Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System

Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel*,
Michael J. Feeley†, Jeffrey S. Chase‡, Anna R. Karlin, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington

Abstract

This paper presents *cooperative prefetching and caching* — the use of network-wide global resources (memories, CPUs, and disks) to support prefetching and caching in the presence of hints of future demands. Cooperative prefetching and caching effectively unites disk-latency reduction techniques from three lines of research: prefetching algorithms, cluster-wide memory management, and parallel I/O. When used together, these techniques greatly increase the power of prefetching relative to a conventional (non-global-memory) system. We have designed and implemented PGMS, a cooperative prefetching and caching system, under the Digital Unix operating system running on a 1.28 Gb/sec Myrinet-connected cluster of DEC Alpha workstations. Our measurements and analysis show that by using available global resources, cooperative prefetching can obtain significant speedups for I/O-bound programs. For example, for a graphics rendering application, our system achieves a speedup of 4.9 over a non-prefetching version of the same program, and a 3.1-fold improvement over that program using local-disk prefetching alone.

1 Introduction

The past decade has seen a two-order-of-magnitude increase in processor speed, yet only a two-fold improvement in disk access time. As a result, recent research has focused on reducing disk stall time through several approaches. One approach is the development of algorithms for prefetching data from disk into memory [7, 27, 22, 29], using hints from either programmer-annotated [27] or compiler-annotated [25] programs. A second approach is the use of memory on idle network nodes as an additional level of buffer cache [13, 14, 16]; this *global memory* can be accessed much faster than disk over high-speed switched networks. A third approach is to stripe files over multiple disks [26], using multiple nodes to access the disks in parallel [18, 9, 3].

This paper presents *cooperative prefetching and caching* — the use of network-wide global memory to support prefetching and caching in the presence of optional program-provided hints of future demands. Cooperative prefetching and caching combines multiple approaches to disk-latency reduction, resulting in a system that is significantly different than one using any single approach alone. In the presence of global memory, a node has three choices for data prefetching: (1) from disk into local memory, (2) from disk into global memory (i.e., the disk and memory of *another* node), and (3) from global memory into local memory. When used together, these options greatly increase the power of prefetching relative to a conventional (non-global-memory) system.

For example, Figure 1a shows a simplified view of a conventional prefetching system. Node A issues prefetch requests to missing blocks *m* and *n* in advance, so that both blocks are available in memory just in time for the data references. In this case, buffers must be freed on node A for blocks *m* and *n* about $2F_D$ and F_D in advance of their use, respectively, where F_D is the disk fetch time. There are two possible problems with this scheme. First, node A's disk may not be free in time to prefetch these blocks without stalling. Second, if prefetched early enough to avoid stalling, blocks *m* and *n* may replace useful data, causing an increase in misses; whether or not this happens depends on how far in advance the data is prefetched (which depends on F_D) and the access pattern of the program.

In contrast, Figures 1b and 1c show two examples of prefetching in a global-memory system. From these scenarios, we see that combining prefetching and global memory has several possible advantages:

- A prefetching node can greatly delay its final load request for data that resides in global memory, thereby reducing the chance of replacing useful local data. In Figure 1b, for example, node A requests that node B prefetch pages from disk into B's memory ahead of time. As a result, node A need not free a buffer for the prefetched data until F_G (the time for a page fetch from global memory) before its use. On a 1Gb/sec network, such as Myrinet, F_G may be up to 50 times smaller than F_D, so this difference is substantial.

- The I/O bandwidth available to a single node is ultimately limited by its I/O subsystem – in most cases, the disk subsystem. However, using idle nodes to prefetch data into

---

*Kimbrel is at the IBM T.J. Watson Research Center.
†Feeley is at the Department of Computer Science, University of British Columbia.
‡Chase is at the Department of Computer Science, Duke University.

This work was supported in part by grants from the National Science Foundation (EHR-95-50429, CCR-9632769, MIP-9632977, and CDA-95-12356), the Advanced Research Projects Agency (F30602-97-2-0226), the National Science and Engineering Research Council of Canada, and from Digital Equipment Corporation, Intel Corporation, Myricom, Inc., and the Open Group. Voelker was supported in part by a fellowship from Intel Corporation, Anderson was supported in part by a fellowship from Microsoft Corporation, and Chase was supported in part by NSF CAREER CCR-96-24857.

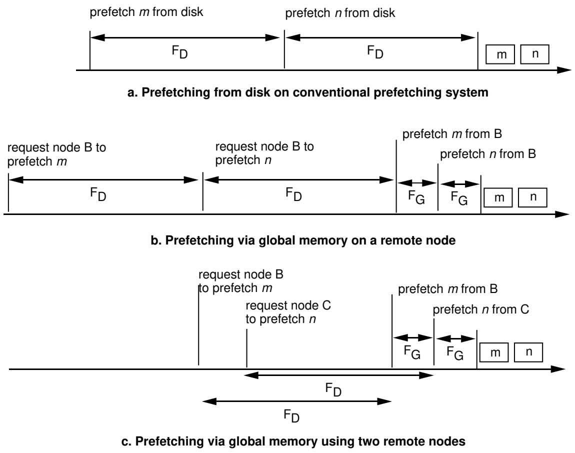

Figure 1: Prefetching in conventional and global-memory systems

global memory greatly increases the available I/O bandwidth by adding in parallel: (1) the bandwidth of the network, (2) the bandwidth of remote disk subsystems, and (3) the execution power of the remote CPUs. Use of this parallelism for the global prefetching shown in Figures 1b and 1c effectively reduces page prefetch time for I/O-bound processes from F_D to F_G.

- Figure 1c shows that distributing prefetch requests among multiple nodes in parallel allows those nodes to delay their own buffer replacement decisions (in this case, node B benefits relative to Figure 1b), thereby making more effective use of their memories.
- With the high ratio of disk latency to global memory latency, a highly-conservative process could choose to prefetch *only* into global memory; the process would fault on reference to a non-resident page, but would still benefit from the 50-fold reduction in fault time.
- Given that there is idle memory and CPU power in the network, a process could afford to prefetch *speculatively*, using idle global pages as the speculative prefetch cache.

While the idea of using network memory for prefetching is conceptually straightforward, it raises a number of questions. For example, how do nodes *globally* choose the pages in which to prefetch from the global memory pool? When should data be prefetched and to what level of the storage hierarchy? When should pages be moved from global memory to local memory? How do we trade off the use of global memory pages for prefetching versus the use of those frames to hold evicted VM and file pages for non-prefetching applications? And finally, how do we value each page in the network, in order to best utilize each page frame?

To answer these questions, we have defined an algorithm for global memory management with prefetching and implemented that algorithm in the DEC UNIX operating system, running on a collection of DEC Alpha workstations connected by a Myrinet high-speed switched network. Our system, called PGMS (Prefetching Global Memory System), integrates all cluster memory use, including VM pages, mapped files, and file system buffers, for both prefetching and non-prefetching applications. It effectively unites techniques from three previous lines of research: prefetching algorithms, including the work of Patterson et al. [27], Cao et al. [7], Kimbrel et al. [22], and Tomkins et al. [29]; the global memory system (GMS) of Feeley et al. [14]; and the use of network nodes for parallel I/O, as in the Zebra system of Hartman and Ousterhout [18]. Our measurements of PGMS executing on the Alpha-based cluster show that prefetching in a global memory system can produce substantial speedups for I/O-bound programs: e.g., for a memory-bound graphics rendering application, PGMS achieves a speedup of 4.3 over a non-prefetching version of the same program, and a 2.8-fold improvement over that program using local disk prefetching alone.

