Implementing Global Memory Management in a Workstation Cluster

Michael J. Feeley, William E. Morgan,† Frederic H. Pighin, Anna R. Karlin, Henry M. Levy
Department of Computer Science and Engineering
University of Washington
and
Chandramohan A. Thekkath
DEC Systems Research Center

Abstract

Advances in network and processor technology have greatly changed the communication and computational power of local-area workstation clusters. However, operating systems still treat workstation clusters as a collection of loosely-connected processors, where each workstation acts as an autonomous and independent agent. This operating system structure makes it difficult to exploit the characteristics of current clusters, such as low-latency communication, huge primary memories, and high-speed processors, in order to improve the performance of cluster applications.

This paper describes the design and implementation of global memory management in a workstation cluster. Our objective is to use a single, unified, but distributed memory management algorithm at the lowest level of the operating system. By managing memory globally at this level, all system- and higher-level software, including VM, file systems, transaction systems, and user applications, can benefit from available cluster memory. We have implemented our algorithm in the OSF/1 operating system running on an ATM-connected cluster of DEC Alpha workstations. Our measurements show that on a suite of memory-intensive programs, our system improves performance by a factor of 1.5 to 3.5. We also show that our algorithm has a performance advantage over others that have been proposed in the past.

1 Introduction

This paper examines global memory management in a workstation cluster. By a cluster, we mean a high-speed local-area network with 100 or so high-performance machines operating within a single administrative domain. Our premise is that a single, unified, memory management algorithm can be used at a low-level of the operating system to manage memory cluster-wide. In contrast, each operating system in today's clusters acts as an autonomous agent, exporting services to other nodes, but not acting in a coordinated way. Such autonomy has advantages, but results in an underutilization of resources that could be used to improve performance. For example, global memory management allows the operating system to use cluster-wide memory to avoid many disk accesses; this becomes more important with the widely growing disparity between processor speed and disk speed. We believe that as processor performance increases and communication latency decreases, workstation or personal computer clusters should be managed more as a multicomputer than as a collection of independent machines.

We have defined a global memory management algorithm and implemented it in the OSF/1 operating system, running on a collection of DEC Alpha workstations connected by a DEC AN2 ATM network [1]. By inserting a global memory management algorithm at the lowest OS level, our system integrates, in a natural way, all cluster memory for use by all higher-level functions, including VM paging, mapped files, and file system buffering. Our system can automatically reconfigure to allow machines to join and in most cases, to depart the cluster at any time. In particular, with our algorithm and implementation, no globally-managed data is lost when a cluster node crashes.

Using our system, which we call GMS (for *Global Memory Service*), we have conducted experiments on clusters of up to 20 machines using a suite of real-world application programs. Our results show that the basic costs for global memory management operations are modest and that application performance improvement can be significant. For example, we show a 1.5- to 3.5-fold speedup for a collection of memory-intensive applications running with GMS; these speedups are close to optimal for these applications, given the relative speeds of remote memory and disk.

The paper is organized as follows. Section 2 compares our work to earlier systems. In Section 3 we describe our algorithm for global memory management. Section 4 details our OSF/1 implementation. We present performance measurements of the implementation in Section 5. Section 6 discusses limitations of our algorithm and implementation, and possible solutions to those limitations. Finally, we summarize and conclude in Section 7.

2 Comparison With Previous Work

Several previous studies have examined various ways of using remote memory. Strictly theoretical results related to this problem include [2, 3, 7, 24]. Leach et al. describe remote paging in the context of the Apollo DOMAIN System [15]. Each machine in

† Author's current address: DECwest Engineering, Bellevue, WA.
This work was supported in part by the National Science Foundation (Grants no. CDA-9123308, CCR-9200832, and GER-9450075), ARPA Carnegie Mellon University Subcontract #381375-50196, the Washington Technology Center, and Digital Equipment Corporation. M. Feeley was supported in part by a fellowship from Intel Corporation. W. Morgan was supported in part by Digital Equipment Corporation.

the network has a paging server that accepts paging requests from remote nodes. This system allowed local users to statically restrict the amount of physical memory available to the paging server.

Comer and Griffioen described a *remote memory model* in which the cluster contains workstations, disk servers, and *remote memory servers* [8]. The remote memory servers were dedicated machines whose large primary memories could be allocated by workstations with heavy paging activity. No client-to-client resource sharing occurred, except through the servers. Felten and Zahorjan generalized this idea to use memory on idle client machines as paging backing store [12]. When a machine becomes idle, its kernel activates an otherwise dormant memory server, which registers itself for remote use. Whenever a kernel replaces a VM page, it queries a central registry to locate active memory servers, picking one at random to receive the replacement victim. Felten and Zahorjan used a simple queueing model to predict performance.

In a different environment, Schilit and Duchamp have used remote paging to enhance the performance of mobile computers [18]. Their goal is to permit small memory-starved portable computers to page to the memories of larger servers nearby; pages could migrate from server to server as the portables migrate.

Franklin et al. examine the use of remote memory in a client-server DBMS system [13]. Their system assumes a centralized database server that contains the disks for stable store plus a large memory cache. Clients interact with each other via a central server. On a page read request, if the page is not cached in the server's memory, the server checks whether another client has that page cached; if so, the server asks that client to forward its copy to the workstation requesting the read. Franklin et al. evaluate several variants of this algorithm using a synthetic database workload.

Dahlin et al. evaluate the use of several algorithms for utilizing remote memory, the best of which is called N-chance forwarding [10]. Using N-chance forwarding, when a node is about to replace a page, it checks whether that page is the last copy in the cluster (a "singlet"); if so, the node forwards that page to a randomly-picked node, otherwise it discards the page. Each page sent to remote memory has a circulation count, N, and the page is discarded after it has been forwarded to N nodes. When a node receives a remote page, that page is made the *youngest* on its LRU list, possibly displacing another page on that node; if possible, a duplicate page or recirculating page is chosen for replacement. Dahlin et al. compare algorithms using a simulator running one two-day trace of a Sprite workload; their analysis examines file system data pages only (i.e., no VM paging activity and no program executables).

Our work is related to these previous studies, but also differs in significant ways. First, our algorithm is integrated with the lowest level of the system and encompasses all memory activity: VM paging, mapped files, and explicit file access. Second, in previous systems, even where client-to-client sharing occurs, each node acts as an autonomous agent. In contrast, we manage memory globally, attempting to make good choices both for the faulting node and the cluster as a whole (we provide a more detailed comparison of the global vs. autonomous scheme following the presentation of our algorithm in the next section). Third, our system can gracefully handle addition and deletion of nodes in the cluster without user intervention. Finally, we have an implementation that is well integrated into a production operating system: OSF/1.

Several other efforts, while not dealing directly with remote paging, relate to our work. Most fundamental is the work of Li and Hudak, who describe a number of alternative strategies for managing pages in a distributed shared virtual memory system [16]. Similar management issues exist at the software level in single address space systems such as Opal [6], and at the hardware level in NUMA and COMA architectures [9, 21]. Eager et al. [11] describe strategies for choosing target nodes on which to offload tasks in a distributed load sharing environment.

3 Algorithm

This section describes the basic algorithm used by GMS. The description is divided into two parts. First, we present a high-level description of the global replacement algorithm. Second, we describe the probabilistic process by which page information is maintained and exchanged in the cluster.

