AN ANALYSIS OF DISTRIBUTED SHARED MEMORY ALGORITHMS

R. E. Kessler
Miron Livny

Computer Sciences Technical Report #825

February 1989

An Analysis of Distributed Shared Memory Algorithms[†]

R. E. Kessler[‡]
Miron Livny

The University of Wisconsin-Madison
Computer Sciences Department
1210 W. Dayton St.
Madison, WI 53706

ABSTRACT

This paper describes results obtained in a study of algorithms to implement a Distributed Shared Memory in a distributed (loosely coupled) environment. Distributed Shared Memory is the implementation of shared memory across multiple nodes in a distributed system. This is accomplished using only the private memories of the nodes by controlling access to the pages of the shared memory and transferring data to and from the private memories when necessary. We analyze alternative algorithms to implement Distributed Shared Memory, all of them based on the ideas presented in [Li86b].

The Distributed Shared Memory algorithms are analyzed and compared over a wide range of conditions. Application characteristics are identified which can be exploited by the Distributed Shared Memory algorithms. We will show the conditions under which the algorithms analyzed in this paper perform better or worse than the other alternatives. Results are obtained via simulation using a synthetic reference generator.

1. Introduction

Two major models of interprocess communication are: message passing and shared memory. In the first model, processes communicate and synchronize by sending and receiving messages whereas in the latter model processes interact by storing and retrieving data from a common address space. Shared memory can be considered a more flexible means of interprocess communication, if for no other reason than the ease at which message passing can be implemented given a shared memory. Moving in the opposite direction, namely from a system that only supports message passing to a shared memory system, is not a simple task. It requires the development of algorithms that will provide a consistent view of the shared space with an affordable overhead. We call algorithms which maintain this view Distributed Shared Memory (DSM) algorithms. The logical advantages of a single address space along with the physical advantages of a loosely coupled message based system have triggered efforts to develop and evaluate such algorithms.

[†] To be presented at The 9th International Conference on Distributed Computing Systems, Newport Beach, California, June 5-9, 1989.
[‡] This work has been supported by graduate fellowships from the National Science Foundation and the University of Wisconsin-Madison

In a distributed system each processor has its own virtual address space that is mapped to its local (private) memory. A processor can access only its local memory, unlike multiprocessor systems such as Cm* [Oust80] and RP3 [Pfis85] which allow different processors to access the same physical memory. As a performance optimization in these local/remote architectures one may migrate data from remote to local memories [Holl88]. Migration is a necessity, not an optimization, when supporting shared memory in a distributed system. One processor can read the content of a location in the shared space that was written by another process only if its content has been transferred by a message from the local memory of the writer to the local memory of the reader. With the help of the access control provided by virtual memory hardware, the need for such a transfer can be detected. It is the responsibility of the DSM algorithm to provide the right information to the virtual memory control hardware and to control the node to node message exchange when implementing a DSM.

Kai Li [Li86a, Li86b, Li88] developed an implementation of a DSM for Apollo workstations and observed good performance for parallel algorithms using the DSM for interprocess communication. Subsequently, DSM implementations have been considered for the Mach [Youn87], and LOCUS [Flei87] operating systems. The increasing interest in DSM algorithms has motivated us to study the performance tradeoffs of this type of distributed algorithms. The DENET [Livn86] discrete event simulation environment allowed the implementation of a detailed simulation model of a distributed system, as well as a flexible synthetic access pattern generator. We feel the generator captures some characteristics of parallel programs which may use shared memory in a distributed environment. Using this simulator, we analyzed the performance of an algorithm introduced by Li as well as several variants of the algorithm. In Section 3, following a section that is devoted to the basic concepts which allow the implementation of a DSM, we describe the four algorithms we analyzed and discuss the differences between them. The flexibility of the simulation model and synthetic access pattern generator enabled us to observe the performance of the algorithms for a wide range of system and access pattern parameters. Section 4 describes the synthetic access pattern generator and the simulation model. Section 5 describes the simulation results from which we derive the conclusions we draw in Section 6.

2. How Does Distributed Shared Memory Work?

DSM algorithms are implemented in software, but, are very similar to hardware *directory* [Agar88, Cens78], and *snooping cache* [Arch86, Good83] schemes used to maintain cache coherency in multiprocessor systems. A strong definition of a coherent memory system is: a system in which a read from a location of the shared memory will return the value of the latest store to that location. A weaker definition would be that a read will eventually return the value of the latest store. Hardware cache coherency schemes maintain coherency on the cache block, which is the basic unit of memory controlled by the cache hardware. The major difference between DSM and hardware cache coherency schemes is that the cost of maintaining a coherent DSM can be

much higher since coherency is maintained on pages, not cache blocks, and network transfers are required to transfer pages. Not only are virtual memory pages typically considerably larger than a cache block, but, network communications will lead to much higher latencies than that of bus-based hardware coherency schemes. An interesting variation of cache coherency schemes is the software and hardware control of coherency in the VMP [Cher86, Cher88] multiprocessor, in which software manages coherency control and caching with hardware support.

DSM algorithms utilize four major ideas to maintain an image of a coherent shared memory:

(1) The DSM is partitioned into pages, the granularity afforded by virtual memory.

(2) The page is the unit on which coherency is maintained. Writes to a given page are strictly ordered and eventually propagated to all nodes.

(3) Copies of each page of the DSM will exist on one or more nodes as per the coherency constraints. At any time at least one copy of each page exists.

(4) Virtual Memory hardware is used to restrict access to shared data. At any time a processor may have: no access, read access, or read&write access to a given page of the DSM. Page faults will occur when a process violates access rights.

The rules enforced by DSM algorithms to efficiently maintain coherency are:

(1) A node should have read&write access to a page of the shared memory if and only if the given node has *the one and only* current copy of the given page.

(2) A node should have only read access to a page if and only if multiple copies of the page exist on different nodes.

In order to maintain these invariants, DSM algorithms keep track of the location of all copies of a given page. The easiest way to do this is to have an *owner* for each page who will know which nodes retain a copy of the page. All requests for changes in access rights to the page will then go through the owner of that page. Ownership can be assigned dynamically and transferred from one node to another.