The remainder of the paper is organized as follows. Section 2 presents our algorithm for prefetching in global memory. Section 3 describes the implementation of our algorithm in the DEC UNIX operating system. Section 4 presents performance results from our prototype. We compare our work to previous research in Section 5 and conclude in Section 6.

2 The Global Prefetching and Caching Algorithm

This section presents the idealized global prefetching and caching algorithm that is the basis of PGMS; Section 3 describes how the PGMS implementation efficiently approximates the algorithm. For the purposes of defining the algorithm, we make several simplifying assumptions. First, we assume a uniform cluster topology with network page transfer cost (F_G) independent of location. Second, we assume uniform availability of disk-resident data to all nodes (e.g., through a network-attached disk [17] or replicated file system [24]) and uniform page transfer time from disk into a node's memory (F_D). For cluster systems using high-speed switched networks, F_G will be significantly smaller than F_D. Third, we assume a centralized algorithm with complete *a priori* knowledge of the reference streams of the applications running on all nodes, including the pages to be referenced, the relative order in which they are referenced, and the inter-reference times.

We begin with a description of the algorithm below. In Section 2.3, we discuss the theory motivating our design.

2.1 Design principles

The goal of our design is to minimize average memory reference time across all processes in the cluster. This goal requires that "optimal" prefetching and caching decisions be made both for individual processes and for the cluster as a whole. The algorithm we use in PGMS has two basic objectives:

- To reduce disk I/Os, maintain in the cluster's global memory the set of pages that will be referenced nearest in the future.
- To reduce stalls, bring each page in the cluster to the node that will reference it in advance of the access.

2.2 Detailed description

In our discussion, we use the term *local page* for a page that is resident on a node using that page, and the term *global page* for a page that is cached by one node on behalf of another. A reference to a global page thus requires a network transfer.

To meet its objectives PGMS must make decisions about both prefetching and cache replacement. Furthermore, the system must make (1) global decisions about which pages to keep in global memory rather than on disk, and (2) local decisions about which data to keep resident in a node's local memory rather than in global memory. The PGMS algorithm thus implements four interrelated policies:

- local cache replacement (transfer of pages from a node's local memory to global memory),

- global cache replacement (eviction from global memory),
- local prefetching (disk-to-local and global-to-local), and
- global prefetching (disk-to-global).

For cache replacement, we modify the algorithms used by GMS [14] to incorporate prefetching. For local cache replacement, we first choose to forward to global memory a global page on the local node (i.e., a page held on behalf of another node); if there is no global page, we choose the local page whose next reference is furthest in the future. For global cache replacement in PGMS, we evict the page in the cluster whose next reference is furthest in the future.

For prefetching decisions we apply a hybrid algorithm, whose goal is to be conservative locally but aggressive with resources on idle nodes. For local prefetching, we adapt the Forestall algorithm of Kimbrel, et al. [22, 29]. Forestall analyzes the future reference stream to determine whether the process is I/O constrained; if so, Forestall attempts to prefetch just early enough to avoid stalling. In our adaptation, we apply the Forestall algorithm to the node's local reference stream and take into account the different access times for network-resident (global) and disk-resident data. This analysis leads to a *prefetch predicate*; when the prefetch predicate is true, Forestall recommends that a page be prefetched either from global memory or from local disk. Whether the page is actually prefetched depends on whether a resident page can be found whose next reference is further in the future.

For prefetching into global memory (disk-to-global) PGMS uses the Aggressive algorithm of Cao et al. [6]. If a page on disk will be referenced earlier than a page in cluster memory, then the disk page is prefetched. To make room, the global eviction policy chooses for replacement the page (in the cluster) whose next reference is furthest in the future.

Computing the local prefetch predicate

The local prefetch predicate indicates when prefetching is needed to avoid additional stalls. In our predicate computation, we assume that all prefetches into a node and all memory references at a node are serialized.

Consider the hinted future reference stream on a node P at a given time T, and let $b[i]$ be the i-th missing page in the hinted reference stream that will be accessed after time T. (Missing pages at time T are those pages that are not in P's local memory or in the process of being prefetched into P's local memory at time T.) Let $t_{b[i]}$ be the time between T and the next access to $b[i]$, assuming no stalls occur between T and this access. Let F_i be the time that will be required to fetch $b[i]$ into local memory: F_i equals F_G (the time to perform a network fetch) if $b[i]$ is currently in global memory and equals F_D (the time to fetch from disk) otherwise. Under these assumptions, we can readily calculate whether or not we need to begin prefetching immediately in order to avoid stalling: prefetching is not yet required if for each j, the time to fetch the first j missing pages ($\sum_{1\leq i\leq j} F_i$) is less than the time until the access to the j-th missing page ($t_{b[j]}$). Therefore, we define the local prefetch predicate to be true if there is some j for which $\sum_{1\leq i\leq j} F_i \geq t_{b[j]}$. Whenever this prefetch predicate is true for some j, node P attempts to prefetch its first missing page.

2.3 Theoretical underpinnings

We begin with a discussion of cache replacement in a three-level memory hierarchy. Then we summarize theoretical results on prefetching as it pertains to PGMS. Finally, we touch upon the problem of buffer allocation among competing processes.

2.3.1 Cache replacement

Our algorithm attempts to minimize the total cost of all memory references within the cluster. The cost of a memory reference depends on whether, at the time of reference, the data is in local memory, in global memory (on another node), or on disk. Typically, a local hit is more than three orders of magnitude faster than a global memory or disk access, while a global memory hit over a Gb/sec network is on the order of 50 times faster than a disk access.

It is well known that in a two-level memory hierarchy such as local memory and disk, the optimal replacement strategy is to replace the page whose next reference is furthest in the future [4]. The analogous replacement strategy for a three-level memory hierarchy (local memory, global memory, disk) such as PGMS is the Global Longest Forward Distance ($GLFD$) algorithm, defined formally as follows.

> On a reference by node A to page g in global memory on node G, bring g into A's memory, where it becomes a local page. In exchange, select a page on A for eviction: if A has a global page, send that page to G, where it remains a global page; otherwise if A has no global page, select the local page whose next reference is furthest in the future on A, and send that page to G, where it becomes a global page. On a reference by node A to page d on disk, read d into A's memory. In exchange, select (1) a page a on A for eviction to global memory, and (2) a page g in the cluster for eviction to disk. Select the page a on A for eviction using the same method described above for a global memory reference. For the cluster-wide eviction, select page g (say on node G) whose next reference is furthest in the future, cluster-wide. Write g to disk, and send a to node G, where it becomes a global page.