3.1 The Basic Algorithm

As previously stated, our goal is to globally coordinate memory management. We assume that nodes trust each other but may crash at any time. All nodes run the same algorithm and attempt to make choices that are good in a global cluster sense, as well as for the local node. We classify pages on a node P as being either *local* pages, which have been recently accessed on P, or *global* pages, which are stored in P's memory on behalf of other nodes. Pages may also be *private* or *shared*; shared pages occur because two or more nodes might access a common file exported by a file server. Thus, a shared page may be found in the active local memories of multiple nodes; however, a page in global memory is always private.

In general, the algorithm changes the local/global memory balance as the result of faults caused by an access to a nonresident page. Node P, on a fault, performs the following global replacement algorithm, which we describe in terms of 4 possible cases:

Case 1: The faulted page is in the global memory of another node, Q. We swap the desired page in Q's global memory with any global page in P's global memory. Once brought into P's memory, the faulted page becomes a local page, increasing the size of P's local memory by 1. Q's local/global memory balance is unchanged. This is depicted in Figure 1.

Case 2: The faulted page is in the global memory of node Q, but P's memory contains only local pages. Exchange the LRU local page on P with the faulted page on Q. The size of the global memory on Q and the local memory on P are unchanged.

Case 3: The page is on disk. Read the faulted page into node P's memory, where it becomes a local page. Choose the oldest page *in the cluster* (say, on node Q) for replacement and write it to disk if necessary. Send a global page on node P to node Q where it continues as a global page. If P has no global pages, choose P's LRU local page instead. This is shown in Figure 2.

Case 4: The faulted page is a shared page in the *local* memory of another node Q. Copy that page into a frame on node P, leaving the original in local memory on Q. Choose the oldest page *in the cluster* (say, on node R) for replacement and write it to disk if necessary. Send a global page on node P to node R where it becomes a global page (if P has no global pages, choose P's LRU local page).

The behavior of this algorithm is fairly straightforward. Over time, nodes that are actively computing and using memory will

Figure 2: Global replacement showing miss in the global cache. The faulted page is read from disk, and the oldest page in the network is either discarded (if clean) or written back to disk.

fill their memories with local pages and will begin using remote memory in the cluster; nodes that have been idle for some time and whose pages are old will begin to fill their memories with global pages. The balance between local and global storage on a node is thus dynamic and depends on its workload and the workload in the cluster. The basic issue is when to change the amount of global store and local storage, both on a node and in the cluster overall. In general, on a fault requiring a disk read, the (active) faulting node grows its local memory, while the cluster node with the oldest page (an "idle" node) loses a page to disk. Global memory grows when the faulting node has no global pages and the oldest page in the network is a local page (i.e., the oldest local page on the faulting node becomes a global page, replacing the oldest cluster page.)

Ultimately, our goal is to minimize the total cost of all memory references within the cluster. The cost of a memory reference depends on the state of the referenced page: in local memory, in global memory on another node, or on disk. A local hit is over three orders of magnitude faster than a global memory or disk access, while a global memory hit is only two to ten times faster than a disk access. Therefore, in making replacement decisions, we might choose to replace a global page before a local page of the same age, because the cost of mistakenly replacing a local page is substantially higher. Which decision is better depends on future behavior. To predict future behavior, a cost function is associated with each page. This cost function is related to LRU, but is based on both the age of the page and its state. Our current implementation boosts the ages of global pages to favor their replacement over local pages of approximately the same age.

3.2 Managing Global Age Information

When a faulted page is read from disk (Cases 3 and 4), our algorithm discards the oldest page in the cluster. As described so far, we assume full global information about the state of nodes and their pages in order to locate this oldest page. However, it is obviously impossible to maintain complete global age information at every instant; therefore, we use a variant in which each node has only approximate information about global pages. The objective of our algorithm is to provide a reasonable tradeoff between the accuracy of information that is available to nodes and the efficiency of distributing that information. The key issue is guaranteeing the validity of the age information and deciding when it must be updated.

Our algorithm divides time into epochs. Each epoch has a maximum duration, T, and a maximum number of cluster replacements, M, that will be allowed in that epoch. The values of T and M vary from epoch to epoch, depending on the state of global memory and the workload. A new epoch is triggered when either (1) the duration of the epoch, T, has elapsed, (2) M global pages have been replaced, or (3) the age information is detected to be inaccurate. Currently, each epoch is on the order of 5–10 seconds.

Our system maintains age information on every node for both local and global pages. At the start of each epoch, every node sends a summary of the ages of its local and global pages to a designated *initiator node*. Using this information, the initiator computes a weight, w_i, for each node i, such that out of the M oldest pages in the network, w_i reside in node i's memory at the beginning of the epoch. The initiator also determines the minimum age, *MinAge*, that will be replaced from the cluster (i.e., sent to disk or discarded) in the new epoch. The initiator sends the weights w_i and the value *MinAge* to all nodes in the cluster. In addition, the initiator selects the node with the most idle pages (the largest w_i) to be the initiator for the following epoch.

During an epoch, when a node P must evict a page from its memory to fault in a page from disk (Cases 3 and 4), it first checks if the age of the evicted page is older than *MinAge*. If so, it simply discards the page (since this page is expected to be discarded sometime during this epoch). If not, P sends the page to node i, where the probability of choosing node i is proportional to w_i. In this case, the page discarded from P becomes a global page on node i, and the oldest page on i is discarded.

Our algorithm is probabilistic: on average, during an epoch, the ith node receives w_i/M of the evictions in that epoch, replacing its oldest page for each one. This yields two useful properties. First, our algorithm approximates LRU in the sense that if M pages are discarded by global replacement during the epoch, they are the globally oldest M pages in the cluster. Second, it yields a simple way to determine statistically when M pages have been replaced; i.e., when the node with the largest w_i receives w_i pages, it declares an end to the epoch.

To reduce the divergence from strict LRU, it is thus important to keep the duration of the epoch T and the value of M appropriate for the current behavior of the system. The decision procedure for choosing these values considers (1) the distribution of global page ages, (2) the expected rate at which pages will be discarded from the cluster, and (3) the rate at which the distributed age information is expected to become inaccurate.† The latter two rates are estimated from their values in preceding epochs. Roughly speaking, the more old pages there are in the network, the longer T should be (and the larger M and *MinAge* are); similarly, if the expected discard rate is low, T can be larger as well. When the number of old pages in the network is too small, indicating that all nodes are actively using their memory, *MinAge* is set to 0, so that pages are always discarded or written to disk rather than forwarded.

† The age distribution on a node changes when its global pages are consumed due to an increase in its local cache size.

3.3 Node Failures and Coherency

Node failures in the cluster do not cause data loss in global memory, because all pages sent to global memory are clean; i.e., a dirty page moves from local to global memory only when it is being written to disk. We do not change the number of disk writes that occur; our system allows a disk write to complete as usual but promotes that page into the global cache so that a subsequent fetch does not require a disk read. If a node housing a requested remote page is down, the requesting node simply fetches the data from disk.

Likewise, since our algorithm deals with only clean pages, coherence semantics for shared pages are the responsibility of the higher-level software that creates sharing in the first place. For instance, in our system shared pages occur when nodes access a common NFS file, e.g., an executable. Thus, the coherence semantics seen by the users of our system are no stronger and no weaker than what NFS provides.

3.4 Discussion

The goal of a global memory management system is to utilize network-wide memory resources in order to minimize the total cost of all memory references. At the same time, it should avoid impacting programs not using global memory. To meet these goals, we believe that the memory management algorithm must use global, rather than local knowledge to choose among the various possible states that a page might assume: local, global, or disk. This knowledge must be efficient to maintain, distribute, and evaluate. In practice, the algorithm must operate without complete information, and must be able to determine when information is out of date.