Two types of faults can occur when access rights are violated: read faults, and write faults. Read faults will occur when a process attempts to read a page to which it has no access (i.e. the node does not have a copy of the page). Write faults will occur when a process attempts to write to a page to which it does not have read&write access (it has either no access or read access).

The *Base* protocol followed upon a **read fault** for a page of the DSM is as follows:

(1) The faulting node requests a copy of the page from the owner of the page.

(2) The owner of the page sends a copy to the faulting node, notes the location of the new copy and changes its local access rights to read-only.

(3) The faulting node receives the page and sets the local access rights of it to read-only.

The Base protocol followed for a **write fault** is as follows:

(1) The faulting node requests read&write access from the owner of the page.

(2) The owner of the page sends the page (if necessary) and the list of nodes with copies of the page to the faulting node (that is, ownership of the page is transferred to the faulting node).

(3) The faulting node sends invalidation messages to all other nodes with a copy of the given page. On receipt of an invalidation message, the copy of the given page is destroyed.

(4) The faulting node sets the local access rights to read&write access and notes that it is the new owner of the page.

Note that with this algorithm it may be possible for read-only non-current copies of a given page to exist for a *short length of time* after another node has obtained write access to the page, depending on whether writes can proceed before invalidations are completed (i.e. acknowledged). If writes are not allowed until the invalidations are completed, the strong definition of a coherent memory given above will be implemented. If execution can continue before the invalidations are completed, higher efficiency may be obtained, but, only the weak definition of coherency can be enforced, and the property of sequential consistency [Lamp79] may be violated.

So far we have not addressed the question of how to locate the current owner of a page during a fault. Li has used both a distributed and a centralized mechanism for finding the current owner of a given page. In the distributed case (the "Dynamic Distributed Manager" [Li86a]) the owner of a page is determined when a page fault occurs by retaining, at each node, a *hint* (or guess) of who the owner of the page is. Requests for changes in access rights are sent to the node that the given hint indicates. When the hint is wrong, the node receiving the request simply forwards the request to the node it thinks is the owner. The hint at a node is changed to point to the node which is obtaining ownership when it is recognized that ownership of a page will be transferred. Li has shown that no loops or deadlocks occur in the distributed tree which develops as a result of these hints.

The centralized mechanism (the "Improved Centralized Manager" and "Fixed Distributed Manager" [Li86a]) uses a known manager for each page to locate the current owner of the page. All request for changes in access rights and ownership are forwarded to the current owner only after passing through the node which is the manager of the page. The manager for a page then is always aware of the current owner of the page. Our analysis of the distributed and centralized mechanisms to determine the owner of the pages has shown the performance difference to be small as compared to the differences in the DSM algorithms specified in the next section. We have therefore decided to consider the method used to determine the owner of a page to be separate and distinct

from the solution to the *double faulting* problem, which is defined in the next section. All the algorithms analyzed in this paper could use either a centralized manager or hints to determine the owner of a page. For consistency, however, all of them use hints.

3. Double Faulting and Distributed Shared Memory Algorithms

Our *Base* algorithm follows the protocol outlined in the previous section for read and write faults and uses hints to determine ownership. With this algorithm, a location in an invalid DSM page that is read and then written by two successive references will incur a pair of faults, a read fault followed by a write fault. We refer to this (read, write) pair of faults to the same page as a *double faulting pair*. If the DSM algorithm could predict that the read fault were to be quickly followed by the write fault, the write fault could be eliminated by taking the equivalent of a write fault when the read fault occurs. Information could be provided by the application to indicate that a page to be read would soon be written (by a hint to the operating system). Prediction without help from the application would require interpretation of the instruction stream causing the fault, particularly difficult since the two faults can be generated from different instructions.

In such a double faulting situation, a node to node page transfer will occur as a result of both the read and the write fault when using the Base algorithm or the "Improved Centralized Manager" algorithm of Li [Li86a]; the difference in these algorithms is that they use different methods to determine the owner of a given page: centralized and hints, respectively. When the access rights to a page are updated from read-only to read&write (during a write fault), an unnecessary page transfer can occur if the transferred page is the same as the previous read-only copy held at the faulting node.

To give an example of why the Base algorithm performs a page transfer during all write faults, whether a previous read-only copy exists at the node or not, consider the following scenario: Two nodes have read-only copies of a page which is owned by a third node. Both nodes write fault on that page at the same time and send request messages to the owner of the page. There may be a very long path from these nodes to the current owner of the page, and, assuming an arbitrary network between the nodes (we only assume that all messages from one node to another are received in the order sent), the delay for each of the messages to reach the owner is undetermined. One of the nodes will receive read&write access (and ownership) to the page first, invalidate other copies of the page, and modify its own copy. The other requesting node may still be waiting to obtain write access to the page, but, unbeknownst to the request message still outstanding, the requesting node no longer has a current copy of the page. A page transfer is completed on every write fault without considering whether the requesting node had an up to date read-only copy of the page at the time the message requesting write access (and ownership) was sent since it is possible for the read-only copy to be invalidated during the time the request for write access is outstanding.

There are alternatives to this double page transfer scenario. One alternative is to adapt a single bus hardware cache coherency scheme to a DSM implementation; the possibility is mentioned in [Good83]. A single bus serves as an excellent broadcast media and serializes all coherency requests. The "owner" of a cache block (page) can be determined by only a single broadcast message, with all of the nodes receiving the message and deciding if they are the owner (as in the "Broadcast Distributed Manager" of Li [Li86a]). A block (page) transfer can be eliminated when writing to a block (page) of which a current read-only copy is held when the request to write (an invalidation of all other copies and transfer of ownership) appears on the broadcast media and the block (page) is not invalidated before the request appears. Such a broadcast scheme is much less appropriate for a DSM implementation than for single bus hardware coherency schemes for the following reasons: (1) Large amounts of processor time would be required to respond to all the broadcast requests; and (2) Ethernet message transfer is not as reliable as single bus messages, immediate acknowledgements would likely be required on all broadcasts. As a final comment, when the communication media is not inherently broadcast, even hardware schemes require double block transfers on successive reads and writes to a block[1] since there is no broadcast serialization of the requests.

