Fast Transparent Cluster-Wide Paging

Michael R. Hines, Jian Wang, Mark Lewandowski, and Kartik Gopalan
Computer Science Department, Florida State University
{mhines,jwang,lewandow,kartik}@cs.fsu.edu

Abstract— **In a cluster with a very low-latency interconnect, the remote memory of nodes can serve as a storage that is faster than local disk but slower than local memory. In this paper, we address the problem of transparently utilizing this cluster-wide pool of unused memory as a low-latency paging device. Such a transparent remote memory paging system can enable large-memory applications to benefit from cluster-wide memory resources without compromising application performance while obviating the need for very large local memory or modifications to legacy applications. There was considerable interest in this subject in the 1990's, which saw a number of remote paging systems built over specialized interconnects such as Myrinet and ATM switches. However, in recent years, this subject has not received the research attention it deserves. With the advent of highly affordable commodity gigabit Ethernet switches in the market, we examine the feasibility and advantages of using gigabit Ethernet to perform fully distributed remote memory paging. We leverage upon our earlier experiences with the Anemone project [11] – a centralized remote paging system – to develop a fully distributed, transparent, Linux-based alternative over gigabit Ethernet to avoid the disk I/O bottleneck. A self-managing pseudo block device module resides in every participating workstation in the cluster and is responsible for a multitude of functions. This module avoids the entire IP stack and user-space overheads, manages the available cluster-wide memory as well as local disk swap devices, eliminates centralized decision making, is completely transparent to the kernel, dynamically adapts to the availability of remote memory, and fully exploits the bandwidth-delay product of the underlying interconnect. Preliminary performance results indicate that the distributed paging system design can yield sequential program execution speedups of up to a factor of 6 when compared to disk-based paging (versus 3 in our in our cenrtalized design) and page-fault latencies in the order of 210 microseconds.**

I. Introduction / Goals

In the effort to virtualize and globalize the resources of a network of workstations in a high performance manner, we tackle the problem of paging to remote memory, with efforts to significantly increase performance. Despite the various other projects out there (most which ended before the turn of the century), we find that most Linux-based clusters still sit idle without a readily available mechanism for remote paging usage over something as common as gigabit Ethernet, in spite of increasing LAN speeds and plummeting access latencies. After mapping out our goals, we will explain our design, a performance evaluation and an enumeration of the differences with prior efforts.

First we identify some terms. We refer to the term of the swap daemon in the operating system as a "pager", that mechanism which handles the swapping in and swapping out of virtual memory pages to disk, or to whichever block device in the system is configured to store swapped out pages from main memory. We say that there is a single paging device in the system, called a module, which is cognizant of both local and remote paging sources, but hides the operational complexity of discovering and using these sources from the pager itself. We refer to any particular commodity workstation that makes up the cluster a "node". We refer to memory-intensive applications as "Large Memory Applications" or LMAs. We require our remote paging system to satisfy the following goals in order to be properly virtualized and automated:

A. *[Autonomy]*

Any remote paging system conceptually has paging client(s) running an LMA and server(s) that store pages in memory. Our design combines the two into a single kernel module. The module keeps constant, updated knowledge of all other available modules in the cluster and autonomously use them without assistance from the user or from the local pager. Thus for every participating node in the cluster, the pager on that machine communicates with the module.

B. *[Transparency]*

The module neither modifies the kernel and nor requires any modifications to LMAs. It appears as a block device to the pager, easily linked into the kernel during runtime.

C. *[Decentralization]*

Participating modules in the cluster operate in a fully distributed mode, without the assitance of a central module responsible for the allocation of remote pages, nor is there a module responsible for managing the location of pages in the cluster. This requires a strategy for continuous resource discovery which we describe later.

D. [Network Support]

Choice of a high-speed layer-2 interconnect can change at whim depending on how much money you have, so our research efforts are based on the need to improve the paging mechanisms themselves in software. Since it is easily obtainable and cheap, our cluster sits onto of 1-gigabit ethernet hardware, permitting us a 4-Kilobyte RTT between two modules of 210 microseconds. Future work, however, may employ the use of RDMA over ethernet (Remote Direct Memory Access).