Our algorithm is intended to meet these needs by using periodically-distributed cluster-wide page age information in order to: (1) choose those nodes most likely to have idle memory to house global pages, (2) avoid burdening nodes that are actively using their memory, (3) ultimately maintain in cluster-wide primary memory the pages most likely to be globally reused, and (4) maintain those pages in the right places.

4 Implementation

We have modified the OSF/1 operating system on the DEC Alpha platform to incorporate the algorithm described above. This section presents the details of our implementation.

Figure 3 shows a simplified representation of the modified OSF/1 memory management subsystem. The boxes represent functional components and the arrows show some of the control relationships. The two key components of the basic OSF/1 memory system are (1) the VM system, which supports *anonymous pages* devoted to process stacks and heaps, and (2) the Unified Buffer Cache (UBC), which caches file pages. The UBC contains pages from both mapped files and files accessed through normal read/write calls and is dynamically-sized; this is similar in some ways to the Sprite file system [17]. At the same level as VM and UBC, we have added the GMS module, which holds global pages housed on the node. Page-replacement decisions are made by the pageout daemon and GMS. A custom TLB handler provides information about the ages of VM and UBC pages for use by GMS.

We modified the kernel to insert calls to the GMS at each point where pages were either added to or removed from the UBC. Similarly, we inserted calls into the VM swapping code to keep track of additions and deletions to the list of anonymous pages.

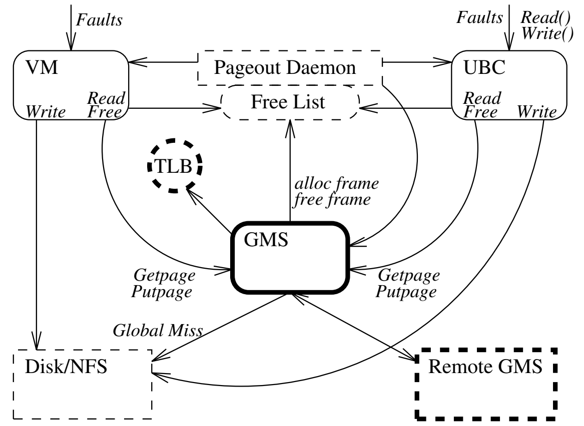

Figure 3: Structure of Modified OSF/1 Memory Management System

Inserting these calls into the UBC and VM modules allows us to track the collection of pages on the node. It also allows us to forward disk reads to the GMS. Disk writes occur exactly as in the original OSF/1 system.

4.1 Basic Data Structures

Our algorithm manages pages in the cluster, which are ultimately backed by secondary storage: either a local disk or an NFS server's disk. Each page of data must be uniquely identified, so that we can track the physical location of that page (or the multiple locations for a shared page). We uniquely identify a page in terms of the file blocks that back it. In OSF/1, pages are a fixed multiple of disk blocks; entire pages, rather than blocks, are transferred between the disk driver and the rest of the kernel. Thus, to identify the contents of a page, it is sufficient to know the IP address of the node backing that page, the disk partition on that node, the inode number, and the offset within the inode where the page resides. We use a 128-bit cluster-wide unique identifier (*UID*) to record this information. We ensure that the page identified by each UID is in one of four states: (1) cached locally on a single node, (2) cached locally on multiple nodes, (3) cached on a single node on behalf of another node, or (4) not cached at all.

We maintain three principal data structures, keyed by UID.

1. The *page-frame-directory* (PFD) is a *per-node* structure that contains a record for each page (local or global) that is present on the node. A successful UID lookup in the PFD yields information about the physical page frame containing the data, LRU statistics about the frame, and whether the page is local or global. An unsuccessful lookup implies that the particular page is not present on this node.

2. The *global-cache-directory* (GCD) is a *cluster-wide* data structure that is used to locate the IP address of a node that has a particular page cached. For performance reasons, the GCD is organized as a hash table, with each node storing only a portion of the table.

3. The *page-ownership-directory* (POD) maps the UID for a shared page to the node storing the GCD section containing that page. For non-shared pages, the GCD entry is always

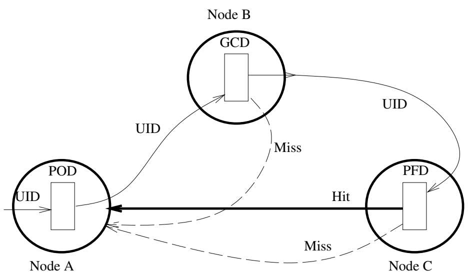

Figure 4: Locating a Page.

stored on the node that is using the page. The POD is replicated on all nodes and is changed only when machines are added or deleted from the cluster.

Finding a page following a fault requires a two-step lookup, as shown in Figure 4. First, the requesting node (e.g., node A in Figure 4) uses the UID to hash into the POD, producing an IP address of the node (node B) implementing the appropriate region of the global-cache-directory. The requester sends a message containing the UID to the GMS on that node, requesting the page identified by the UID. The GMS on the node does a lookup in the global-cache-directory, which results in either a miss or a hit. In case of a miss, a message is sent to the original node. In case of a hit, the GCD node forwards the request to the node that contains the page-frame-directory for the UID (node C), which replies to the requesting node. Figure 4 shows three nodes involved in the lookup. When a faulted page is not shared, nodes A and B are identical, thereby decreasing the number of network messages needed. This distribution of the page management information is somewhat similar to the handling of distributed locks in VAXclusters [14].

The *page-ownership-directory* provides an extra level of indirection that enables us to handle the addition or deletion of nodes from the cluster without changing the hash function. A central server running on one of the workstations is responsible for updating the indirection table and propagating it to the rest of the cluster. This updating must occur whenever there is a reconfiguration. Parts of the global-cache-directory database are redistributed on reconfiguration as well. However, none of these operations are critical to the correct operation of the cluster. In the worst case, during the redistribution process, some requests may fail to find pages in global memory and will be forced to access them on disk.

4.2 Collecting Local Age Information

An important part of our algorithm is its page aging process, which provides information for global decision making. Unfortunately, it is difficult to track the ages of some pages in OSF/1. For pages belonging to files accessed via explicit read/write requests, these calls can provide the required age information. However, access to anonymous and mapped file pages is invisible to the operating system. Furthermore, the OSF/1 FIFO-with-second-chance replacement algorithm provides little useful global age information for our purposes, particularly on a system that is not heavily faulting—exactly the case in which we wish to know the ages of pages.

In order to collect age information about anonymous and mapped pages, we modified the TLB handler, which is implemented in PALcode [20]. Once a minute, we flush the TLB of all entries. Subsequently when the TLB handler performs a virtual-to-physical translation on a TLB miss, it sets a bit for that physical frame. A kernel thread samples the per-frame bit every period in order to maintain LRU statistics for all physical page frames.

4.3 Inter-node Communication

Between nodes we use simple non-blocking communication. In our current implementation, we assume that the network is reliable and we marshal and unmarshal to and from IP datagrams directly. This is justified primarily by the increased reliability of modern local area networks such as AN2 that have flow control to eliminate cell loss due to congestion [23]. To date, we have not noticed a dropped packet in any of our experiments. However, our implementation can be readily changed to use any other message-passing package or transport protocol.

