1994

Reducing page-fault delays in a MIMD distributed virtual shared memory

Thomas P. Browne
Lehigh University

AUTHOR:

Browne, Thomas P.

TITLE:

Reducing Page-Fault Delays in a MIMD Distributed Virtual Shared Memory

DATE: October 9, 1994

Reducing Page-Fault Delays in a MIMD Distributed Virtual Shared Memory

by

Thomas P. Browne

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

August 11, 1994

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

8-25-94.

Date

Thesis Advisor

Chairperson of Department

ACKNOWLEDGEMENTS

I would like to thank Professor Spezialetti for all her help and encouragement during the past year on this thesis as well as with my master's program in general.

My thanks also to Ali Erkan for his help with communication protocols and his communication routine library which were used in some of the timing programs that I used for this paper.

TABLE OF CONTENTS

0. Abstract

With the growing use of networks of personal computers and workstations, creating applications which can take advantage of using more than one machine on the network has become more practical and desirable. The use of a distributed virtual memory shared across machines of the network would present the programmer with a familiar environment for program control and synchronization. However, most distributed virtual shared memories suffer serious performance problems due to the delays caused by memory page faults on local machines.

This paper suggests an approach to minimize these delays through an educated allocation of the physical memory as local permanent memory, virtual permanent memory and temporary or "cache" memory, and through a careful distribution of memory pages amongst the machines. A step by step process to reduce delays through the duplication of shared memory pages is presented, along with a set of conditions for determining which pages to duplicate and when to stop adding pages into the permanent memory.

The paper then suggests a method for handling page fault requests through a pre-determined priority and a process for determining these priorities in an attempt to equalize the delay on each node in the network.

1. Introduction to Distributed Virtual Shared Memory

With the greater affordability and power of workstations currently available, networks of small workstations are becoming more prevalent in business world environments. If several machines on a network were able to work together on a single problem, their combined resources could provide a considerable performance improvement when compared with solving the problem using only a single workstation. Such co-operation between nodes, however, would require that more than one machine have access to certain parts of data.

This desire to have machines co-operate efficiently led to the development of a distributed virtual shared memory (or DVSM) model. This model provides the system with a view of a single shared address space which is available to some (or all) of the machines in the network. The application programmer can access this memory just as if it were local memory, but the same data would be available to all of the machines in the DVSM environment. However, there is not an actual physically shared memory in the network. This view of a shared memory is provided through the distribution of memory amongst the local physical memories of the machines and a communication system which supports the exchange and update of this data between machines.

The advantage of a DVSM is that it provides the programmer with a familiar environment for the exchange of data and the synchronization of processes, as well as often resulting in more understandable code. A DVSM relieves the programmer of the necessity of dealing with low level communication protocols which are often more complex than other communication techniques, such as parameter passing, and result in confusing code when compared to the virtual memory environment.

The most important consideration in a DVSM system is that it must guarantee a coherent view of the shared data across all of the machines. In other words, like

hardware caches, a read to any active copy of the data on any of the machines in the network must always return the same value -- the value of the last update to that shared address. In their paper [7], Kessler and Livny suggest four major ideas that DVSM algorithms use to maintain the abstraction of a coherent memory:

> (1) The [DVSM] 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 [DVSM] 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.

It is in the handling of these memory pages and in maintaining the system's coherency that DVSMs experience significant communication delays which can degrade their performance.

2. The Page Fault Problem in DVSM

While a DVSM environment makes work easier for the programmer, a shared virtual memory has one serious problem that is not a consideration with a physical shared memory multiprocessor systems: the communication delays incurred by the system.

There are two properties of a DVSM that require communication over the network, and therefore delays. First, a DVSM needs to share data with other machines in the network, and secondly it must assure that the data is consistent across the entire network.

The first property is encountered when one workstation requires data from a portion of the shared memory which the workstation does not currently hold, or what is known as a read page-fault. Whenever a program needs to access a remote page of memory, the DVSM system needs to determine who has a current copy of that page, and then requests a transfer of that memory across the network. However, unlike hardware cache handling, the amount of data transferred is usually on the order of kilobytes as opposed to only a few words. The transfer of this much data can result in considerable delays especially on a congested or unreliable network.

Finding ways to improve the performance of shared memory systems has been the focus of a lot of research [2-4, 6-10, 12-14]. One of the ways discussed, and the one this paper considers, is the initial reading of additional data at the start of execution. The reading of additional pages before execution will reduce the need for later communication as some of the page faults that shared memory accesses would have generated have been eliminated and combined into one communication. Since the overhead of communication is often a significant part of the overall communication

delay (see Table 1), any reduction in the number of actual page faults will result in an improved performance.

The maintenance of data consistency amongst multiple copies of shared data also results in considerable communication overhead. This overhead, a write page-fault, results when a process attempts to write to the shared memory. Unlike a read fault, this overhead is encountered whether the writing process already has a copy of the memory page or not. Whenever any page of the shared memory is changed, the shared memory system must guarantee that all other active copies of the page are either made consistent with the new data, or that they are removed from the system so that the now invalid data will no longer be used.

While there are some similarities between the behavior of shared memory and hardware caches, techniques for maintaining consistency in caches may not work well in a shared memory environment. A common approach to cache coherence is to update the main memory copy of the block, and then "invalidate" all of the cache copies, so no one will use the now out-of-date information. While this performs satisfactorily for hardware caches, a shared virtual memory often deals with a much larger amount of information that would be invalidated -- as much as several kilobytes as opposed to only a few bytes. The result is that any further access to the data on that page, even the data that was not changed, will generate additional read page-fault and will require the entire page to be resent over the network.

In a cache, this delay is sufficiently short, because only a small amount of "correct" data has been invalidated, and there will likely be few accesses to that data. In a shared memory system however, there is a far greater chance of that page being needed again on one or more nodes, thus generating new page-faults, and increasing the amount of time lost to communication delays.

