PAPER

A False-Sharing Free Distributed Shared Memory Management Scheme

Alexander I-Chi LAI[†], ***Student Member***, **Chin-Laung LEI**[†], ***and*** **Hann-Huei CHIOU**[†], ***Nonmembers***

SUMMARY Distributed shared memory (DSM) systems on top of network of workstations are especially vulnerable to the impact of false sharing because of their higher memory transaction overheads and thus higher false sharing penalties. In this paper we develop a dynamic-granularity shared memory management scheme that eliminates false sharing without sacrificing the transparency to conventional shared-memory applications. Our approach utilizes a special threaded splay tree (TST) for shared memory information management, and a dynamic token-based path-compression synchronization algorithm for data transferring. The combination of the TST and path compression is quite efficient; asymptotically, in an n-processor system with m shared memory segments, synchronizing at most s segments takes $\mathbf{O}(s \log m \log n)$ amortized computation steps and generates $\mathbf{O}(s \log n)$ communication messages, respectively. Based on the proposed scheme we constructed an experimental DSM prototype which consists of several Ethernet-connected Pentium-based computers running Linux. Preliminary benchmark results on our prototype indicate that our scheme is quite efficient, significantly outperforming traditional schemes and scaling up well.

key words: *distributed shared memory, aggressive consistency, distributed synchronization, threaded splay tree, false sharing*

1. Introduction

Along with the rapid development of the networks of workstations (NOW) in recent years, software distributed shared memory (DSM) [26], [28], [33], [35] becomes more and more important than before. A software DSM typically uses the paging capability of modern CPUs to provide a virtual shared memory space on the top of NOW, hiding the communication details and offering a transparent programming environment. Such a design is an economic alternative to dedicated multiprocessor systems, with additional benefits like flexibility and scalability. However, those traditional paging-based designs inevitably induce *false sharing* [11], [17], [36], a phenomenon arises when multiple hosts update different parts of a large memory unit (a page) and thus cause that unit unnecessarily *thrashing* among them. The DSM/NOW performance is particularly vulnerable to the impact of false sharing, because the network latency (and the false sharing penalty) is generally much higher than the access time of true hardware shared memories. Previous solutions to this problem usually rely on extra false-sharing-avoiding mechanisms controlled either manually [5]–[7], [18], [19], [27], [37] or automatically [11], [17], [30], [31], [36]. Unfortunately, the manual approaches increase programming difficulty and severely reduce the compatibility to existing applications, and the automatic ones cannot totally prevent false sharing.

In fact, false sharing comes from the combination of both caching and a fixed, large granularity (unit size): When hosts are caching and competing for different pieces of information in one single page, false sharing occurs. Since caching is very important to performance, a *dynamic granularity* scheme thus becomes the choice to eliminate false sharing. Also, such a scheme must retain high scalability, low communication overhead, and sufficient programming transparency to be suitable for DSM/NOW environments. To fulfill all these requirements, we propose a new high-performance, dynamic-granularity shared memory management scheme which, to the best of our memories, is the first one specifically designed for DSM/NOW environments. The key mechanism of our scheme is a dynamic token-based synchronization algorithm that utilizes *path-compression* technique [9], [20]. In our algorithm, one shared memory segment is treated as a token passing from one host to another. To achieve best scalability and performance, our algorithm does not assign any central or fixedly distributed manager hosts to control the token locations. Instead, each host keeps its own guess about the probable "owners" of the tokens; when one host wants to access a specific token, it simply requests the token from the probable owner who either grants the token (if it actually owns that token) or forwards the request to the next probable owner (if not). All hosts along the forwarding route will then "compress the path" by updating their owner information to indicate the new token owner. In addition, our algorithm is capable of dynamically splitting/merging tokens at run-time to reflect the status/ownership changes of address-adjacent segments, and also to suppress external fragmentation of conventional segmentation schemes. The status information is controlled by a shared memory directory called *threaded splay tree* (TST), which is a special data structure based on splay trees [32] by linking neighboring leaf nodes bi-directionally to trace the status change of adjacent

Manuscript received November 26, 1998.
Manuscript revised May 28, 1999.

[†]The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, 10764, R.O.C.

memory segments. Moreover, the splaying property of TST perfectly caches frequently accessed memory segments near the root, further improving the overall efficiency of our algorithm. Analytically, our scheme, issuing s accesses to an n-processor system with at most m shared memory segments takes $\mathbf{O}(s \log m \log n)$ amortized computation steps and generates $\mathbf{O}(s \log n)$ communication messages, respectively. Also note that our scheme does not need any special hardware nor software supports, meaning that our scheme is compatible to both page- and segmentation-based memory management systems and most CPU families, operating systems, compilers, and programming languages.

To investigate the real-world performance characteristics of our scheme, we constructed an experimental DSM platform called *ULTRA* (for *U*niversal *L*ayer for *T*ransparent *R*emote *A*ccess), which is currently developed on several Ethernet-connected Pentium-based computers running Linux. The DSM transparency in *ULTRA* is retained by a flat, directly-accessible memory layout and critical-section style programming interfaces, with only minimal syntax modifications that locks data memory segments directly instead of an extra *mutex* (*mut*ual *ex*clusion) synchronization variable. In addition, the *ULTRA* memory consistency semantics also incorporates other performance optimization features, such as concurrent writes to the same location and read-only locks. This new consistency semantics is called the *aggressive consistency* (AC), to name after its concurrent-write support. Preliminary benchmarking results on this prototype (with concurrent-write and other optimization features disabled to get the raw performance) show that our memory management scheme is very efficient, outperforming other traditional DSM management schemes by up to 800% and delivering up to 6X speedup on our 8-host NOW configuration for computation intensive applications.

The rest of this paper is organized as follows. In Sect. 2 we survey related works. In Sect. 3 we present the TST data structure and the dynamic token-based synchronization algorithm used in our scheme. Section 4 demonstrates the *ULTRA* prototype system as well as the preliminary benchmark results. Finally, conclusions and perspectives are given in Sect. 5.

2. Related Works

Conventional page-based DSM systems provide a transparent, directly accessible shared memory space via the paging capability in modern CPUs. To reduce the virtual memory processing overhead, page-based DSM systems often utilize some kind of relax consistency semantics like the release consistency (RC) [16], [38] or the lazy release consistency (LRC) [21], a more efficient implementation of RC. TreadMarks [3], [4], [21] is one representative of the LRC systems. Constructed as a library on top of Unix systems, TreadMarks takes advantage of Unix virtual memory management system calls `mmap()` and `mprotect()` to control access to shared pages. Moreover, TreadMarks utilizes a mechanism called *diff* that compares and collects modifications to different parts of the same page to reduce false sharing [4] and enhance performance. Another RC-based example, Munin [5], [6], is constructed as an extension of its underlying operating system. Performance of Munin benefits from a access-pattern-specific synchronization mechanism using various algorithms for different access patterns such as producer-consumer, migratory, etc.; however it is the responsibility and risk of programmers to use the proper access pattern for each data type, which also implies that conventional parallel applications need substantial modifications to be ported to Munin. Another relax consistency semantics used by Midway [7], [37] is the entry consistency (EC). In EC shared data will be synchronized at a processor when that processor locks a *mutex* that "guards" the data; that is, the guarded data will be fetched together with the associated *mutex*. The association between the "guard" *mutex* and shared data must be established either explicitly by the programmer or automatically by a more convenient but less precise run-time prediction mechanism [30]. Some other researchers have reported that EC and LRC are equipotential in terms of performance [1]. In general, page-based DSM systems provides a more transparent and convenient programming support than object-oriented designs, but they cannot avoid false sharing because of page granularity.

