistency annotations in the model are necessary. For example, the extra overhead of a REQUEST message compared to that of a NONE message in the current implementation is the cost of handling a vector timestamp at the sender and receiver. Currently, this adds two bytes per node to the message. It incurs a direct cost of between 750 and 2350 extra processor cycles, about 5 and 15 microseconds, respectively. Since variation is most likely due to cache misses, an accurate accounting of the total cost should include the additional cache misses incurred to bring back the data that were replaced. In our current environment, these costs are dwarfed by the costs of system calls on OSF/1, the UDP/IP protocol stack, and message latencies on the Ethernet. On the other hand, these times are a substantial fraction of the time necessary to pass a small message on a current-generation multicomputer interconnect or on a streamlined ATM. Furthermore, while the vector timestamp is small compared to typical Ethernet packets, it is a large part of an ATM frame. Thus, while making the distinction between NONE and REQUEST is overkill in our current environment, we believe it is worth retaining in the design.

Similarly, it would eliminate a potential source of programming errors if all user-level messages were RELEASE messages. In our current implementation, the additional overhead of a RELEASE message compared to that of a NONE message is about 30 microseconds plus the time to process the write notices that the message contains. The overhead per write notice has a wide variation, depending on how much work needs to be done. For the lock and hybrid versions of TSP, the average costs per write notice are 42 and 52 microseconds, respectively. For the two versions of Quicksort, the costs are 125 and 141 microseconds, and for Water, they are 94 and 95 microseconds. These costs are large enough that they result in measurable differences when all messages are marked RELEASE in these programs. On systems with high-performance interconnects, the additional overhead will be more important.

Our experiments with Quicksort showed a significant, but not crippling, penalty when all messages were marked RELEASE. Running the hybrid versions of the TSP and Water applications with all messages marked RELEASE, however, increased execution times by only 2.4% and 1.4%, respectively.[6] The explanation is that most of the time the only benefit of not using a RELEASE message is the 30-microsecond fixed overhead. This is because, in these

[6] For TSP, it meant sending twice as much data, because RELEASE messages are bigger than REQUEST messages. Since the messages were small in the first place, this had no significant effect on the total execution time.

applications, most of the write notices will be transmitted eventually, so avoiding the use of a RELEASE message in one place merely delays the transmission of some number of write notices until a later RELEASE message is sent. This effect would occur in Figure 1 if it were extended with a request for a lock from $P1$ to $P2$ followed by a RELEASE message from $P2$ to $P1$ in reply. The savings realized by marking the first request from $P2$ REQUEST rather than RELEASE are illusory because the same write notices will be piggybacked onto the later RELEASE message. In the programs that we studied, the cost of acting on a specific write notice does not change much over time, and all nodes do become mutually consistent, so there cannot be a large impact.

There will be a high penalty for using RELEASE messages unnecessarily in cases in which there is fine-grain false sharing and a frequent exchange of messages that do not need to synchronize memory. Despite the multiple-writers protocol, the synchronizing effect of using RELEASE annotations on the non-synchronizing messages will force the shared pages constantly to be invalidated and "unnecessary" diffs to be requested. Thus, we know that artificial examples can be constructed to illustrate the importance of non-synchronizing messages to the message-driven consistency model. We have not yet seen such an example arise in practice.

6 Discussion and Conclusions

The message-driven memory consistency model is intended to provide a foundation for constructing systems that integrate message-passing and shared memory such that each of these abstractions can be used for purposes for which it is best suited. Message-passing provides an explicit communication mechanism that is especially well suited for constructing process coordination mechanisms. The shared memory abstraction provides a single global name space for shared data. Coherent caching techniques provide an implicit communication mechanism to move and replicate data to the processors that access them. Our early experiences with CarlOS indicate that the model has fulfilled our expectations.

We report performance measurements for CarlOS based on our early experiences with a few applications and a very small cluster of high-performance workstations connected by Ethernet. The main benefit from using our message-driven consistency model is derived from the use of high-level process synchronization and scheduling mechanisms. Compared to locks and barriers, using these mechanisms reduces the number of messages exchanged. While message sizes increase, there is a net reduction in the total amount of data transferred. The asynchrony of message transmission also helps to overlap computation and communication. Even then, there still is a substantial amount of idle time associated with remote operations. CarlOS is designed to support user-level multithreading to address this problem as well as to improve the distributed programming model by providing function shipping and thread migration mechanisms in a well structured user interface. We currently have two multithreading packages under construction.

The message-driven consistency model provides several annotations that are intended to provide a means of limiting "unnecessary consistency". Our initial experiences with a few applications running on our prototype have shown limited benefit to this strategy. To a certain extent, this is an example of Amdahl's law in action. Given high overheads and latencies in the operating system and network, the memory consistency code itself does not contribute much to overall execution time for the applications that we measured. Even doubling this consistency overhead would have a small effect. In other contexts, such as more modern networks and finer-grain applications, consistency code overhead may be a larger part of overall execution time, and the choice of annotations will become more important.

While synchronizing (RELEASE) messages have substantially higher overhead than non-synchronizing ones, using more than the required number of RELEASE messages had a smaller-than-expected effect on execution time in our experiments. In these cases, the choice of annotation mostly affects *when* a particular consistency operation is performed rather than *whether* it is performed. Even for these programs, however, the choice of annotations gives the programmer some degree of control over when the program will incur consistency overhead. We currently are investigating alternative implementations of synchronization mechanisms designed to move consistency operations out of the critical path.

The performance figures that we report were measured on a very small, four-node cluster of powerful workstations connected by Ethernet. Our immediate plans include evaluating the performance of CarlOS on larger clusters and more modern networks.

Acknowledgements

We are very grateful to Alan Cox and Willy Zwaenepoel of Rice University for giving us access to TreadMarks. The equipment was provided by a grant from The Faculty of Natural Sciences, University of Copenhagen. Lars Jarnbo Pedersen profiled CarlOS using fine-grained timers. The anonymous reviewers and our shepherd, Michael Scott, provided very valuable comments. Special thanks to Niki Fowler for weeding out phrases too literally translated from Danish and for smoothing out the text in general. We would also like to thank Marc Shapiro and Julien Maisonneuve at INRIA Rocquencourt (France) for granting us access to their computing facility.

References

[1] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] John B. Carter. Efficient distributed shared memory based on multi-protocol release consistency. Ph.D. thesis, Rice University, September 1993.

[3] 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*. To appear.

[4] Jeffrey 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 12th ACM Symposium on Operating System Principles*, pages 147–158, December 1989.

[5] Digital. Alpha architecture handbook. Digital Press, 1992.

[6] M. Dubois, C. Scheurich, and F.A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, May 1986.

[7] Sandhay 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 244–255, May 1993.

[8] 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*, pages 15–26, May 1990.

[9] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

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

[11] Peter Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Conference*, pages 115–132, January 1994.

[12] Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, May 1992.

[13] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B-H. Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, San Diego, May 1993.

[14] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the Alewife multiprocessor. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, pages 195–206, July 1993.

[15] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenbaum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Conference on Computer Architecture*, pages 302–313, April 1994.

[16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[17] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, June 1991.

[18] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[19] Thomas von Eicken, David E. Culler, Seth Copen Goldstein, and Karl Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[20] Larry D. Wittie, Gudjun Hermannsson, and Ai Li. Eager sharing for efficient massive parallelism. In *1992 International Conference on Parallel Processing*, pages 251–255, St. Charles, IL, August 1992.