This indicates that an approach other than page invalidation should be considered. Wilson, LaRowe and Teller [14] demonstrate that in most cases an update based scheme will perform better than a page invalidation scheme for a DVSM even if the actual delay of performing the updates takes longer than the invalidation message would because it will not generate the additional number of page-faults that the invalidation would incur.

These schemes will be considered in more depth in section five, where a system model for evaluating these proposals will be discussed.

3. SPMD Solution to Reduce Page Fault Delays

Reducing the overall delay in program execution resulting from page fault communication was the basis of work by Clancey and Francioni at Michigan Technological University. Their work [3], though, presents a simplified system which assumes that the same program is executing on each node of the network, although such an assumption is never stated. The results of examining this single program - multiple data environment will be used to extend the ideas to a more general MIMD system in sections 6 and 7.

Clancey and Francioni present a system where local memory is partitioned into two areas: the "home set" and the "copy set". The "home set" is the collection of pages that are owned by that node, and that will always be available on that node for the duration of the program execution -- there is no migration of page ownership across the network. The "copy set" is that portion of memory that is used for temporary storage of faulted for pages. All page replacement occurs in the copy set, never in the home set, and only occur as a result of page faults.

In order to better understand the behavior of their techniques and measure their gain, a general equation representing the total length of time to service all of the page faults was derived [3]:

$$Cost_{method} = r_{method}\,(t_{fs} + X_{method}\,t_{hop}) + OH_{method} \quad (1)$$

where r_{method} = total number of page faults using the distribution method

t_{fs} = the time required by the faulting node and the home node of the faulted-for page to service the fault

X_{method} = the average round-trip distance in hops from the faulting node to the home node

t_{hop} = average internode transmit time of a message

OH_{method} = overhead time required to implement the distribution method

The first method Clancey and Francioni propose for improving performance is derived from this equation. Since many programs exhibit a consistent pattern of memory accesses even with different data sets, this behavior could be used to attempt to reduce the number of page faults. If a certain page is often faulted for by a program, it generates a lot of network messages, and therefore delays, which would not occur if the page had been in the home set initially. If a page were initially loaded on each node, it would never generate a page fault. The overhead of transferring that page would have only been incurred once at the start of execution. Initially loading a faulted for page reduces the total number of page faults by some factor, r_{hs}. Because each node is running the same program, similar behavior is expected on each computer, and the page can be duplicated on every node so that a page fault is never generated in the system.

However, if a page is added to the home set of a node, then the corresponding copy set must be reduced by one page since there is a constant amount of available memory. Reducing the size of the copy set, though, can generate new page faults due to new page replacements that are required. The number of these new page faults, r_{cs}, could outweigh the benefit of the page duplication by causing page thrashing in the copy set.

The performance gained by this technique is dependent on how many more page faults are eliminated by the duplication of a page in the home set than are generated by the reduction in the copy set size. If duplicating a page creates more page faults than it removes, performance will suffer, and even if the number of page faults is very close there would be a slight performance loss due to the added overhead of loading the page on every node. The number of pages that can be duplicated in this manner before this performance loss occurs varies between applications.

The second technique that Clancey and Francioni present has its basis in the distributed computing idea of load balancing. When a node generates a page fault, not only is its own program delayed, but it forces a delay in the execution of the node that has to service the request. Load balancing suggest that ideally this delay should be distributed equally over all of the nodes so that no single process is delayed extraordinarily by servicing requests, and thus resulting in a slower execution of the entire system.

The cost associated with this technique is in a one time analysis of determining if there is a consistent behavior in the pages that are requested independent of the data set. Determining this can be done by running the program multiple times with different types of data sets and seeing if a pattern of pages faulted for occurs. If no pattern can be determined than this technique can not be used; however, if there is a tendency for the same pages to be faulted for more often than others, this method can be applied.

To achieve the best performance, the number of page fault requests serviced by each machine in the network should be about equal. An arbitrary distribution of pages across the network is unlikely to result in the optimal selection. Pages should be distributed so that the pages which caused the most faults are all on different nodes. This distribution will attempt to minimize the longest amount of time that any one node spends servicing page faults rather than executing its own process.

Clancey and Francioni implemented these two techniques separately and recorded performance improvement ranging from 2% to 80% by duplicating pages and from 2% to 6.1% by reorganizing the distribution of pages across the network.

4. Problems Resulting from MIMD Applications

While the techniques proposed by Clancey and Francioni work for a SPMD application, in the more general and flexible MIMD environment, there are some important considerations that their methods do not address. Most of these considerations result from the fact that in a MIMD environment, each node could be executing a different program with a completely different behavior than other programs. In a SPMD application, it is assumed that most of the nodes will follow a similar faulting behavior. In MIMD environments however, each node has a different program with different memory requirements and faulting pattern. Therefore the duplication of pages on each node could result in data that is never used being stored on machines at the expense of more useful information.

Another major difference between SPMD and MIMD environments is the type of information that the shared memory stores. In a SPMD environment, the program instructions as well as some data will be in the shared memory, since all nodes must use the same program. Most data is kept locally on each machine, and only critical data or synchronization flags are usually stored in the shared memory.

In a MIMD environment though, each node is running a different program. The instructions therefore wouldn't need to be in the shared memory since no other node is likely to use them. However, unless each node has some form of local permanent storage, for instance a local hard drive or floppy disk, the program instructions will need to be read through the network, probably from a file server. In a MIMD system, it is primarily the data which is shared by all of the nodes that is stored in the virtual memory. Since different types of information (instructions and data) are stored in different locations (file server and shared memory) a MIMD system has to consider different types of page faults.

