Anemone: Adaptive Network Memory Engine

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

I. Motivation and Problem Statement

Large-memory high-performance applications such as scientific computing, weather prediction simulations, database management, and graphics rendering applications are increasingly claiming a prominent share of the cluster-based application space. Even though improvements in Dynamic Random Access Memory (DRAM) technology have yielded an enormous increase in the memory capacity of off-the-shelf systems, the memory-intensive applications continue to hunger for even larger amounts of memory. The issue is not whether one can provide enough DRAM to satisfy these applications; rather the more DRAM one provides, the more these applications ask for. These large-memory applications quickly hit the physical memory limit and start paging to the local disk. As a consequence, application execution time is degraded due to higher disk latency involved in paging operations.

At the same time, while memory resources in one machine in a LAN-based cluster might be heavily loaded, large amounts of memory in other machines might remain idle or under-utilized. Since affordable Gigabit LANs with attractive cost-to-performance ratios are now very common, one could significantly reduce access latencies by first paging over a high-speed LAN to the unused memory of remote machines and then resort to disk-based paging only as the last resort when remote resources are full. Thus, remote memory access can be viewed as another level between the traditional memory hierarchy memory and the disk. In fact, typical remote memory paging latencies of 100 to 500μs can be easily achieved whereas the latency of paging to slow local disk (especially while paging in) is typically around 2 to 10ms, depending upon seek and rotational overheads. *Thus remote memory paging could potentially be one to two orders of magnitude faster than paging to disk [13].*

II. Research Objectives

Our primary research objective, called the ***Anemone*** (Adaptive NEtwork MemOry engiNE) project, is to harness a large collective pool of memory resources across nodes in a high-speed LAN. The ultimate performance objective of Anemone is to enable memory hungry applications to gain transparent, fault-tolerant, and low-overhead access to potentially "unlimited" memory resources. The novel aspect that distinguishes the Anemone project from prior efforts is that no modifications are required either to the memory-intensive applications or to the end-hosts, thus eliminating painstaking and expensive end-system modifications. Instead, Anemone exploits the standard features that arrive bundled with these systems - the RAM disk (or memory-resident disk interface) and the Network File System (NFS) protocol - in a novel manner to provide transparent and low-latency access to remote-memory.

Virtualization is emerging as a fundamental paradigm that will drive the provisioning and management of resources in future large scale clusters. Virtualization is the ability to subdivide a pool of physical resources, such as computation, memory, storage, and bandwidth, into higher level logical partitions, each of which can be assigned to individual users of the resource pool. This high-level view of a cluster's resources greatly simplifies their provisioning and management. In this work, we specifically tackle the challenge of virtualizing the *distributed memory resources* in a cluster to support the memory needs of high-performance applications.

III. State of the Art

While the advantages of accessing memory over the network have been explored in prior works, their acceptance by high performance application community in running real-world memory-intensive applications has been scant in spite of their enormous potential benefits. The primary reason is that all earlier approaches advocate significant modifications to the end-systems that need the extra memory. These modifications range from application-specific programming interfaces that programmers must follow to changes in the end-host's operating system (OS) and device drivers.

In early 1990's the first two remote paging mechanisms were proposed [4], [8], both of which employed a customized remote paging protocol between clients and servers and incorporated extensive OS changes. Global Memory System (GMS) [7] provides network-wide memory management support at the lowest level of the OS for activities such as paging, memory mapped files and file caching. The Network RamDisk [9] offers remote paging with data replication and adaptive parity caching. Network RamDisk is implemented as a device driver inside the OS, thus increasing the installation and management complexity at each node in the LAN. Dodo [10] provides a user-level library based interface that a programmer that coordinates all data transfer to and from a remote memory cache. Samson [12] is a dedicated memory

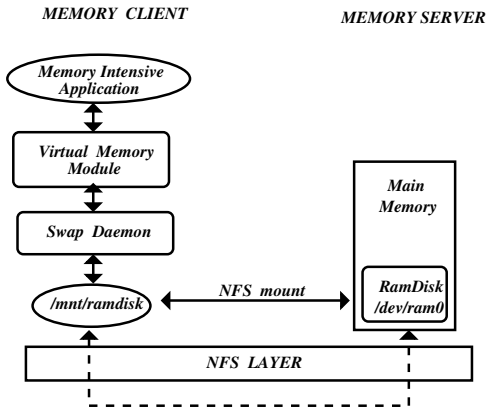

Fig. 1. A Basic Remote Memory Access Architecture using RamDisk & NFS. Clients transparently utilize remote memory instead of local disk.

