Managing Server Load in Global Memory Systems

Geoffrey M. Voelker, Hervé A. Jamrozik†,
Mary K. Vernon*, Henry M. Levy, and Edward D. Lazowska

Department of Computer Science and Engineering
University of Washington

*Computer Sciences Department
University of Wisconsin - Madison

Abstract

New high-speed switched networks have reduced the latency of network page transfers significantly below that of local disk. This trend has led to the development of systems that use network-wide memory, or *global* memory, as a cache for virtual memory pages or file blocks. A crucial issue in the implementation of these global memory systems is the selection of the target nodes to receive replaced pages. Current systems use various forms of an approximate global LRU algorithm for making these selections. However, using age information alone can lead to suboptimal performance in two ways. First, workload characteristics can lead to uneven distributions of old pages across servers, causing increased contention delays. Second, the global memory traffic imposed on a node can degrade the performance of local jobs on that node.

This paper studies the potential benefit and the potential harm of using load information, in addition to age information, in global memory replacement policies. Using an analytic queueing network model, we show the extent to which server load can degrade remote memory latency and how load balancing solves this problem. Load balancing requests can cause the system to deviate from the global LRU replacement policy, however. Using trace-driven simulation, we study the impact on application performance of deviating from the LRU replacement policy. We find that deviating from strict LRU, even significantly for some applications, does not affect application performance. Based upon these results, we conclude that global memory systems can gain substantial benefit from load balancing requests with little harm from suboptimal replacement decisions. Finally, we illustrate the use of the intuition gained from the model and simulation experiments by proposing a new family of algorithms that incorporate load considerations as well as age information in global memory replacement decisions.

1 Introduction

With new high-speed switched networks, such as ATM, the latency of a page transfer from a remote node is significantly less than the latency for reading the page from local disk. This change in the efficiency of storage access has led to the development of systems that use network-wide memory, or *global* memory [2, 3–9]. By managing memory globally in the network, active nodes can benefit from "idle" memory available on other nodes; in effect, the memories of idle or lightly-loaded machines are used as a cache for replaced memory pages or file blocks of active nodes. A global memory system attempts to exploit all memory on local-network nodes, in order to reduce the average access time of page faults and file block reads for all applications.

A crucial issue in the implementation of global memory systems is the selection of the target nodes to receive evicted (i.e., paged-out) pages. For example, in the xFS cooperative caching system [3], pages displaced from one node are sent randomly to other nodes, using an algorithm called *N-chance* to recover from bad selections. In the GMS global memory system [4], the system maintains global information describing the ages of pages on all cluster nodes; using this information, GMS attempts to implement an approximation to network-wide global LRU replacement. Studies of GMS show that the use of global information improves performance relative to a random algorithm. Recently, Sarkar and Hartman [11] have proposed a fully distributed information exchange (piggybacked on global page forwarding operations) to gather less complete but perhaps more up-to-date page age information, leading to a different approximation of global LRU replacement.

In any case, a global memory system seeks to choose the "best" target node to receive a displaced page. However, using page age information alone for this selection, or even using an optimal page replacement strategy, can lead to suboptimal performance, in two ways. First, dynamic workload characteristics can lead to short-term concentrations of old or young global pages at one or a few nodes providing global memory, which can increase the latency for remote page requests of the applications using global memory. Second, the load imposed on any given node that is handling global memory traffic may degrade the performance of the local jobs on that node. For example, experiments with GMS showed that a node supplying pages to seven active nodes used 56% of its CPU cycles just for servicing the remote page requests. This situation runs counter to the ideal cluster design, where local performance is also key.

This paper studies the potential benefit and the potential harm of using memory server load information, in addition to age information, in global memory replacement policies. By considering load information in the replacement algorithm, global memory systems can reduce contention at servers and limit CPU utilization by remote requests. There are tradeoffs to the use of load information, however: if load considerations cause replacement of a "young" page, then this replacement may increase the number of disk accesses relative to the original replacement algorithm. Using an analytic

† Author's current address is Amazon.com, Seattle, WA.

This research was supported in part by grants from the National Science Foundation (EHR-95-50429, CCR-9024144, CCR-9200832, CCR-9632769, and MIP-9632977), from Digital Equipment Corporation, and from Intel Corporation.

queueing network model and trace-driven simulator, we study this tradeoff in detail in several ways.

First, the analytic model shows the extent to which uneven request distribution among servers can degrade memory request performance for different idle node configurations. Second, the model shows the benefits that can be gained by load balancing requests, ignoring the impact of altering the replacement policy. Using the simulator, we study the qualitative impact on application performance of deviating from the LRU replacement policy and of using new idle nodes in a load-balanced fashion. We find that deviating from strict LRU, even significantly for some applications, does not affect application performance. Based upon these results, we conclude that global memory systems can gain substantial benefit from load balancing requests with little harm from suboptimal replacement decisions, as long as newly idle nodes are used more aggressively than strict load balancing would allow. Finally, we illustrate the use of the intuition gained from the experiments by proposing a new family of algorithms that incorporate load considerations as well as age information in global memory replacement decisions.

The results of our study are applicable to global memory systems that implement approximate global LRU replacement using varying types of global information. Furthermore, the general insights about the benefit of balancing load at minimal costs due to less perfect replacement decisions, as well as the new family of algorithms suggested by these insights, can also be applied to global memory systems that use replacement algorithms different than LRU.

The remainder of this paper is organized as follows. Section 2 describes the global memory systems we study in this paper. Section 3 describes the queueing network model and trace-based simulator used in this study. In Section 4 we present performance results that show the extent to which load balancing can benefit global memory systems, and the impact of deviating from LRU replacement. Section 5 studies the performance issues involved in using non-idle nodes, in addition to idle nodes, as memory servers. We then present our new family of global memory replacement algorithms in Section 6 that incorporate server load information as well as page age information into the page replacement decision. Finally, Section 7 summarizes our results and conclusions.

2 Global Memory Systems

Our study is based on global memory systems such as the *Global Memory Service* (GMS) described and implemented in [4] or the recently proposed cooperative caching systems for file I/O [3, 11]. The use of global memory is transparent to applications and is managed entirely by the operating system. GMS, for example, implements global memory at the lowest level in the operating system; thus global memory is available for virtual memory backing store, mapped files, and the file buffer cache.

Each node in a global memory system can contain both *local* pages and *global* pages. A local page is one that contains data for an application executing on the same machine; a global page is one that contains data for an application executing on another machine. The balance between local and global pages on a node changes over time, depending on the behavior of that node and others. When a node becomes idle, its pages will age, and eventually will be replaced by younger global pages displaced from other nodes. When an idle node begins computing again, it will increase its use of local memory, forwarding or discarding global pages as it does so. Eventually, if the applications on the node have high memory demands, the node will fill its memory with local pages and will begin to use global memory on other nodes as it replaces its oldest local pages.

The global memory across the nodes in the cluster forms a shared cache between local memory and disk. (Note that pages in local memory are never shared). Coherency in this cache can be handled, for example, by flushing dirty local pages to disk before placing them in global memory at another node, as is done in GMS. Pages in global memory can then be shared among nodes without any locking. In practice, the code pages of an application are shared by multiple instances of the application executing on different nodes, while the data pages are private.

Current global memory systems [4, 11] maintain approximate global age information for both local and global pages, and use this age information to manage the memory in the cluster as a single resource. When a node needs to replace a page, it locally chooses a target node based on its global age information. The replaced page is then sent to that target node, where it becomes a global page. We describe the gathering of global age information further in Section 6.