E. [Computation]

The module avoids computational overhead as much as possible. This includes minimization of data copying, the employment of data structures for efficient lookups during a page-fault, eliminating the effect of context switching by restricting the module to kernel space, and avoiding excessive protocol processing by eliminating the use of the IP stack. All prior remote paging efforts have pitfalls in one or more of these areas, but our design optimizes on them all which is critical for high performance. Our methods for minimizing computational overhead are described later.

F. [Global memory]

The module both stores pages in the memory of other modules in the cluster and stores pages on behalf of other modules, depending on the activity of that module. This also means a module should employ a method for memory reclamation of remotely stored pages for the pager that it is serving. The module load balances its page-out requests to remote modules and uses the disk as a backing store if there are no available modules. While GMS [7] does this, our approach is completely transparent, requiring a strict interpretation of the behavior of the local pager and no kernel modifications.

G. [Reliability]

Network paging introduces more components capable of failure than the disk does, so modules should provide support for failure recovery. However, because these mechanisms have already been strongly addressed in [9] and [12], we are not targeting reliability but we leave it as a necessary component, nonetheless. Possible methods include both mirroring/replication and the use of parity.

II. Architectural Design

This section examines our design goals in detail. We first begin by explaining the structure of a module and how it interacts with the system and then address each individual design goal of a module. Figure 1 illustrates a module. It contains

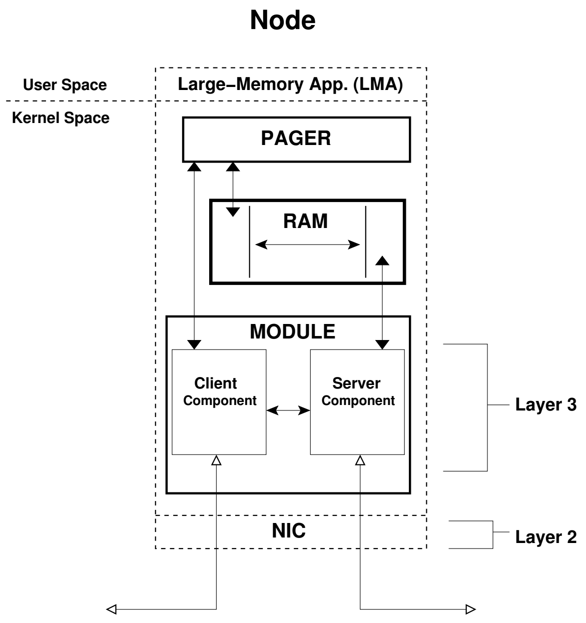

Fig. 1. The interactions and components of a node and module in our remote paging system.

both a client component and server component, where the former accepts paging requests from the pager and directs them to appropriate remote server component, and the latter accepts remote paging requests from the network and stores or retrieves them in memory.

A. Component Structure

The client component exposes a block device interface to which the pager makes page-in or page-out requests. All physical pages are represented by a ”page” structure. When a page is swapped out, it is wrapped in a ”bio” (block I/O) structure and handed to the client component of the module, accompanied by an offset into the block device. The client component keeps a hashtable where it stores the location of each remotely-stored page, keyed by the offset of the page. If an entry does not exist, a location is chosen in a dynamic manner (described later) and an entry is created. The page is then sent out to the chosen remote node. The server component also contains a hashtable, keyed by the offset of the page belonging to which ever source-node sent that page, and the node’s address. The page is then copied into main memory. The module is then easily linked into the kernel during runtime. Neither the kernel nor the LMA needs to be modified or recompiled, allowing the module to operate transparently.

B. Component Interaction

An idle node means there are no LMAs running. Server components become alive (accepting remote paging requests) only when a node is idle and the client component is dormant. The transition of a server component from alive to dormant requires that it start freeing up space. This can be detected by the current "speed" of the pager, i.e. the amount of block I/O (Page-OUT) requests coming from the pager, which can be expressed as some X requests per second.