4.4 Addition and Deletion of Nodes

When a node is added to the cluster, it checks in with a designated master node, which then notifies all the existing members of the new addition. The master node distributes new copies of the page-ownership-directory to each node, including the newly added one. Each node distributes the appropriate portions of the global-cache-directory to the new node. In the current implementation, the master node is pre-determined and represents a single point of failure that can prevent addition of new nodes. It is straightforward to extend our implementation to deal with master node failure through an election process to select a new master, as is done in other systems [14, 18].

Node deletions are straightforward as well. The master node checks periodically for the liveness of the other nodes. When it detects a crashed node, it redistributes the page-ownership-directory. As with addition, global-cache-directories are also redistributed at that time.

4.5 Basic Operation of the Algorithm

Section 3 described the basic operation of the algorithm in terms of a series of page swaps between nodes (see Cases 1–4). However, in the actual implementation, the swaps are performed in a somewhat different fashion.

On a fault, the faulting node allocates a frame from its free-frame list and executes a remote *getpage* operation for the missing page. The inter-node interaction for a getpage operation is conceptually identical to the scenario shown in Figure 4: on a hit, the actual data is returned; on a miss, the requesting node goes to disk (or to the NFS server). The global-cache-directory is updated with the new location of the page and the page-frame-directory structures on the requester and the remote node are updated suitably. If the remote node was caching the page on behalf of another node, i.e., it was a global page, then the remote node deletes its entry from the page-frame-directory. (Recall from Section 3 that there needs to be only one copy of a global page in the system, because the cost of remotely accessing a global page is independent of which node it is on.) The only other possibility is that the requested page is a local shared page, in which case the remote marks it as a duplicate. In either case, the requesting node adds an entry.

The getpage operation represents one half of the swap; the second half is executed at a later time in the context of the pageout daemon. As getpage operations are executed on a node, the free list shrinks, eventually causing the pageout daemon to wakeup.

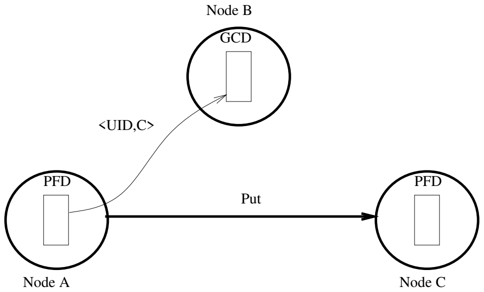

Figure 5: Executing a Putpage Operation.

The pageout daemon, in concert with the GMS, the UBC, and the VM managers (see Figure 3), attempts to evict the oldest pages on the node. These evictions are handled by the GMS module, which has age information about all these pages. Some of the pages are pushed to other idle nodes (using a *putpage* operation); others are discarded because they are older than the *MinAge* value for the current epoch. Those putpage operations that go to other nodes may eventually cause older pages on those nodes to be forwarded or discarded.

When a node performs a putpage operation, apart from sending the page to the target node, it also updates the global-cache-directory at the node that is responsible for the affected page. The target node and the sending node each update their page-frame-directory structures. If the evicted page is a shared global page for which a duplicate exists elsewhere in the cluster, then it is simply discarded.

The target node for a putpage operation is selected by executing the algorithm described in Section 3.2. Figure 5 depicts the major steps involved in a putpage operation.

5 Performance

This section provides performance measurements of our implementation of global memory management from several perspectives. First, we present microbenchmarks that give a detailed breakdown of the time for fundamental global memory management operations. Second, we look at the speedup obtained by memory-intensive applications when using our system. Third, we examine the effectiveness of our algorithm under different global loading parameters. All experiments were carried out using DEC Alpha workstations running OSF/1 V3.2, connected by a 155 Mb/s DEC AN2 ATM network. The page size on the Alpha and the unit of transfer for our measurements is 8 Kbytes.

5.1 Microbenchmarks

Here we report simple measurements to evaluate the underlying performance of our current implementation. The experiments were done using eight 225-MHz DEC 3000-700 workstations in the environment above, each with a local disk and connection to one or more NFS file servers that export shared files, including program binaries. All of these measurements were taken on otherwise idle machines.

Table 1 itemizes the cost of a getpage operation. The getpage cost depends on whether the page is shared or non-shared and whether there is a hit or miss in the global cache. As described in Section 4.1, if the page is non-shared, the GCD (global-cache-directory) node is the same as the requesting node.

Operation	**Latency in μs**			
	Non-Shared Page		**Shared Page**	
	Miss	Hit	Miss	Hit
Request Generation	7	61	65	65
Reply Receipt	-	156	5	150
GCD Processing	8	8	59	61
Network HW&SW	-	1135	211	1241
Target Processing	-	80	-	81
Total	15	1440	340	1558

Table 1: Performance of the *Getpage* Operation (μs)

The rows provide the following information. *Request Generation* is the time for the requesting node to generate a getpage request, including access to the page-ownership-directory to contact the GCD node. *Reply Receipt* is the requester time spent to process the reply from the remote node; e.g., for a hit, this includes the cost of copying data from the network buffer into a free page and freeing the network buffer. *GCD Processing* accounts for the total time spent in the GCD lookup operation as well as forwarding the request to the PFD node. *Network HW&SW* is the total time spent in the network hardware and the operating system's network protocol. The bulk of this time is spent in hardware; when sending large packets, the total latency introduced by the sending and receiving controllers is comparable to the transmission time on the fiber. *Target Processing* refers to the time on the remote node to do a lookup in the page-frame-directory and reply to the requesting node. *Total* is the sum of the rows and represents the latency seen by the initiating node before the getpage operation completes.[†] To put these numbers in perspective, on identical hardware, the cost of a simple user-to-user UDP packet exchange for requesting and receiving an 8 Kbyte page is about 1640 μs.

The *Total* line for the first column shows the cost of an unsuccessful attempt to locate a non-shared page in the cluster. In this case, the requested page is not in the global cache and a disk access is required. This cost thus represents the overhead we add to an OSF/1 disk access. The OSF/1 disk access time varies between 3600 μs and 14300 μs, so this represents an overhead of only 0.4–0.1% on a miss.

Table 2 shows the costs for a putpage operation. As shown in Figure 5, a putpage updates the global-cache-directory as well as the page-frame-directory structures on the source and target nodes. In the case of shared pages, the sending node might need to initiate two network transmissions—one to the GCD and another to the PFD; this time is reflected in *Request Generation*. In a putpage operation, the sending node does not wait for a reply from the target before returning from a putpage call; therefore, in the bottom row we show *Sender Latency* rather than the total. For shared pages, this latency is the same as *Request Generation*; for non-shared pages, the latency also includes the *GCD processing*, because the GCD is on the same node.

Putpage requests are typically executed by a node under memory pressure to free frames. An important metric is therefore the latency for a frame to be freed; this consists of the sum of the *Sender Latency* and the time for the network controller to transmit a buffer, so that the operating system can add it to the free list. For 8 Kbyte pages, the combined transmission and operating system

[†] We could reduce the getpage hit latency 200 μs by eliminating a copy and short circuiting the network-packet delivery path, but have not yet included these changes.