Another potential cure to the double faulting problem is the naive one given by the *Hot Potato* algorithm. This algorithm solves the problem by treating all faults (read or write) as if they were a write fault. The implication of this algorithm, however, is that *one and only one* copy of each page may exist at a given time. This copy is stored and owned by the node who has read&write access rights to the page. In case of a double faulting pair, the first (read) fault gains read&write access to the page. The second (write) fault does not occur since the page has been granted read&write access after the first fault.

The Hot Potato algorithm shows us how to solve the double faulting problem if we are willing to live with a single copy DSM. The question which then arises is whether it is necessary to give away the advantages of a multi-copy (read-only) DSM in order to solve the double faulting problem? The next algorithms indicate that the double page transfer can be eliminated in a multi-copy DSM.

The "Dynamic Distributed Manager" algorithm of Li [Li86a], which we will call the *Li* algorithm, allows for multiple read-only copies to exist. The Li algorithm differs from the Base algorithm only in when ownership of a page is transferred. The Li algorithm transfers ownership during both a read and a write fault, whereas, the Base algorithm transfers ownership only during a write fault. When a node first reads and then writes to the same page, the first (read) fault will obtain both a copy of and ownership to the page. On a write fault occurring soon after the read fault, an invalidation of other pages is all that is necessary to allow read&write access since the local

[1] The hardware coherency protocol of the Wisconsin Multicube [Good88] is an example of one which requires a double block transfer on successive reads and writes to the same block.

node is the owner of the page.

The Li algorithm is an improvement over the Base algorithm when considering double faulting pairs in that it eliminates the page transfer necessary for the write fault when it occurs soon after the read fault. Unfortunately, it does not minimize the number of page transfers when the write does not occur soon after the read. When a write fault is taken at a node which has a read-only copy of the page, but, is not the owner, unnecessary page transfers can still result. Furthermore, transferring ownership on a read fault is not always the best decision. When single-writer/multiple-readers (producer-consumers) sharing is taking place, the ownership transfer is a poor choice. In this case, ownership is transferred to the readers when they obtain copies of the page, whereas, it would be preferable for the single writer to remain owner at all times. With the hint based means of determining the current owner of a page, longer forwarding strings will result with this unnecessary ownership transfer.

The *Shrewd* algorithm eliminates *all* unnecessary page transfers with the help of a sequence number per copy of a page. Each time read&write access is newly obtained, the sequence number of the new copy is the sequence number of the previous copy incremented by one. Each time read-only access to a page is obtained, the sequence number of the new copy is the same as the sequence number of the old copy. On each write fault at a node with a previously existing read-only copy, the sequence number of the copy is sent with the request for read&write access. When arriving at the node which is the owner of the page, the owner compares the sequence number of its copy of the page to the sequence number in the incoming request. If they are equivalent, the requesting node can be allowed read&write access to the page without a page transfer. Page transfer minimization is *guaranteed* since the sequence number is a direct indication of the validity of the previously existing copy of the page at the requesting node. In the case of a double faulting pair, the first (read) fault gains a copy of the page, while the second (write) fault is likely to find that a page transfer is unnecessary since the sequence numbers are equivalent.

Alternative algorithms which provide the guaranteed page transfer minimization of the Shrewd algorithm while transferring ownership only on write faults were also considered. A particular possibility is an *optimistic* algorithm in which it is assumed that a page transfer will be unnecessary when a node is requesting write access to a page which it already has a read-only copy of. Ownership transfer would occur during the write fault. If the copy were still available (not invalidated) after the write request returned from the previous owner of the page to the requesting node, it would be the current copy of the page. It is possible that the copy at the requesting node could be invalidated before ownership is transferred to the requesting node. In that case, the requesting node would retrieve the up to date copy of the page from the previous owner. This optimistic algorithm would thus require separate transfers for the ownership to and copy of the page in that situation, whereas, the Shrewd algorithm transfers both at the same time. We do not consider this optimistic algorithm further since it can only require more messages than the Shrewd algorithm.

The difference between the Shrewd and Base algorithms is that the Shrewd algorithm eliminates page transfers when a node faults on a page which it has an up to date copy of. The difference between the Shrewd and Li algorithms is the means by which the double faulting problem is solved; the Shrewd algorithm solves it by sequence numbers and the Li algorithm solves it by ownership transfer. Neither the Shrewd or the Li algorithm can be said to be superior to the other in all cases. Our simulations, in section 5, help to highlight the differences between the two algorithms and the conditions which make one or the other perform better.

4. The Simulation Model

When faced with the choice between alternative algorithms, the most important question a system designer must ask is: what algorithm is best for my system? The answer to that question depends on many factors within the system, as well as the characteristics of the applications using the system. The goal of this study is to understand the implications of the differences in the Base, Hot Potato, Li, and Shrewd algorithms presented in section 3 and to attempt to determine under what conditions these algorithms perform better or worse.

A simulation model has been developed which consists of a synthetic reference generator, a processor model, and a network model. This model, described below, was used to analyze the performance of the different DSM algorithms. The major advantage of this simulation model is its flexibility. It can be used to simulate many potential distributed system configurations and it allowed us to uncover the behavior of the DSM algorithms under widely varying application access patterns as well.

4.1. The Synthetic Reference Generator

The generator views the parallel program as executing localities [Denn72], or *phases*. The execution of each process in the program is considered to be a sequence of phases through which it flows. Each phase references a subset of the address space of the program in a characteristic pattern. The memory references of a process are considered to access one of two types of memory: *private*, and *shared*. The private memory can only be accessed by the process to which it belongs and is assumed to be mapped to the local memory of a node without causing any page faults. The shared memory of the program is mapped to the DSM. References to this type of memory can and will cause page faults.

The reference string of a process is assumed to consist of two types of phases: shared and private. The duration of a phase is called a *burst*. Each burst has the following characteristics: *percentage*, *placement*, *read to write ratio*, *locality*, and *length*. During a private burst the process accesses only its private memory whereas both shared and private references are made during a shared burst. The burst percentage determines how many accesses during a shared burst are to shared memory. Each shared burst has a placement around which the references to the shared memory will be centered. The distribution of the placement for different shared bursts is

uniform across the shared memory space. The address of each shared memory reference during a shared memory burst is determined by a normal distribution with the mean being the placement of the burst and the locality of the burst being the standard deviation. Note that the placement is constant during a given burst. By increasing (decreasing) the locality of a burst, a larger (smaller) locality of reference within a burst is simulated. Each shared memory reference is chosen to be either a read or a write according to the read to write ratio.