Evaluation of speed changes are controlled by a exponential weighted moving average (EWMA) in order not to cause the server component to free up pages too aggressively. We use an EWMA with a multiplier of 10 percent. Changes in speed signal different events to occur within the module:

1) If the speed increases, the client component becomes alive and informs the server component to go dormant and free up remotely-stored pages by sending them back to the source node that owns them.
2) If the speed hits zero, the server component then becomes alive and accepts remote paging requests. At this point, we also perform reclamation of remotely-stored pages whose system usage counts have been set to zero. The client component sends out requests for remote server components to free up space which local LMAs no longer use. The pages to be freed are identified as a whole by using a periodically changing per-client "session id".
3) The server component also issues a "soft-state" timeout for each page it is housing. This requires the client components to periodically send keep-alives to all servers that house its pages, one message per server. If an LMA exits, this allows modules to free up old pages.

As the two components interact, the physical memory space used by one or more LMAs and that used by the server component will be dominated by one or the other, but not both. Either a server component that is coming alive will "push left" to take up available physical memory space or an LMA that is becoming more active will push right.

C. Autonomy and Decentralization

This goal of the system requires that modules have some way of discovering both which nodes in the cluster have running modules for usage and exactly how loaded they are. We solve this by having the server-component of all participating modules advertise their internal status in a broadcasted message to all other nodes in the cluster. We fix this advertisement to be strictly periodic every Z seconds, so that the network is not congested with advertisement traffic. Modules accept these updates and keep a list of all available remote nodes in the cluster along with their status, including a flag indicating whether or not the module is accepting any remote paging requests, the amount of memory used by the module's server component, and the total amount of available memory contained within that node. Upon removal of the module from a node, a withdrawal request is sent, instructing other modules not to consider that node anymore.

When a node's pager swaps out a new page, the module traverses its list of remote modules and selects the least-loaded one, by dividing the amount of used memory by the total memory. The client component sends out the page and puts and entry in the hashtable. This allows for load-balancing among the modules in the cluster. (Future work may load-balance among CPU or I/O intensive nodes.) During transience, the period of time between updates received from a particular module, a client makes these choices the same way, but if a module becomes full and cannot service the request, the acknowledgment to that request is negative and the client tries again to store that page on a different module. If all nodes are full, the client sends the request to the disk's swap partition that the pager would have used in the common case. These mechanisms allow modules to behave autonomically within the cluster while at the same time eliminating the need for a centralized module to keep the responsibility for global paging management and allocation. Central modules add extra time to the RTT of a complete request and we want to eliminate that, particularly during page-faults.

D. Network Support and Computation

Since this system operates solely within a cluster, we do not need the routing functionality provided by the IP layer. Modules do not operate in userspace nor on top of sockets so they do not need the concept of a "port", eliminating layer 4 UDP. The fragmentation functionality provided by IP is eliminated by Ethernet "jumbo frames", capable of supporting frame sizes in excess of 9 kilobytes. For these reasons, we avoid the kernel's IP stack altogether and register a new networking layer with the kernel upon initialization. Nodes are addressed directly by their MAC address, learned by listening to advertisement messages described earlier. Figure 2 illustrates the structure of a request packet used between any two modules in our system. This new layer replaces layer 3 IP and sits right on top of the local network interface's device driver, issuing and receiving packets as quickly as possible. Using this method, the client component replaces the 4-kilobyte in-memory copying needed to fragment a packet. Furthermore, the two copies used between the pager and the client component and the client and the network interface is the same number of copies used if the disk were used instead of a module.