Operation	Latency in μs	
	Non-Shared Page	Shared Page
Request Generation	58	102
GCD Processing	7	12
Network HW&SW	989	989
Target Processing	178	181
Sender Latency	65	102

Table 2: Performance of the *Putpage* Operation (μs)

Access Type	Access Latency in ms	
	GMS	No GMS
Sequential Access	2.1	3.6
Random Access	2.1	14.3

Table 3: Average Access Times for Non-shared Pages (ms)

Access Type	Access Latency in ms			
	GMS Single	GMS Duplicate	NFS Miss	NFS Hit
Sequential Access	2.1	1.7	4.8	1.9
Random Access	2.1	1.7	16.7	1.9

Table 4: Average Access Times for Shared Pages (ms)

Operation	CPU Cost μs	Network Traffic bytes/s
Initiator Request	$78 \times n$	$25 \times n$
Gather Summary	3512	$154 \times n$
Distribute Weights	$45 \times n$	$(108 + 2 \times n) \times n$

Table 5: Normalized overhead for age information for 2-second epoch

overhead is about 300 μs.

Table 3 compares the average data-read time for non-shared pages with and without GMS. For this experiment, we ran a synthetic program on a machine with 64 Mbytes of memory. The program repeatedly accesses a large number of anonymous (i.e., non-shared) pages, in excess of the total physical memory. In steady state for this experiment, every access requires a putpage to free a page and a getpage to fetch the faulted page. The average read time thus reflects the overhead of both operations.

The first row of Table 3 shows the average performance of sequential reads to non-shared pages. The numbers shown with no GMS reflect the average disk access time; the difference between the sequential and random access times indicates the substantial benefit OSF gains from prefetching and clustering disk blocks for sequential reads. Nevertheless, using GMS reduces the average sequential read time by 41% for non-shared pages. For non-sequential accesses, GMS shows a nearly a 7-fold performance improvement. Here the native OSF/1 system is unable to exploit clustering to amortize the cost of disk seeks and rotational delays.

Table 4 shows the data-access times for NFS files that can be potentially shared. In this experiment, a client machine with 64 Mbytes of memory tries to access a large NFS file that will not fit into its main memory, although there is sufficient memory in the cluster to hold all of its pages. There are four cases to consider.

In the first case, shown in the first column of Table 4, we assume that all NFS pages accessed by the client will be put into global memory. This happens in practice when a *single* NFS client accesses the file from a server. For the most part, the behavior of the system is similar to the experiment described above: there will be a putpage and a getpage for each access. In this case, the pages will be fetched from global memory on idle machines.

The second case is a variation of the first, where two clients are accessing the same NFS file. One client has ample memory to store the entire file while the other client does not. Because of the memory pressure, the second client will do a series of putpage and getpage operations. The putpage operations in this case are for shared pages, for which copies already exist in the file buffer cache of the other client (i.e., they are *duplicates*). Such a putpage operation causes the page to be dropped; there is no network transmission. The average access cost in this case is therefore the cost of a getpage.

The next two cases examine the cost of a read access when there is no GMS. In the first case, we constrain the NFS file server so that it does not have enough buffer cache for the entire file. A client read access will thus result in an NFS request to the server, which will require a disk access. In the final case, the NFS server has enough memory so that it can satisfy client requests without accessing the disk. Here, the cost of an access is simply the overhead of the NFS call and reply between the client and the server. Notice that an NFS server-cache hit is 0.2 ms faster than a GMS hit for a *single*. This reflects the additional cost of the putpage operation performed by GMS when a page is discarded from the client cache. In NFS, discarded pages are dropped as they are in GMS for *duplicates*, in which case GMS is 0.2 ms faster than NFS.

5.2 Bookkeeping Overheads

This section describes the cost of performing the essential GMS bookkeeping operations, which include the periodic flushing of TLB entries as well as the overhead of collecting and propagating global page age information.

On the 225-MHz processor, our modified TLB handler introduces a latency of about 60 cycles (an additional 18 cycles over the standard handler) on the TLB fill path. In addition, since TLB entries are flushed every minute, with a 44 entry TLB, we introduce a negligible overhead of 2640 (60×44) cycles per minute. In practice, we have seen no slowdown in the execution time of programs with the modified TLB handler.

Collecting and propagating the age information consists of multiple steps: (1) the initiator triggers a new epoch by sending out a request to each node asking for summary age information, (2) each node gathers the summary information and returns it to the initiator, and (3) the initiator receives the information, calculates weights and epoch parameters, and distributes the data back to each node.

The three rows of Table 5 represent the CPU cost and the network traffic induced by each of these operations. For steps one and three, the table shows the CPU overhead on the initiator node and the network traffic it generates as a function of the number of nodes, n. The CPU cost in step two is a function of the number of pages each node must scan: 0.29 μs per local page and 0.54 μs for each global page scanned. The overhead shown in the table assumes that each node has 64 Mbytes (8192 pages) of local memory and that 2000 global pages are scanned. We display network traffic as a rate in bytes per second by assuming a worst-case triggering interval of 2 seconds (a 2-second epoch would be extremely short). Given this short epoch length and a 100-node network, CPU overhead is less than 0.8% on the initiator node and less than 0.2% on other nodes, while the impact on network bandwidth is minimal.

5.3 Execution Time Improvement

This section examines the performance gains seen by several applications with the global memory system. These applications are memory and file I/O intensive, so under normal circumstances, performance suffers due to disk accesses if the machine has insufficient memory for application needs. In these situations we would expect global memory management to improve performance, assuming that enough idle memory exists in the network. The applications we measured were the following:

Boeing CAD is a simulation of a CAD application used in the design of Boeing aircraft, based on a set of page-level access traces gathered at Boeing. During a four-hour period, eight engineers performed various operations accessing a shared 500-Mbyte database. We simulated this activity by replaying one of these traces.

VLSI Router is a VLSI routing program developed at DEC WRL for microprocessor layout. The program is memory intensive and can cause significant paging activity on small-memory machines.

Compile and Link is a partial compile and link of the OSF/1 kernel. By far, the most time is spent in file I/O for compiler and linker access to temporary, source, and object files.

OO7 is an object-oriented database benchmark that builds a parts-assembly database in virtual memory and then performs several traversals of this database [4]. The benchmark is designed to synthesize the characteristics of MCAD design data and has been used to evaluate the performance of commercial and research object databases.

Render is a graphics rendering program that displays a computer-generated scene from a pre-computed 178-Mbyte database [5]. In our experiment, we measured the elapsed time for a sequence of operations that move the viewpoint progressively closer to the scene without changing the viewpoint angle.

Web Query Server is a server that handles queries against the full text of Digital's internal World-Wide-Web pages (and some popular external Web pages). We measured its performance for processing a script containing 150 typical user queries.

To provide a best-case estimate of the performance impact of global memory management, we measured the speedup of our applications relative to a native OSF system. Nine nodes were used for these measurements: eight 225-MHz DEC 3000 Model 700 machines rated at 163 SPECint92 and one 233-MHz DEC AlphaStation 400 4/233 rated at 157 SPECint92. The AlphaStation had 64 Mbytes of memory and ran each application in turn. The other eight machines housed an amount of idle memory that was equally divided among them. We varied the total amount of idle cluster memory to see the impact of free memory size. Figure 6 shows the speedup of each of the applications as a function of the amount of idle network memory.