In general, there are two classes of jobs that run in a global memory system: *local jobs*, whose memory requirements are completely satisfied by local memory resources on the node on which they run, and *global jobs*, which benefit from the use of global memory while executing. Given these two job classes, we can separate the nodes in the system into three classes:

- *local nodes*, which are running local jobs only,
- *global nodes*, which are running global jobs (and possibly local jobs as well), and
- *idle nodes*, on which no jobs are running, but which may provide global pages for global jobs on other nodes.

Note that in general a global node is running exactly one global job and perhaps also some local jobs as well.

Idle and local nodes can act as *memory servers* for global nodes, in the sense that a portion of their memory can be allocated as global memory. Global jobs running on global nodes make global memory requests, and these requests are sent to the idle or local nodes managing the global memory for these jobs. These requests come in two forms, *getpage* requests and *putpage* requests. A getpage is a request to read a global page from another node's global memory, e.g., in response to a page fault or file cache miss. The global node thus sends a getpage request to the remote node to ask for the page, and the remote node returns it. A putpage is a request to store a page in another node's global memory, e.g., as the result of a page replacement decision. The global node sends a putpage request along with the page to be stored.

In GMS, a node cannot both be a global memory server and a global memory client. Furthermore, in any global memory system, a global node that is an active global memory client will not simultaneously have enough memory to be a significant memory server for other nodes.

3 Model and Simulator

This section describes the queueing network model and the simulator we use in this study. Our first goal is to characterize the performance of getpage requests (remote page requests) in the face

Application	Description	Input	Mem Refs (10^6)	Footprint (pages)	Z (ms)
m3	Modula 3 compiler	smalldb	87	773	2.7
ld	Linker	Digital Unix 3.2	102	6807	0.36
atom	Binary rewriting tool	kproc tool on gzip	117	1175	1.5
render	Graphics renderer	model of tree	245	1433	4.1
gdb	GNU debugger	init phase	0.5	138	0.17
007	OODB benchmark	synthetic workload	–	–	5.0

Table 1: The applications used in this study.

of contention at memory servers. By doing so we can show the potential benefits of load balancing global memory requests across servers to reduce contention. We use a queueing network model and Approximate Mean Value Analysis (AMVA) for this experiment, because it is a natural tool for measuring the effect of contention on getpage response time. The queueing network analysis is also useful for studying the interference that local jobs experience from global memory requests. The AMVA model does not, however, take into account the effect of deviating from a strict LRU replacement policy when managing global pages. Therefore, our second goal is to measure the effect on application performance when global pages are not replaced using strict LRU. We use the simulator for this experiment because it allows us to measure and compare deviations from strict LRU in terms of the number of page faults and execution time for a set of applications.

The applications used with the AMVA model and in simulations in this paper are summarized in Table 1. For each application, the table gives the name and a description of the application, the input used to drive it, the number of memory references made and memory footprint, and the average time between global memory requests.

The number of memory requests is the total number of loads and stores made by the application. The memory footprint is the total number of pages touched by the application during its execution; in other words, this is the total number of local and global pages that would be required to prevent the application from having to page to disk. The average time between global memory requests is the resulting fault rate to global memory when GMS is configured to be able to hold 1/4 of an application's footprint in local memory and 3/4 of the footprint in global memory. We chose this ratio of local to global memory as a reasonable approximation to the resources applications will likely find available in real global memory systems. These parameters were all derived from simulation experiments, except for the 007 benchmark [1] whose parameters were obtained from the GMS prototype. Note that, since we don't have a trace for 007, we don't have measurements of the number of memory references or its memory footprint.

3.1 MVA Model of Global Memory Requests

We use the 2-class queueing network model shown in Figure 1 to model global memory request behavior, and use approximate mean value analysis to compute performance metrics of interest. The model is designed to measure the effect of contention on getpage latency and the CPU utilization of memory servers. To accomplish this, a closed class of customers in the network represents the global nodes, each executing one global job that alternates between executing and sending a synchronous getpage request to an idle or local node. The open class of customers represents the asynchronous putpage requests that are issued by the global nodes. The effect of server contention on getpage requests can be estimated by solving for the response time of the getpage requests. Memory server utilization can be estimated by solving for the utilization of the server queues in the network.

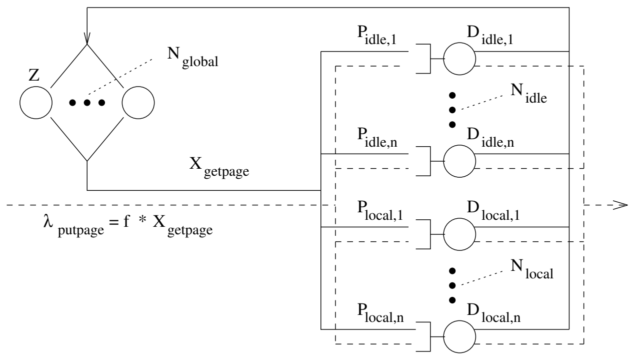

Figure 1: Queueing network model of the Global Memory Service.

The input N_{global} denotes the number of global nodes. Each global node is executing a single global job as well as possibly some local jobs. The center with average delay Z corresponds to time the global nodes spend executing between a getpage response and the next getpage request by the global job. The other queues correspond to idle and local nodes in the system; these are the nodes that serve the getpage and putpage requests. Note that in the simple case we assume that global nodes are homogeneous; that is, all global nodes have the same average delay between getpage requests and the same routing probabilities for the requests. Routing probabilities are perhaps expected to be homogeneous because they depend on the page forwarding algorithm, which is the same for all nodes. However, the model is easily extended to handle non-homogeneous request rates or routing probabilities.

Since putpage requests are asynchronous operations (a process can continue to execute while the page is being sent to the server), we model them as an open customer class. Their arrival rate is denoted by $\lambda_{putpage}$, which is a constant f times the getpage throughput $X_{getpage}$, where f is an input to the model. Note that $X_{getpage}$ is an output of the model, which means that we must solve the model iteratively to determine the correct value for $\lambda_{putpage}$ as well as the desired output measures.

Both getpage and putpage requests flow into and out of a subsystem consisting of two sets of service centers, one set corresponding to the idle nodes in the system and the other corresponding to the local nodes in the system. Note that, although we distinguish between idle and local service centers in the model notation and routing probabilities, each type of node services getpage and putpage requests

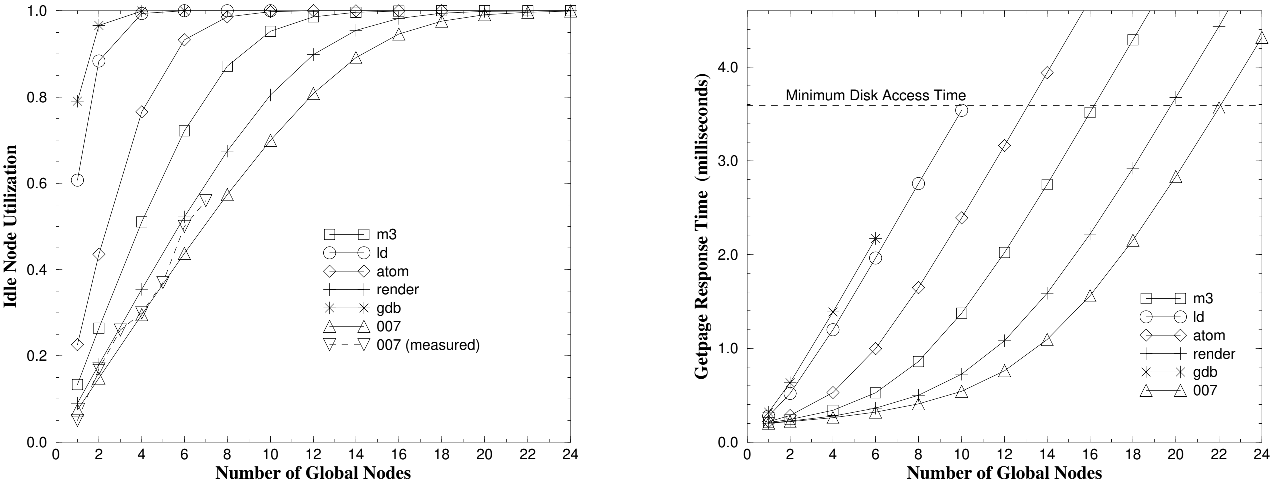

Figure 2: Utilization of the idle node and response time of a getpage request versus number of global nodes.