The different types of page faults result in different delays on the system. In SPMD, all page faults result only in the access of memory on a remote machine. In MIMD, page faults looking for data will still only access remote memory, but when there is a page fault resulting from needing program instructions, there is the additional i/o delay of reading from storage besides the regular communication delay to a remote machine. Therefore when considering how to reduce delays, rather than just reducing page faults, the types of page faults being generated and eliminated need to be examined.

Another major consideration for a MIMD environment is the data coherency problem. In SPMD, most of the shared memory is program code, which is not changed. Most data is stored locally, with only important data that is needed by all of the machines in the shared memory. Since most of the shared information is instructions, there are probably few writes to the shared memory, reducing the importance of an "efficient" coherence protocol. With very few changes to memory, a simple invalidate scheme might be sufficient.

In MIMD environments, the shared memory is made up almost entirely of data. Depending on the application, there might be a significant number of writes to the shared memory. For that reason, an inefficient coherence protocol would drastically hurt performance as a large number of unnecessary messages and page-faults might be generated. As an application becomes more write intensive, the importance of the coherency protocol increases.

The need for different pages by each node, the different delays caused by the two different types of page faults, and maintaining the consistency of data across a changing number of machines are the three main considerations in reducing the page-fault delays in a MIMD system that are not as crucial in a SPMD environment.

5. MIMD System Model

Before examining how techniques similar to those proposed for SPMD could be used in a MIMD environment, this author makes some assumptions about the machines on the network, the properties of the communication network, the memory handling of the machines and the shared memory system implementation:

1. For this analysis, a system of N identical computers which are connected by a single high-speed backbone along with a single file server is considered [Figure 1].

2. Each of these N computers has a finite amount of physical memory which is large enough to allow for efficient execution of most applications using paging of memory. In other words, most applications can not fit entirely in memory, and will generate some number of page-fault requests. Each user workstation is assumed to have only one user process at a time.

3. None of the user workstations have any form of permanent storage. All stored files reside on the network file server.

4. The file server is not a user workstation, and has no application processes on it. The file server's only purpose is to service requests for stored data from a central location.

The communication behavior of the network needs to be considered next.

1. All inter-process communication occurs on a single communication backbone via message passing. There is no physical shared memory, or alternative method of data transmission besides the backbone.

2. The communication backbone allows one message at a time to be transmitted. Requests for control of the backbone are executed sequentially in the order the requests were made to either the high priority queue or the low priority queue, which are maintained by the hardware controller of the backbone. Page requests are queued at a low priority, while responses to page requests are queued at a higher priority. This priority scheme assures that data will be made available to a requestor as soon as possible, allowing execution of those processes which have been delayed the longest to resume as soon as possible.

3. The backbone supports a multi-casting protocol so that a single message can be accessed off the network by some, or all, of the workstations in the network.

4. Each workstation is assumed to have network hardware which examines the messages on the network and determines if a message is for that workstation without slowing down the execution of the main CPU. Only if the message is directed for that workstation is the processing of the CPU interrupted while the message is handled and acknowledged.

5. The backbone is considered efficient -- there is no significant delay in message transmission once control has been granted, and the distance between workstations on the network has negligible effect on the time of transmission.

6. The communication network is assumed to be robust. No messages are lost or corrupted during transmission although the network recognizes such problems if they occur. Re-transmissions are not required often enough to add significantly to the network delay.

The memory behavior of the local machines and the shared-memory system, also needs to be considered.

1. Each workstation has its own physical memory which is handled in pages of a definite, unchanging size. Any portion of this memory may or may not be allocated into the shared memory of the entire system.

2. The shared memory system allows each workstation to contribute a different amount of memory to the system, depending on that workstations needs. This allows better optimization of the entire system.

3. There is some mechanism on each workstation that allows memory pages to be indicated as permanent -- such pages will never be considered for removal during a page replacement operation and will always be resident for the duration of the application.

4. The shared memory system provides each workstation with a permanent table which for each page of shared memory lists all of the nodes in the network that the page permanently resides on.

5. The page replacement policy of the entire system is a least recently used (LRU) algorithm.

One assumption is made about the applications -- that each page of the shared memory is used on at least two different workstations within the time of the application. This prohibits the performance degradation of adding shared memory overhead to data that should have been kept in local memory on the only machine that needed it.

With these basics of a network and system, analysis of how to apply the SPMD ideas to a MIMD system is possible.

6. MIMD Solution without Consistency Concerns

To simplify the analysis, first consider a situation where data consistency problems never occur. Such a situation could occur, for example, in an application where all the data is initially generated and updated by one process. Only after all updates are completed is the data then used by multiple workstations in a read-only situation. Since no data is shared until all changes to the shared memory are complete, there is no danger of using outdated information, and since there is only one process making updates, there is no race condition or synchronization problem of assuring the proper ordering of updates.

One example of such an application system is in the area of image processing and pattern recognition. In pattern recognition, images must often be filtered to examine only the relevant data, segmented into regions where the desired patterns might exist, and finally each region needs to be examined in detail. If an image is loaded into the shared memory of the system, one workstation might be responsible for performing all of the filtering, and updating of the image. This same workstation would then examine the image for important regions and isolate them -- a process known as image segmentation. Only after these regions, which might overlap [see Figure 1], have been identified would other workstations begin the task of attempting to recognize some feature within the region. In fact, any application where a large amount of data is processed sequentially, and the results are examined in individual discrete parts or as a whole by different techniques that do not change the data, would qualify as this type of system.

The techniques being applied are attempting to minimize the average delay to the system. On any node there are two types of delays that will be encountered -- the

total delay that occurs from waiting on page faults, T_{wait} and the total delay that arises from servicing page faults from other nodes, T_{ser}.