Shared and private bursts are of the same length on the average. On completing a burst, a choice is made as to the type, length, and placement of the next burst depending upon the *private/shared ratio*, *mean burst length*, and placement distribution (uniform across the shared memory), respectively. The burst length of a shared burst is the number of accesses to the shared memory. Since shared accesses occur interleaved with private accesses, the actual number of processor references during a burst is larger than the burst length.

Although the shared memory is only accessed during shared bursts; since bursts appear back to back over the long term it will appear as though private and shared memory references occur interspersed over all time. Locality of reference is simulated within, but not across, shared bursts.

This synthetic access pattern is not intended to be, nor should it be taken as, a realistic access pattern. It is a tunable tool used to determine how the various DSM algorithms perform under varying circumstances to gain insight as to the conditions under which one algorithm performs better than another. It is unlikely that the synthetic access pattern can realistically be said to characterize any given real access patterns. However, it contains interesting characteristics which will be exploited in the simulations, emphasizing the differences in the DSM algorithms. The absence of an attempt to model interprocess synchronization and primitives may be the most limiting characteristic of this synthetic application model.

4.2. The Processor and Network

The network models a multicast message passing network (token ring, bus, or ethernet) which can send messages at the rate of 10 Mbit/sec with a channel dead time of 3 microseconds between each message. Messages can be of arbitrary length, though in this simulation they are assumed to be either a short packet or a page packet. A short packet is 32 bytes, includes all header information necessary to transfer a message on the network, and is assumed of sufficient size to carry all messages other than page transfers. The size of a page packet is the size of a short packet plus the size of a page (in all the simulations here, 4 Kbytes). The channel is assumed to be error free and nodes are assumed to buffer each and every message without error[2]. Acknowledges are assumed unnecessary. The network model could be extended to account for acknowledgements and random channel errors; however, it is unlikely that this would change the fundamental results of this study.

[2] That is, it is assumed that the network interface at a node can buffer multiple incoming messages without error.

References by the processor to its local memory are assumed to take a fixed amount of time per reference (except on a page fault). The processor model attempts to take into account the processor time necessary to receive and return page requests, forward page requests, invalidate pages, and page fault. Page transfers (both to and from a node) are assumed to take a considerable amount of processor time due to needed copying of data.

5. Results

The simulations isolate the effects of the values of the workload parameters of table 1 by changing the value of a parameter while holding others constant. In so doing, specific characteristics of the DSM algorithms can be analyzed in detail. It was difficult to choose the "correct" values of the parameters in the model since there is no published information to guide us. We simply chose values we thought were appropriate.

Parameters			
Workload		CPU Time Taken	
Name	Default	Operation	Time
Number of Nodes	5	Forward Message	.8 msec
Reference Rate	1 MHz	Send Page	5 msec
Page Size	4 Kbytes	Invalidate Page	.5 msec
Shared Memory Size	50 pages	Page Fault	.5 msec
Burst Percentage	25 %	Receive Page	5 msec
Private/Shared Ratio	40 to 1		
Mean Burst Length	100 Accesses		
Locality	20 Bytes		
Placement Distribution	Uniform		

Table 1. Parameters for the Workload and the CPU Time Taken.

The performance metric used in this paper, *processing power*, is intended to indicate the effective number of processors of the parallel synthetic access pattern running on the system using the given DSM algorithm. Processing power is defined here as the sum of the processor references of the synthetic access pattern at all nodes divided by the maximum number of references achievable by a single processor in the same amount of time running at the memory access rate (assuming no page faults, of course).

5.1. Read to Write Ratio

Simulation results for varying read to write ratios are shown in Figure 1. Varying the read to write ratio is important in examining the effectiveness of the algorithms in solving the double faulting problem. It also is important to determine what can be gained by allowing multiple read-only copies of a given page of the DSM to exist at multiple nodes. The curve for the Hot Potato algorithm is a straight line since its performance is independent of the read to write ratio. The characteristic trough[3] in the Base, Shrewd, and Li curves of Figure 1 is due to

[3] Note that the curve of all algorithms pass through the same point at a read to write ratio of 0.

the presence of double faulting pairs. In the lower portions of the trough double faulting pairs are more frequent. Pages faulted on a burst are first being faulted as a read fault; soon after a write fault is taken in the same burst. This is opposed to either the situation where the write fault is the only fault taken during a burst (since a write occurs before a read), or a read fault is taken during a burst and no subsequent write fault occurs within the same burst. The better performance to the left of the trough is because the first fault is more often a write fault during a burst, due to the low read to write ratio. The better performance to the right of the trough is because more often a read fault occurs and no write fault occurs, allowing for the increased concurrency which multiple read-only copies allows.