in the same way due to the priority of global requests. That is, since operating systems typically manage memory in the kernel, a global memory system will service requests in the kernel. Consequently, when a global request arrives at a local node, the request gets top priority at the local node and is immediately scheduled ahead of any local work on that node. As with idle nodes, the only interference a global request will encounter on a local node will be due to other global requests being serviced at that node at the arrival instance. Given their priority, global requests can therefore impact the performance of local jobs on the node; this impact is investigated in Section 5.

The number of idle nodes is denoted by N_{idle}, and the number of local nodes is denoted by N_{local}; both parameters are inputs. The service demands of the idle and local service centers are defined to be $D_{idle,i}$, $1 <= i <= N_{idle}$, and $D_{local,l}$, $1 <= l <= N_{local}$, respectively. Note that these service demands are inputs that apply to both getpage and putpage requests.

There are also two sets of probabilities associated with the service centers, $P_{idle,i}$, $1 <= i <= N_{idle}$, and $P_{local,l}$, $1 <= l <= N_{local}$. These parameters correspond to the probability that a request will go to the associated service center. With these probabilities, we can control the distribution of requests among the service centers within a set as well as the distribution between the two sets of service centers. The former enables us to model arbitrary request distributions among idle nodes in Section 4.2, and the latter to model uneven request distributions between idle and local nodes in Section 5. Note that both getpage and putpage requests share these probabilities, and that the sum of both sets of probabilities equals 1. These probabilities are also inputs.

3.2 Model Parameterization and Validation

We parameterize the model using workload parameters both from an implementation of the Global Memory Service [4] and from results from our simulator, which is described subsequently in Section 3.3.

Based upon measured values reported in [4], we set the service demands of both idle and local service centers to 194μs. Since we assume putpage and getpage requests have the same demand, and the page size never changes, this service demand never changes for any of our experiments. Based upon simulation results, we have found that, in a cluster running a heterogeneous workload, the number of getpage and putpage requests are nearly equal. We therefore set $f = 1$ to correspond to an equal number of putpage and getpage requests. The think time Z is set according to the application being modeled; values for each application are listed in Table 1. Unless otherwise specified, the probability distributions are uniform across all service centers. The other inputs, such as the number of each kind of nodes, are set specifically for each experiment with the model.

Using these inputs and the average request rate for the 007 benchmark, we can now perform an experiment similar to the one performed in [4] and compare the results. In the experiment, we simply vary the number of global nodes (N_{global}) making requests to a single idle node ($N_{idle} = 1$). The experiment performed on the prototype measures the utilization of the idle node as the number of global nodes ranges from 1 to 7.

The results of this experiment from the MVA model, as well as the prototype, are shown in Figure 2. This graph plots the utilization of the idle node and response time of a getpage request versus the number of global nodes in the system. To put the getpage response times in perspective, the fastest disk page fault time of 3600μs reported in [4] is shown on the response time graph. As with the prototype, the model exhibits linear growth in idle node utilization when there are fewer than 8 global nodes. Furthermore, with 7 global nodes, the measured idle node utilization of the prototype was 56%, and the model reports a utilization of 51%, which is in surprisingly close agreement, particularly considering the level of abstraction in the model compared to the actual system implementation. This degree of agreement between measured system performance and the MVA model calculations gives a reasonable amount of assurance that the model is generating good results regarding the impact of contention on the average performance metrics we are interested in.

3.3 Trace-Driven Simulator

Our simulator models the memory behavior of applications executing in a global memory environment. It takes as input memory reference traces generated from applications instrumented by Atom [13]. It then simulates these memory references and models the effect of

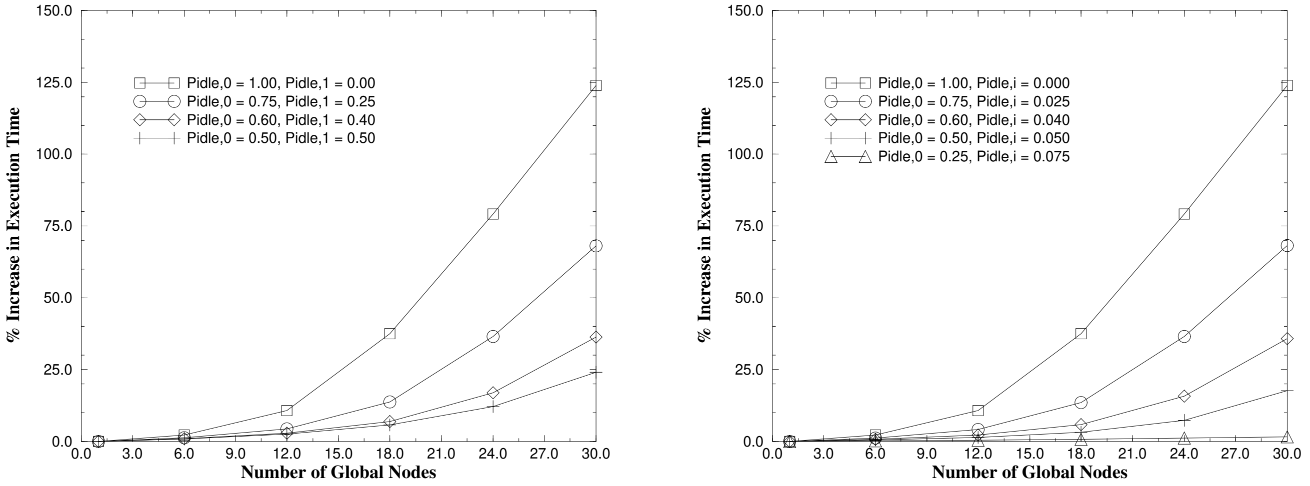

Figure 3: Percent increase in execution time due to uneven distributions of requests across idle nodes. The first graph corresponds to a network with 2 idle nodes; with probabilities $P_{idle,0}$ and $P_{idle,1}$, requests go to one idle node or the other. The second graph corresponds to a network with 11 idle nodes; with probabilities $P_{idle,0}$ and $P_{idle,i}$, requests go to the first and each of the remaining 10 idle nodes, respectively.

paging to both local disk and to remote memory. Paging policy is determined by a configurable memory management module; an LRU policy with global knowledge is used by default. Once all references have been consumed, the simulator produces as output a complete description of the paging behavior of the applications that generated the traces. This description details, among other things: the number and type of page faults, the number of faults overlapped with others, the contribution of the application's CPU time to total execution time, the time spent waiting for subpages, and the time spent waiting for pages. More details on the simulator, including the capability to simulate subpage fetching policies not used in this paper, can be found in [8].