So, the total delay can be expressed as:

$$Cost = T_{wait} + T_{ser} \quad (2)$$

The delay incurred in servicing a single request is the time it takes for a node to recognize a request, create the reply and send the reply to the requesting node. This is the same delay as is encountered by the requesting node except for the amount of time it takes to generate the initial request message and transmit it over the network. Since the entire delay for the request can be measured, and has been in Table 2 [Remote Memory Access Time], and the time for creating and sending a short request message has also been identified in Table 1 [Data Transmission Time], the delay of servicing a single request is the difference in time between the delay in the requesting node, and the amount of time it takes the servicing node to receive the request. The total delay to the servicing node can be expressed as:

$$T_{ser} = N_{req} * (T_{mem} - T_{req}) \quad (3)$$

where N_{req} is the number of requests serviced by a node, T_{mem} is the delay incurred by a node faulting for a page of shared memory (generating and sending a request, as well as waiting for the reply to be created and transmitted), and T_{req} is the time required to generate and send a short request message across the network.

Since there are two types of page faults, the total delay that these faults generate is dependent not only on the number of faults, but on the type of each fault. Each fault for program instructions will incur a delay to access the file server, while each fault for shared memory will incur a delay to access remote memory. Both of these times can be measured [Table 2]. The total delay from page faults can be written as:

$$T_{wait} = (F_p * T_{file}) + (F_m * T_{mem}) \quad (4)$$

where F_p is the number of page faults for program instructions, T_{file} is the delay of requesting and reading one page from the file server and F_m is the number of page faults for shared memory that occurred.

In considering the initial distribution of pages, two requirements must be met. First, all nodes are loaded with the first page of program instructions -- it is assumed that this page will include the actual starting instruction for the program. Second, every page of the shared memory must belong in the "home set" of at least one workstation.

In order to determine which pages of the shared memory should be placed on which workstation, it is necessary to know how many times each workstation faults for each page. This requires multiple executions of the entire application with different pages being assigned to each workstation every time. Every workstation must have been missing each page during at least one of these executions so it can be determined roughly how many times it will be faulted for -- this is only an approximation because page faults might change based on which pages were included in the home set.

In order to meet the requirement of having each page of shared memory on at least one workstation, a page is initially assigned to the home set of the node that faulted for it the most. This will eliminate the most number of page faults possible, as well as reducing the number of service requests that will be required of the home node. Since each node has different behavior, and will be responsible for varying amounts of page faults and requests serviced, each node will have a different cost. This delay can be expressed as:

$$Cost_i = (F_{p,i} * T_{file}) + (F_{m,i} * T_{mem}) + (N_{req,i} * (T_{mem} - T_{req})) \quad (5)$$

for each node i in the network.

When considering the effect of loading an additional page on a workstation, the effect of adding another program page is different from adding a page of shared memory.

On a given node i, if the most faulted for program page is loaded into the permanent memory, the total number of faults for program pages, $F_{p,i}$, will be reduced by a factor of $F_{p,elim}$, the number of times that program page was faulted for. As in the SPMD system, adding an additional page to the permanent set will reduce the available copy set though, generating new page faults. These new page faults will be some mix of faults for program pages generated by adding a program page to the home set, FP_p, and faults for shared data pages generated by duplicating a program page, FP_m.

The delays eliminated and generated can be expressed by the formulas:

$$T_{elim} = F_{p,elim} * T_{file} \quad (6)$$

$$T_{gen} = (FP_p * T_{file}) + (FP_m * T_{mem}) \quad (7)$$

Since both T_{file} and T_{mem} are measurable, it is possible to define a ratio R such that:

$$R = T_{file} / T_{mem} \quad (8)$$

Since this algorithm is trying to reduce the delay in the system, a new program page should only be loaded into the home set if the delay eliminated is greater than the delay generated by reducing the copy set. By combining this relationship with equations 6, 7 and 8, it can be determined that the program page in question should only be added to the home set if the following condition is met:

$$F_{p,elim} > FP_p + (FP_m / R) \quad (9)$$

On the other hand, if the most faulted for shared memory page was added to the home set of node i, the number of memory page faults $F_{m,i}$ will be reduced by the number of faults that page had generated, $F_{m,elim}$. This reduces the total delay by:

$$T_{elim} = F_{m,elim} * T_{mem} \quad (10)$$

As in the program page case, a new delay will also be introduced because of page faults generated by the reduced copy set where FM_p is the number of additional program page faults and FM_m is the number of additional shared memory page faults. This delay is of the same form as equation 7.

Following the same logic as in the previous example, a shared data page should only be duplicated if the delays eliminated are greater than the delays created by the smaller copy set. Using equations 7, 8 and 10, this relationship can be expressed as:

$$F_{m,elim} > (FM_p * R) + FM_m \quad (11)$$

So in order to duplicate a page of shared memory, it needs to not only eliminate more faults than it creates, it has to eliminate enough faults to counter the added delay of accessing the file server instead of shared memory.

Following these two relationships (equations 9 and 11), pages should be added to a workstation one at a time until these relationships fail. At that point, the maximum number of page faults has been eliminated, and the final total delay of a node can be expressed by combining equations 5, 6, 7, and 10:

$$Cost = T_{file} * (F_p - F_{p,elim} + FP_p + FM_p) + T_{mem} * (F_m - F_{m,elim} + FP_m + FM_m) + T_{ser}$$

$$Cost = T_{mem} * (R * (F_p - F_{p,elim} + F'_p) + F_{m,i} - F_{m,elim} + F'_m) + T_{ser} \quad (12)$$

$$\text{where } F'_p = FP_p + FM_p \text{ and } F'_m = FP_m + FM_m \quad (13,14)$$