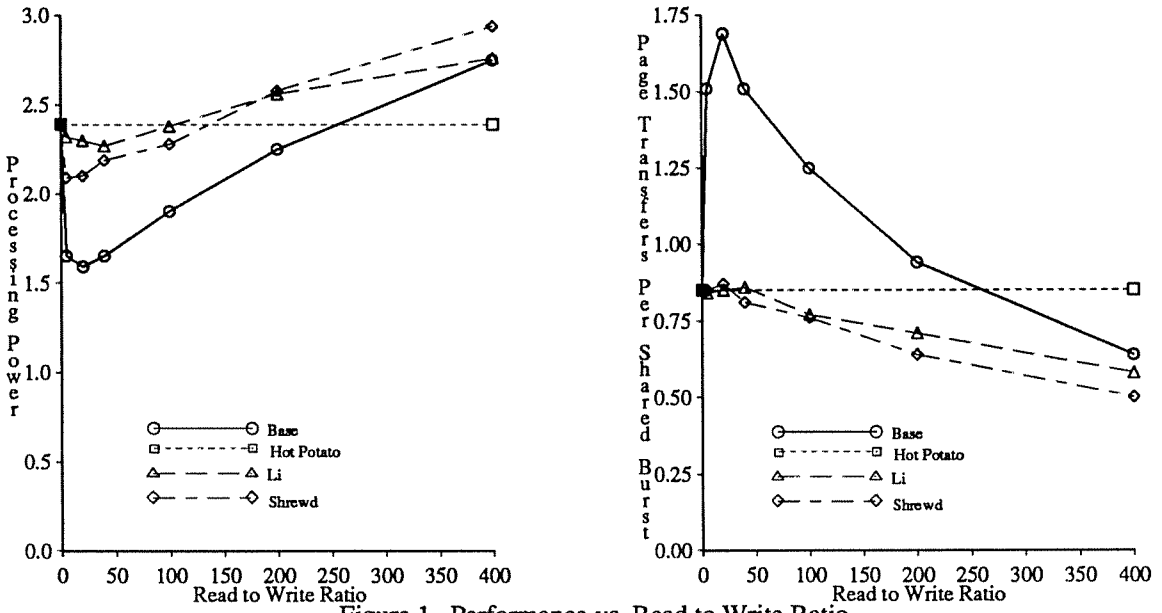

Figure 1. Performance vs. Read to Write Ratio.

This figure shows the performance of the algorithms for varying read to write ratios with a five node synthetic access pattern. The left graph shows the processing power for the varying read to write ratios, the right shows the number of page transfers per shared burst for the read to write ratios.

The Base algorithm requires more network page transfers than the other algorithms, particularly at small read to write ratios, resulting in its poor performance in this area. Figure 1 shows that the Base algorithm near the peak is forcing approximately twice the number of page transfers per shared memory burst as the other algorithms. The Hot Potato algorithm performs poorly at large read to write ratios since it does not allow for read-only copies of the pages of the DSM to exist at multiple nodes in the system. Both the Li and Shrewd algorithms reduce the number of page transfers per shared burst, yet retain the potential for parallel access which is allowed by multiple read-only copies.

The Li algorithm performs better than the Shrewd at smaller read to write ratios since write faults (just after a read fault) can often be handled without request from a remote node, simply by invalidating other copies of the

page. This can be done particularly quickly in this simulation since invalidations are not acknowledged. Note, however, how the Shrewd algorithm performs better than the Li algorithm at larger read to write ratios. This occurs since nodes are write faulting on pages which they have read-only copies of, but don't have ownership of. The Li algorithm is unable to detect that page transfer is unnecessary in this situation, often the case when a write fault does not occur during the same burst as the read fault which obtained the read-only copy of the page.

5.2. Number of Nodes

Performance of the algorithms versus the number of nodes is indicated by Figure 2. It shows the Hot Potato and Li algorithms are close in performance at a read to write ratio of 5, with a slight advantage to the Hot Potato. They are followed by the Shrewd algorithm, then the Base algorithm. The Base algorithm performs only about half as well as the leaders for large numbers of nodes. For a read to write ratio of five the double faulting problem is prevalent so the algorithms which solve this the best, the Hot Potato and the Li, perform the best. At a read to write ratio of 400 the Shrewd and Li algorithms perform best since they allow multiple read-only copies, followed by the Base and Hot Potato algorithms. Figure 2 shows that the network is being saturated around 15 nodes, which is roughly where the processing power curves saturate as well, indicating how network utilization is very important to the performance of a DSM algorithm. The Shrewd and Li algorithms have lower network utilizations than the Base algorithm. It is for this reason that they perform better than the Base algorithm over the range of conditions studied.

The fact that the Li algorithm is superior in performance to the Shrewd algorithm for a read to write ratio of 5 in Figure 2 (reference the upper left graph) indicates that transferring ownership on a read fault is a good idea under those conditions. The performance difference is due to the fact that the Shrewd algorithm must transfer ownership during the write fault of a double faulting pair, whereas, the Li algorithm simply needs to invalidate other copies of the page. When the network utilization is high, the delay in execution caused by this required ownership transfer results in significant performance degradation, though it has little effect on the network utilization.

5.3. Other Relationships

The relationship of burst length to processing power is indicated in Figure 3. An access pattern with a longer burst length will make longer, less frequent accesses to the DSM. A longer burst length results in a large improvement in performance for the given synthetic access pattern. This occurs since the longer burst length means that there are less bursts per unit time. Since a burst typically implies a page transfer, less bursts result in a big win. Applications using DSM will likely perform better if they limit their accesses to the shared memory to less frequent, longer accesses. The algorithms intended to solve the double faulting problem performed well over