4 Issues in Load Balancing Memory Requests

This section studies the potential benefits and also the potential harm that can result when global memory requests are load balanced across a given set of memory servers. First, we discuss the conditions under which uneven request distributions occur in global memory systems. Then, using the analytic model, we study the degree to which uneven load degrades the performance of remote page requests, or conversely how much load balancing can improve request performance by eliminating contention. Load balancing the requests also forces possibly undesirable deviations from the strict LRU replacement policy. These deviations can potentially negate the benefits of load balancing. In Section 4.3 we use the simulator to study the impact on application performance when page replacements deviate from a strict LRU replacement policy.

4.1 Sources of Contention

Server contention arises when the total request rate to global memory servers by all global nodes is sufficiently high that the requests interfere with each other. On average, each global node will issue global memory requests relatively infrequently; otherwise application performance suffers. However, a number of situations can lead to sudden short-term increases in the request rate to one or more of the servers. As one example, during the course of their execution applications go through periods where their peak fault rate is significantly higher than their average fault rate. As another example, when nodes transition from one state to another, they can produce increased traffic to the servers. When a node transitions from idle to local or global (e.g., because the user starts to use the node again), the global pages it has been storing for other global nodes need to be moved to other servers as those pages become dislocated by local processes on the node. Considering that a node may be serving thousands of pages on contemporary machines, this transition can quickly result in thousands of putpages to the remaining servers in the cluster.

When a node transitions from local or global to idle (e.g., some time after the user stops using the node), its pages become immediately available to the rest of the cluster. Since these pages are likely the oldest in the cluster, putpage requests will begin to concentrate at the newly idle node. This situation can have long-term effects as well. For example, assume one server holds a large set of pages of the same age; when that set of pages becomes the oldest in the cluster, putpage requests will concentrate at that node as those pages are replaced by newer ones. The combination of a concentration of oldest pages at one node, and another node operating at high fault rate due to a transition in its application behavior or its node state, may lead to particularly significant server contention.

4.2 Benefits of Load Balancing

To study the potential benefit of load balancing global requests, we use the analytic model to quantify the relative performance of getpage requests (page faults) for varying degrees of imbalance among servers. In particular, we quantify:

- how the contention produced by situations with load imbalance degrades getpage performance,
- how load balancing improves getpage performance,
- the number of idle nodes required to serve various numbers of global nodes under balanced conditions.

4.2.1 Potential improvement in execution time

We study the first two issues with the following experiment. We consider a system with two idle nodes and a variable number of global nodes. Each of the global nodes is running the 007 benchmark described in Section 3.2. We then vary the probabilities $(P_{idle,0}, P_{idle,1})$ so that requests will go to one idle node or the other with various distributions. For this experiment, the distributions range from (0.5, 0.5) to (1.0, 0.0), covering the range from completely balanced to completely unbalanced. We then solve the model for getpage response time R for each of the distributions and, using R and the think time Z, compute the increase in execution time for the application. Note that since the 007 benchmark has a larger value of Z than the other applications we have traced (see Table 1), the estimates for performance degradation due to load imbalance (and the estimate of potential performance improvement that could be realized by load balance) will be conservative for this set of applications.

Because the first system is small compared to what we expect to find in real global memory systems, we consider a second system similar to the first but larger. Instead of two idle nodes, though, the second system has 11 idle nodes. The distributions for this experiment range from (0.25, 0.075) to (1.0, 0.0); in this system, the first probability $P_{idle,0}$ is for the first idle node, and the second probability $P_{idle,i}$ is the probability for each of the 10 remaining idle nodes. With these distributions, we are able to model a large cluster that has old memory heavily skewed to one node. Such a skew reflects situations such as when a node becomes a newly idle node in the cluster.

Figure 3 graphs the results of the experiment on the two systems. Each curve corresponds to a request distribution, and shows the increase in execution time of the 007 application as the number of global nodes increases. There are two important observations to note from these graphs. The first is that skew increases execution time. With a skew of (0.75, 0.025) at 18 nodes, for example, the execution time of all applications making global memory requests increases by 12%. The second is that the total number of idle nodes matters little when the skew is primarily to one node. With a similar skew of (0.60, 0.040) at 24 nodes, the execution time increases by nearly the same amount.

4.2.2 Required number of global memory servers

Another issue involving request contention at memory servers is the relationship between the number of global nodes and the number of idle nodes in the cluster. At the two extremes, a large number of global nodes will overwhelm a small number of idle nodes, and a relatively large number of idle nodes are going to smoothly handle a small number of global nodes. But typical clusters are not likely to be at the two extremes. The flip side of the load balancing question is how many servers do we need to balance the load? At one extreme, a large number of idle nodes will effectively handle a small number of global nodes even in the presence of skew in the requests (see Figure 3). At the other extreme, a large number of global nodes will overwhelm a small number of idle nodes even if load is balanced. However, during the operation of a cluster there will typically be significant fluctuations between the two extremes. A relevant question when faced with a moderate number of idle nodes and a significant skew in the requests by the default replacement policy is, to how many nodes should we redistribute some of the skewed requests?

Figure 4 provides insight into this issue for a wide range of system configurations between the two extremes. Each curve in the graph corresponds to a particular number of idle nodes in the

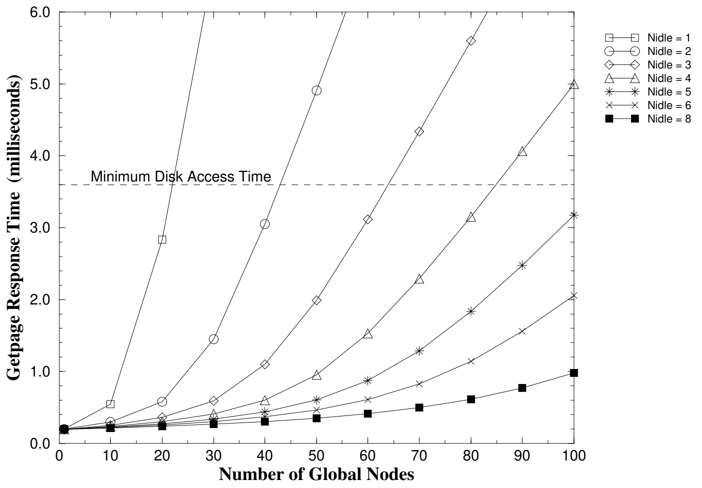

Figure 4: Response time of getpage requests versus the number of global nodes as the number of idle nodes in the network, N_{idle}, varies from 1 to 8. For each solution of N_{idle} we use the parameters for the 007 benchmark. Requests are balanced across idle nodes.

cluster, and plots getpage response time as the number of global nodes increases. The global nodes are configured to run the 007 benchmark, which gives optimistic results for our set of applications in the sense that the other applications will have response times that are greater than the ones shown in the figure. The dashed horizontal line at 3.6 ms corresponds to the minimum disk access time, as described in Section 3.1. Note that requests are again balanced evenly across idle nodes.

This graph shows that, when requests are balanced across memory servers, more than three idle nodes are needed to provide good service to medium or large numbers of global nodes. On the other hand, with eight balanced idle nodes the getpage response time only doubles when there are 60 global nodes in the cluster. This indicates that, with large clusters, traffic of the global nodes can be spread across a relatively small number of idle nodes.