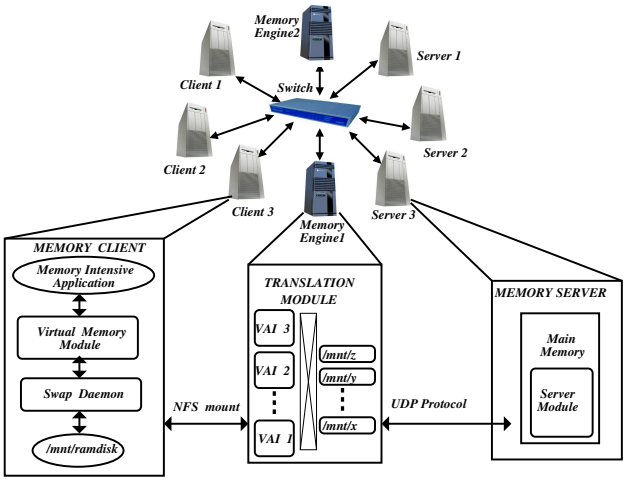

Fig. 2. Architecture of Anemone. Memory clients mount a Virtual Access Interface (VAI) from the Memory Engine, which corresponds to memory spaces hosted by back-end memory servers. The Memory Engine multiplexes client paging requests among back-end memory servers.

server that actively attempts to predict client page requirements and delivers the pages just-in-time to hide the paging latencies. The drivers and OS in both the memory server and clients are extensively modified. Simulation studies for a load sharing scheme that combines job migrations with the use of network RAM are presented in [13]. The NOW project [1] performs cooperative caching via a global file cache. Other efforts include software distributed shared memory (DSM) systems [6], and alternative cache replacement/replication strategies [3], [5].

IV. Architecture & Research Methodology

In contrast to prior efforts, Anemone maintains complete client transparency and does not require any OS, device driver or application level modifications in the clients' configuration. It exploits the standard Network File System (NFS) [11] protocol which is available by default with most end systems.

A. Basic Architecture: A Memory Client mounts a remote Linux "RamDisk" from a Memory Server and configures it as a swap device. A RamDisk is a portion of physical memory that one can set aside for access as a regular disk partition via the standard device interface (such as `/dev/ram0, /dev/ram1` etc. in Linux). The server can then export this RamDisk through NFS to be mounted. When the client mounts the RamDisk onto its local file-system, it can create a file inside the mounted RamDisk and configure the file as the primary swap device or allow applications to create memory-mapped files on the mount-point. Using this new swap space, applications on the memory client can transparently swap to remote memory instead of to disk. In Figure 1, this basic architecture is demonstrated. As a fallback option, one can also configure a local disk partition as a secondary swap device in the unlikely event that remote RamDisk becomes full. Next, we can augment this basic architecture in several ways more appropriate for enterprise scale applications.

B. The Memory Engine - An Agent for End-host Transparency: The Anemone project explores the design space of full-scale distributed remote memory paging by way of the fundamental entity - the *memory engine*. Simply speaking, the engine acts as a bridge between memory clients on one side and servers hosting the remote memory resources on the other side. The memory engine may not provide any memory resources; rather it helps pool together and present a unified access interface for distributed memory resources across the cluster. Figure 2 shows the architecture of Anemone. The memory engine is a dedicated device connected to the same switched Gigabit network as other memory clients and back-end servers. The client-side face of the engine consists of the *virtual access interface* (VAI), a pseudo device that can be NFS mounted by remote memory clients and configured as either a swap device or a memory mapped file. Swapped pages from the client are then directed over NFS to the remote VAI. Note that VAI is not a real device but only a convenient logical device through which clients can transparently access the distributed memory resources. On the server-side, instead of using RamDisks, we use a simple, reliable light-weight UDP-based communication protocol between the engine and the server. The server accepts pages through this protocol and stores them in a local in-memory data structure (such as a hash table). The memory engine tracks the memory resource usage on individual servers and transparently multiplexes client page requests to these back-end servers through the communication protocol.