In contrast to page-based ones, object-oriented DSM systems represent the shared memory as a shared container of language-level objects and hide remote accesses in the member functions associated with the objects, meaning that the shared memory is not directly visible to applications but accessible through special function calls. Some of them are built on top of platform-independent communication facilities such as the parallel virtual machine (PVM) [15], [34] for higher interoperability. For example, in the C Region Library (CRL) [18], [19] the consistency unit is a "region," an object-like entity with a unique ID and must be explicitly locked via system calls to translate the ID into a temporary address for application use. A CRL incarnation on the MIT Alewife machine is reported to achieve speedups within 15% of those provided by Alewife's native hardware [19]. Two other object-oriented examples, the Phosphorus [12] and the Adsmith [27], share some common features such as lock/unlock and barrier synchronization primitives, shared memory accesses via system calls, and using PVM as the underlying communication environment. Phosphorus offers a debugging facility to trace the underlying PVM messages. The consistency semantics of Adsmith is a special variation of RC with synchronization primitives conforming sequential consistency (SC), rather than processor consistency (PC). Moreover, for false sharing control

Adsmith employs a special *multi-writer* protocol which acts like the *diff* mechanism in TreadMarks. Overall, object-oriented DSM designs have intrinsically better data migration and false sharing control, at the cost of sacrificing programming simplicity.

Existing false sharing reduction mechanisms rely on extra supports of compilers or other software tools. One approach, known as *padding*, is to allocate independent data into different pages to break the spatial relationship among the falsely shared data. This approach is applicable to both the static [36] and the dynamic [25] data, yet both require compiler/run-time library support. Another direction is to break temporal locality (instead of spatial locality) by adjusting the execution behaviors of programs, making processors access the same coherence unit at different time. *Loop rescheduling* [11], [31] is the representative of this category, Still, if the run-time system is not capable of dynamically rescheduling the execution flow, the compiler support is indispensable. The other method is clustering the processors by *processor-data affinity* [17], [30]. In contrast to the previous two approaches, the affinity information between processors and data is usually obtained at run-time. However, this method can only reduce rather than totally eliminate false sharing, because the affinity-based prediction cannot be 100% precise and the mis-clustered processors must access the remote memory at a high cost. In summary, these approaches require additional compiler or other software support, and they cannot completely eliminate false sharing.

3. The Scheme

In this section we propose a new false-sharing-free shared memory management scheme. Our goal is to integrate the advantages and avoid the drawbacks of previous page- and object-based DSM designs; that is, utilizing dynamic granularity of the object paradigm while preserving the transparency of page-based designs. We first describe the syntax of synchronization primitives from parallel applications' viewpoints, followed by the details and performance analyses of the special data structure and the algorithm of our scheme.

3.1 Syntax of Synchronization Primitives

In previous relax consistency models like LRC and EC, entering and leaving a critical section are represented by acquiring (locking) and releasing (unlocking) a dummy *mutex* (for *mut*ual *ex*clusive) variable, respectively. As an example, if we want to access some shared variable `X` in previous relax consistency models we must write the code as follows:

```
// LOCK_X is a dummy mutex variable
ACQUIRE LOCK_X
// X is the shared data item
.....(access X)
RELEASE LOCK_X
```

We notice that the dummy *mutex* variables are unnecessary and potential sources of false sharing: Observe that conventional DSM systems have to keep track of what data are used within a critical section by trapping page faults, i.e. their consistency granularity must be one page, inducing false sharing inevitably. Furthermore, in many DSM designs the *mutex* variables are a kind of limited system resource competed by all applications, and the management (allocating, releasing, etc.) overhead of *mutex* use is a substantial overhead to the system. The solution to these problems is therefore to adopt a distinct syntax of synchronization primitives: The operand of `ACQUIRE/RELEASE` should be the target data item itself, rather than an extra *mutex*. For example, in our scheme the previous code piece may be transformed as:

```
ACQUIRE X
.....(access X)
RELEASE X
```

If existing high-level languages such as C/C++ are used, programmers can explicitly `ACQUIRE/RELEASE` the starting addresses with the lengths of the target variables, like this:

```
ACQUIRE(&X, sizeof(X));
.....(access X)
RELEASE(&X, sizeof(X));
```

Such an approach saves not only the *mutex* resource space but also the binding overhead of the *mutex* and the very data it guards. A more crucial advantage is that this syntax prevents false sharing, because the runtime system can precisely (up to the resolution of 1 byte) identify the address ranges involved in the critical section without the restriction of fixed page size. This also implies our new syntax does not depend on any virtual memory mechanism; nevertheless, shared data can still be directly manipulated within critical sections, retaining better compatibility to existing parallel applications than most object-based synchronization models. In addition, we add the following enhancements to this new syntax for higher performance and flexibility:

- *Locking multiple shared data items at a time.* Such a feature ensures the atomicity of critical section entrance and exit, eliminating potential race conditions caused by a series of `ACQUIRE` operations. In practice, the requested memory segments will be handled in the order of their starting addresses to prevent deadlocks.
- *Read-only optimization.* The `ACQUIRE` primitive

can be optionally tagged as read-only, allowing multiple read-only `ACQUIRE` operations to be launched simultaneously on the same data. If a read-only `ACQUIRE` operation is untagged, it can be simply treated as the original ones, and the correctness of the program still remains intact.

- *Backward compatibility.* Our syntax can emulate other relax consistency models, such as LRC, to guarantee backward compatibility to existing parallel applications. This can be done via explicitly declaring special *mutex* variables: When applications `ACQUIRE` such *mutex* variables, the associated critical section is treated conventionally. Alternatively, a compiler or the run-time system can detect whether a variable is acquired but left unused within a critical section; if so, obviously this variable must be a *mutex*, and the whole critical section should be handled in the conventional manner.
- *Supporting multi-writer concurrent writes.* This feature is designed to comply with the ideal Parallel Random-Access Machine (PRAM) computation model extensively studied in the literature. Since concurrent writes are performed in parallel and non-exclusively, they must be placed *outside* critical sections and handled by a coherence-on-demand strategy, in which the results of such operations will keep private until being explicitly requested by other processors. The temporary inconsistency caused by conflict concurrent writes will then be resolved by a new unary synchronization primitive called `SYNC`: more specifically, different values of a concurrently written data item will be unified by the freshest one at the `SYNC` point. In practice, a scalar or vector timestamp mechanism [14], [24], [29] is sufficient to implement `SYNC`, whose details are beyond the scope of this paper.

We call this new enhanced consistency model the *aggressive consistency* (AC) for its support of concurrent writes. The complete AC definition is listed below, in which the bound and free accesses represent the memory operations inside and outside critical sections, respectively. An axiomatic yet equivalent definition can be found in our previous works [22], [23].

(AC-1) Before any bound memory access is performed with respect to any other processor, the associated `ACQUIRE` must be performed.

(AC-2) Before an `ACQUIRE` is allowed to be performed with respect to any other processor, the most recent `RELEASE` of the same shared object must be performed with respect to the processor issuing the acquire operation, if either `ACQUIRE` or `RELEASE` is in read-write mode.