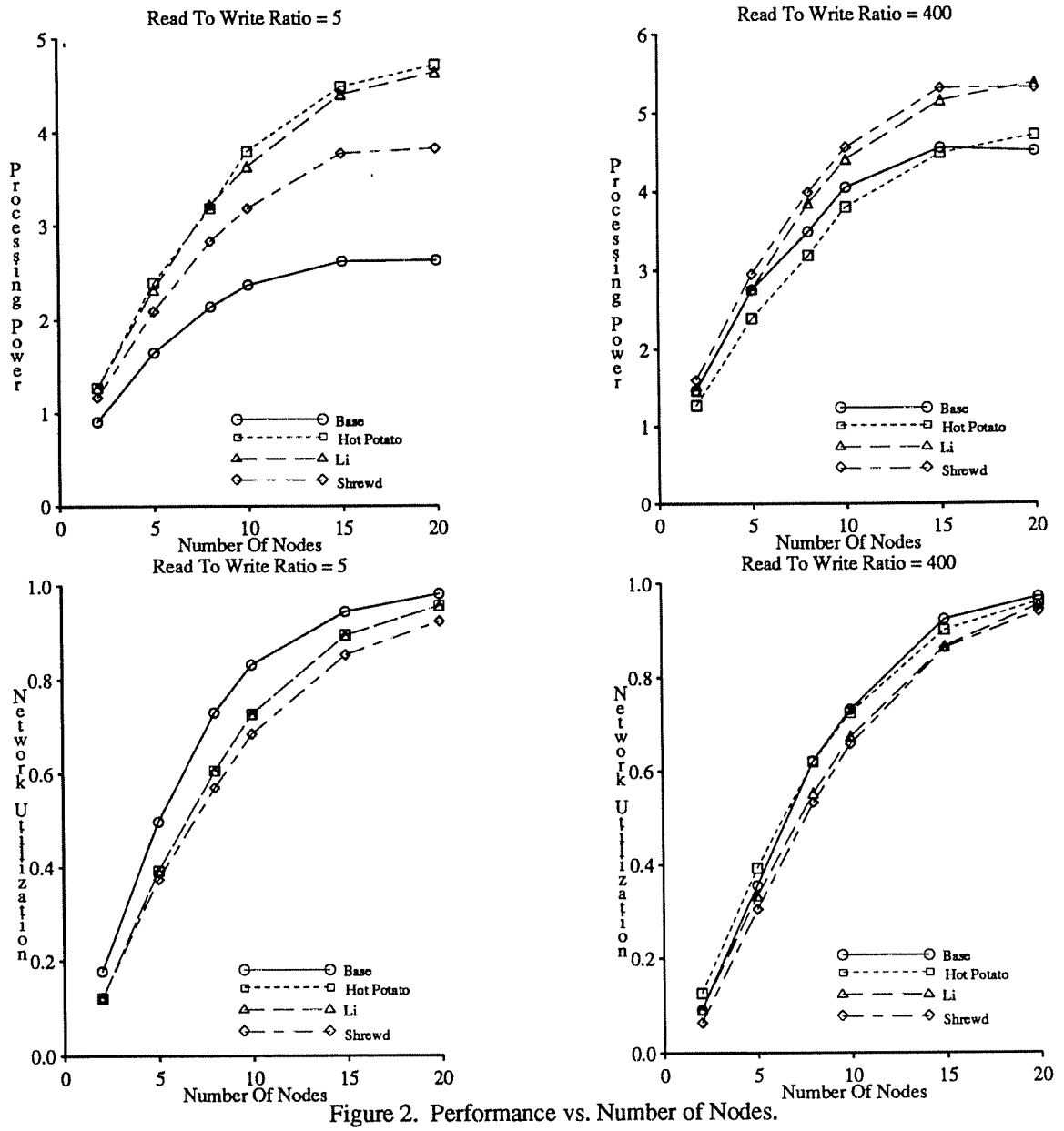

Figure 2. Performance vs. Number of Nodes.

This figure shows the performance of the algorithms versus the number of nodes for read to write ratios of 5 (left) and 400. The top figures show the processing power and the bottom figures show the network utilization.

the spectrum of burst lengths for a read to write ratio of five. At a read to write ratio of 400, the algorithms which allow multiple read-only copies perform well for small burst lengths since most of the bursts are read-only. As the burst length increases above the read to write ratio, the double faulting problem becomes prevalent, so the algorithms which solve it perform better.

Figure 4 indicates that performance decreases as the size of the locality of reference increases. One reason for this is that it is likely that multiple pages may have to be transferred during a given burst. Another is that it is likely to increase contention among nodes for a page. That is, multiple nodes may be accessing the same page of

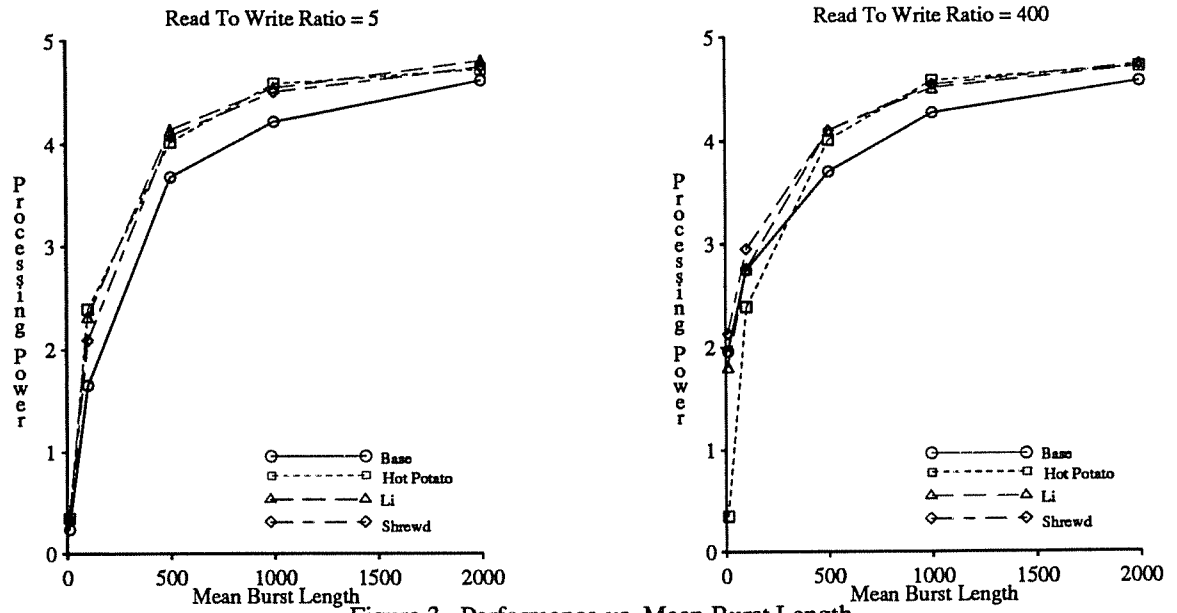

Figure 3. Performance vs. Mean Burst Length.

This figure shows the Performance of the algorithms versus the mean burst length for read to write ratios of 5 (left) and 400 on five node synthetic access patterns.

the DSM at the same time. The general downward slope of the curves is due to the multiple page transfers needed per burst. The fact that the performance of the Shrewd, Li and Base algorithms increases relative to the performance of the Hot Potato algorithm as the size of localities increase are an indication that these algorithms perform better in the presence of this contention since they allow multiple read-only copies of a page to exist.

The fact that the Shrewd algorithm significantly outperforms the Li algorithm at a read to write ratio of 400 in Figure 4 is an indication that the Li algorithm performs some unnecessary page transfers. Note how the Base algorithm performs similar to the Li algorithm under these conditions. The guaranteed page transfer minimization of the Shrewd algorithm is very useful when pages are being read and will not soon be written.