4.3 Potential Harm of Load Balancing

The tradeoff with load balancing global memory requests is the impact on the global page replacement algorithm. The page replacement algorithms in current global memory systems implement some form of approximate LRU replacement [3, 4, 11]: when they have to make a page replacement decision, they strive to replace the oldest page in the system. However, based on the MVA results, there is potential benefit to modifying the replacement decision to balance the request load across servers. This means that, due to balancing, pages can be replaced at some position possibly relatively high on the LRU stack instead of at the bottom. Such suboptimal replacements can potentially degrade application performance.

In this section we study this issue in further detail. First we look at the relative frequencies of accesses to pages in the LRU stack for several of the applications in Table 1, as well as a number of additional applications described below. Then we measure the effect on application performance when the replacement policy deviates from strict LRU. Note that it has long been known that for many applications approximate implementations of LRU replacement have nearly the same performance as exact LRU; what we study here is the potential impact of repeatedly and deliberately replacing pages

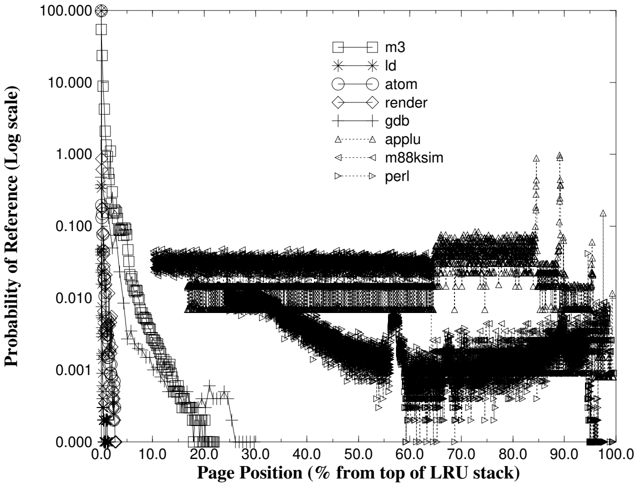

Figure 5: Probability of reference for pages in the LRU stack.

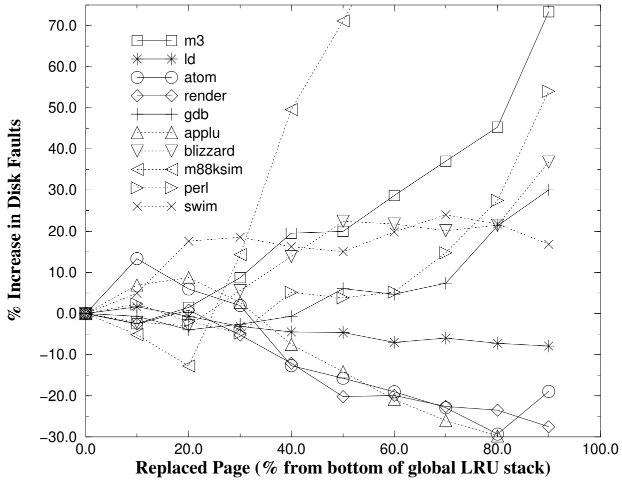

Figure 6: Effects of suboptimal LRU.

that are as high as 50–90% up in the LRU stack. We use the trace-driven simulator for our experiments, since an MVA model cannot capture the impact of specific replacement policy decisions.

For the LRU experiments, we measure the behavior of an additional set of applications to broaden the scope of the results. The additional applications are from the study in [7], and are summarized in Table 2. For each application, the table gives the length of the traced application (in millions of instructions) and the amount of total memory used by the application (in number of 4K pages).

The memory behavior of these applications was simulated based upon compressed traces of page references. Note that the simulator used for these traces is different than the one in 3.3; since the traces are incompatible with the GMS simulator, we implemented another simple simulator to measure the LRU stack behavior of these applications. The format of the page reference traces, as well as the constraints imposed on simulations using the traces, are fully described in [7]. We describe how these constraints interact with our LRU simulator when we describe the results of our experiments.

4.3.1 The LRU stack

To quantify the relative importance of a page to an application, we simulated the individual execution of each application on a single node. For each execution, the node had enough memory to hold all the pages the application needed. After simulating the memory behavior of each application on its inputs, the simulator calculated the fraction of memory references that occurred to each position in the LRU stack. Due to the method by which page references were compressed in the compressed traces, references to the top of

Application	Description	Length	Pages
applu	Solve parabolic/elliptic PDEs	1068	3631
blizzard	DSM binary rewriting tool	2122	3908
m88ksim	Microprocessor simulator	10020	4838
murphi	Protocol verifier	1019	2345
perl	Interpreted scripting language	18980	9836
swim	Shallow water simulation	438	3754

Table 2: Additional applications used for LRU experiments.

the stack (which is always in memory) were not recorded. Thus, for those applications, the simulator cannot calculate the fraction of references to positions in the top 10–20% of the stack; furthermore, the fraction of references for lower positions in the stack are the fraction of the *recorded references* that occurred to each position.

Figure 5 shows the results of this experiment. Each curve corresponds to the LRU stack of an application. For each position in the LRU stack, the probability that it was referenced is plotted on a log scale. So that clarity is not entirely sacrificed, only three of the compressed trace applications, applu, m88ksim, and perl, are shown in the figure. These three applications belong to three different types of applications reported in [7], respectively. Interestingly, for each compressed trace application not shown, its curve would overlie the curve shown for the application in its class.

The GMS applications are clustered on the left of the graph due to very low probability of accessing pages in the bottom 70% of the stack. Note that, for these applications, the pages on the bottom 95% of the LRU stack are each accessed with less than 1% probability. The curves for the compressed trace applications begin around 20% from the top of the LRU stack due to the constraints imposed by the method in which the traces were generated. Note that, none of the positions measured are accessed with greater than 1% probability; furthermore, the probabilities for the portion of the LRU stack that could be accurately simulated are inflated because the references to the top 10–20% of the stack, missing in the compressed trace, could not be counted in the denominator.

These long tails of positions with low probability of reference have promising implications for load balancing. Since load balancing global memory requests will result in page replacements above the bottom of the LRU stack, but always the oldest page on a give node, these suboptimal replacements may not severely degrade the performance of the application.

4.3.2 Deviating from LRU

To quantify the effect on overall execution time of deviating from strict LRU, we simulated the execution of the GMS applications running on one global node and paging to one idle node. The global node had 1/4 of the total memory needed by each application, and the idle node had 1/2 of the total memory needed. We modified the replacement policy of the simulator to replace the page at a specific

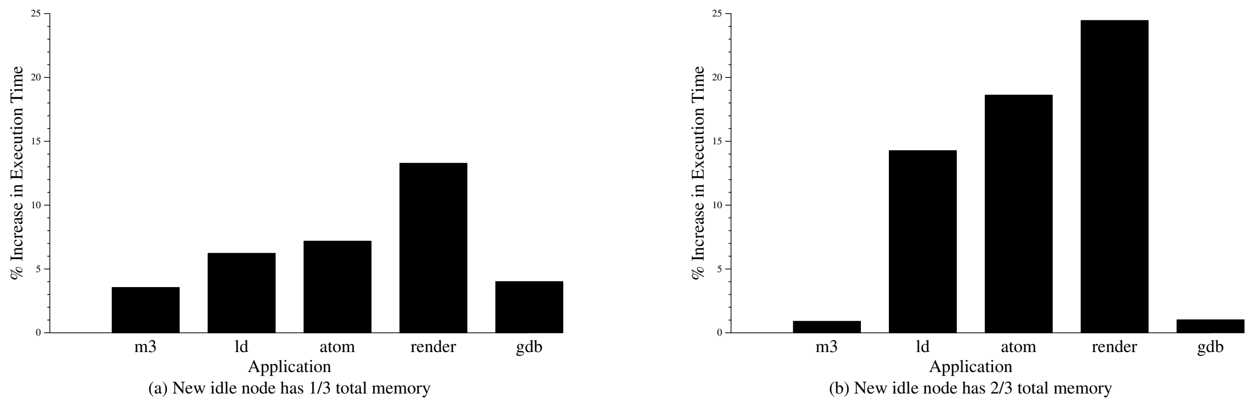

Figure 7: Upper bound on the increase in execution time for naively balancing the load across a new idle node, as compared with global LRU.

position from the bottom of the LRU stack, instead of replacing the oldest page. This position was varied from the oldest page to the youngest page in *global memory*, denoted by 0% to 100% from the bottom of the LRU stack, respectively. Replacements are only varied within global memory because load-balancing only affects pages in global memory.