(AC-3) Before an exclusive `RELEASE` is allowed to be performed with respect to any other processor, all previous bound memory accesses after the associated `ACQUIRE` in program context must be performed.

(AC-4) Before an ordinary operation is performed, all previous `SYNC` operations to the same object accessed must be performed.

(AC-5) All synchronization primitives follow the partial causal order; that is, they either follow dependency order, or they are in the same program except that `ACQUIRE`-s can bypass outstanding `RELEASE`-s if they are mutually independent.

3.2 The Shared Memory Directory

On the implementation side, a directory is necessary for managing shared memory in distributed or hierarchical memory systems like DSM/NOW. The memory directory keeps track of the information about each shared memory piece, including the ownership (i.e. which host processor has the freshest copy exclusively), locking status (who is acquiring or waiting for the memory piece), and so on. Moreover, if dynamic granularity is adopted the shared memory directory must be able to split a segment into two or more (if the application is accessing a small portion of that segment) and merge multiple adjacent segments into one (if the application is simultaneously using those segments as a whole), in contrast to object-oriented designs in which an object is accessed as an entity and there is no need for splitting and merging memory pieces. Briefly speaking, this directory must fulfill the following requirements:

- Keeping track of the locking and ownership status of the segments;
- Supporting dynamic granularity and segment splitting / merging; and
- Achieving high operation efficiency.

The directory structure we devise is called the *Threaded Splay Tree* (TST), which is an augmented version of the original splay tree [32]. In a TST, each node represents one shared memory segment with the starting address, the length, a flag of locking status, and other ownership-specific information to satisfy the requirements stated above. The starting address acts as the key value that determines the organization in a TST. In addition, each node has two extra threading links respectively pointing to its in-order traversal predecessor and successor (also the order of starting addresses) to support the segment splitting/merging functionality; that is, the TST is a combination of the splay tree and the doubly linked list. Such a configuration can be adaptive to both temporal locality (via splaying the most recently used shared memory segment to the root) and spatial locality (via thread tracking, without traversing the tree first), achieving better performance. Also note that splaying the TST does not affect the threads at all, as splaying never changes the in-order traversal sequence of the TST. An example of a segmentation mapping and its corresponding TST structure is

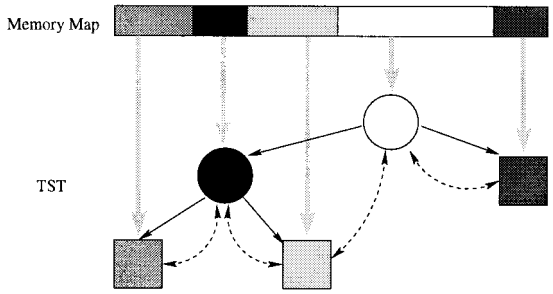


Fig. 1 An example of a shared memory segmentation map and its corresponding TST. The thick arrows are splay-tree edges and dotted bi-directional arrows are thread links.

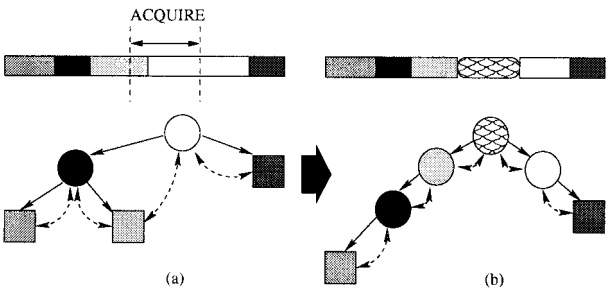


Fig. 2 When one `ACQUIRE` is applied to the TST in the previous example (a), a new node is generated by merging the fragments of the involved nodes and splayed to the root (b).

depicted in Fig. 1.

Looking up nodes in a TST is performed as follows. When one processor requests a memory range spanning one or more segments, the node representing the first segment (in terms of the starting address) will be splayed to the root for access, followed by its successors (if any) which can be tracked via the thread links. Merging and splitting segments are virtually look-up operations with node replacement. Since the thread links are not affected (as the address order of memory segments remains invariant), the amortized cost of an operation involving at most s nodes in an m-node TST is clearly $\mathbf{O}(s \log m)$; if the s nodes correspond to a contiguous address range of s memory segments, the amortized cost can be further reduced to $\mathbf{O}(s + \log m)$. Figure 2 depicts how an `ACQUIRE` operation manipulates the TST directory shown in the previous example.

3.3 A TST-Based Dynamic Token-Chasing Synchronization Algorithm

Now we present a dynamic token-chasing implementation of the kernel operations of AC, i.e. the `ACQUIRE` and the `RELEASE` synchronization primitives. In this scheme, there is no so-called "home" directory host that manages all shared memory blocks; instead, each shared memory segment is treated as a single, unique token. The processor holding the memory "token" has the exclusive right to access its contents. Like most token-chasing algorithms, our approach requires that the communication messages are *processor consistent*; i.e., from the viewpoint of a processor (say) P, all messages involving P should be totally ordered, while the ordering of other messages is irrelevant.

To get maximal performance, our approach is based on a highly efficient algorithm utilizing the path compression technique to accelerate the most time-consuming token-searching phase. The one we choose as the basis is that proposed by Chang, Singhal and Liu [9], [10], [20] (abbreviated as the CSL algorithm), which is generally the most efficient algorithm among proposed path compression ones, to our best knowledge. For a system of n processors, the CSL algorithm generates $\mathbf{O}(\log n)$ messages per request, and the actual number of messages is usually far fewer than that upperbound due to the effectiveness of path compression.

The key idea of the original CSL algorithm is described as follows. Each processor maintains a guess (called `dir`) of the probable token owner. If a processor neither holding nor requesting the token receives a request, it forwards this request to the processor indicated by `dir`, and then sets `dir` to point to the new requester because the new requester will eventually be the true token holder. When a processor requests the token, it sends a request message to the processor indicated by `dir`. It then sets an additional pointer, `next`, to NIL. If a processor holding or waiting for the token receives a request, and its `next` pointer is NIL, it sets `next` to the new requester. Otherwise, it forwards the request to the node indicated by `dir`, and "compresses the path" for further requests by setting `dir` to the new requester. Among the processors that hold the token or are requesting, their `next` pointers form a distributed queue of the pending requesters. If a processor is waiting and its next pointer is NIL, this processor is effectively at the end of the waiting queue. If next is not NIL, the end of the waiting queue is at or beyond the processor pointed to by `dir`. Since the requester will become the one at the queue end, it is appropriate to set the `dir` variable to that requester. When the token holder releases the token, it sends the token to the processor pointed to by `next`, if `next` is not NIL. Otherwise, the current holder just keeps the token.