As long as the relationships 9 and 11 were followed, it can be demonstrated that this process will reduce the total delays due to page faults. By substitution in equation 12 the following relationship can be derived:

$$F_{p,elim} > FP_p + (FP_m / R) \quad (9)$$
$$FP_p < F_{p,elim} - (FP_m / R)$$

$$Cost = T_{mem} * (R * (F_p - F_{p,elim} + F'_p) + F_{m,i} - F_{m,elim} + F'_m) + T_{ser} \quad (12)$$

by substitution

$$Cost < T_{mem} * (R * (F_p - F_{p,elim} + F_{p,elim} - (FP_m/R) + FM_p) + F_m - F_{m,elim} + F'_m) + T_{ser}$$

$$Cost < T_{mem} * (R * (F_p - (FP_m/R) + FM_p) + F_m - F_{m,elim} + F'_m) + T_{ser}$$

$$Cost < T_{mem} * ((R * F_p) - FP_m + (R * FM_p) + F_m - F_{m,elim} + F'_m) + T_{ser}$$

$$Cost < T_{mem} * ((R * F_p) - FP_m + (R * FM_p) + F_m - F_{m,elim} + FP_m + FM_m) + T_{ser}$$

$$Cost < T_{mem} * ((R * F_p) + (R * FM_p) + F_m - F_{m,elim} + FM_m) + T_{ser}$$

$$F_{m,elim} > (R * FM_p) + FM_m \quad (11)$$

$$FM_m < F_{m,elim} - (R * FM_p)$$

by substitution

$$Cost < T_{mem} * ((R * F_p) + (R * FM_p) + F_m - F_{m,elim} + F_{m,elim} - (R * FM_p)) + T_{ser}$$

$$\sim Cost < T_{mem} * (R * F_p + F_m) + T_{ser} \quad (15)$$

Since the initial cost was of the form $T_{mem} * (R * F_p + F_m) + T_{ser}$, the total delay has been reduced as long as the relationships above are followed.

So far, only the delays for page fault requests have been considered. By duplicating shared memory pages, the delay do to servicing page faults from other workstations might also be affected since a node besides the original node might now be able to handle the request. There are several possible schemes for determining which node should handle a page request if more than one workstation has that page in its home set, including nearest neighbor, random selection, queues of servicing nodes and pre-defined priority.

In a nearest neighbor scheme, the node nearest the requesting node would service the request. In this backbone network however, there is no significant

difference in the communication time between any two workstations, so there is only a negligible benefit of choosing one workstation over the other. The only benefit would be that all of the requests could be split amongst the duplicating workstations reducing the load on each machine. However this technique does not consider the current load already on a workstation, so it might be sending requests to an already overloaded node.

In random selection, the requesting node would randomly choose amongst the nodes that own the desired page [as is stored in the table provided by the shared memory system] and request the page from that node. This has the same benefit that nearest neighbor does in that it provides for splitting the load of servicing requests over multiple workstations. However, it suffers from the same drawback in that nodes that are already very busy might be sent more work by this technique. The benefit of this technique is can not be expressed algebraically since it is not possible to predict where work will actually be sent.

In a scheme based on keeping queues of servicing nodes, the theory is that a list of all the nodes with each page is kept either locally or centrally. Each time a fault for a given page occurs, the first node on the list services the request, and is then placed at the end of the list. This would provide the most even distribution of requests for each page amongst nodes that have a given page. Unfortunately, it is difficult to keep the global list without forcing each request to check a table stored at one central location, creating a communication bottleneck as workstations need to request where the page is before they can actually request the page -- resulting in two request-reply pairs per page-fault.

If local queues were kept on each node and only its own faults were circled on that list, there is the danger that all of the requesting nodes will make requests to

workstation P at the same time, and then they will all make requests to workstation Q at the same time and repeat this same pattern resulting in temporary bottlenecks at each of the servicing workstations, and increasing the delays as each requesting node has to wait for all the replies to be sent.

All three of these techniques have one other fault. They are looking at distributing the work for each page across all of the available nodes. However if one page is faulted for very often, then even splitting this work could result in significant delays since the servicing node might have other shared memory pages which it is also getting a portion of the requests to service. The final technique of pre-defined priority attempts to address this problem.

The theory behind the pre-defined priority scheme is very different from the basis of the three other schemes. Where the other techniques attempt to equally divide the load of each page across all the nodes that have it, pre-defined priority looks at all of the workstations and tries to equalize the total delay in each node. This technique requires that a measure of the total delay for each node be maintained. Since T_{mem} and T_{file} are measurable and the final number of page faults after page duplication can be determined, the value of T_{wait} for each node can be calculated.

T_{ser} is the variable that can be changed in an attempt to equalize the total delay, *Cost*, over the system. Rather than trying to split all of the requests for the same page across the network, pre-defined priority assigns pages to specific nodes and that node then serves all requests for that page. By determining the average number of times a page is faulted for through multiple executions of the application, a pattern might be found that will allow workstations with a lower delay service more requests than the workstations that are already heavily delayed.

In order to accomplish this, a record is kept of the "total current delay" that has been assigned to a workstation already. Initially this is the delay caused by the page faults that node generates. To this factor, the delay of servicing requests that have been assigned to the node will be added. First, if any page of shared memory exists only on one node, that node is assigned to service all requests for that page, and this delay is included into the "total current delay" record. After all of the single copy pages have been assigned, the remaining pages are assigned in decreasing order of page faults that have to be serviced. Performing the technique in this order allows the largest delay encountered to be assigned as early as possible so that the delay on that node can be equalized by assigning later pages to other nodes. In order, each page is assigned to the node owning that page which has the lowest total current delay. The delay for these requests is then added to the workstation's total current delay, making it less likely to accept another page. This algorithm is performed until each page of the shared memory has been assigned to a workstation.

Unfortunately, since there is no control of the initial page distribution, this technique could still result in an imbalance in the average delay of the system -- some node could finish with a total current delay much higher, or much lower than most of the network. This imbalance can be fine tuned if necessary to try and equalize the performance.