For the compressed trace applications, we simulated a memory capable of holding 3/4 of the total amount required by an application; the 3/4 was chosen so that the 1/4 and last 1/2 of the memory would correspond to the breakdown between local and global memory of the GMS simulations. The compressed trace simulator was likewise modified to replace pages at a specific position from the bottom of the LRU stack, again varying from oldest to youngest in the global memory portion of the LRU stack.

Figure 6 shows the results of these simulations. Each curve corresponds to an application and plots the percent increase in number of disk faults versus the position in the global LRU stack which was selected for page replacement. The GMS applications are shown with solid lines, and the compressed trace applications are shown with dotted lines.

These curves show two important effects. The first is that some applications (ld, atom, render, and applu) improve their performance when the policy deviates from LRU. For these applications LRU is not the optimal replacement policy; deviating from LRU therefore improves their performance. The second is that, for most of the other applications, replacing pages up to 30% or more from the bottom of the LRU stack does not significantly degrade performance. With m3, for example, replacing pages 30% up from the bottom of the stack only introduces 9% more disk faults (which translates to an increase in execution time of only 2%).

With these results, we should expect that application performance may not significantly degrade from deviations from strict LRU due to load balancing. Note, though, that one application, m88ksim, is sensitive to bad replacements once pages are persistently replaced at least 25% up the global LRU stack. We conclude from this that there is a limit to which LRU ordering can be ignored, but that some deviation from LRU will not cause significant performance degradation.

4.3.3 Naive Load Balancing

Given the results of the previous experiments, a reasonable question one might ask is whether a global memory system could just ignore page age information and naively balance putpage requests across the available idle nodes. After all, such an approach should lead to a relatively uniform distribution of oldest pages across the idle nodes in the long run. We investigate a possible drawback of that approach, related to suboptimal use of global memory when a node transitions to the idle state, in the following experiment.

To quantify the effect of load balancing when new idle nodes join the cluster, we simulated a system with one global node running an application and three idle nodes being used as memory servers. For the first third of the execution time, only two of the idle nodes are used by the global node. Then, the third idle node joins the system and the application can begin to use its idle memory.

We use two different policies for introducing the memory on the new idle node into the system. In the first policy, the idle memory is made immediately available to the application; this policy corresponds to strict LRU (the new memory is the globally oldest memory). In the second policy, the idle memory is made available to the application at a fixed rate. In this case, every third fault goes to the new idle node; this policy corresponds to strict load balancing (the new idle node receives one third of the requests).

We then measure the increase in execution time of the second policy over the first to compare the effect of load balancing new memory into the system. Note that we would expect contention at the new idle node to be significantly higher for the LRU policy than for the load balancing policy, but that contention is not modeled in the simulator. Therefore, this experiment is conservative in that the increase in application execution time would be less if contention were modeled.

The results of the experiment are shown in Figure 7. Each graph has a bar for each of the simulated GMS applications, and the height of the bar shows the percent increase in execution time as a result of using the load balancing policy for new idle nodes. The two graphs each correspond to a different cluster configuration. In the first configuration, the global node holds 1/3 of the memory required by the application, the first two idle nodes each hold 1/6, and the new, third idle node holds the remaining 1/3. In the second configuration, the global node and the first two idle nodes each hold 1/9 of the total memory required, and the new, third idle node holds the remaining 2/3 of the memory.

Although this experiment measures maximum possible increase in execution time, rather than the actual increase that would occur in the presence of contention, it appears that naive load balancing does not use the memory available at a new node aggressively enough. One could also expect that other short term concentrations of old pages would likewise not be replaced aggressively enough under this naive load balancing policy. Thus, although global LRU algorithms might be improved by load balancing global memory requests, indications are that they should not be replaced with a load balancing strategy that ignores page age information altogether.

5 The Cost of Global Memory on Local Nodes

In a global memory system node, the balance between local and global pages changes dynamically. A key issue, then, is how and when to change that balance. In GMS, the balance depends on the ages of the pages on a node relative to the ages of (replaced) pages on other nodes. However, there is a tension that is not considered in this calculation, namely, the CPU time cost to a node that is storing global pages.

This section explores the performance issues involved in using local nodes (in addition to idle nodes) as memory servers. Using local nodes as memory servers can benefit global jobs in two ways: by making more global memory available, and by providing more memory servers to help distribute load. Previous work has quantified the benefit of the increased global memory [4], so we concentrate here on the benefit of having more servers with which to distribute and balance the remote memory load.

Using the analytic model, we perform an experiment on a system composed of 10 idle nodes, 10 local nodes, and a variable number of global nodes. We vary the number of global nodes (N_{global}) in the network to vary the request rates from global nodes to the idle and local nodes. We then partition the load from the requests between the idle nodes and the local nodes such that a request will go to a local node with probability P_{local}, and a request will go to an idle node with probability $P_{idle} = 1 - P_{local}$.

In this experiment, we characterize the benefit of using local nodes to distribute load in terms of the improvement in response time of getpage requests, and we characterize the cost in terms of the utilizations on the local nodes. For a given $P_{local} = p$, we first solve the model with $P_{local} = 0$ and derive the response time of getpage requests. We then solve the model with $P_{local} = p$ and derive both the response time of getpage requests and the utilization of the local node. From the two response times, we then calculate the percent improvement in getpage response time at the cost of a particular utilization of the local node.

Figure 8 shows the results of this experiment when P_{local} is incrementally varied from 0.05 to 0.70. Symbols are placed on the curves for each value of N_{global}, which ranges from 1 to 36 for each curve in increments of 3 (except for the first increment, which is by 2). We again used the inputs derived from the 007 benchmark to parameterize the request rates from global nodes.

From the figure we see that, for each P_{local}, improvement in response time is dramatic as N_{global} increases and the local node starts to shoulder the burden of global requests. However, improvement peaks at around 18 global nodes for our inputs, and then starts to decrease. Improvement decreases after 18 global nodes because this is the region where the idle node response time curve for $P_{local} = p$ enters its heavy load region. Since the idle node response time curve for $P_{local} = 0$ is already in heavy load, the differences between the curves, and therefore the percent improvement, decreases as the number of global nodes increases.