To enable the CSL algorithm in our dynamic-granularity implementation, the first thing to do is adding the two `dir` and `next` pointers into each node of the TST directory of each processor so that the CSL algorithm can use shared memory blocks as tokens. Still, there remains one more challenge: In AC, the shared memory blocks may be dynamically split or merged, implying that the tokens will be generated or destroyed on the fly. This could totally ruin the execution of the CSL algorithm. Fortunately, such an obstacle can be overcome by making the processor in-order traverse the distributed forest formed by the `dir` pointers. More pre-

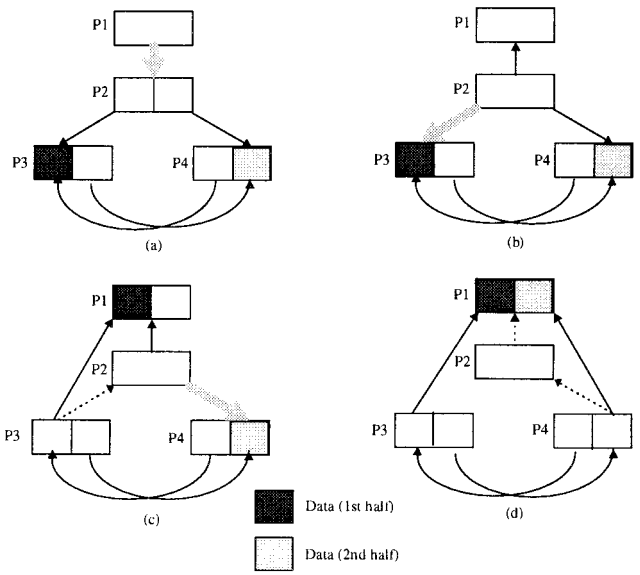


Fig. 3 An execution example of the token-based AC algorithm. The solid thin arrows are the dir pointers. (a) Initially P1 sends the request (the gray thick arrow) to P2; (b) P2 finds out that the block is divided to 2 pieces, so it first forward the split request to P3, the owner of the first half; (c) P3 acknowledges the forwarded request (the dotted arrow), so P2 can forward the other split request to P4, the owner of the second half; (d) P4 acknowledges the forwarded request. Notice that data are asynchronously sent to P1, the original requester.

cisely, when one processor splits a memory into several smaller pieces, it will launch a new CSL operation per each sub-segment, gather and merge the responses, and return the collective result to the source requester (if any). In practice, the acknowledgment and the actual data of each memory sub-segment can be separated and transferred asynchronously for better performance. We show an execution example of our algorithm in Fig. 3.

The pseudo codes of our algorithm with enhancement of separated acknowledgment and data transfer are listed below. In the listing a memory segment in the TST is represented by a range $[s, s+l]$ where s and l is the starting address and the length of the very segment, respectively. Note that the following algorithm can be easily extended to handle read-only `ACQUIRE` operations, which is left to the reader.

Initial Settings

- All shared memory blocks are exclusively owned by one processor and others set the `dir` pointers to be the initial owner.
- All processors are idle; i.e. outside critical sections.

Upon Acquiring the Range $[s, s+l]$

1. Look up the TST to transform the range $[s, s+l]$ into a list of memory blocks (LMB) $\{[s_0, s_0+l_0]$, $[s_1, s_1+l_1]$, $[s_2, s_2+l_2]$, $\ldots\}$ that are possibly owned by processor P_0, P_1, P_2, $\ldots$, respectively.
2. Merge the nodes of LMB into a single node $[s, s+l]$ in the TST, with the lock flag set true, `next` = NIL, `dir` = NIL, respectively.
3. Let $i = 0$ and repeat the following sub-steps until the LMB is empty:
   3.1 Check if P_i is the requester itself; if so, go to step 3.4.
   3.2 Send out an acquire-writing request $[s_i, s_i+l_i]$ to its probable owner P_i;
   3.3 Wait until the acknowledgment of $[s_i, s_i+l_i]$ arrives.
   3.4 Remove $[s_i, s_i+l_i]$ from the LMB; set $i = i+1$.
4. Enter the critical section when all data within $[s, s+l]$ arrives.

Upon Receiving an Acquire $[s, s+l]$ Request

1. Look up the TST to transform the range $[s, s+l]$ into a list of memory blocks (LMB) $\{[s_0, s_0+l_0]$, $[s_1, s_1+l_1]$, $[s_2, s_2+l_2]$, $\ldots\}$ that are possibly owned by processor P_0, P_1, P_2, $\ldots$, respectively.
2. Merge the nodes of LMB into a single node $[s, s+l]$ in the TST, with `dir` = the initiator of the acquire operation.
3. Let $i = 0$ and repeat the following sub-steps until the LMB is empty:
   3.1 If the receiver itself is neither P_i nor requesting the block $[s_i, s_i+l_i]$, or the `next` pointer of the block is not NIL, then forward the request to the node indicated by the `dir` pointer of block and go to step 3.4.
   3.2 If the receiver itself is currently requesting the block $[s_i, s_i+l_i]$ and `next` is NIL, set `next` = the initiator of the acquire operation and go to step 3.4.
   3.3 Send an acknowledgment to the previous forwarder and the contents of $[s_i, s_i+l_i]$ to the initiator of the acquire operation.
   3.4 Remove $[s_i, s_i+l_i]$ from the LMB; set $i = i+1$.

Upon Releasing the Range $[s, s+l]$

1. Look up the TST to transform the range $[s, s+l]$ into a list of memory blocks (LMB) $\{[s_0, s_0+l_0]$, $[s_1, s_1+l_1]$, $[s_2, s_2+l_2]$, $\ldots\}$ that are possibly owned by processor P_0, P_1, P_2, $\ldots$, respectively.
2. Let $i = 0$ and repeat the following sub-steps until the LMB is empty:
   2.1 If the `next` pointer of the block $[s_i, s_i+l_i]$ is not NIL, send the data of the block to the processor indicated by the `next` pointer and reset the `next` to NIL; otherwise, let `dir` = the releasing processor itself.
   2.2 Reset the lock flag of the corresponding TST node to be false.
   2.3 Remove $[s_i, s_i+l_i]$ from the LMB; set $i = i+1$.
3. Leave the critical section.

3.4 Analysis

Theorem 1: The TST-based synchronization algorithm guarantees mutual exclusion.

Proof (Sketch)**:** Suppose two memory segments M_1 and M_2 are respectively requested by two host processors P_1 and P_2 at the same time, and $M_1 \cap M_2 \neq \Phi$. Let $M_x = M_1 \cap M_2$ and call the previous M_x owner P_x. Since messages are processor-consistent, P_x must first receive and handle the `ACQUIRE` request from either P_1 or P_2, but not simultaneously. However, if P_x handles the request from P_1 first, it can never grant the request from P_2 because P_x will no longer own M_x; or *vice versa*. Since all shared memory segments are initially owned by one processor, we conclude that $M_1 \cap M_2 = \Phi$. □

Theorem 2: The TST-based synchronization algorithm is deadlock-free.

Proof (Sketch)**:** Since the `ACQUIRE` requests are propagated in the direction pointed by the dir pointers. A deadlock occurs if and only if there exist some host processors whose dir pointers form a cycle; more specifically, a requesting host P_0 is deadlocked if and only if there exist n ($n \geq 0$) host processors, say P_1, P_2, P_3, ... , P_n, such that $P_0 \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \ldots \Rightarrow P_n \Rightarrow P_0$ where $\Rightarrow$ represents the dir pointer. Let M_i ($i = 1, 2, \ldots, n$) be the memory segment held by P_i and being waited for by P_{i-1}, and M_0 be the memory segment held by P_0 and being waited for by P_n. Because in our algorithm memory segments are requested in the order of their starting addresses, the starting address of block M_n must be larger than those of M_{n-1}, M_{n-2}, ... , M_1 and M_0, whileas the starting address of M_0 must be larger than that of M_n, an obvious contradiction. Thus our algorithm is guaranteed deadlock-free. □

The efficiency of our algorithm is highly dependent on the communication network topology. Below we show the analytical result for typical LAN-based NOW systems and fully-connected multicomputers.