If a node is delayed much less than most of the system, the last page of shared memory that had been duplicated can be removed from the home set of that node. A more faulted for page can then be placed into the shared memory. This node can then be assigned to service requests for the new page. This removes a heavy delay from another node and moves it to this less delayed node. In addition to the new service requests, more page faults will be generated as it will need to request the page that had

been removed from the home set. While this will result in a few more requests to the node that will have to service requests, but the less delayed workstation will be delayed more and the node that had been servicing the requests will be reduced in an attempt to better distribute the load of the delay across the network.

If a node is heavily delayed, and there is no node that is less delayed than all of the other workstations, there are still some techniques which might give some benefit. If the node is delayed because it is forced to wait often, and is still required to service requests because it has the only copy of a page of shared memory, these pages can be duplicated on other nodes, thus eliminating the delays on the loaded workstation even as it will force greater delays on the network. If the node is delayed mainly because of servicing requests for particular pages, these pages can also be split across other nodes.

This technique will sometimes fail however since there will be applications when the pattern of page faults is so random that a different page is faulted for many times during each execution. If that happens, an already busy node might become even more seriously delayed. This is a problem, however, that can not be solved with a static priority scheme.

7. MIMD Solution with Consistency Concerns

With a technique usable for a simple read-only application, it becomes easier to examine a more complex system where writing to the shared data pages is not limited, giving rise to concerns about maintaining data consistency. Maintaining data consistency is very important in any form of shared memory and has been the focus of research of many people [1, 2, 5-12]. In many cases, this research has focused on physical shared memory multiprocessor systems, but in a virtual shared memory the efficiency of the coherency protocol becomes more vital because the delays inherent in the required communication is greater.

Since the methods given in section 6 only consider the reads and page faults to the system, the added consideration of maintaining consistency does not change these procedures, but merely adds another delay into the total delay of the system. Therefore, it is not necessary to consider whether page faults are for reads or writes when attempting to reduce page faults as they will be treated in the same manner.

There are a number of common methods of guaranteeing data consistency within a system. This paper will consider four of them: ownership migration, invalidation, pessimistic write-through, and optimistic write-through.

In the usual implementation of ownership migration [10-12], the system keeps only one copy of the shared memory as allowing writes to the data. The workstation that owns the write-enabled page of memory also keeps track of every node that has a copy of that page. When a node that does not own the write-enabled page needs to write to that page, the workstation sends a message to the owner of the page and requests ownership. If the owner is not writing to the page, it will transfer the write-enable permission to the requesting page along with the list of all copies of that page. If

the owner is writing, the requesting node must wait until the owner is ready to transfer the page.

There are problems with this solution though, relating to the fact that ownership is associated with pages. If two nodes are attempting a series of writes to discrete sets of memory locations that happen to reside on the same page, either one process will be forced to wait for the entire series of writes by the other node to be completed before being granted control, or ownership will thrash back and forth between the two processes resulting in increased traffic on the communication network.

The second technique that is available is a combination invalidation and update scheme [1]. As already discussed, an invalidation scheme is very inefficient because of the number of unnecessary page faults it generates. Assume that the network has a page length of 1 Kb, and a word length of 4 bytes, there are 256 words on a page. Under this combination scheme, a write to one word will require the update of all other nodes that have that page in their home set, and the invalidation of that page in copy sets of nodes. This invalidation will not only prevent the use of the outdated data, it will prevent the use of 255 words of data that were still valid. Any access to those 255 words will now generate an additional page fault, even though the data the node had was correct. Since the pages that have been invalidated would likely be used again, the additional page faults will degrade the performance of the system.

The method of pessimistic write-through updates works on the premise that any write is likely to create consistency problems, so no write to the system is allowed until every other node in the network says it is all right to write, enforcing a strict definition of coherence [1, 10, 11]. Any node that desires to write to the shared memory broadcasts a message to all other nodes requesting permission for the write, and then waits until it receives responses from all nodes. Each of the other nodes receives the

request and determines if the write can proceed. If the write does not interfere with the node's own process, it replies to the requesting node that it may proceed, and continues its own execution. If the write might cause conflicts because the node is using that address, the non-requesting node needs to pause its own execution and allows the writing node to proceed. After the requesting node receives replies from all other nodes, it broadcasts the update, and then all processes are allowed to resume their execution.

While this technique guarantees data consistency it is extremely inefficient. For a single write, N+1 messages are required -- 1 request, N-1 replies, and 1 write update. This is not only an excessive delay to the writing node, but it also ties up the communication network, prohibiting other messages from being sent.

The final method is an optimistic write-through update scheme [6, 10-12]. In this technique, when a node wishes to write to shared memory, it broadcasts the update to all other nodes [some implementations may delay this broadcast for better use of the communication network]. Each workstation will read the message from the network and will update the page accordingly if the workstation has it in either its home set or its copy set.

The problem inherent in this technique is that due to the communication delay, a workstation might have already used the outdated value by the time it learns of the change. To deal with this, each workstation maintains a small log in memory of the instructions it has performed. When it receives a write, it examines this log to determine if it has used the outdated value. If not, execution can continue as normal. If the invalid data had already been used, the workstation must "rollback" and undo the instructions that it performed with the invalid data and all instructions since that use. A complication arises though in that this rollback might require the undoing of a write that

had been performed to the shared memory. This can result in the situation known in distributed computing as cascading rollback. One write request can result in the necessity to rollback other processes, which can force other rollbacks, which can force still other rollbacks.

This technique however takes the optimistic approach that rollbacks will rarely occur since a node is only forced to undo instructions if it had read from the same address being updated -- not just from the same page. Since each workstation is running a different program, there is only a small chance that any two workstations will be accessing the same memory address in the small amount of time that it takes for the transmission of the update over the computer network. Even if such a conflict occurs and a rollback is required, a second rollback is only required if a write to the shared memory needs to be undone. Unless the application being considered is write intensive to the shared memory, this situation becomes less likely, and with each rollback that is required, the chance that another rollback will be required decreases. So this optimistic approach assumes that this type of conflict will rarely occur, and allows the cascading rollbacks to proceed if they are absolutely required.