The relationship of the private to shared burst ratio to performance is indicated in Figure 5. The higher the private to shared burst ratio, the less often shared memory is accessed. Figure 5 clearly indicates the less shared memory accesses, the better the performance. Applications using DSM which use the DSM less may have better performance than those which use it more. While this may be contrary to the reason for using shared memory versus using message passing, one must realize that DSM can perform poorly when used in the wrong circumstances.

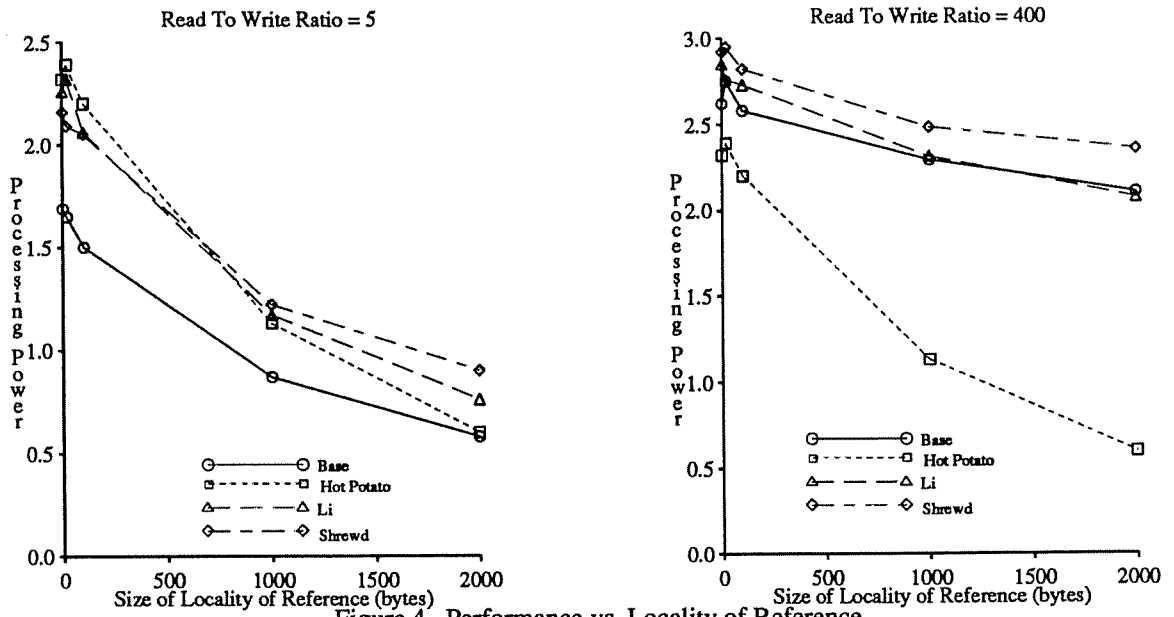

Figure 4. Performance vs. Locality of Reference.

This figure shows the Performance of the algorithms versus the locality of reference for read to write ratios of 5 (left) and 400 on five node synthetic access patterns.

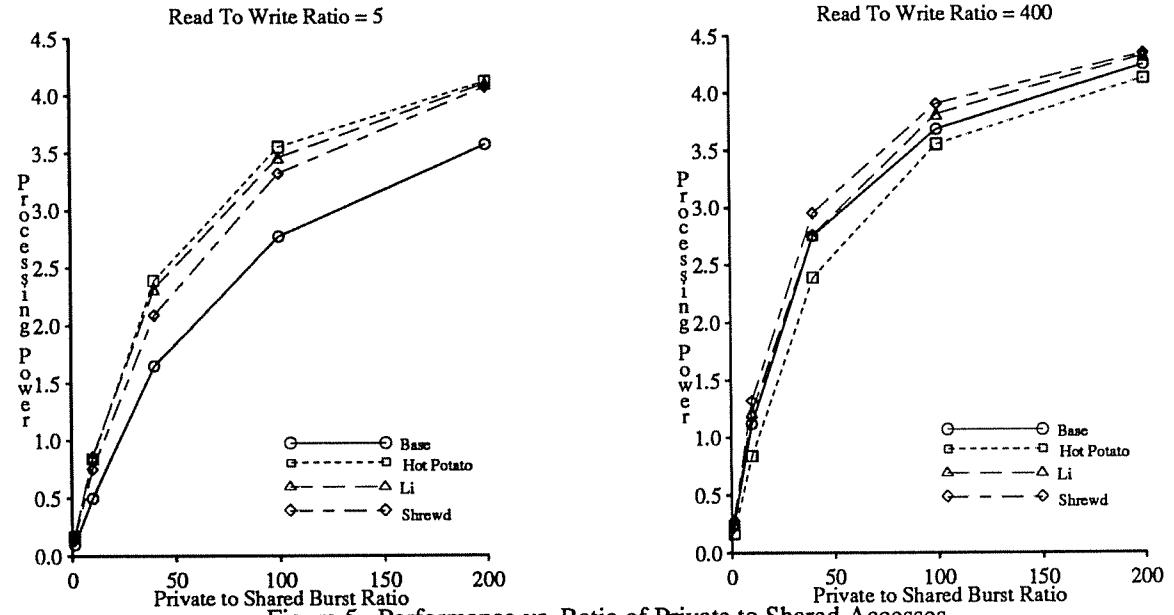

Figure 5. Performance vs. Ratio of Private to Shared Accesses.

This figure shows the Performance of the algorithms versus the ratio of private to shared accesses for read to write ratios of 5 (left) and 400 on five node synthetic access patterns.

6. Conclusions

This study brings out major characteristics of DSM and issues a DSM implementation should address. Several algorithms were presented here to implement DSM: Base, Hot Potato, Shrewd, and Li, each has its

tradeoffs. The algorithms analyzed other than the Base algorithm attempt to minimize the implications of the double faulting problem outlined in section 3. Solving the double faulting problem reduces the number of page transfers and decreases network utilization. Reducing usage of the network is important since network operations are expensive and often the major determinant of the performance of distributed systems.

The naive Hot Potato algorithm provides a very good solution to the double faulting problem, but, performs poorly as compared to the other algorithms when read sharing can occur. This is due to the implication of the Hot Potato algorithm that at any time one and only one copy of any given page of the DSM exists. This disallows the possibility of read-only copies of a page on multiple nodes and the increased parallel activity which can occur as a result. A pathological scenario for the relative performance of the Hot Potato algorithm is the case when areas of the shared memory are accessed read-only, likely a common occurrence. In this case, the other algorithms will clearly be preferred over the Hot Potato.