The surprising result from this experiment is that the utilization of the local node approaches an asymptotic value as the number of global nodes in the network increases. In other words, for a given P_{local}, the utilization of the local node by global requests will never

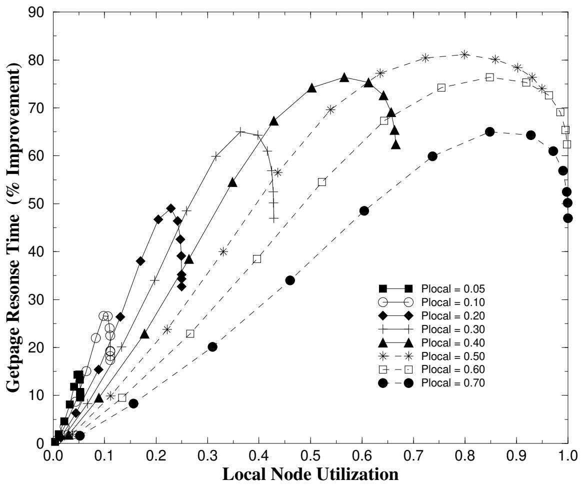

Figure 8: Percent improvement in the response time of getpage requests when a local node serves a fraction P_{local} of all global requests versus the utilization of the local node. Symbols are placed on the curves for each value of N_{global}, which ranges from 1 to 36 for each curve. The curves are solid when local nodes are given less precedence than idle nodes ($P_{local} < 0.50$), and are dashed when given equal or higher precedence ($P_{local} >= 0.50$).

go above a constant value, no matter what the rate of requests might be.

The intuitive reason why the local node utilization asymptotes at a value less than 1.0 when $P_{local} < 0.5$ is due to the fact that the arrival rate at local nodes will always be less than the arrival rate at idle nodes. Therefore, when the arrival rate at the idle nodes reaches a maximum, the arrival rate to local nodes also reaches a maximum, but a maximum that is smaller than the one for idle nodes. Since idle and local nodes have the same service demand, the smaller arrival rate to local nodes is not enough to fully utilize the local nodes, and so the utilization maximizes at a value less than 1.0. Once $P_{local} >= 0.5$, though, the rate to local nodes is equal to or greater than the rate to idle nodes, and the local node utilization increases to 1.0.

6 A New Family of Global Page Replacement Algorithms

In this section we use the insights gained from the experiments in section 4 to design a new family of global page replacement algorithms that incorporate load considerations as well as age information. The goal is to achieve a close approximation to global LRU replacement while bounding the global memory traffic load on local nodes and balancing the global memory traffic across the idle nodes.

We begin by pointing out that the ideal target for balancing the load across the idle nodes is not necessarily equal global request traffic at each node; in order to fully utilize the available global memory, the target relative load for each idle node is instead proportional to the total amount of memory on the node. In many systems the amount of memory per node will not vary greatly, and in any case we assume that the higher relative load at the servers that have more memory is generally outweighed by the advantage of caching as many recently used pages as possible in global memory.

The new algorithms can be easily modified if experience were to prove otherwise. For now, it is the additional unevenness in short term loads caused by (persistent) high concentrations of same-aged pages in one or more servers, as discussed in section 4.1, that the new class of global page replacement algorithms seeks to minimize.

We also point out that if the global replacement policy uses only the target relative loads (or weights) when selecting servers for global page replacement, then in the long-term the weights will reflect the relative proportion of pages of a given age at each node, and thus replacing the locally oldest page at the selected server node should be a close approximation of global LRU. On the other hand, the results of section 4.3.3 showed that strict use of target relative loads is not aggressive enough in taking advantage of temporary concentrations of very old global pages, such as the pages that become available when a node transitions from global or local to idle or when the overall load on a local node decreases. What's needed is an algorithm that uses age information to select those pages aggressively, and yet also uses target relative loads to balance the concentrations of same-age pages over time. As well, the weights for local and idle nodes should be continuously bounded to protect local jobs running on the local nodes and to avoid overloading any given idle node.

With these factors in mind, we propose a new family of global page replacement algorithms. We first describe the algorithms assuming that all required global system parameters are known and that all global nodes use an equal share of the global memory capacity. We then describe how global nodes that have greater need for global memory can temporarily "borrow" server capacity from global nodes that have lower need. Finally we briefly discuss how the global parameters are communicated.

6.1 The Basic Global Page Replacement Algorithm

Let M_i denote the total amount of memory on node i, measured in pages, and let G_i denote the number of pages on node i that can be replaced without degrading the performance of the local jobs on i, if any. For an idle node, $G_i = M_i$; for a local node, $0 \leq G_i < M_i$. A global node has $G_i = 0$. The *target weight* for node i, w_i, equals $G_i / \sum G_j$, where the summation is over each node j in the system. This is the desired relative frequency of selecting node i for global page replacement based solely on load balancing considerations. The notation for the proposed page replacement algorithm is summarized in Table 3.

Regarding global page age information, two schemes have previously been proposed. In GMS, a dynamically selected *initiator* node periodically collects page age information from each node in the network and computes a set of weights that reflect the fraction of the M globally oldest pages that reside at each node. (M is an adjustable parameter of the GMS algorithm.) These weights, which we will call *age weights*, are then distributed to each node and used to select nodes for putpage requests, thus implementing approximate global LRU page replacement. Alternatively, in the global page replacement algorithm proposed by Dahlin and Sarkar [11], each memory server piggybacks the age of its oldest page on its responses to global memory requests. In that algorithm, each global node maintains a list of the most recent page age information it has received from each server, sorted in order of decreasing age. The global node then selects the memory server at the top of the list for its next global page replacement. We choose to use age weights rather than simply the age of the oldest page on each server, as the weights are more easily adapted to accommodate load-balancing considerations. However, we leave open the possibility that the weights might be estimated using global page age information that can be piggybacked on responses to global memory requests. This is discussed further in section 6.3 below.

When a global node needs to replace a page in its local memory, it first chooses a memory server based upon its current age weights (a_i) and current target weights (w_i) for the memory servers, as follows. Let b denote a parameter of the algorithm called the desired *load balancing rate* for the system, $0 \leq b \leq 1$. Node i is selected for the global page replacement with frequency

$$f_i = bw_i + (1-b)a_i. \tag{1}$$

If $b = 0$ the algorithm approximates global LRU replacement without regard to load balance, as in GMS. If $b = 1$ the algorithm simply balances global requests across the available memory servers without regard to age information. For in between values of b the algorithm responds to new concentrations of globally oldest pages but gradually balances the number of similar age pages across the servers. The closer b is to zero, the more aggressively it will take advantage of newly idle memory. The closer b is to one, the more aggressively it will balance the global memory load.

Two additional constraints, in the form of upper bounds, are imposed on the memory server selection frequencies for global page replacement. First, for local nodes, we derive a simple bound to protect the local jobs running on the server from excessive interference by global memory requests. Let U_i^* denote an upper bound on utilization of local node i by global memory requests, and S denote the average time to serve a global memory request. Assuming each of the N_g global nodes uses an equal share of the total allowed server utilization, then the maximum rate that a global node can send global memory requests to local node i is $\frac{1}{S}\frac{U_i^*}{N_g}$. Thus, the following equation defines an upper bound on the selection frequency for local node i by global node j:

$$\frac{f_i}{T_j} \leq \frac{U_i^*}{N_g S}. \tag{2}$$

where T_j is the recent average time between the global memory requests issued by global node j. Multiplying each side of the equation by T_j gives the desired upper bound on f_i.