As Figure 6 shows, global memory management has a beneficial impact on all the applications. With zero idle memory, application performance with and without GMS is comparable. This is in agreement with our microbenchmarks that indicate GMS overheads are only 0.4–0.1% when there is no idle memory. Even when

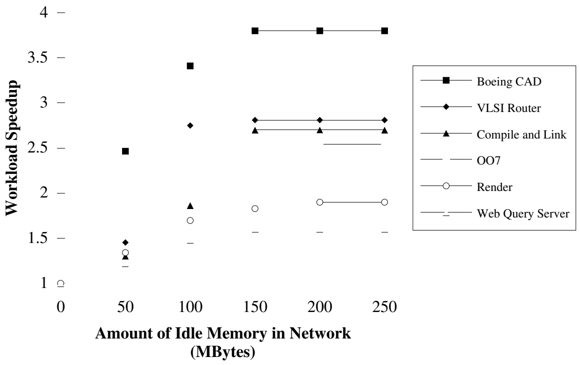

Figure 6: Workload Speedup with GMS

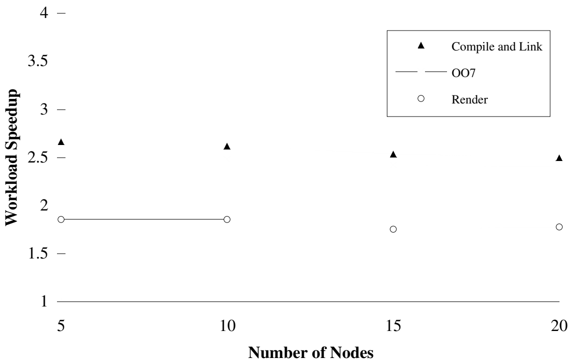

Figure 7: Workload speedup as we vary the number of nodes. Two fifths of the nodes are idle and the remainder run a mix of three workloads.

idle memory is insufficient to meet the application's demands, our system provides relatively good performance. Beyond about 200 Mbytes of free memory in the cluster, the performance of these applications does not show any appreciable change, but at that point, we see speedups of from 1.5 to 3.5, depending on the application. These speedups are significant and demonstrate the potential of using remote memory to reduce the disk bottleneck.

Figure 6 shows the speedup of each application when running alone with sufficient global memory. To demonstrate that those benefits remain when multiple applications run simultaneously, competing for memory in a larger network, we ran another experiment. Here we varied the number of nodes from five to twenty; in each group of five workstations, two were idle and each of the remaining three ran a different workload (OO7, Compile and Link, or Render). The idle machines had sufficient memory to meet the needs of the workloads. Thus, when running with twenty nodes, eight were idle and each of the three workloads was running on four different nodes. The results of this experiment, shown in Figure 7, demonstrate that the speedup remains nearly constant as the number of nodes is increased.

5.4 Responsiveness to Load Changes

Our algorithm is based on the distribution and use of memory load information. An obvious question, then, is the extent to which our system can cope with rapid changes in the distribution of idle pages in the network. To measure this, we ran a controlled experiment, again using nine nodes. In this case, the 233-MHz AlphaStation

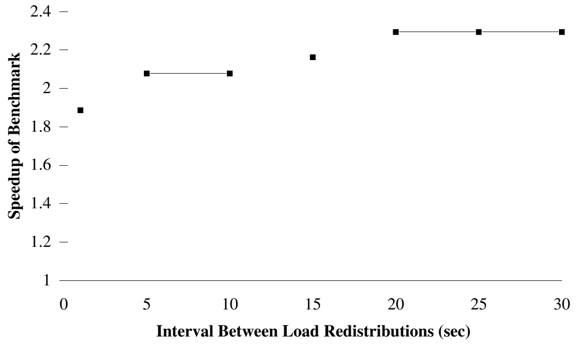

Figure 8: Effect of varying the frequency with which nodes change from idle to non-idle on the performance of the OO7 benchmark.

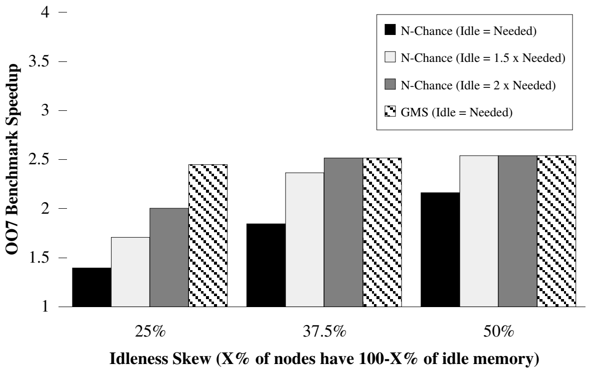

Figure 9: Effect of varying distribution of idleness on performance of OO7 benchmark.

ran a memory-intensive application (we chose OO7 for this and several other tests, because its relatively short running time made the evaluation more practical); the other eight nodes were divided into two sets: those with idle memory and those without idle memory. The total idle memory in the network was fixed at 150% of what is needed by OO7 for optimal performance, and 4 of the 8 nodes had most of the idle memory. For the experiment, every X seconds we caused an idle node to switch with a non-idle node.

Figure 8 shows the speedup of the OO7 benchmark (again, relative to a non-GMS system) as a function of the frequency with which the load changes occurred. As the graph indicates, our system is successful at achieving speedup even in the face of frequent and rapid changes. At a maximum change rate of 1 second, we still see a speedup of 1.9 over the non-global system, despite the fact that the change requires a swap of 70MB of data between the non-idle and idle nodes, which adds significant stress to the system. At lower change rates of 20 to 30 seconds, we see only a small (4%) impact on the speedup.

5.5 The Effect of Idle Memory Distribution

Another key question about our algorithm is the extent to which it is affected by the distribution of idle pages in the network. To measure this impact, we constructed an experiment to vary the number of nodes in which idle pages are distributed. Again, we measured the speedup of the OO7 benchmark on one node, while we controlled the distribution of idle pages on 8 other nodes. We chose three points in the spectrum, arranging in each case that $X\%$ of the nodes contained $(100 - X)\%$ of the idle memory. The cases we consider are (1) 25% of the nodes had 75% of the free memory, (2) 37.5% of the nodes housed 62.5% of the free memory, and (3) 50% of the nodes held 50% of the free memory. Case (1) is the most skewed, in that most of the free memory is housed on a small number of nodes, while in case (3) the idle memory is uniformly distributed.

For comparison, we show results for our algorithm and for N-chance forwarding, the best algorithm defined by Dahlin et al.[10]. It is interesting to compare with N-chance, because it differs from our algorithm in significant ways, as described in Section 2. To the best of our knowledge, there is no existing implementation of the N-chance algorithm described by Dahlin et al. [10]. Consequently, we implemented N-chance in OSF/1. We made a few minor modifications to the original algorithm. Our implementation is as follows.

Singlet pages are forwarded by nodes with an initial recirculation count of N = 2. When a singlet arrives at a node, the victim page to be evicted is picked in the following order. If there are sufficient pages on the free list such that allocating one will not trigger page reclamation (by the pageout daemon), we allocate a free page for the forwarded page and no page is evicted. Otherwise, we choose in turn: the oldest duplicate page in the UBC, the oldest recirculating page, or a very old singlet. If this process locates a victim, it is discarded and replaced with the forwarded page. If no victim is available, the recirculation count of the forwarded page is decremented and it is either forwarded or, if the recirculation count has reached zero, dropped.

The modifications improve the original algorithm while preserving its key features: (1) a random choice of a target node is made without using global knowledge, and (2) singlets are kept in the cluster at the expense of duplicates, even if they might be needed in the near future.