The Shrewd algorithm retains the parallel potential of multiple read-only copies, yet significantly reduces the number of node to node page transfers as compared to the Base algorithm. In our simulations it performs best when the read to write ratio is high, with still some write sharing. The Shrewd algorithm will consistently outperform the Base algorithm.

The Li algorithm is a good solution to the double faulting problem, yet, the ownership transfer it requires can at times be counter-productive. When a node is obtaining a read-only copy of a page and will not subsequently write that page, it is preferable that the node does not become the owner of the page. The Li algorithm, however, will always make the node the owner of the page. Nevertheless, the Li algorithm performs very well relative to the other algorithms which allow for multiple read-only copies in our simulations, particularly for small read to write ratios. The major reason for this is that it is a very good solution to the double faulting problem when the write fault occurs soon after the read fault. Our simulations show that algorithms which allow multiple read-only copies of a page while solving the double faulting problem will result in better performance than those which do not.

A simple variant of the Shrewd and Li algorithms would be one which combines the properties of both to achieve the advantages of both. Since they use separate mechanisms to solve the double faulting problem and reduce network utilization, the ideas of both can be combined into an algorithm which has the good performance of the Li algorithm at lower read to write ratios while retaining the page transfer minimization provided by the Shrewd algorithm at higher read to write ratios. This algorithm will certainly result in less page transfers than the Li algorithm. We are confident that this variant will prove useful and expect to examine it in the future.

Our simulations have shown the Shrewd and Li algorithms to have superior performance to the Base algorithm over a wide range of application characteristics. In some cases we observed they obtained twice the

performance of the Base algorithm. The algorithms consistently perform over 10-20 percent better than the Base algorithm under most conditions studied. While our synthetic access pattern generator is not a real application, we feel it captures significant characteristics of applications which will use a DSM, notably, locality of reference and sharing. Its tunability was most useful in bringing out the differences in the DSM algorithms. We expect performance gains will be seen in practice by using the algorithms which solve the double faulting problem rather than the Base algorithm.

Finally, we present some general application characteristics which can be exploited by all DSM algorithms:

(1) Designating certain areas of the shared memory as read-only. This allows the pages of the DSM to exist on multiple nodes. This cannot be exploited by the Hot Potato algorithm, however.

(2) Less frequent, longer term shared memory access. This can reduce the level of contention among nodes for the shared memory and the number of page transfers needed.

(3) Differing locality of reference across nodes. This allows the pages of the DSM to be partitioned among the nodes. This can be particularly useful if the application is careful about its placement of data. It is helpful to place independent objects on separate pages in order to reduce contention for pages.

7. Acknowledgements

The authors would like to thank Mark Hill for his reviewing and suggested improvements. The authors would also like to thank the reviewers for pointing out some serious flaws in a previous version of this paper.

8. References

[Agar88] A. Agarwal, R. Simoni, J. Hennessy and M. Horowitz, An Evaluation of Directory Schemes for Cache Coherence, *The 15th Annual Symposium on Computer Architecture*, Honolulu, Hawaii (1988), 280-289.

[Arch86] J. Archibald and J. Baer, Cache Coherence Protocals: Evaluation Using a Multiprocessor Simulation Model, *ACM Transactions on Computer Systems*, 4, 4 (November, 1986), 273-298.

[Cens78] L. M. Censier and P. Feutrier, A New Solution to Coherence Problems in Multicache Systems, *IEEE Transactions on Computers*, c-27, 12 (December, 1978), 1112-1118.

[Cher86] D. R. Cheriton, G. A. Slavenburg and P. D. Boyle, Software-Controlled Caches in the VMP Multiprocessor, *Proceedings of the 13th International Symposium on Computer Architecture*(1986), 366-374.

[Cher88] D. R. Cheriton, A. Gupta, P. D. Boyle and H. A. Goosen, The VMP Multiprocessor: Initial Experience, Refinements, and Performance Evaluation, *The 15th Annual Symposium on Computer Architecture*, Honolulu, Hawaii (1988), 410-421.

[Denn72] P. J. Denning, On Modelling Program Behavior, *Proceedings of the AFIPS Spring Joint Computer Conference*(1972), 937-944.

[Flei87] B. D. Fleisch, Distributed Shared Memory in a Loosely Coupled Distributed System, *Proceedings of the ACM SIGCOMM '87 Workshop*, Stowe, Vermont (August 1987), 317-327.

[Good83] J. R. Goodman, Using Cache Memory to Reduce Processor-Memory Traffic, *Proceedings of the 10th International Symposium on Computer Architecture*(1983), 124-131.

[Good88] J. R. Goodman and P. J. Woest, The Wisconsin Multicube: A New Large Scale Cache-Coherent Multiprocessor, *The 15th Annual International Symposium on Computer Architecture*(1988), 422-431.

[Holl88] M. A. Holliday, Page Table Management in Local/Remote Architectures, *International Conference on Supercomputing*(1988), 1-8.

[Lamp79] L. Lamport, How To Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers*(September 1979), 690-691.

[Li86a] K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, To appear in ACM Transactions on Computer Systems. *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*(August 1986), 229-239.

[Li86b] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. dissertation, Yale University, Department of Computer Science, YALEU/Dept. of Computer Science/RR-492, (September 1986).

[Li88] K. Li, IVY: Shared Virtual Memory System for Parallel Computing, *International Conference on Parallel Processing*(1988).

[Livn86] M. Livny, *DENET Users Guide, Version 1.0*, Computer Sciences Department, University of Wisconsin-Madison, (1986).

[Oust80] J. K. Ousterhout, D. A. Scelza and P. S. Sindhu, Medusa: An Experiment in Distributed Operating System Structure, *Communications of the ACM*, 23, 2 (February, 1980), 92-105.

[Pfis85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton and J. Weiss, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *Proceedings of the 1985 International Conference on Parallel Processing*(1985), 764-771.

[Youn87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron, The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *11th Symposium on Operating Systems Principles*(November 1987).