Theorem 3: In an n-processor system with at most m shared memory segments, the TST-based synchronization algorithm processes a synchronization request involving at most s segments at the cost of $\mathbf{O}(s \log m \log n)$ amortized computation steps and $\mathbf{O}(s \log n)$ communication messages, respectively.

Proof (Sketch)**:** Immediately from the facts that (1) a memory segment is virtually a token in the CSL algorithm, yielding $\mathbf{O}(\log n)$ messages per request; and (2) each host processor along the searching path takes $\mathbf{O}(\log m)$ amortized computation steps in querying its own TST. □

Corollary 4: In an n-processor system with at most m shared memory segments, the TST-based synchronization algorithm processes a synchronization request involving at most s contiguous segments at the cost of $\mathbf{O}(s \log n + \log m \log n)$ amortized computation steps and $\mathbf{O}(s \log n)$ communication messages, respectively.

4. ULTRA: An Experimental DSM System

In order to examine the real-world performance of our proposed scheme, we constructed a prototype software DSM system called *ULTRA* (for Universal Layer for Transparent Remote Access) that implements AC. The primary design goal of *ULTRA* is to provide both a testbed of our proposed AC scheme and an experimental development platform of DSM applications. The first version of *ULTRA* is developed in C language on several Ethernet-connected Intel Pentium-based computers running Linux, a robust, freeware Unix clone. We also conducted a benchmark test of several typical parallel applications on our prototype to measure the performance. The preliminary results are quite encouraging, especially considering that we have not even added any advanced performance optimization techniques in this incarnation of *ULTRA*. Below we first briefly describe the current *ULTRA* implementation, then present and evaluate the benchmark results.

4.1 Overview

The first implementation of *ULTRA* is designed as an intermediate layer between shared-memory applications and the underlying operating system. Programs running on *ULTRA* consist of one or more parallel threads or "light-weight processes," which are embodied by Unix processes that share a single DSM address space controlled by the *ULTRA* runtime system. The internal organization of the current *ULTRA* implementation, depicted in Fig. 4, consists of three main components: the network listener (NL) that picks up from the network the necessary packets, the Application Programming Interface (API) library that bridges the run-

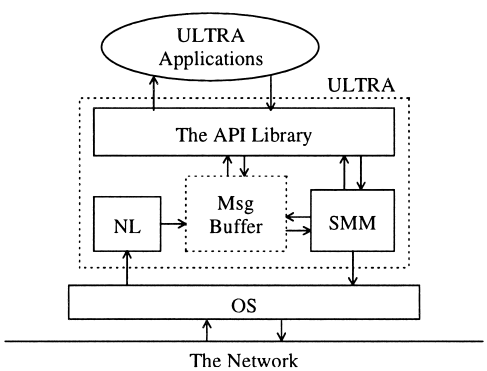


Fig. 4 The block diagram of the ULTRA system organization.

Table 1 Highlights of *ULTRA* API calls.

Function Name	Meaning
vm_init( )	Initialize the *ULTRA* system
vm_allocate( )	Allocating a segment
vm_free( )	Freeing an allocated segment
vm_close( )	Quit the *ULTRA* system
vm_acquire( )	Lock shared memory segments
vm_release( )	Unlock segments
vm_sync( )	The `SYNC` operation
vm_barrier( )	Blocking rendezvous
vm_createthread( )	Create a new thread
vm_threadexit( )	Terminate a thread
vm_getthreadid( )	Get a unique thread ID

time system and application programs, and the shared memory manager (SMM). These modules communicate one another by an internal message buffer, which is implemented by a combination of a standard Unix message queue and a block of Unix shared memory for large amount of data transfer.

The heart of the current *ULTRA* implementation is the SMM, which is implemented as a daemon process that continuously monitors the message buffer for incoming requests. It implements most of the data structures and algorithms presented in previous sections except that we deactivate the asynchronous acknowledgment and data transfer feature to measure the raw performance of our algorithms.

We use remote procedure calls (RPC) as the underlying inter-process communication facility; the reasons of adopting RPC are reliable message delivery, guarantee of data packet ordering, and programming simplicity. In fact, we implemented a single data transferring routine to handle both remote read and write requests, because the only difference of a read and a write is the direction of data movement. The `SYNC` operation in this version of *ULTRA* is also implemented by the simplified approach. On the other hand, we incorporated additional system management functions (see Table 1); some of them are directly built into the API library rather than integrated in the SMM.

4.2 Performance Evaluation

In this subsection we measure and compare the performances of our *ULTRA* prototype with another software DSM management scheme. The system we compare is Adsmith [27], which is also a variable-granularity software DSM system with *multi-writer* protocol for false sharing control as well as other performance features. A series of benchmark tests were conducted on both systems to evaluate the performance improvements of our new scheme over traditional approaches.

4.2.1 The Experimental Environment

Our testing platform consists of 8 Ethernet-connected Pentium/100 MHz machines running Linux kernel 2.0.36. Each node is equipped with 32 MBytes of EDO-RAM and a 3COM 3C509B Ethernet adapter. The version of Adsmith is 1.8.0*i* running on PVM 3.4. All executables including the OS kernel, the two DSM systems, and the benchmarking programs are made using GNU C 2.7.2.3. During the benchmark process we also enabled *bulk transfer*, in addition to the *multi-writer* protocol, for Adsmith to maximize its network throughput. Table 2 lists some "mini" benchmark results of the two DSM systems on our testbed, which indicate that *ULTRA* is substantially faster than Adsmith in terms of basic operation and internal bookkeeping costs.

Table 2 Mini-benchmark results (Units: milliseconds).

Action / System	Dummy SysCall	1 KB data transfer	4 MB data transfer
ULTRA	6.15	10.18	5240.0
Adsmith	7.23	11.35	5877.7

The benchmark suite we use consists of three parallel applications that vary widely in their computation and communication complexities. The first program, *MSort*, is a parallel implementation of the merge sort algorithm that consists of two phases. First, by the number of processors *MSort* divides the original data array (2^{21} integer elements in this test) into roughly equal-sized chunks, and assigns each processor to sequentially sort one chunk respectively. Next, *MSort* pairwisely merges those intermediate sorted chunks in parallel until the whole array is completely sorted. Such a technique is frequently used for parallelizing other divide-and-conquer algorithms.

The second application in the suite is *Relax*, a parallel implementation of the 2-D successive over-relaxation algorithm. The whole algorithm is an iterative process on a matrix of 4K×1K elements, repeated 100 times. During each iteration, *Relax* replaces each matrix element with the average of its neighboring elements. Such an algorithm is the key step in the finite elements method in numerical and scientific computations. In this incarnation, we partition the whole matrix into several horizontal strips and assign one strip to exactly one processor; thus sharing only occurs across the boundaries of neighboring strips.

The third benchmark program is *MatMul*, which performs parallel matrix multiplication of two 768*768 square matrices. The computation of *MatMul*, based upon the "classical" triple-loop approach, can be described as follows. The product matrix is partitioned into roughly equal-sized horizontal bands by the number of available processors, and each processor is responsible for computing one band. In the same style the multiplicand and multiplier matrices are also partitioned horizontally (along the row) and vertically (also along the row if the target matrix has been transposed in advance), respectively. Each processor then takes and holds its corresponding multiplicand and product

Table 3 Benchmark results of *ULTRA* (Units: seconds).