The effect of this algorithm is to (1) maintain in the cluster as a whole the set of pages that will be accessed soonest and (2) maintain on each node the set of pages that will be accessed soonest by processes running on that node. While this algorithm is not always optimal, it is near optimal as shown by the following theorem (whose proof we omit for reasons of space):

Theorem

Consider a global memory system with local memory access cost F_L, global memory access cost F_G, and disk access cost F_D, where $F_L < F_G < F_D$. Let OPT be the offline page replacement algorithm minimizing total memory access cost. We denote by $C_{OPT}(R)$ the total memory access cost incurred by OPT on reference stream R, i.e.

$$C_{OPT}(R) = |R|F_L + O_G(R)F_G + O_D(R)F_D,$$

where $O_G(R)$ (resp. $O_D(R)$) denotes the number of global memory references (resp. disk references) made by OPT on R. Similarly, denote by $C_{GLFD}(R)$ the total memory access cost incurred by $GLFD$ on input R. Then for any R,

$$C_{GLFD}(R) \leq C_{OPT}(R)\left(1 + 3(F_G/F_D)\right).$$

The theorem implies that the $GLFD$ algorithm is near optimal whenever the ratio of network access time to disk access time is small. For example, in a fast network such as the Myrinet where $(F_G/F_D) \leq 0.02$, the total I/O overhead incurred by the $GLFD$ paging algorithm is within 6% of optimal. Therefore, in PGMS we use $GLFD$ as the cache replacement algorithm.

2.3.2 Prefetching strategy

Effective prefetching into local memory eliminates stall time while minimizing computational overhead. Previous studies of prefetching [6, 7] have shown that for a fully-hinted process with a single disk, the Aggressive prefetching algorithm achieves near-optimal reduction in stall time. Unfortunately, Aggressive's early prefetching may result in suboptimal replacements, which can increase the total number of I/Os performed. Although these I/Os are overlapped with computation, a significant overhead (the computational overhead of issuing fetches) can result. The Forestall algorithm has been shown in practice to match the reduction in I/O stall achieved by the Aggressive algorithm, while avoiding the computational overhead of performing unnecessary fetches [22, 29]. Forestall is therefore the method of choice for local prefetching.

In contrast to local prefetching, disk stall time is much more important than computational overhead for disk-to-global prefetching, where the prefetching is performed by otherwise idle nodes. By analogy with the problem of prefetching from a single disk into a single memory [6], the problem of prefetching from multiple disks into global memory, under the assumption that disk-resident data is available uniformly on all disks, can be shown to achieve near-optimal reduction in disk stall time. [1] Further, where the pages to be evicted will not be referenced until significantly later, if ever, aggressive prefetching's drawback of less accurate cache replacement decisions is relatively unimportant. Little harm results from displacing these pages aggressively in order to gain the benefits of prefetching. [2]

There are two other important reasons to prefetch aggressively into global memory. First, pressure on *local* memory is significantly reduced through aggressive global prefetching. Indeed, the times at which the local prefetch predicate for a process is true depends directly on how many of the process' missing pages are in global memory (as opposed to disk); the greater the fraction of missing pages that are in global memory, the *later* the times at which the predicate will first be true. Delaying the times at which the local prefetch predicate is true allows better replacements to be made on a busy node running the hinted process, reducing unnecessary fetches and associated overhead on that node. Second, hinted processes cannot rely on access to the full CPU and disk bandwidth of idle nodes, because of competition with other prefetching processes for these resources. Aggressive prefetching gives these processes some leeway for dealing with this uncertainty whereas more conservative global-memory prefetching could result in unnecessary stall.

2.3.3 Allocating buffers among competing processes

Our assumption of complete advance knowledge of the combined reference streams of the applications in the cluster allows us to view each node as executing a single process. This simplifies the algorithm and conceptual framework. In practice, any prefetching system must allocate buffers among multiple independent processes with differing hint capabilities.

Policies for allocating buffers among competing processes on a single node have been extensively studied [27, 29, 8, 7]. These studies show that proper buffer allocation among competing processes on a single node must consider working set sizes, hinted reference patterns, cache behavior of unhinted processes, variability of inter-reference CPU times between different processes, the prefetching and cache replacement policies used and processor scheduling. For example, the benefit of the prefetch recommendations made by hinted processes can be compared to the cost of LRU cache replacement decisions for unhinted processes [27, 29]. An interesting direction for future research is to analyze these algorithms in the context of a prefetching global memory system.

Processes on different nodes will also compete for global memory and prefetching resources. The prefetching system must similarly allocate resources among competing nodes. As noted above, the aggressive prefetching policy in PGMS reduces the impact of this competition on individual prefetching processes, both by reducing the likelihood of disk faults and by reducing the uncertainty resulting from independent competing prefetching requests.

2.4 Summary

We have outlined an algorithm for prefetching and caching in global memory systems. PGMS prefetches into local memory conservatively (delaying as long as possible) and into global memory aggressively. The objective of this two-pronged scheme is to maintain valuable blocks in local memory, while sacrificing global blocks to speedup prefetching. We make this tradeoff because it has been shown that in a global memory system performance is relatively insensitive to which of the oldest global pages are replaced [30]. Therefore, we replace the least valuable global pages in order to reduce stall time through prefetching, without risking local performance. *The ability to make this tradeoff is the key advantage of combining prefetching and global memory.*

3 Implementation

The previous section presented an idealized algorithm for prefetching in global memory systems. We now describe the prototype PGMS implementation that approximates the ideal algorithm for cooperative prefetching and caching. In brief, we implemented PGMS by taking the Digital-UNIX-based GMS global memory system [14], adding prefetching support, and then implementing an approximation to the prefetching algorithm presented above. We begin by giving an overview of GMS for background.

3.1 Overview of GMS

GMS is a global memory system for a clustered network of workstations. The goal of GMS is to use global memory to increase the average performance of all cluster applications. Programs benefit from global memory in two ways. First, on a page replacement, the evicted page is sent to global memory rather than disk; a reload of that page may therefore occur much faster. Second, programs benefit from access to shared pages, which may be transferred over the network rather than from disk.

GMS is implemented as a low-level operating system layer, underneath both the file system and the virtual memory system. All *getpage* requests issued by both the VM and file systems to fetch pages from long-term storage, and all *putpage* requests issued to send pages to long-term storage, are intercepted by GMS. Each page in the GMS system has a network-wide unique ID, determined by its location on disk (i.e., the UID consists of the IP address, device number, inode number, and block offset). GMS maintains a distributed directory that when given the UID for a page, can locate that page in global memory, if it exists. The key structures of that database are: (1) a per-node *page-frame-directory* (PFD) that describes every page resident on the node, (2) a replicated *page-ownership-directory* (POD) that maps a page UID to a manager node responsible for maintaining location information about that page, and (3) the *global-cache-directory* (GCD), a distributed cluster-wide data structure that maps a UID into the IP address of

[1] This is in sharp contrast to the case where different pages reside on different disks, in which case aggressive prefetching can be far from optimal.

[2] It should also be noted that in contrast to the results of [29], a page cannot be prefetched into global memory and then evicted before it is referenced: a page chosen for prefetch into global memory is always the then soonest non-resident page to be referenced by any process in the cluster, so the next global fault will not occur until after that page is referenced.