For each idle node, we derive a simple upper bound on the selection frequency to avoid overloading the node. In this case, we observe that the maximum utilization of the idle node by global memory requests is 100%. Plugging in 100% for U_i^* in equation 2 yields the following bound on selection frequency for idle nodes:

$$\frac{f_i}{T_j} \leq \frac{1}{N_g S}. \tag{3}$$

This assumes again that all global nodes use an equal share of the global memory capacity. The bounds can be modified for unequal use of memory capacity, as described below. Note that the above bound is simpler and somewhat more aggressive than the bound that would be derived by setting estimated memory server capacity (N* $\leq N_g$) and using standard asymptotic bounds techniques to estimate $N*$. [10]

For each memory server, if the applicable upper bound on selection frequency is lower than the selection frequency originally computed from the target and age weights, then the selection frequency

Symbol	Meaning
M_i	total memory available at node i, in pages
G_i	number of pages at node i that can be replaced without harming local jobs, if any
a_i	the age weight for node i, (i.e., the fraction of globally oldest pages at node i)
w_i	the target weight for node i, (i.e., node i share of the balanced load)
f_i	selection frequency for node i
b	load balancing rate
T_i	recent average time between global requests issued by node i
S	average time to serve one global memory request at a server node
N_g	current number global nodes
U_i^*	upper bound on the utilization of local node i by global memory requests

Table 3: Notation used in discussion of new family of algorithms.

is set to the bound. Selection frequencies are then recursively re-computed for the other nodes using new target and age weights that are renormalized for the smaller set of server nodes that can be selected as well as for the remaining total selection frequency needed.

6.2 Borrowing Memory Capacity

The algorithm outlined above assumes that each global node uses an equal fraction of the total global memory capacity. In practice, however, this will rarely be the case. At any given point time, there is likely to be great variability in the total request rate for global memory among the global nodes. The basic algorithm approximates LRU while balancing server load even for this case, but the upper bounds on the selection frequencies will limit the throughput of the applications that have the greatest memory demands. We can improve the performance of those large applications by allowing them to "borrow" server capacity that would otherwise go unused by the global nodes that have lower global memory demands, as follows.

Define the unused capacity of a server node, U_e, i, to be the difference between its maximum possible utilization by global memory requests, U_i^* or 1.0 depending on the node type, and its recent utilization by global memory requests. Let each server node piggyback the maximum of this value and zero on each response it sends to a global memory request. Each global node can maintain the latest value of U_e, i that it has received from each server node i. The global node can then add the following term to the right hand side of equation 2 to increase the bound on its selection frequency for node i: $U_e, i/(N_g - 1)$. Note that this term assumes that the excess capacity will be shared equally with (at most) $N_g - 2$ other global nodes in the system. If one or more global nodes in the system continue to require less than their fair share of the global memory server capacity, the nodes that require more than their fair share will gradually grab the unused capacity. Note also that if the applications that haven't been using their fair share every suddenly require their fair share (or more), they are guaranteed to get at least their fair share immediately.

6.3 Parameters of the New Page Replacement Algorithms

Table 3 gives the parameters used in the new global page replacement algorithms. In this section we briefly discuss how the global nodes might determine the parameters that are needed to compute the memory server selection frequencies.

The total memory available at node i (M_i) and the average time to serve one global memory request (S) are system parameters that change infrequently and are easy to provide as inputs to the algorithm. Similarly, each global node j can easily measure the recent average time between the global requests it issues (T_i).

The current number of global nodes (N_g) can be determined by a dynamically selected (and possibly replicated) "status node", or can be estimated by each memory server and piggybacked on responses to global memory requests.

The desired load balancing rate (b) is a tunable parameter for the algorithm. Future research includes determining good values of this parameter for various workloads. It may even be useful to investigate the viability of dynamically tuning b at run-time using measured impact on global memory paging rates.

The upper bound on the utilization of local node i by global memory requests (U_i^*) might simply be specified by the system administrator, or might be adjusted according to the measured recent utilization of node i by its local jobs. Again this is the subject of future research.

The number of pages at node i that can be replaced without harming local jobs, if any, (G_i) is equal to M_i for idle nodes. For local nodes, further research is needed to determine precisely how this value should be estimated.

The age weights (a_i) can be determined by an initiator node, as in GMS, or might instead be estimated from global page age information that is piggybacked on responses from global memory servers. In this latter case, for example, the memory server might report the age of its xth oldest page, where x is a parameter of the algorithm. Each global node would maintain a table of the age reported by each memory server. The age weight for node i would then be estimated as the reported age of the xth oldest page on node i, divided by the sum of the reported ages of the xth oldest page on each other node.

7 Conclusions

High-speed switched networks have reduced the latency of network page transfers to below that of local disk. This technology change has made global memory systems practical. A crucial issue in the implementation of global memory systems is the selection of the target nodes to receive evicted pages. Existing systems have selected targets based on the ages of their memory pages, effectively selecting the oldest pages in the network for replacement. In this paper, we study the impact of this decision, particularly on the nodes holding old pages; these nodes become global memory servers for highly-active nodes, which may interfere with their local application performance. We also examine (1) the impact of changing the target node selection criteria in order to reduce interference on local performance, and (2) the effect of deviating from the global LRU replacement algorithm used in some global memory systems.

Using an analytic queueing network model, we studied how uneven request distributions produce memory server contention, even when the cluster has a large number of servers. Load balancing requests to produce an even distribution across servers minimizes server contention and improves request latency. With the model, we have also seen that global memory systems will not scale if there are only a few memory servers available. However, when requests are load balanced across servers, we find that eight memory servers are sufficient to handle the load in the 70 node cluster we modeled.

Using a trace-driven simulator, we have demonstrated that the global LRU page replacement decision is relatively *insensitive* to exactly which of the oldest pages is replaced. Therefore, load balancing global page replacements should not degrade application performance if the deviation from strict LRU is moderate.

Overall, we conclude from these experiments with the model and simulator that load balancing global memory requests will reduce contention at memory servers and improve application performance. This led us to the design of a new family of global memory management algorithms that use a small set of measured system parameters, as well as two tunable parameters (i.e., the load balancing rate, b, and the stack location for exchange of global page age information, x) to balance the load across the servers while still implementing approximate LRU global page replacement. Further research includes investigating the performance of this family of algorithms and comparing the performance against GMS as well as other methods for balancing server load.

Acknowledgments

We would especially like to thank Gideon Glass and Pei Cao for the use of their traces in the LRU stack experiments. We would also like to thank the anonymous reviewers whose comments helped improve the quality of this paper.

References

[1] M. J. Carey, D. J. Dewitt, and J. F. Naughton. "The 007 benchmark." In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1993.

[2] Douglas Comer and James Griffioen. "A new design for distributed systems: The remote memory model." In *Proceedings of the USENIX Summer Conference*, June 1990.

[3] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. "Cooperative caching: Using remote client memory to improve file system performance." In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, November 1994.

[4] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. "Implementing global memory management in a workstation cluster." In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[5] Edward W. Felten and John 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.

[6] Michael J. Franklin, Michael J. Carey, and Miron Livny. "Global memory management in client-server DBMS architectures." In *Proceedings of the 18th VLDB Conference*, August 1992.

[7] Gideon Glass and Pei Cao. "Adaptive Page Replacement Based on Memory Reference Behavior." In *Proceedings of ACM SIGMETRICS 1997*, June, 1997.

[8] 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*, pp. 258–267, October 1996.

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

[10] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. "Quantitative System Performance." Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[11] Prasenjit Sarkar and John H. Hartman. "Efficient cooperative caching using hints." In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

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

[13] Amitabh Srivastava and Alan Eustace. "ATOM: A system for building customized program analysis tools." *DEC Western Research Lab Technical Report 94/2*, March, 1994.