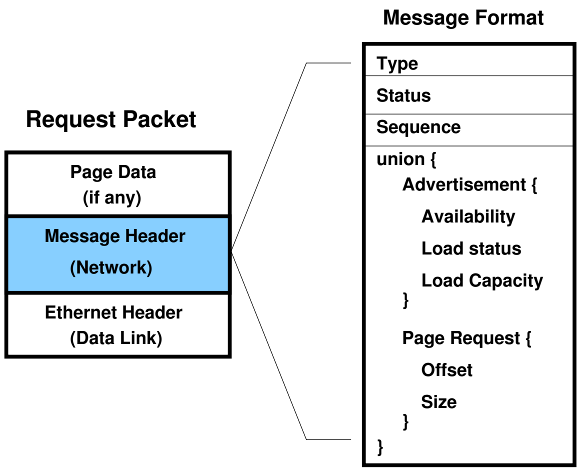

Fig. 2. The structure of a request packet used between modules and summary of the fields contained within the message header of the request.

Since we are not using TCP, we must finally implement our own flow control. One of the many knobs that we use in our transmission strategy includes the client's window size. This window is a modified sliding window that does not guarantee in-order delivery. The retransmission rate can significantly slow down the application, because the swap daemon must provide guarantees to the application that it will get its pages when needed. Ideally, a high performance system will include the ability to change the window size in response to RTT and/or remote-module workloads, but for now a fixed window size has sufficed in getting initial results.

E. Summary and Focus

We've described the details used to implement each of the design goals of our system. A module achieves transparency through its linkability to the kernel, high performance by avoiding the kernel networking stack, hashtable employment, and adaptation to increases and decreases in paging speed. The global paging system achieves scalability and autonomy by its completely decentralized approach and strict separation of each of the functions of the module.

III. Performance

In this section we do a preliminary performance evaluation. We are actively developing this decentralized version of the system based on our performance estimations of our earlier centralized Anemone prototype [11]. We focus on answering two key performance questions: What speedups do real-world, unmodified applications (LMAs) get? How much networking processing overhead is introduced inside the module? We first break down our measured overheads of networking and computational overheads. Then we show the speedups over disk-based paging gained by the previous Anemone prototype and how we predict significant improvements with this new system.

A. Testbed and Operation

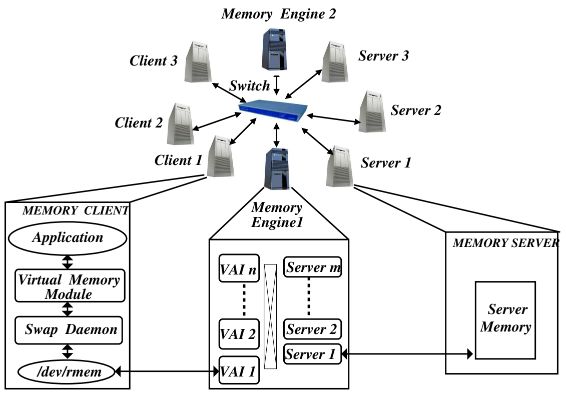

Fig. 3. The architecture of the Anemone system, a centralized approach to remote paging.

Figure 3 illustrates the centralized architecture of Anemone. Our testbed is made up of four 2.6 ghz Intel Xeon machines each containing one Intel PRO-1000 gigabit NIC. We run our experiments using one of the machines running LMAs and the other 3 server-only machines. The LMA machine is a low-memory machine containing only 256 MB of memory. The remaining machines each contain one, two, and three gigabytes of memory respectively. For disk-based tests, the LMA machine uses a Western Digital WD400BB disk with 40GB capacity and 7200 RPM speed. Anemone does not have combined client/server modules. Clients and servers are separated onto different machines. Anemone is still just as transparent as our current work. Furthermore, Anemone employs a central machine which we call an ”engine”. All clients and servers connect to the engine. Clients do all of their remote paging directly through the engine which in turn redirects those pages to appropriate servers in a load-balanced manner. This requires 4-hops in the RTT of a page-fault.

Component	Avg Latency	Std. Deviation
Round Trip Time	496.7 usec	9.4 usec
Engine/Client Comm.	235.4 usec	8.2 usec
Client Computation	3.36 usec	0.98 usec
Engine Computation	5.4 usec	1.1 usec
Engine/Server Comm.	254.6 usec	6.2 usec
Server Computation	1.2 usec	0.4 usec
Disk:	9230.92 usec	3129.89

TABLE I
PAGE-FAULT SERVICE TIMES AT INTERMEDIATE STAGES OF ANEMONE.