The delay involved in a write under this scheme is relatively uniform across the network, assuming that there are very few roll backs. When the non-writing node receives the request, it is delayed for the time it takes to receive the message and perform the update, T_{upd}. The writing node on the other hand needs to delay for the time it takes to send the short update message, T_{req}. If the number of writes on a node i is W_i, and the total number of writes is W, the delay due to the consistency protocol on node i can be expressed as:

$$T_{consistent,i} = W_i * T_{req} + (W - W_i) * T_{upd} \quad (16)$$

Unless there is some way to optimize the program to reduce the number of writes in the system, or an even more optimistic approach is taken for grouping writes before sending the update, there is no method of reducing this delay, so it does not effect the decisions made in the duplication and placement of pages in the shared memory.

8. Conclusions and Future Work

This paper demonstrates that the delays on a distributed virtual shared memory MIMD system can be reduced using techniques very similar to those first proposed by Clancey and Francioni for a SPMD system. Through a pre-production use examination of the memory behavior of an application system, pages of shared memory and local memory can be assigned so as to distribute the expense of the shared memory across the network as evenly as possible.

The reduction in the system delays is based on the number of faults that have been removed as opposed to the page faults that have been generated by the change in the system. By building the final configuration of the system step by step and re-evaluating its performance in terms of several relationships, this method will produce an efficient, if not optimal configuration.

This method requires multiple executions of the system before the configuration is complete. The amount of time that this method requires suggests that it would only be of value for large production systems where the same application is performed many times on different sets of data.

Areas for future consideration would include an actual implementation of this technique to measure the true effect on performance of these techniques; the consideration of a dynamic priority scheme for servicing page-fault requests, so that a node can request to have its own priority either raised if its current load is less than most of the network, or to have its priority lowered if it is being delayed seriously by service requests; and the evaluation of the optimistic write-through coherency scheme to determine how often rollbacks and cascading rollbacks will occur, and if necessary determine solutions to these problems that will not seriously degrade the performance of the entire system.

9. References

[1] Lothar Borrmann and Martin Herdieckerhoff, "A Coherency Model for Virtually Shared Memory", *Proceedings of the 1990 International Conference on Parallel Processing, Vol II,* August 1990, 252-257.

[2] H. Bunjevac, "Adaptive Algorithm For Distributed Shared Memory Management", *Proceedings of the 15th International Conference on Information Technology Interfaces*, June 1993, 245-250.

[3] Patrick M. Clancey and Joan M. Francioni, "Distribution of Pages in a Distributed Virtual Memory", *Proceedings of the 1990 International Conference on Parallel Processing, Vol II,* August 1990, 258-263.

[4] Karim Harzallah and Kenneth C. Sevcik, "Hot Spot Analysis in Large Scale Shared Memory Multiprocessors", *Proceedings of Supercomputing '93 Conference*, November 1993, 895-905.

[5] T.-S. Jou and R. Enbody, "A Scalable Snoopy Coherence Scheme on Distributed Shared-Memory Multiprocessors", *Proceedings of Supercomputing '92 Conference*, November 1992, 652-660.

[6] P. Keleher, A. L. Cox and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory", *Computer Architecture News, Vol 20, Issue 2*, May 1992, 13-21.

[7] R. E. Kessler and Miron Livny, "An Analysis of Distributed Shared Memory Algorithms", *Proceedings of the 9th International Conference on Distributed Computing Systems*, May 1989, 498-505.

[8] T. Kolarik, "Cooperative Computing in Loosely-Coupled Distributed Systems", *Proceedings SHARE Europe Spring Meeting, Distributed Applications*, April 1993, 359-362.

[9] M. Mizuno, M. Raynal, G. Singh and M. L. Neilsen, "An Efficient Implementation of Sequentially Consistent Distributed Shared Memories", *IFIP Transactions A, Vol: A-44*, 1994, 145-154.

[10] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", *Computer, Vol 25, No. 8*, August 1991, 52-60.

[11] Umakishore Ramachandran, Mustaque Ahamad and M. Yousef A. Khalidi, "Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer", *Proceedings of the 1989 International Conference on Parallel Processing, Vol II,* August 1989, 160-169.

[12] Michael Stumm and Songnian Zhou, "Algorithms Implementing Distributed Shared Memory", *Computer, Vol. 23, No. 5,* May 1990, 54-64.

[13] Josep Torrellas, Monica S. Lam, and John L. Hennessy, "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates", *Proceedings of the 1990 International Conference on Parallel Processing, Vol II,* August 1990, 266-270.

[14] Andrew W. Wilson Jr., Richard P. LaRowe Jr., Marc J. Teller, "Hardware Assist for Distributed Shared Memory", *Proceedings of the 13th International Conference on Distributed Computing Systems,* May 1993, 246-255.