# of Processors	*MSort*	*Relax*	*MatMul*
1	37.17	241.44	344.93
2	19.58	129.94	176.65
3	16.15	98.10	121.77
4	15.10	78.82	95.52
5	14.23	69.32	79.00
6	13.71	60.18	68.28
7	13.24	54.25	62.47
8	12.92	50.33	59.13

Table 4 Benchmark results of Adsmith (Units: seconds).

# of Processors	*MSort*	*Relax*	*MatMul*
1	99.94	849.84	408.93
2	109.33	482.61	224.02
3	116.68	329.51	167.20
4	117.51	253.56	144.55
5	118.73	208.88	135.96
6	119.32	179.65	133.84
7	118.64	161.68	137.17
8	119.93	150.83	142.63

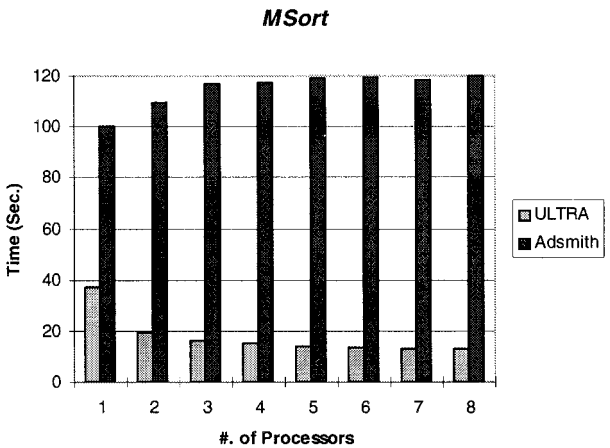


Fig. 5 Execution times of *Msort* (lower is better).

bands, and fetches one different multiplier band a time in a round-robin ordering to compute the product.

4.2.2 Benchmark Results and Analyses

Tables 3 and 4 summarize the execution times of the benchmark suite on *ULTRA* and Adsmith respectively. Overall, we find that *ULTRA* consistently and significantly outperforms Adsmith in all three benchmarks by a very large margin from some 20% (*MatMul*, 1 processor) to over 800% (*MSort*, 8 processors). We also discover that *ULTRA* scales up better than Adsmith does, showing positive speedups for up to 8 processors in all benchmarks while the other degenerates in 2 cases out of 3. Performance characteristics of each individual application are detailed as follows.

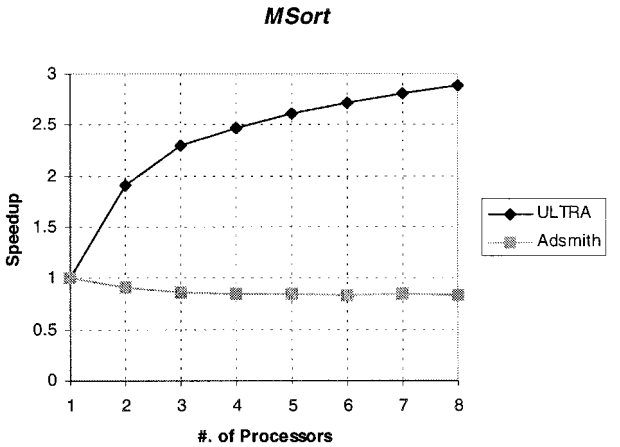


Fig. 6 Speedups of *Msort* (higher is better).

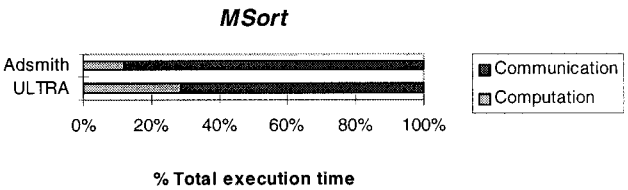


Fig. 7 Breakdown of 8-processor execution times of *MSort*.

MSort

Figures 5 and 6 summarize the elapsed execution times and speedups of *MSort* respectively. From these figures we notice the unusually large performance lag of Adsmith behind our *ULTRA* prototype. After examining the system and the benchmark programs we find that, to our surprise, the direct reason of such a performance difference is indeed *false sharing*. Due to the object-oriented nature of Adsmith, an array can only be refreshed and/or flushed as a whole, causing serious false sharing during the merging phase of *MSort*. Worse, the multi-write protocol is unable to reduce this kind of false sharing because the protocol has nothing to do with the granularity. To measure and alleviate such an impact on system speed, we manually restructured *MSort* by substituting several small arrays for the primary data array and then re-executed the benchmark. The outcomes are dramatic: the modified *MSort* runs almost twice as fast as the original version, completing the sorting task in 56.45 seconds on a single processor and 63.72 seconds on 8 processors. Such a steep improvement not only explains the peculiar performance characteristics, but also validates the effectiveness of our false-sharing-free memory management scheme.

Another issue worthy to note is that the problem size is too small for the testing configuration, thus the communication time soon dominates the total execution time, as Fig. 7 shows. Observe that the throughput of *MSort* soon saturates on both systems when the number of processors increases, and the even degrades on Adsmith regardless of whether the *MSort* program

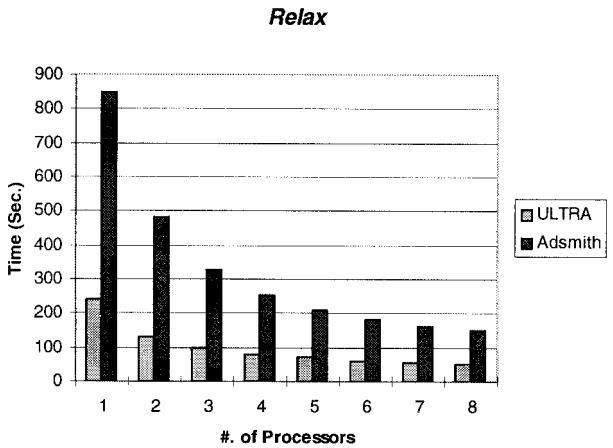


Fig. 8 Execution times of *Relax* (lower is better).

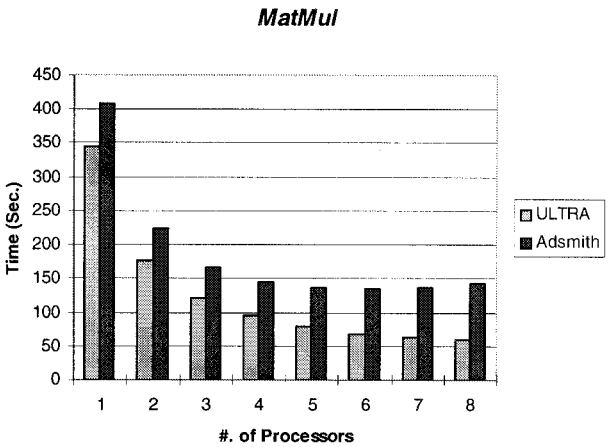


Fig. 11 Execution times of *MatMul* (lower is better).