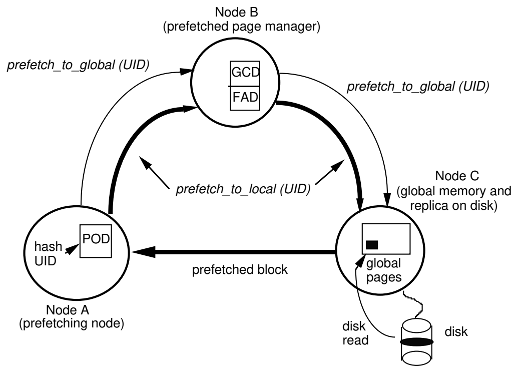

Figure 2: Communications for prefetch into global memory

a node caching a particular page. On a *getpage* request, GMS finds the manager node for that page and sends a request to that node; the manager checks whether that page is cached in the network, and sends a message to the caching node if so, requesting that it transfer the page to the original requester.

3.2 Mechanisms for implementing prefetching

PGMS extends the GMS implementation in three key ways. First, PGMS adds operations for prefetching blocks into any node's memory from disk or global memory. Second, PGMS modifies GMS mechanisms for distributing and maintaining global page information. Third, the PGMS policy module follows a hinted application through its predicted reference stream and uses epoch-based prefetch information for scheduling prefetching operations. The remainder of this section describes these three key aspects of our implementation.

At the lowest level, prefetching in PGMS is handled by two operations. The *prefetch_to_local* operation prefetches a page into local memory – if that page is in global memory, the page is fetched over the network, otherwise it is read from local disk. The *prefetch_to_global* operation prefetches into global memory from the disk on the global node. In our current prototype, prefetched files are replicated on the disks of multiple cluster nodes, thus allowing each of these nodes to prefetch from the same file independently. (Alternatively, the files could be striped across the disks.)

PGMS stores file replication information in a distributed directory called the *file-alias-directory* (FAD). For each replicated file, the FAD contains an entry that lists the IP address and local file name for each node storing a replica. The FAD entry for a file is stored on the manager node for the file's blocks (the FAD entry for shared files is replicated on every manager node). The FAD serves two key purposes: (1) PGMS uses the FAD to pick the nodes used to prefetch a file, and (2) the FAD extends the GMS UID to permit on-disk replication. While pages in GMS are named by a UID, the UID does not allow a page to be replicated on multiple disk locations, because each copy would be given a different name. In PGMS, a replicated file is assigned a primary location that determines the UID assigned to each of its pages; the FAD is then used in conjunction with the UID to support aliasing of a file to multiple storage locations.

Figure 2 shows the communications and data structures for the most general prefetch to global memory (a shared page). The thin arrows show the actions performed when node A issues a *prefetch_to_global* request. The request is first directed to the page's managing node, which knows if and where the page is cached in the network. If the page is not cached, PGMS picks the prefetching node from among the idle nodes that replicate the page's backing file. When a node receives a prefetch request, it reads the page from disk and caches the page in its own memory (as illustrated by node C in Figure 2).

The thick arrows show the actions performed later when the page is prefetched from global memory on node C to local memory on node A. Both actions must pass through the manager, because in the interim the global page may have moved.

3.3 Approximating the prefetching and caching algorithm

Our goal in approximating the algorithm of Section 2 is to provide a reasonable tradeoff between accuracy and efficiency. The key issue is guaranteeing the validity of global knowledge used by the algorithm and deciding when it must be updated.

We give only a high-level description of our algorithm approximation here due to length considerations. Our approach is similar to that used in GMS. The algorithm divides time into *epochs*, where each epoch has a maximum duration T (in practice, T is between 5 and 10 seconds). A coordinator node is responsible for collecting and distributing global information at the beginning of each epoch; the coordinator for the next epoch is elected as the "least loaded" node at the start of the current epoch.

At the start of the epoch, each node sends to the coordinator its CPU load and a summary of its *buffer values*. The CPU load on a node is an estimate of the CPU utilization seen by locally running processes. The value of a buffer, or equivalently the value of a page, is an estimate of the time until the next reference to the page stored in that buffer. The time until the next reference to a page is estimated on a per-process basis as follows. Future inter-reference CPU time is estimated from inter-reference CPU times measured in the recent past scaled by the percentage of time that the process was scheduled on the processor. The estimated time until the next reference to a hinted page is then the number of hinted references preceding it multiplied by the estimated future inter-reference CPU time. For unhinted processes, the time until the next reference to a page is estimated to be the time since the previous reference.

Using the information collected and the recent rates of evictions and prefetches, the coordinator computes a weight w_i for each node, representing the number of buffers on node i that are candidates for replacement by global prefetch requests and putpages (evictions) from other nodes during the epoch. Nodes whose CPUs are fully utilized are assigned a w_i value of 0, regardless of whether or not they have buffers of low value. The coordinator also determines the maximum buffer value, *MaxValue*, that will be replaced in the new epoch. To start the epoch, the coordinator sends the weight vectors w_i, and the value *MaxValue*, to all nodes in the cluster. The epoch terminates when either (1) the duration of the epoch, T, has elapsed, (2) $\sum_i w_i$ global pages have been replaced, or (3) the buffer value information is detected to be inaccurate.

During the epoch, nodes perform replacement and prefetching as follows:

- **Replacements:** When a page on a node must be replaced, the node selects its least-valuable page, p, for eviction. The node then forwards p to node i, where p becomes a global page in i's memory, replacing i's least valuable page. The target node i is chosen with probability proportional to w_i/N ($N = \sum_i w_i$). (If p is a shared page and a copy exists in another node's local memory, then p is simply discarded.) Roughly then, over an epoch the system will replace the N least valuable pages in the network.

- **Prefetching into local memory:** For each node j, whenever the prefetch predicate for a hinted process on j is true, and there are buffers on j of lower value than the ones that it wishes to consume for prefetching, the node issues prefetch requests, replacing its least-valuable pages.

- **Prefetching into global memory:** Processes running on each node, j, issue requests in round-robin order to other nodes k with $w_k > 0$ and request prefetches into global memory on their behalf. The request includes an estimate of the value of each page to be prefetched. To avoid global-memory thrashing, the system limits each node to B simultaneous outstanding disk prefetch requests, where B is a system parameter (we set $B = 8$ in our prototype). A node with fewer than B outstanding requests may issue a new batch of global prefetch requests.

  A given node k may receive several competing disk-to-global prefetch requests from other nodes. It initiates the prefetch request for the page p of highest value and, upon initiation of the prefetch, sends an acknowledgement to the requesting processor, say node j. This enables node j to update its hinted reference stream to indicate that p will soon be stored in global memory. The acknowledgement is also used to inform node j how many of the w_k low-value buffers have not yet been replaced.

The w_i values determine the rate at which global prefetch requests and evictions to global memory arrive at node i. We choose these values to meet two goals. First, we wish to minimize the overhead on each node due to global memory requests and to balance this load across the various nodes in the network. Load balance is important for global prefetching in order to achieve a high degree of I/O parallelism. Second, we wish to ensure that buffers supplied for eviction to global memory and global prefetching are of sufficiently low value, so that we do not replace useful data. This is particularly important for limiting any negative impact from aggressive or speculative prefetching into global memory.

4 Performance Measurements and Analysis

This section presents the measured performance of the PGMS system. We begin with low-level measurements of the implementation, and then present the performance of the prefetching mechanisms for several workloads. We then examine the performance characteristics of various aspects of PGMS in detail, using a rendering application as an example.