Appendix: Tables and Figures

Message Size	Data Transmission Time	Request & Data Transmission Time
1 byte	175 uSeconds	1945 uSeconds
10 bytes	178 uSeconds	1972 uSeconds
100 bytes	215 uSeconds	2051 uSeconds
1 kilobyte	372 uSeconds	3331 uSeconds
2 kilobytes	830 uSeconds	4729 uSeconds

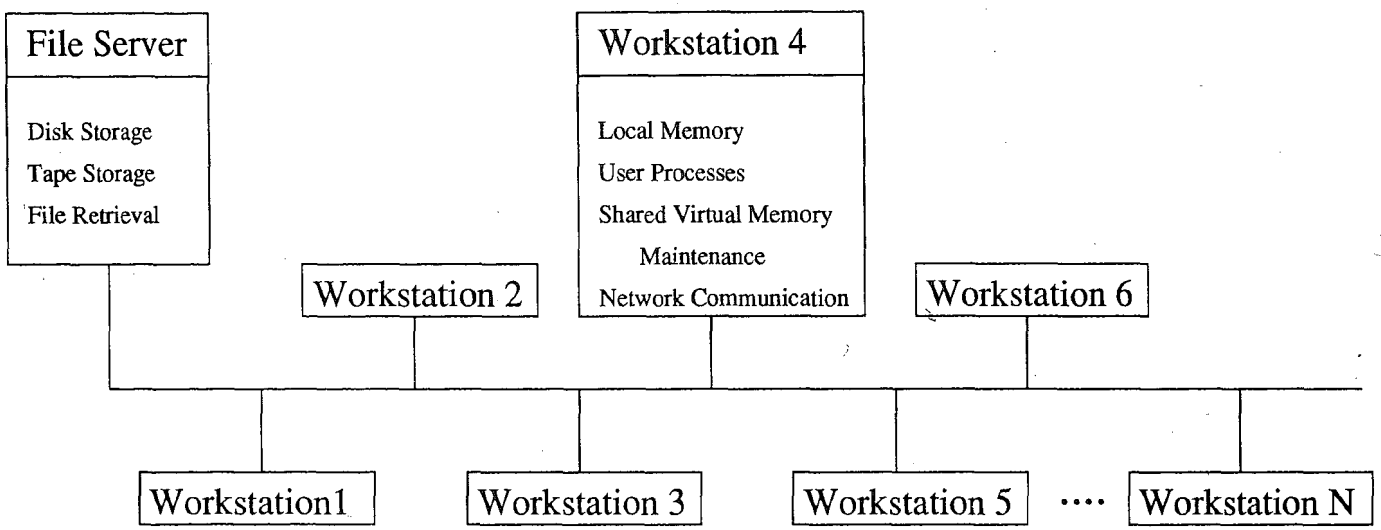
[Times based on 3000 trials on a network of Sun/4 workstations]

Table 1: Communication Delays Caused by Message Transmission

Page Size	File Server Access Time	Remote Memory Access Time	Ratio
100 bytes	5506 uSeconds	2051 uSeconds	2.68
1 Kilobyte	5589 uSeconds	3331 uSeconds	1.68
2 Kilobytes	5725 uSeconds	4729 uSeconds	1.21

[Times Based on 3000 trials on a network of Sun/4 workstations]

Table 2: Comparison of File versus Remote Memory Page Fault Delays



All Workstations have identical resources and responsibilities.

Figure 1. Network Configuration, Resources and Responsibilities

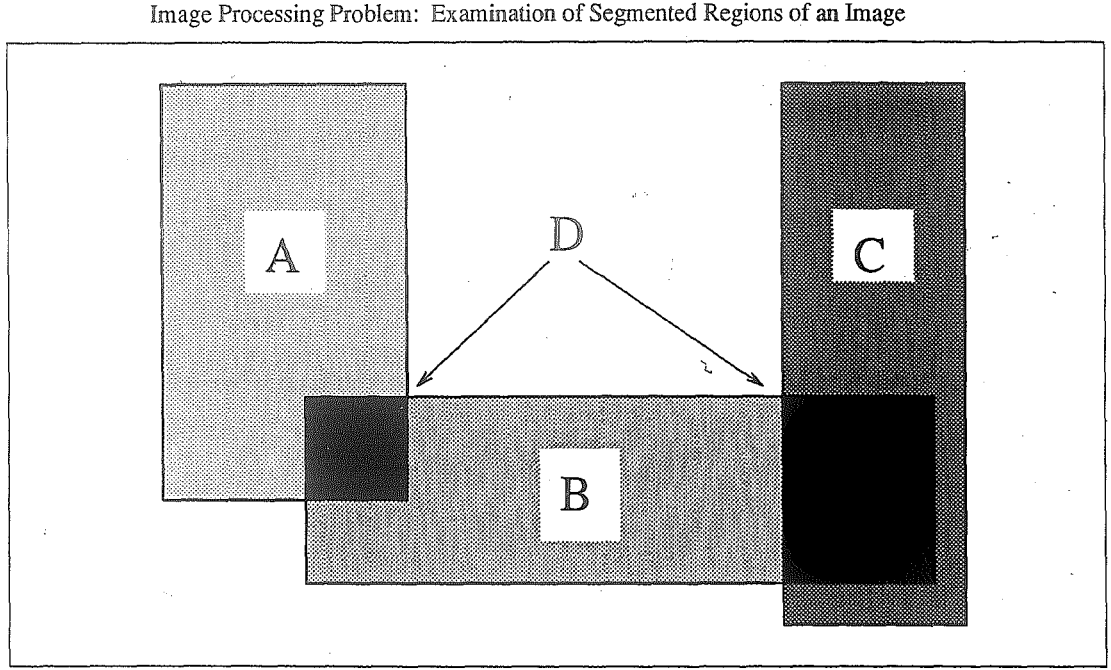


Figure 2: Sample of Multiple Readers of Shared Memory Data

Vita

Thomas Browne was born in Plainfield, New Jersey on January 26, 1967 to Charles and Virginia Browne, and has lived in Allentown, Pennsylvania for most of his life.

He graduated from Lehigh University in 1989 with a Bachelors of Science in computer science and minors in theatre and applied mathematics. He graduated summa cum laude and is a member of Phi Beta Kappa.

He attended the University of Pittsburgh and graduated in 1992 with a Master of Arts in technical theatre.

After completing his Masters degree in computer science, Thomas plans to pursue his PhD at Georgia Institute of Technology.

END OF TITLE