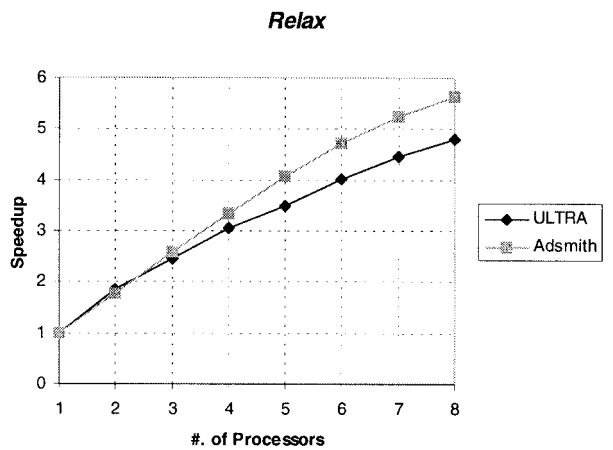


Fig. 9 Speedups of *Relax* (higher is better).

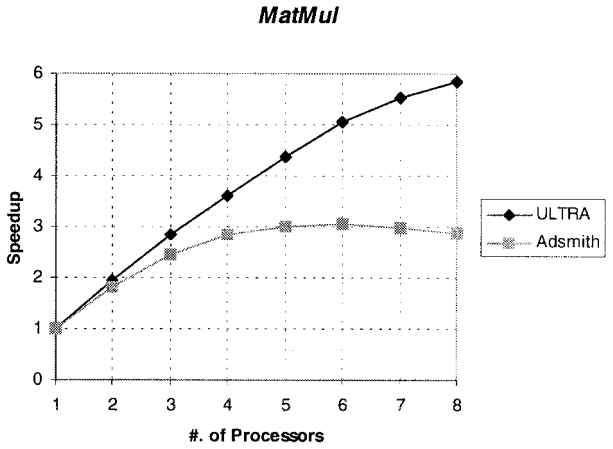


Fig. 12 Speedups of *MatMul* (higher is better).

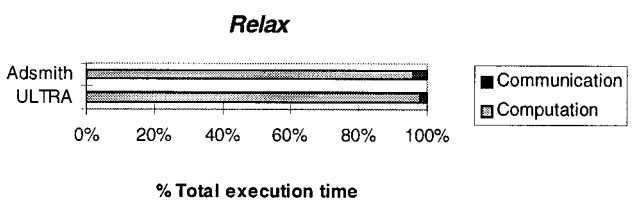


Fig. 10 Breakdown of 8-processor execution times of *Relax*.

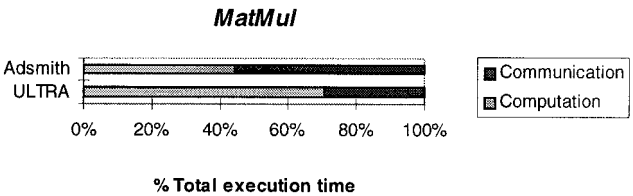


Fig. 13 Breakdown of 8-processor execution times of *MatMul*.

has been restructured or not. Fortunately, the better efficiency of basic operations (see Table 2) still keeps *ULTRA* from the performance degradation that Adsmith undergoes. Although this implies that the raw bandwidth of Ethernet is adequate for *ULTRA* on up to 8 processors, a faster communication medium such as the 100 Mbps Fast Ethernet should undoubtedly enhance both systems' speeds.

Relax

The scenario is different for *Relax*, whose execution times on both DSM systems are depicted in Fig. 8. Since the computation-to-communication ratio of this application is high (refer to Fig. 10) and its shared access patterns are regular, *Relax* is obviously far better scalable than *MSort*—from Fig. 9 we observe about 5x speedup on 8 processors for both DSM systems. In fact, in this case Adsmith even exhibits slightly better scalability than *ULTRA*, because the internal caching management expense of Adsmith is proportional to (i.e. scalable to) the amount of shared data held by each processor, thus contributing to its scalability. Nevertheless, *ULTRA* still executes *Relax* 200% to 250% faster than Adsmith in terms of raw speed.

MatMul

In Figs. 11 and 12 we present respectively the execution times and speedups of *MatMul*. We observe that at first, *MatMul* scales up well on both systems, and the performance edge of *ULTRA* over Adsmith is about 20%, not as large as in the previous two scenarios. When the number of processors grows, however, the performance on *ULTRA* continues to increase steadily to finally reach 6x speedup on 8 processors, yet the throughput on Adsmith quickly saturates and even starts to degenerate after the fifth processor is added. A closer inspection reveals that most network traffics are generated when the multiplier matrix is accessed. Recall that in *MatMul* the multiplier matrix is partitioned into vertical bands and each processor accesses one different band a time in a round-robin fashion. Since in C language a multi-dimensional array is indeed a collection of possibly noncontiguous subarrays, accessing a band of the multiplier matrix will induce multiple small synchronization transactions to that band's constituent subarrays, which are especially time-consuming on Adsmith due to its higher function invocation costs. In contrast, the AC syntax of *ULTRA* supports acquiring multiple noncontiguous memory blocks atomically, greatly minimizing the run-time overheads and thus leading to better computation-to-communication ratio and higher scalability, as Fig. 13 shows.

5. Conclusions and Future Works

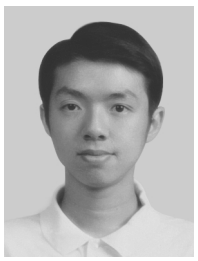
The key of eliminating false sharing is utilizing dynamic granularity, without inducing side effects such as programming difficulty and external memory fragmentation. In our proposed dynamic granularity scheme, we define a new memory consistency model AC for memory usage transparency, and we adopt segment splitting/merging mechanism to suppress external fragmentation. Performance analyses of our scheme show that the combination of the splay-tree based TST directory and the path-compressing token chasing algorithm is very efficient, yielding synchronization complexity of only a polynomial of logarithmic scale of system sizes. Meanwhile, the benchmark results on our *ULTRA* prototype indicate that our scheme also delivers good real-world performances, surpassing other software DSM designs significantly even with advanced speed optimization features disabled. The high synchronization efficiency also balances the computation/communication ratio, meaning that our scheme scales up well along with the growth of the NOW system size. Another advantage of our scheme is platform-independency, indicating that its implementations such as *ULTRA* will enjoy high portability.

A second generation of *ULTRA*, the *ULTRA-II*, is now under development to fully exploit the massive power of NOW by utilizing performance enhancements such as asynchronous data transfer, the *diff* mechanism [3], [4], [21], and data prefetching [23] techniques. In addition, we will incorporate more features such as real-time and multimedia support into *ULTRA-II*.

