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, the US-Israel Binational Science Foundation (BSF), 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.

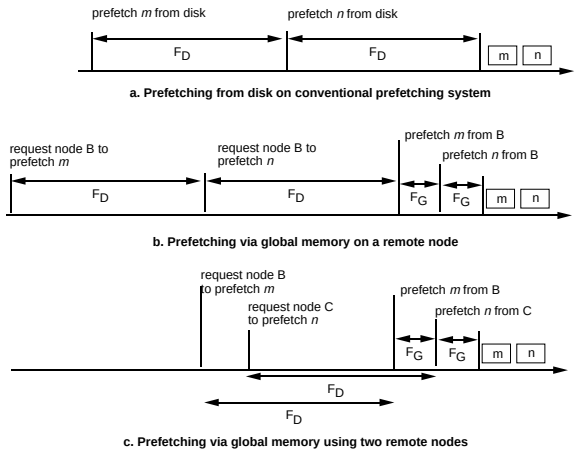

Figure 1: Prefetching in conventional and global-memory systems

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 ($\sum_{1 \le i \le j} F_i$) 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 $\sum_{1 \le i \le j} F_i \ge t_{b[j]}$. 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,

$$C_{GLFD}(R) \le C_{OPT}(R)\,(1 + 3(F_G/F_D))\,.$$

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 $(F_G/F_D) \le 0.02$, 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.

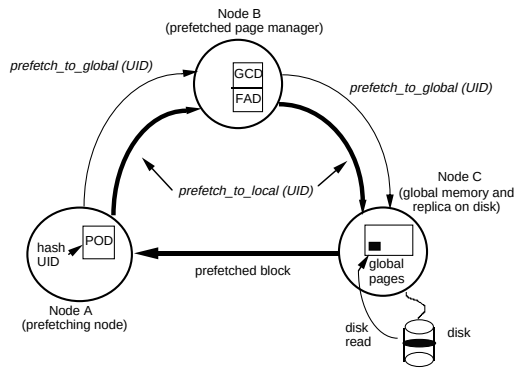

Figure 2: Communications for prefetch into global memory

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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.