4.1 Experimental testbed

All measurements are from 266 MHz DEC AlphaStation 500 (Alcor) systems running Digital Unix 4.0, connected by a 1.28 Gb/s Myrinet network [5]. All nodes in each experiment use M2F-PCI32 (LANai-4) Myrinet adapters attached to a full-crossbar SW8 Myrinet switch. The disks from which all experimentes are performed are 7200 RPM ST32171W Seagate Barricuda drives. Pages and file blocks are 8KB, and reading a random 8KB page from disk takes an average of 13ms.

For optimum network performance we used Trapeze [31, 2] firmware for the Myrinet adapters. Trapeze uses an adaptive message pipelining technique called cut-through delivery [31] to minimize transfer latencies on the network in a manner similar to GMS subpages [20]. Using Trapeze, GMS can perform an 8KB page fault from remote memory in 165μs on platforms capable of delivering the full bandwidth of the 33 MHz 32-bit PCI bus. The Alcor is limited to 66 MB/s in the receiving direction, which increases raw page transfer times to 187μs. Including the PGMS overhead of generating a request and processing the reply, a global-to-local prefetch takes 370μs.

Operation	Node	Time (μs)		
		CPU	Net	Total
Request	Requester	67.9	—	67.9
	Manager	53.9	35	88.9
Prefetch	Prefetcher	46.3	187	233.3
Receive	Requester	22.3	—	22.3
Access	Requester	153	—	153

Table 1: Micro measurement of *prefetch_to_local()* operation (median times from 50 iterations). Note that, due to pipelining effects, the requester and prefetcher overlap part of their overheads.

Operation	Node	Time (μs)		
		CPU	Net	Total
Request	Requester	7.6	—	7.6
	Manager	40	35	75
Prefetch	Prefetcher	146	—	146

Table 2: Micro measurement of the *prefetch_to_global()* operation (median times from 50 iterations). Note that the prefetch request sent to the prefetcher is an asynchronous operation.

4.2 Microbenchmarks

Tables 1 and 2 detail the costs of the PGMS prefetching operations for the common case where the requester and manager are the same node. For a *prefetch_to_local* that returns a page from global memory, the times are divided into four components: issuing the prefetch request to the manager and target nodes (*Request*), processing the message and sending the prefetched page to the requester (*Prefetch*), receiving the page (*Receive*), and *Access*, the time to install the page in the local page map (as is normal for any read). In our PGMS experiments, all *putpage* and *getpage* operations (including *prefetch_to_local*) copy the page once on the client; these copies can be eliminated with an optimization [2]. The *prefetch_to_global* operation has two components: the time to generate the prefetch request (*Request*), and the time to initiate the disk request into remote host memory and process it when it completes (*Prefetch*). Where appropriate, the values are broken into the CPU overhead for each node (*CPU*), the overhead and latency of communication (*Net*), and the total overhead and latency (*Total*).

4.3 Application-level performance of PGMS

To characterize the application-level performance of PGMS under a wide range of workloads, we hinted a number of synthetic benchmarks and real applications and ran them on the PGMS prototype. We measured the following programs:

OO7 is an object-oriented database benchmark that builds and traverses a parts-assembly database [10]. Our experiments traverse an existing 100MB database mapped into memory, accessing approximately 65MB of data.

Render is a display engine that renders a computer-generated scene from a pre-computed 178MB database of tracing data [11]. Our experiments perform a sequence of operations to move the viewpoint progressively closer to the scene without changing the viewpoint angle, accessing approximately 100MB of data.

Hotcold accesses 512 random pages from a 20MB "hot" mapped file region, then 256 random pages from a 80MB "cold" region, and repeats for 20 phases. Every group of four pages read are accessed sequentially.

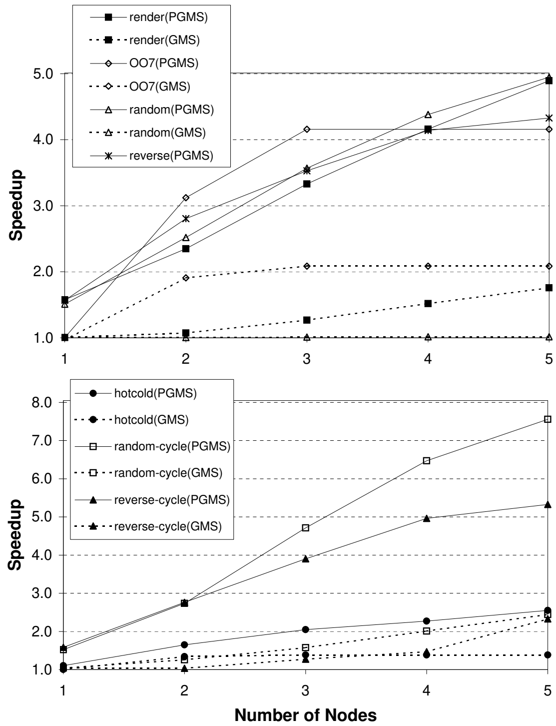

Figure 3: Application speedup on GMS and PGMS

Random randomly accesses all of the pages of a 100MB file.

Reverse sequentially reads a 100MB file in reverse.

The synthetic benchmarks first issue hints to inform the kernel of the pages or blocks they will access. In the experiments presented here, each benchmark computes for 250μs between each page or block access. To explore the effects of data reuse in the synthetic file benchmarks, we also measured "cyclic" variants that access their data sets three times before terminating.

Figure 3 shows the speedup of the synthetic benchmarks and applications on clusters ranging from 1 to 5 nodes. The top graph shows the speedup of the applications and synthetic benchmarks; the bottom graph shows the speedup of the benchmarks run cyclically. The speedups are relative to performance on a single node with no global memory and no prefetching other than the Digital Unix readahead mechanism for files accessed sequentially. All programs run on a single node with 64MB of memory, about 32MB of which is available for prefetching and caching. The other nodes in the cluster are idle and act as memory and disk servers, each again with about 32MB of available global memory and a single disk. For comparison, we also show performance under GMS (dashed curves).

For these applications, PGMS achieves speedups of 4 to 7 on 5 nodes. Using the Render application as an example, the Y intercept on the Render curve (Number of Nodes = 1) is the speedup for a single isolated node that is prefetching from its local disk only. In this case, we see that local-disk prefetching achieves a speedup of 1.6 over the non-prefetching version of the application. The 2-, 3-, 4-, and 5-node measurements show what happens as we add 1, 2, 3, and 4 idle nodes to the network, respectively. The curve demonstrates that PGMS is able to achieve nearly linear speedup when up to 4 idle nodes are used to service a memory-intensive application. With 4 idle nodes, PGMS achieves a speedup of 4.9 over a non-prefetching version of Render and 3.1 over a single node

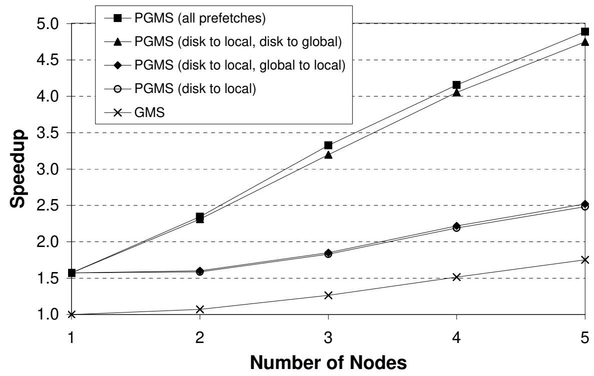

Figure 4: Speedup of Render application on PGMS

with local-disk prefetching. Although we were not able to perform the bulk of our experiments on a cluster larger than 5 nodes, we did investigate Render on larger clusters and found that its performance levels off at a speedup of 5.6 on 7 nodes.