Figure 9 shows the effect of varying idleness in the cluster. Here, for each point of the idleness-distribution graph we show four bars—three for N-chance and one for GMS. The captions on the N-chance bars indicate, for each case, the total amount of idle memory that was distributed in the network. For example, for Idle = Needed, the network contained exactly the number of idle pages needed by the application, while for Idle = 2 $\times$ Needed the network contained twice the idle pages needed by the application. The GMS bar is measured with Idle = Needed. From the figure, we see that for degrees of idleness that are more skewed, it is difficult for N-chance to find the least loaded nodes. For example, in the case of 25% of the nodes holding 75% of the pages, GMS achieves a substantial speedup relative to N-chance in all cases, even if the N-chance algorithm is given *twice* the idle memory in the network. As we move along the graph to 37.5%, where 3 of the 8 nodes hold 63% of the idle pages, N-chance is equivalent to GMS only when given twice the idle memory. Obviously, when there is plenty of free memory in the cluster, the exact distribution of idle pages becomes secondary and the performance of N-chance is comparable to that of our algorithm, as long as it has more pages than are needed.

The effectiveness of the GMS system and its load information is in fact shown by two things: first, it is superior to N-chance with non-uniform idleness distributions, even given fewer idle pages; second, and perhaps more important, the performance of GMS, as show by the graph, is more-or-less *independent* of the skew. In other words, GMS is successful at finding and using the idle pages in the network, even though it had no excess idle pages to rely on.

The differences between these algorithms are mostly due to the random nature of the N-chance algorithm. Since N-chance chooses

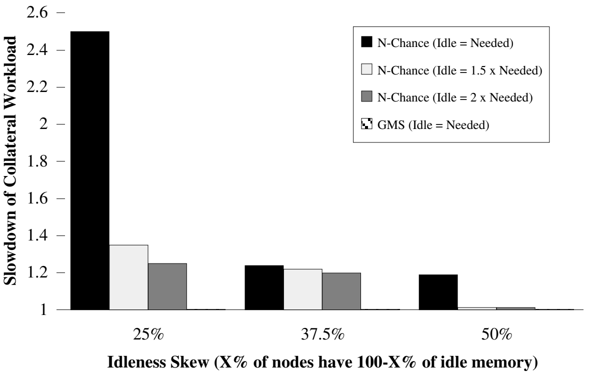

Figure 10: Effect of varying distribution of idleness on the performance of a program actively accessing local memory, half of which consists of shared data that is duplicated on other nodes.

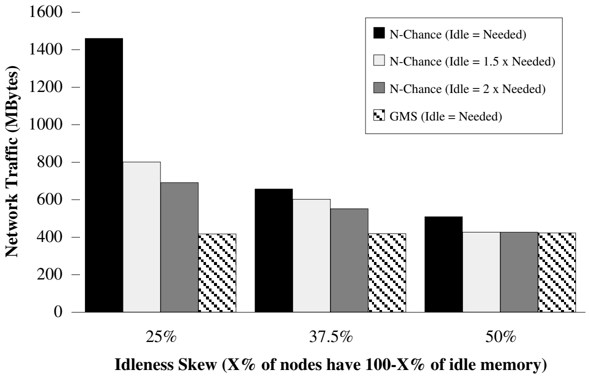

Figure 11: Effect of varying distribution of idleness on network activity.

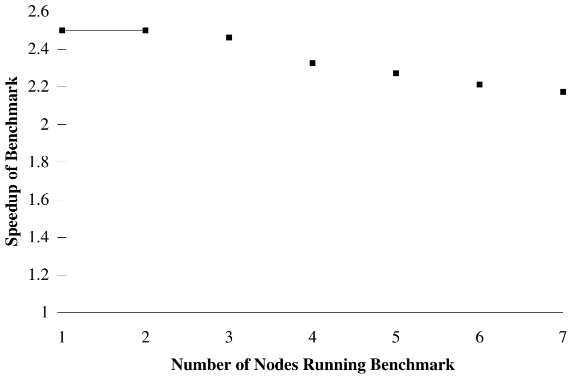

Figure 12: Performance impact of a single idle node serving the remote memory needs for multiple client nodes.

a target node at random, it works best if idle memory is uniformly distributed, which will not always be the case.

5.6 Interference with Non-Idle Nodes

A possible negative effect of global memory management, mentioned above, is that the global management may negatively impact applications running on non-idle nodes in the cluster. A heavily paging node can interfere with other non-idle nodes by adding management overhead, by replacing needed pages, or by causing network traffic.

As one test of this effect, we ran an experiment as before with 9 nodes. One node ran the OO7 benchmark, generating global memory traffic. On the remaining 8 nodes, we distributed idle pages as in the previous section, creating different skews of the idle pages. However, in this case, all the *non-idle* nodes ran a copy of a synthetic program that loops through various pages of its local memory; half of the pages accessed by this program are shared among the various running instances, while half are private to each instance.

Figure 10 shows the average slowdown of the synthetic programs for both GMS and the three N-chance tests given various distributions of idle memory, as before. The slowdown gives the performance decrease relative to the average performance when OO7 was *not* running. For all the distributions tested, GMS causes virtually no slowdown of the synthetic program when OO7 is generating global memory traffic. In contrast, N-chance causes a slowdown of as much as 2.5, depending on the amount of idle memory and its distribution. For 3 idle nodes (37.5%), N-chance causes a slowdown of 1.2, even when twice the needed idle memory is available. When the idle memory is uniformly distributed, N-chance again does as well as GMS, as long as there is additional idle memory in the cluster.

Figure 11 shows, for the same experiment, the network traffic in megabytes measured during the test, which took 100-200 seconds. Here we see the network impact of the two algorithms for the various idle page distributions. For the more skewed distribution of 25%, GMS generates less than 1/3 of the network traffic of N-chance, given the same number of idle pages in the network. At twice the number of idle pages, N-chance still generates over 50% more traffic than GMS. Not until the idle memory is uniformly distributed does the N-chance network traffic equal the GMS traffic.

The reasons for these differences are two-fold. First, N-chance chooses nodes at random, as previously described, so remote pages from OO7 are sent to the non-idle nodes. Second, N-chance attempts to keep singlets in the network by displacing duplicates, even if they have been recently referenced. This second effect is evident in this test, i.e., the active, shared, local pages on the non-idle nodes are displaced by remote pages sent (randomly) to those nodes by the N-chance algorithm. When the program on the non-idle node references one of these displaced pages, a fault occurs and the page is retrieved from one of the other nodes or from disk. To make room for the faulted page, the node must discard its LRU page by forwarding it or dropping it if it is a duplicate. This additional paging increases network traffic and reduces the performance of all programs running in the cluster.

5.7 CPU Load on Idle Nodes

The experiment in the previous section demonstrates the success of GMS in avoiding nodes without idle memory. However, even nodes with substantial amounts of idle memory may be "non-idle," i.e., may be running programs executing in a small subset of local memory. A global management system such as ours does place CPU overhead on such a node in order to use its idle memory, through the getpage and putpage requests that it must respond to.

To evaluate the extent of this impact under heavy load, we performed an experiment in which only one node had idle memory. We then monitored the CPU load on that node as we increased the number of clients using its memory for their global storage. Again,

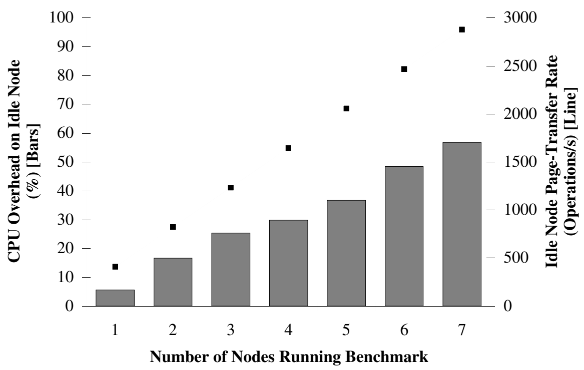

Figure 13: Impact of multiple clients on CPU performance of an idle node.

we used OO7, increasing the number of OO7 client nodes from one to seven, ensuring in all cases that the idle node had sufficient memory to handle all of their needs.