Figure I summarizes the results of both the networking and computational overheads of this system. Instead of a 210 microsecond RTT, we have an almost 500 microsecond RTT. Disk-based page-faults go up to 10ms. The computation values are measured by timestamping the request when it is generated and subtracting the timestamp when the request is completed. For the engine, this time includes the difference between the time the page-out/page-in request is received and the time when it finally responds to the client with an ACK or a data page. For the server, it is simply the difference between time the request for the page arrives and the time the reply is transmitted back to the engine. These values are negligible compared to round trip communication latency, barely exceeding 5 microseconds. The majority of the latency involved is spent in communication time.

B. *Single and Multiple LMAs*

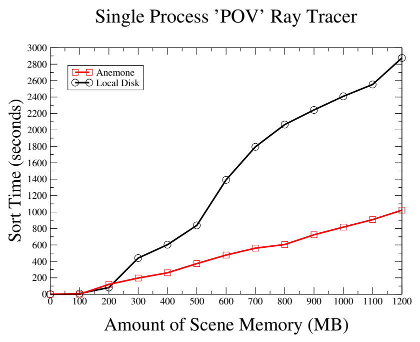

Fig. 4. Comparison of execution times of POV-ray for increasing problem sizes.

This section evaluates the performance improvements seen by a graphics ray-tracing program called POV-Ray [14]. For space reasons, we only present one of many LMAs we have tested. Using POV-Ray, we rendered a scene within a square grid of 1/4 unit spheres. The size of the grid was increased gradually to increase the memory usage of the program in 100 MB increments. Figure 4 shows the completion times of these increasingly large renderings up to 1.2 GB of remote paging memory versus the disk using an equal amount of local swap space. The figure clearly demonstrates increasing application speedups with increasing memory usage and that execution time increases by a factor of up to 2.9 . Multiple concurrently executing LMAs tend to stress the system by competing for computation, memory and I/O resources and by disrupting any sequentiality in the paging activity. Figures 5 show the execution time comparison of Anemone and disk as the number of POV-ray processes increases. The execution time measures the time interval between the start of the multiple process execution and the completion of last process in the set. Each process consumes 100MB of memory. As the number of processes increases, the overall remote paging usage also increases. Remote-paging reacts very well to concurrent LMAs and the total execution time increases very slowly. This is because, unlike disk based paging, remote paging encounters an almost constant paging latency over the network even as the paging activity loses sequentiality of access. With 12 concurrent memory-intensive processes, Anemone achieves a speedups of a factor of 7.7 .

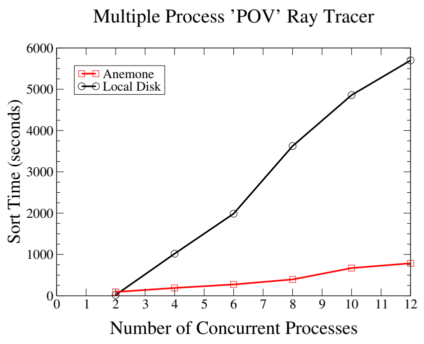

Fig. 5. Comparison of execution times of multiple LMAs for increasing problem sizes.

C. *Decentralized Performance Projections*

The most obvious performance advantage of a decentralized design over Anemone includes the elimination of the engine, literally cutting the RTT of a page-fault in half to 210 microseconds, as we have evaluated in our currently running prototype. Furthermore, the architecture has a number of design improvements over Anemone, many stemming from the necessity to get rid of the engine. We project that single-process LMAs will jump to a speedup of at least a factor of 6 while concurrent LMAs will similarly double to at least a factor of 14. Many kinds of applications are expected to benefit, including mathematical operations like sorting and cacheaware algorithms, simulation programs like network and particle simulations or even standard programs like browsers, compilations and high-performance benchmarks.