Comparing the GMS and PGMS lines, we see that GMS achieves moderate speedups when the benchmarks access their data cyclically, yielding hits in global memory as pages are reused. However, parallel prefetching allows PGMS to achieve significant speedup for both non-cyclic and cyclic data access.

A common workload absent from these graphs is a sequential benchmark. Digital Unix and the Alpha I/O hardware is heavily optimized for sequential access, doing clustered readahead prefetches that provide user-level bandwidths of up to 8.3MB/s on our hardware. The PGMS prototype currently does not take advantage of these optimizations, and as a result performs no better than Digital Unix when doing sequential accesses.

The following sections focus on the behavior of a single application, Render, to examine the performance characteristics of various aspects of PGMS in detail. We chose Render because of its data size and relatively complex data access pattern. Render is a difficult application for global memory systems because of its large number of cold misses and its poor data locality, which reduces the value of both local and global data caching.

4.4 Performance breakdown of prefetch types

Figure 4 shows the performance of the hinted Render application with various prefetching options. The top curve shows the speedup of Render under PGMS relative to its performance on a single node with neither prefetching nor global memory, as described in the previous section.

The bottom curve of Figure 4 shows the performance of Render under raw GMS, i.e., global memory is used only to hold pages evicted from the active node's memory, without PGMS prefetching. No prefetching of any type is performed. With 4 idle nodes, full PGMS outperforms GMS by a factor of 2.8 for this application.

The middle curves show the effect of selectively enabling both types of PGMS prefetching in order to quantify the performance contribution of each type. The second curve from the bottom shows that adding local-disk prefetching to the basic GMS system improves performance by approximately 57% on a single node. As idle nodes are added to cache evicted pages, the speedup increases linearly. This second curve uses no global-to-local prefetching: page fetches from global memory are demand faulted, as in the original GMS. The third curve from the bottom, which nearly overlaps the second, shows that enabling global-to-local prefetching provides only slight benefit. This is for two reasons. First, global

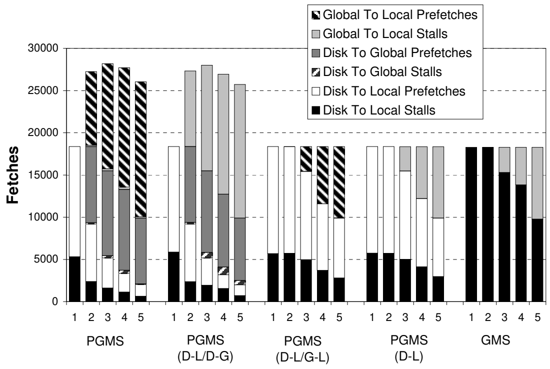

Figure 5: Prefetch request and stall breakdown for Render

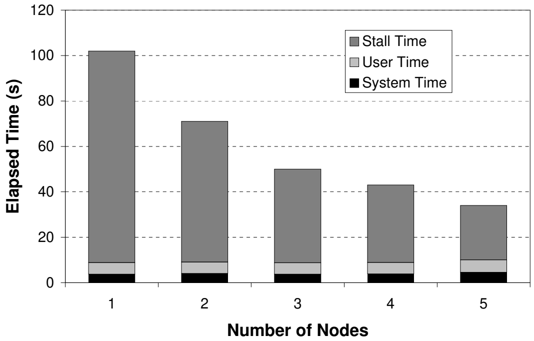

Figure 6: Execution time detail for Render

memory holds only evicted pages, and the GMS curve (bottom curve) shows that the benefit of caching evicted pages for Render is modest. Second, global memory faults using Trapeze/Myrinet are very fast relative to a disk access, reducing the I/O stall time to be saved by prefetching from global memory to local memory.

Finally, the top two curves show the large gains from adding disk-to-global prefetching in PGMS. The second curve from the top uses disk-to-global prefetching, but without global-to-local prefetching. This configuration suffers the cost of demand faults from global memory; however, global memory is used more effectively as a buffer for parallel prefetching of disk data ahead of its access. Thus, the benefit relative to the previous curve (with global-to-local prefetching but without disk-to-global prefetching) comes from replacing disk stalls with global memory stalls. The top curve shows the final performance improvement from eliminating most of these global stalls with global-to-local prefetching. Again, the incremental gains from global-to-local prefetching are much smaller than the gains from using global memory on a fast network to begin with.

4.5 Detailed breakdown of prefetch types

Figure 5 shows a detailed breakdown of the total number of fetches performed in the 1- through 5-node cases for each of the configurations shown in Figure 4, in order from fastest (on the left) to slowest (on the right). For each prefetch type we show counts of both the prefetch requests that arrived in time to avoid a stall (*prefetches*) and those that did not (*stalls*).

Note that the height of each bar shows the number of fetches, and not performance. In particular, the different bar components have different performance costs, and some blocks are fetched twice in the faster PGMS configurations: once from disk into global memory, and once from global memory to local memory. The three rightmost sets of bars show the slower configurations with no disk-to-global prefetching, counting only fetches into the active node's memory. Since the size of the active node's memory is the same across all experiments, the total number of fetches remains constant. The two leftmost sets of bars show additional fetches resulting from PGMS disk-to-global prefetching.

The rightmost set of bars shows Render's performance on raw GMS. Render's working set is sufficiently large and its locality sufficiently poor that the addition of the first idle node does not yield any hits in the global cache; i.e., none of the pages evicted from local memory and cached in global memory are accessed again before being evicted from global memory. With the addition of the second idle node, the size of the global cache grows enough to eliminate some disk stalls.

The next two sets of bars from the right show the effects of adding disk-to-local and global-to-local prefetching to GMS, respectively. The second set of bars from the right show that the addition of local disk prefetching to GMS eliminates many of the local disk stalls not already eliminated by the global cache. This is in part because of the large number of cold misses in our Render test. The middle set of bars show that the addition of global-to-local prefetching eliminates all of the remaining stalls on global memory fetches, improving performance further.

The leftmost two sets of bars show the effect of adding disk-to-global prefetching, in order to benefit from the combined effects of global memory and I/O parallelism from the use of disks on the added nodes. As nodes are added, disk-to-local prefetches and stalls are reduced to their lowest levels, replaced by disk-to-global and global-to-local fetches. In the leftmost set, for full PGMS, the global-to-local fetches are prefetched, eliminating all but a few I/O stalls occurring at the start of the experiment. With 4 nodes, global memory is large enough to store Render's entire data set, and the number of disk-to-global prefetches drops significantly, leading to a reduction in the total number of fetches performed.