References

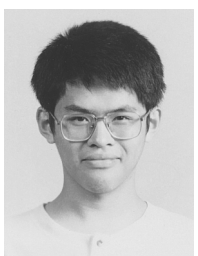
[1] S.V. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel, "A comparison of entry consistency and lazy release consistency implementations," Proc. HPCA-2, pp.26–37, Feb. 3-7, 1996.
[2] A. Agarwal and A. Gupta, "Memory-reference characteristics of multiprocessor applications under Mach," Proc. Conference on Measurement and Modeling of Computer Systems, pp.215–225, May 1988.
[3] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: shared memory computing on networks of workstations," Computer, vol.29, no.2, pp.18–28, Feb. 1996.
[4] C. Amza, A. Cox, K. Rajamani, and W. Zwaenepoel, "Tradeoffs between false sharing and aggregation in software distributed shared memory," Proc. 6th Symp. on PPoPP, pp.90–99, June 18-21, 1997.
[5] J.K. Bennett, J.B. Carter, and W. Zwaenepoel, "Munin: distributed shared memory based on type-specific memory coherence," Proc. 2nd Symp. on PPoPP, pp.168–176, 1990.
[6] J.K. Bennett, J.B. Carter, and W. Zwaenepoel, "Implementation and performance of Munin," Proc. 13th ACM Symp. on OS Principles, pp.152–164, Oct. 1991.
[7] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon, "The Midway distributed shared memory system," Proc. IEEE COMPCON 93, pp.528–537, Feb. 1993.
[8] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso, Programming under Mach, Reading, Addison-Wesley Inc., 1993.
[9] Y.I. Chang, M. Singhal, and M.T. Liu, "An improved O(log(n)) mutual exclusion algorithm for distributed systems," Proc. 1990 ICPP, pp.III295-302, 1990.
[10] R. Chow and T. Johnson, Distributed Operating Systems & Algorithms, Addison-Wesley Longman Inc., 1997.
[11] J.-H. Chow and V. Sarkar, "False sharing elimination by selection of runtime scheduling parameters," Proc. 1997 ICPP, pp.396–403, Sept. 1997.
[12] R.C. Dantart, I. Demeure, P. Menuier, and V. Bartro, "Phosphorus: A tool for shared memory management in a distributed environment," Technical Report from Ecole Nationale Supérieure des Télécommunications Département Informatique, Dec. 1995.
[13] M. Dubois, C. Scheurich, and F.A. Briggs, "Synchronization, coherence, and event ordering in multiprocessors," Computer, vol.21, no.2, pp.9–21, Feb. 1988.
[14] C. Fidge, "Logical time in distributed computing systems," Computer, vol.24, no.8, pp.28–33, Aug. 1991.
[15] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mangcheck, and V. Sunderam, "PVM 3 users guide and reference manual," Technical Report, Oak Ridge National Laboratory, May 1994.
[16] K. Gharachorloo, A. Gupta, and J.L. Hennessy, "Two techniques to enhance the performance of memory consistency models," Proc. 1991 ICPP, vol.1, pp.355–364, Aug. 1991.
[17] K.P. Hung, N.H. C. Yung, and Y.S. Cheung, "Reduction of false sharing by using process affinity in page-based distributed shared memory multiprocessor systems," IEEE Int'l Conf. on Algorithms and Architectures for Parallel Processing, pp.383–390, June 1996.
[18] K.L. Johnson, J. Adler, and S.K. Gupta, "CRL version 1.0 user documentation," Technical Report CRL-SOSP-15,

MIT Laboratory for Computer Science, 1995.
[19] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach, "CRL: high-performance all-software distributed shared memory," Proc. 15th ACM Symp. on OS Principles, pp.213–228, Dec. 1995.
[20] T. Johnson and R. Newman-Wolfe, "A comparison of fast and low overhead distributed priority locks," J. Parallel and Distributed Computing, vol.32, no.1, pp.74–89, Jan. 1996.
[21] P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," Ph.D. Dissertation, Dept. of Computer Science, Rice University, 1995. An earlier version of this dissertation was published in Proc. ISCA-19, pp.13–21, May 19–21, 1992.
[22] A. I.-C. Lai and C.-L. Lei, "The formalization and hierarchy of memory consistency models," Proc. International Computer Symposium, pp.1060–1066, Dec. 1994.
[23] A. I.-C. Lai and C.-L. Lei, "Data prefetching for distributed shared memory systems," Proc. HICSS-29, vol.1, pp.102–110, Jan. 1996.
[24] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," CACM, vol.17, no.8, pp.453–455, Aug. 1974.
[25] J.W. Lee and Y.K. Cho, "An effective shared memory allocator for reducing false sharing NUMA multiprocessors," IEEE Int'l Conf. on Algorithms and Architectures for Parallel Processing, pp.373–382, June 1996.
[26] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," ACM Trans. Computer Syst., vol.7, no.4, pp.321–359, Nov. 1989.
[27] W.-Y. Liang, C.-T. King, and F. Lai, "Adsmith: An object-based distributed shared memory system for network of workstations," IEICE Trans. Inf. & Syst., vol.E80-D, no.9, pp.899–908, Sept. 1997.
[28] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," Computer, vol.24, no.8, pp.52–60, Aug. 1991.
[29] M. Raynal and M. Singhal, "Logical time: Capturing causality in distributed systems," Computer, vol.29, no.2, pp.49–56, Feb. 1996.
[30] C.B. Seidel, R. Bianchini, and C.L. Amorim, "The affinity entry consistency protocol," Proc. 1997 ICPP, pp.208–217, Aug. 1997.
[31] B. Simons, V. Sarkar, M. Breternitz, and M. Lai, "An optimal asynchronous scheduling algorithm for software cache consistency," Proc. HICSS-27, vol.2, pp.502–511, Jan. 1994.
[32] D.D. Sleator and R.E. Tarjan, "Self adjusting binary search trees," JACM, vol.32, no.3, pp.652–686, 1985.
[33] M. Stumm and S. Ziou, "Algorithms implementing distributed shared memory," Computer, vol.23, no.5, pp.54–64, May 1990.
[34] V.S. Sunderam, "PVM: A framework for parallel distributed computing," from Concurrency: Practice and Experience, vol.2, no.4, pp.315–339, Dept. of Math and Computer Science, Emory University, Atlanta, Dec. 1990.
[35] M.-C. Tam, J.M. Smith, and D.J. Farber, "A taxonomy-based comparison of several distributed shared memory systems," ACM OS Review, vol.24, no.3, pp.40–67, May 1990.
[36] J. Torrellas, M.S. Lam, and J. Hennessy, "False sharing and spatial locality in multiprocessor caches," IEEE Trans. Comput., vol.43, no.6, pp.651–663, June 1994.
[37] M.J. Zekauskas, W.A. Sawdon, and B.N. Bershad, "Software write detection for a distributed shared memory," Proc. OSDI'94, pp.87–100, Nov. 1994.
[38] R.N. Zucker, "Relaxed consistency and synchronization in parallel processors," Ph.D. Dissertation, Dept. of Computer Science and Engineering, Univ. of Washington, Dec. 1992.



Alexander I-Chi Lai was born in Taipei, Taiwan on May 5, 1969. He received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1991. Currently, he is a Ph.D. candidate and also the network administrator at the Department of Electrical Engineering at National Taiwan University. His research interests include distributed operating systems, distributed algorithms, and Internet computing.

Chin-Laung Lei was born in Taipei, Taiwan on January 9, 1958. He received the B.S. degree in electrical engineering from National Taiwan University in 1980, and the Ph.D. degree in computer science from the University of Texas at Austin in 1986. From 1986 to 1988, he was an assistant professor of the computer and information science department at the Ohio State University, Columbus, Ohio, U.S.A. In 1988, he joined the department of electrical engineering, National Taiwan University, where he is now a professor. His current research interests include network and computer security, parallel and distributed processing, operating system design, and formal semantics of concurrent programs. Dr. Lei is a member of the Institute of Electrical and Electronic Engineers and the Association for Computing Machinery.



Hann-Huei Chiou was born in Chang-Hwa, Taiwan on July 6, 1974. He received the M.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan in 1999. Currently, he is working toward the Ph.D. degree in the Department of Electrical Engineering at National Taiwan University. His research interests include distributed operating systems, distributed algorithms, and Internet technology.