IV. Related Work

Our work is not in any way a DSM (Distributed Shared Memory) [6] systems. Paging to remote memory, however has been well-researched in the 1990s, but has not received the attention it deserves in spite of increasing LAN speeds and plummeting access latencies. Our approach is indeed a combination of already-tested techniques, but is also a restructured approach to virtual memory paging. To the best of our knowledge, our system is also the first attempt to evaluate the feasibility of remote paging over commodity Gigabit Ethernet LANs that support jumbo frames.

We first saw [4] and [8], both of which incorporated extensive OS changes on top of 10Mbps Ethernet. The Global Memory System (GMS) [7] was designed to provide network-wide memory management support for paging, memory mapped files and file caching. This system was also closely built into the end-host operating system over Myrinet. GMS employed a distributed approach, but still required kernel changes as well as a centralized accounting module. The Dodo project [10], [1] provides a user-level library based interface that a programmer can use to coordinate all data transfer to and from a remote memory cache, requiring legacy applications to be changed at user-level.

Reliable Remote Memory Pager [12] implements the closest system to ours over 10mbps Ethernet. However, their servers are user-level applications and not in the kernel and do not use Linux. The work in [13] implements another remote paging system at user-level based on the Nemesis operating system and the Network Ramdisk [9] offers remote paging with data replication and adaptive parity caching by means of a device driver based implementation. Their servers, however are not kernel-based, but written as user-applications. Samson [15] is a dedicated memory server that actively attempts to predict client page requirements but requires both drivers and the FreeBSD operating system to be extensively modified. The NOW project [2] performs cooperative caching via a global file cache [5] in the xFS file system [3], but the exact performance and implementation details are unclear from their publications.

V. Conclusions

This paper presented the design and preliminary evaluation of a high performance remote paging system. We've designed a highly adaptive system aimed at transparently reducing as much paging overhead as possible in a fully-distributed and self-managing manner. Our system implements a number of new and old ideas and we are quickly implementing a working prototype based upon promising performance numbers from our prior Anemone project. When compared to disk-based paging, we project single-process application speedups of up to factor 6 and multiple-process speedups of up to 14, even when competing against modern disks that are optimized with high RPM, and have large local caches to improve spatial locality.

References

[1] A. Acharya and S. Setia. Availability and utility of idle memory in workstation clusters. In *Measurement and Modeling of Computer Systems*, pages 35–46, 1999.

[2] T. Anderson, D. Culler, and D. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, 1995.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. of the 15th Symp. on Operating System Principles*, pages 109–126, Copper Mountain, Colorado, Dec. 1995.

[4] D. Comer and J. Griffoen. A new design for distributed systems: the remote memory model. *Proceedings of the USENIX 1991 Summer Technical Conference*, pages 127–135, 1991.

[5] M. Dahlin, R. Wang, T.E. Anderson, and D.A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Operating Systems Design and Implementation*, pages 267–280, 1994.

[6] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proc. of Intl. Parallel Processing Symposium, San Juan, Puerto Rico*, pages 153–159, April 1999.

[7] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. *Operating Systems Review, Fifteenth ACM Symposium on Operating Systems Principles*, 29(5):201–212, 1995.

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

[9] M. Flouris and E.P. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Cluster Computing*, 2(4):281–293, 1999.

[10] S. Koussih, A. Acharya, and S. Setia. Dodo: A user-level system for exploiting idle memory in workstation clusters. In *Proc. of the Eighth IEEE Intl. Symp. on High Performance Distributed Computing (HPDC-8)*, 1999.

[11] M. Lewandowski M. Hines and K. Gopalan. Implementation experiences in transparently harnessing cluster-wide memory. Computer Science Department, Florida State University, `http://www.cs.fsu.edu/~mhines/anemone/`, 2005.

[12] E.P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *USENIX Annual Technical Conference*, pages 177–190, 1996.

[13] I. McDonald. Remote paging in a single address space operating system supporting quality of service. Tech. Report, Dept. of Computing Science, University of Glasgow, Scotland, UK, 1999.

[14] POV-Ray. The persistence of vision raytracer, 2005.

[15] E. Stark. SAMSON: A scalable active memory server on a network, Aug. 2003.