4.6 Elapsed time breakdown

Figure 6 shows a breakdown of the execution time for Render on the active node under full PGMS as a function of the number of nodes cooperating. The user CPU time component is constant since the program executes the same computation in all configurations; our prefetching techniques only reduce I/O stall time. In the first bar, we see that, despite prefetching from the local disk, stall time is substantial on a single node. Adding additional nodes reduces stall time enormously, in exchange for a small increase in system time that includes the time spent in the PGMS prefetching code and the network driver.

Figure 7 gives more insight into Render's behavior by showing the causes of the PGMS prefetch requests. For a single node, disk fetches occur for two reasons: the cold misses are the unavoidable disk reads for the first access to each block in the dataset, while capacity misses are caused by insufficient memory to cache the entire dataset. Capacity misses are unchanged when a second node is added, since (as previously noted) the second node provides insufficient global memory to deliver global cache hits given Render's working set size and locality. However, with the second node these capacity misses now result in disk-to-global rather than disk-to-local prefetches, followed by global-to-local prefetches as shown in the top portion of the second bar. As additional nodes are added there is sufficient global memory to reduce capacity misses, and the count of global-to-local prefetches rises due to effective GMS caching of evicted pages in the idle cluster memory.

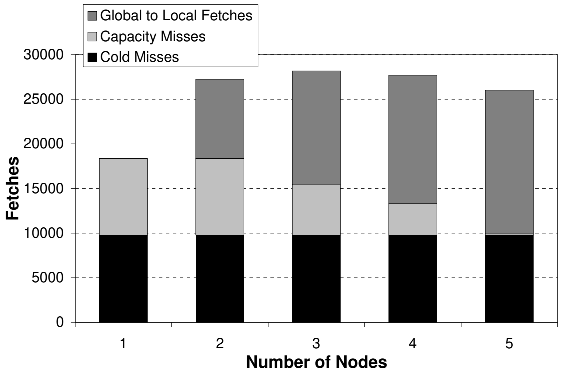

Figure 7: Breakdown of cold and capacity misses for Render

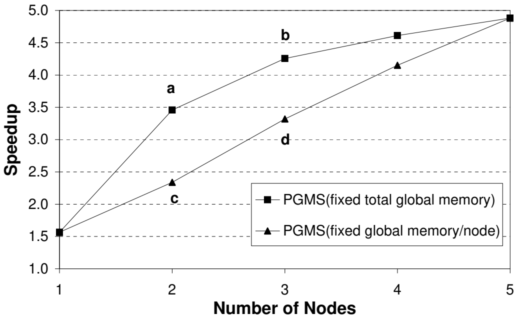

Figure 8: Fixed total global memory size vs. fixed per-node global memory size

4.7 The impact of global memory size

A key question in global memory systems is the tradeoff between adding more global memory to an existing node vs. adding more nodes. To explore this tradeoff, we ran an additional set of experiments, shown in Figure 8. The lower curve shows the PGMS data previously presented; in these measurements, each idle node added an additional 32MB of global memory to the cluster. In the new experiment shown by the upper curve in Figure 8, we kept global memory size constant, independent of the number of idle nodes (i.e., total global memory size was always 128MB, divided evenly among the idle nodes). Comparison of points *a* and *c* shows that increasing memory on the single idle node (by 96MB) had a substantial performance impact, improving speedup by 48%. With 2 idle nodes (points *b* and *d*), increasing global memory size from 64MB to 96MB improved speedup by 28%. These improvements are not surprising, given what we've seen of Render's locality. By comparing points *a* and *b*, we can isolate the impact of an additional CPU and disk, independent of global memory size. In this case, we see an improvement of 23% due to the additional parallelism and bandwidth added by that CPU and disk resource. Thus, speedup in PGMS is caused both by the addition of global memory and by the additional parallelism provided by idle nodes. The exact benefit of either will of course depend on the nature of the applications.

4.8 Interaction of competing processes

It is interesting to examine the effect of prefetching when a prefetching process competes for resources. The left side of Figure 9 shows the effect of running two competing processes simultaneously on the same node. In one case we run two unhinted Renders; in the other case we run one hinted and one unhinted Render. There is one idle node in the system, and thus the two processes in each case are competing for both local and global resources. The line on the bars shows the performance for each application when running standalone in the same configuration. (While we showed two instances of the same program in Figure 9, experiments using different programs show similar results.)

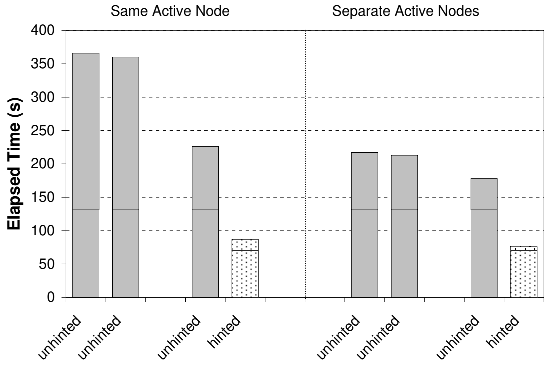

Figure 9: Elapsed times for two Render processes executing simultaneously.

Running the two unhinted applications together takes longer than running them sequentially, i.e., they suffer from the competition for memory and disk, causing each to run slower than 2 times its single-node time. Running the unhinted application along with a hinted application reduces its execution time substantially (compared to the previous example); in this case the hinted process is essentially unaffected by the unhinted process and completes quickly, leaving all of local and global memory for the unhinted process to use. The result is nearly optimal, as in shortest-job-first scheduling.

The right side of Figure 9 shows two similar experiments, however in this case each application runs on a different active node; they again share a single idle node, and thus they compete for global memory only. For the two unhinted applications running simultaneously, the elapsed time of each of the processes is within 1% of the elapsed time of an unhinted process running by itself in a system with one idle node with half the global memory, i.e., the sharing of the idle node is nearly perfect. For the hinted/unhinted experiment the results are similar to the single-node case; the hinted application finishes quickly, leaving the global resources to the unhinted application, which finishes relatively quickly afterwards.

4.9 Summary

In this section we presented measurements of Render in various system configurations. From these measurements we saw the following results for this application:

1. Global prefetching and caching can provide significant speedup over local-disk prefetching alone.

2. Disk-to-global prefetches can substantially improve performance by adding additional disk parallelism, and by turning disk misses into global memory fetches, which have much lower latency.

3. Compared to prefetches from disk, global-to-local prefetches are less significant because global fetch time is small (and shrinking with newer network technologies).

4. For some applications, increasing global memory size may have more impact than adding processors.

5. Running hinted and unhinted applications together does no harm to the unhinted application (when compared to running with another unhinted program). Hinted applications make more effective use of resources.

6. Applications experience both cold misses and capacity misses, depending on the memory requirements of the application and the configuration of the system. Caching in global memory reduces the cost of capacity misses; prefetching reduces the cost of both cold and capacity misses. The Render application that we used has a large number of cold misses, and thus is aided more by prefetching than global memory caching.

Taken together, these results show the advantages of combining prefetching and global memory in a cluster.

5 Comparison to Previous Work

Our work unites several independent threads of previous research in prefetching, global memory systems, and network I/O. We describe below key research efforts in these areas and briefly contrast our efforts with those.