Figure 12 shows that when seven copies of OO7 were simultaneously using the remote memory of the idle node, the average speedup achieved by GMS was only moderately lowered. That is, the applications using the idle node's memory did not seriously degrade in performance as a result of their sharing a single global memory provider.

On the other hand, Figure 13 shows the result of that workload on the idle node itself. The bar graph shows the CPU overhead experienced by the idle node as a percentage of total CPU cycles. As well, Figure 13 plots the rate of page-transfer (getpage and putpage) operations at the idle node during that execution. From this data, we see that when seven nodes were running OO7 simultaneously, the idle node received an average of 2880 page-transfer operations per second, which required 56% of the processor's CPU cycles. This translates to an average per-operation overhead of 194 μs, consistent with our micro-benchmark measurements.

6 Limitations

The most fundamental concern with respect to network-wide resource management is the impact of failures. In most distributed systems, failures can cause disruption, but they should not cause permanent data loss. Temporary service loss is common on any distributed system, as anyone using a distributed file system is well aware. With our current algorithm, all pages in global memory are clean, and can therefore be retrieved from disk should a node holding global pages fail. The failure of the initiator or master nodes is more difficult to handle; while we have not yet implemented such schemes, simple algorithms exist for the remaining nodes to elect a replacement.

A reasonable extension to our system would permit dirty pages to be sent to global memory *without* first writing them to disk. Such a scheme would have performance advantages, particularly given distributed file systems and faster networks, at the risk of data loss in the case of failure. A commonly used solution is to replicate pages in the global memory of multiple nodes; this is future work that we intend to explore.

Another issue is one of trust. As a cluster becomes more closely coupled, the machines act more as a single timesharing system. Our mechanism expects a single, trusted, cluster-wide administrative domain. All of the kernels must trust each other in various ways. In particular, one node must trust another to not reveal or corrupt its data that is stored in the second node's global memory. Without mutual trust, the solution is to encrypt the data on its way to or from global memory. This could be done most easily at the network hardware level [19].

Our current algorithm is essentially a modified global LRU replacement scheme. It is well known that in some cases, such as sequential file access, LRU may not be the best choice [22]. The sequential case could be dealt with by limiting its buffer space, as is done currently in the OSF/1 file buffer cache. Other problems could exist as well. The most obvious is that a single badly-behaving program on one node could cause enormous paging activity, effectively flushing global memory. Of course, even without global memory, a misbehaving program could flood the network or disk, disrupting service. Again, one approach is to provide a threshold limiting the total amount of global memory storage that a single node or single application could consume.

If only a few nodes have idle memory, the CPU load on these nodes could be high if a large number of nodes all attempt to use that idle memory simultaneously. If there are programs running on the idle machines, this could adversely affect their performance. This effect was measured in the previous section. A possible solution is to incorporate CPU-load information with page age information, and to use it to limit CPU overhead on heavily-loaded machines.

In the end, all global memory schemes depend on the existence of "a reasonable amount of idle memory" in the network. If the idle memory drops below a certain point, the use of global memory management should be abandoned until it returns to a reasonable level. Our measurements show the ability of our algorithm and implementation to find and effectively utilize global memory even when idle memory is limited.

7 Conclusions

Current-generation networks, such as ATM, provide an order-of-magnitude performance increase over existing 10 Mb/s Ethernets; another order-of-magnitude—to gigabit networks—is visible on the horizon. Such networks permit a much tighter coupling of interconnected computers, particularly in local area clusters. To benefit from this new technology, however, operating systems must integrate low-latency high-bandwidth networks into their design, in order to increase the performance of both distributed and parallel applications.

We have shown that global memory management is one practical and efficient way to share cluster-wide memory resources. We have designed a memory management system that attempts to make cluster-wide decisions on memory usage, dynamically adjusting the local/global memory balance on each node as the node's behavior and the cluster's behavior change. Our system does not cause data loss should nodes fail, because only clean pages are cached in global memory; cached data can always be fetched from disk if necessary.

The goal of any global memory algorithm is to reduce the average memory access time. Key to our algorithm is its use of periodically-distributed cluster-wide age information in order to: (1) house global pages in those nodes most likely to have idle memory, (2) avoid burdening nodes that are actively using their memory, (3) ultimately maintain in cluster-wide primary memory the pages most likely to be globally reused, and (4) maintain those pages in the right places. Algorithms that do not have these properties are unlikely to be successful in a dynamic cluster environment.

We have implemented our algorithm on the OSF/1 operating system running on a cluster of DEC Alpha workstations connected by a DEC AN2 ATM network. Our measurements show the underlying costs for global memory management operations in light of a real implementation, and the potential benefits of global memory management for applications executing within a local-area cluster.

Acknowledgments

Jeff Chase was closely involved in early discussions of this work. Comments from Brian Bershad, Jeff Chase, Dylan McNamee, Hal Murray, John Ousterhout, Chuck Thacker, and the anonymous referees helped improve the quality of the paper. The authors would like to thank Ted Romer for his help with Alpha PALcode, Dylan McNamee and Brad Chamberlain for the Render application, Jeremy Dion and Patrick Boyle for the VLSI router, Steve Glassman for the Web Query Server, Vivek Narasayya for help with the OO7 benchmark, and Ashutosh Tiwary for the Boeing CAD traces. We would also like to thank Hal Murray for his help with the AN2 network, and the DEC SRC community for donating workstations for our experiments.

References

[1] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, Nov. 1993.

[2] M. Bern, D. Greene, and A. Raghunathan. Online algorithms for cache sharing. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, May 1993.

[3] D. Black and D. D. Sleator. Competitive algorithms for replication and migration poblems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[4] M. J. Carey, D. J. Dewitt, and J. F. Naughton. The OO7 benchmark. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, May 1993.

[5] B. Chamberlain, T. DeRose, D. Salesin, J. Snyder, and D. Lischinski. Fast rendering of complex environments using a spatial hierarchy. Technical Report 95-05-02, Department of Computer Science and Engineering, University of Washington, May 1995.

[6] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, Nov. 1994.

[7] M. Chrobak, L. Larmore, N. Reingold, and J. Westbrook. Page migration algorithms using work functions. Technical Report YALE/DCS/RR-910, Department of Computer Science, Yale University, 1992.

[8] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Proceedings of the Summer 1990 USENIX Conference*, pages 127–135, June 1990.

[9] A. L. Cox and R. J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[10] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.

[11] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5), May 1986.

[12] E. W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, Department of Computer Science and Engineering, University of Washington, Mar. 1991.

[13] M. J. Frankling, M. J. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *Proceedings of the 18th VLDB Conference*, August 1992.

[14] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. VAX-clusters: A closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, May 1986.

[15] P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1(5):842–857, Nov. 1983.

[16] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[17] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, Feb. 1988.

[18] B. N. Schilit and D. Duchamp. Adaptive remote paging. Technical Report CUCS-004091, Department of Computer Science, Columbia University, February 1991.

[19] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, Oct. 1991.

[20] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, One Burlington Woods Drive, Burlington, MA 01803, 1992.

[21] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[22] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, July 1981.

[23] C. P. Thacker and M. D. Schroeder. AN2 switch overview. In preparation.

[24] J. Westbrook. Randomized algorithms for multiprocessor page migration. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 7, 1992.