Early studies showed the possibility of remote paging using dedicated or temporary paging servers [12, 15, 19]. More recent efforts have examined the use of network memory in more dynamic environments. Dahlin et al. [13] describe the use of remote memory in the XFS file system, which permits file system clients to benefit both from idle memory and shared file pages on other nodes. The GMS system described by Feeley et al. [14] places a global memory layer underneath both the file and virtual memory systems, which allows both to benefit transparently from "old" pages in the network. Sarkar and Hartman [28] showed how hints could be used to approximate global information in remote memory systems such as GMS and XFS. Franklin et al. [16] evaluate a client-server DBMS system in which the server can fetch requested data from clients' memories. In all of these systems, applications benefit from shared data that exists in global memory, or from evicting pages to global memory. Our work builds on systems like XFS and GMS. In contrast, however, PGMS uses idle cycles on network node to aggressively prefetch data for future requests into global memory, in order to reduce future stalls. This greatly increases the total hardware bandwidth (CPU, network, disk, and memory) available to highly-active applications.

Several studies of prefetching considered file system resource management for a single processor with a parallel disk array when application-disclosed access patterns (hints) are available. Researchers have shown that many I/O-intensive applications have predictable access patterns and can therefore provide such hints [27, 1, 23], while Mowry et al. [25] showed that the compiler can automatically generate hints. Patterson et al. [27] manage allocation of cache space and I/O bandwidth between multiple processes, a subset of which are hinted; they apply cost-benefit analysis to estimate the impact of alternative buffer allocations. Cao et al. [7] considered the integration of prefetching, caching, and disk scheduling in the single-disk case; their Aggressive prefetching strategy is provably near-optimal for a single disk [6], but can be suboptimal for data striped across multiple disks [21]. Kimbrel et al. [22] studied combined prefetching and caching strategies for multiple-disk systems executing a single process; their Forestall algorithm adapts the aggressiveness of prefetching to the extent to which performance is limited by I/O stalls. Tomkins et al. [29] compared the performance of the LRU-SP algorithm [8, 7] and cost-benefit algorithms for allocating I/O and cache resources among prefetching and non-prefetching processes. PGMS builds on several of these results, in particular, Cao et al.'s Aggressive algorithm and Kimbrel et al.'s Forestall algorithm. PGMS differs from all of this work, however, in that it performs prefetching in the context of a three-level storage hierarchy (local memory, global memory, and disk), and attempts to benefit from the parallelism available on multiple nodes and disks in the network.

Finally, efforts such as Zebra [18], TickerTAIP [9], and XFS [3] use multiple nodes to increase parallelism for remote file access. PGMS differs from these in its use of active prefetching and caching into remote memory.

6 Conclusions

This paper presented PGMS, a system using cooperative prefetching and caching in a network-wide global memory system. Cooperative prefetching permits multiple network nodes with idle CPU cycles and memory pages to cooperate in prefetching on behalf of active nodes. This prefetching to global memory can reduce stall time without the risks of aggressive prefetching on the active nodes.

We have developed a hybrid algorithm for PGMS that combines aggressive prefetching into global memory with more conservative prefetching into local memory. We designed and implemented an approximation to that algorithm on the Digital UNIX operating system running on a small cluster of Myrinet-connected DEC Alpha workstations. Our results show that significant speedups can be achieved using cooperative prefetching for memory-bound applications. As well, we quantify the impact of various types of prefetching: disk-to-local, disk-to-global, and global-to-local.

As network technology advances, we expect that the ratio between disk and network transfer times will continue to increase, and therefore using network resources to reduce the I/O bottleneck will be even more crucial in future generations. The PGMS experiments show how the integration of prefetching and global memory technologies permits multiple idle network resources to be used in parallel to benefit memory-bound jobs on active nodes in the cluster.

Acknowledgements

We are especially grateful to Andrew Gallatin for spending a long weekend running the final benchmarks for this paper on the Duke Myrinet/Alpha cluster. Darrell Anderson, Andrew Gallatin and Ken Yocum contributed Trapeze-related code and patiently answered many questions about Digital Unix internals and the workings of Trapeze and Myrinet. Alec Wolman, Darrell Anderson, Gretta Bartels, and Wei Jin commented on drafts of this paper.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O-intensive parallel applications. In *Proc. of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, May 1996.

[2] Darrell Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrin et. In *Proceedings of the 1998 USENIX Technical Conference*, June 1998.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. *ACM Trans. on Computer Systems*, 14(1), February 1996.

[4] L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 1966.

[5] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K Su. Myrinet - a gigabit-per-second local area network. *IEEE Micro*, February 1995.

[6] P. Cao, E. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1995.

[7] P. Cao, E. Felten, A. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. on Computer Systems*, 14(4), November 1996.

[8] P. Cao, E. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proc. of the First Symp. on Operating Systems Design and Implementation*. USENIX, November 1994.

[9] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. *ACM Trans. on Computer Systems*, 12(3), August 1994.

[10] M. J. Carey, D. J. Dewitt, and J. F. Naughton. The oo7 benchmark. In *Proceedings of the ACM SIGMOD Internation Conference on Management of Data*, May 1993.

[11] 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, University of Washington, May 1995.

[12] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Proc. of the USENIX Summer Conf.*, June 1990.

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

[14] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Sympoisum on Operating Systems Principles*, December 1995.

[15] E. 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, March 1991.

[16] M. Franklin, M. Carey, and M. Livny. Global memory management in client-server DBMS architectures. In *Proc. of the 18th VLDB Conf.*, August 1992.

[17] G. Gibson, D. Nagle, K. Amiri, F. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, June 1997.

[18] J. Hartman and J. Ousterhout. The Zebra striped network file system. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, December 1993.

[19] L. Iftode, K. Petersen, and K. Li. Memory servers for multicomputers. In *Proc. of the IEEE Spring COMPCON '93*, February 1993.

[20] Hervé A. Jamrozik, Michael J. Feeley, Geoffrey M. Voelker, James Evans II, Anna R. Karlin, Henry M. Levy, and Mary K. Vernon. Reducing network latency using subpages in a global memory environment. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[21] T. Kimbrel and A. Karlin. Near-optimal parallel prefetching and caching. In *Proc. of the 1996 IEEE Symp. on Foundations of Computer Science*, October 1996.

[22] T. Kimbrel, A. Tomkins, R. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-drive comparison of algorithms for parallel prefetching and caching. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*. USENIX, October 1996.

[23] D. Kotz and C. Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1), January 1993.

[24] E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proc. of the 7th Conf. on Arhcitectural Support for Programming Languages and Operating Systems*, October 1996.

[25] T. Mowry, A. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*. USENIX, October 1996.

[26] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks. In *Proc. of the 1988 ACM SIGMOD Conf. on Management of Data*, June 1988.

[27] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. of the 15th Symp. on Operating Systems Principles*, December 1995.

[28] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*. USENIX, October 1996.

[29] A. Tomkins, R.H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In *Proc. of the ACM International Conf. on Measurement and Modeling of Computer Systems*, June 1997.

[30] G. Voelker, H. Jamrozik, M. Vernon, H. Levy, and E. Lazowska. Managing server load in global memory systems. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, June 1997.

[31] Kenneth G. Yocum, Jeffrey S. Chase, Andrew J. Gallatin, and Alvin R. Lebeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–252, August 1997.