Even though the Memory Engine does introduce an additional level of indirection in comparison to the basic architecture, it provides a much greater benefit by hiding all the complexity of managing global memory resources from the memory clients, and the complexity of managing client requirements from the memory servers. Additionally, the three different latency reduction mechanisms discussed later in Section IV-E can hide much of the overheads due to

indirection.

C. Virtualizing The Collective Memory Resource Pool: The Memory Engine is ideally placed to virtualize the collective back-end memory resources. Each VAI carries associated attributes such as logical capacity of the partition, its priority relative to other VAIs, reliability requirements, and current back-end servers storing VAI contents. A *translation module* at the memory engine maps the contents of each VAI to back-end servers. Thus, each client is attached to a single VAI while the translation module hides and efficiently manages the underlying virtualization complexities. The translation module resides between the data-link layer and the IP layer of the memory engine's protocol stack as a pluggable module that interacts with the runtime kernel. The translation module intercepts the clients' page requests from NFS and maps each one to a back-end server. For instance, if a client sends a page-out request to its VAI, the translation module intercepts, interprets, and schedules the request and sends it to a chosen back-end server. Similarly, for a page-in request, the translation module first identifies the server where the requested page is stored. Subsequently, the translation module itself can either retrieve and return the page to the client or take advantage of the *split page-in* latency reduction mechanism described later.

Anemone's virtualization techniques also enables the Memory Engine to transparently perform fine-grained memory resource management. Three specific resource management dimensions are being explored in Anemone: (1) Instead of dedicating a single back-end server's entire allocated memory space to each client, the Memory Engine could transparently *multiplex* the resources so that a server can hold the pages of multiple clients. This also has the benefit of distributing the load on the servers, keeping all servers equally utilized. (2) The Memory Engine can seamlessly absorb the increase/decrease in global paging capacity as servers constantly join or leave the network, thus insulating the memory clients from fluctuations in total system capacity. (3) When a particular server needs to reclaim its memory resources for its own applications, the Memory Engine can transparently migrate the pages stored at that server to other less loaded servers (using the communication protocol as described), thus adapting to load variations.

D. Scalability of Transparent Remote Memory Access: One may argue that a single Memory Engine can become a performance bottleneck if it has to service a large number of clients. We can easily address this bottleneck by placing multiple engines in the cluster (as shown in Figure 2), which share a common domain-level name but have different IP addresses. Clients can be transparently redirected to mount VAIs from the least loaded engine by employing a customized Domain Name Service (DNS) based load-balancing mechanism, which is just a one-time initialization overhead.

E. Latency Reduction Mechanisms: The most attractive feature of remote paging is that data transfer latencies over a network are orders of magnitude smaller than disk transfer latencies. Even though the Memory Engine can yield significant performance improvements without any special optimizations, it is still important to minimize network access latencies.

Page-out vs. Page-in: Writes (Page-out) can be delayed by allowing the swap daemon to schedule and buffer them based on the usage status of local memory. However, Reads (Page-in) are more critical because an application absolutely cannot continue after a page-fault without the required page. Thus, reads are the prime target of optimization efforts.

We are investigating three specific latency reduction mechanisms: (1) Split Page-in, (2) Concurrent paging and (3) Memory Engine Page Cache.

- **Split Page-in**: This technique optimizes a potentially expensive two-step transfer of large memory pages into a direct single-step transfer from the back-end server to the memory client, avoiding an extra hop through the Memory Engine.
- **Concurrent Paging**: This technique exploits the inherent concurrency in interacting with multiple back-end servers. A memory page can be broken into multiple packets that can be exchanged simultaneously between the Memory Engine and multiple back-end servers.
- **Memory Engine Page Cache**: In this technique, the Memory Engine can maintain an active *Page Cache* that contains copies of the most frequently accessed client pages. If a client's page-in request is successfully located in the page cache, the Memory Engine does not need to contact any back-end servers, thus reducing the page-in latency by almost half.

F. Compression and Its Impact on Latency and Storage: Another attractive option is to let the Memory Engine compress the swapped pages before transferring them to back-end servers, storing them in compressed form. Upon a page-in request, either the server or the engine can decompress the pages before returning them to the client. Compression at the Memory Engine provides two advantages: (1) The communication overhead between memory engine and back-end server is minimized and (2) The back-end server's total storage capacity greatly increases. The main challenge will be to balance the computation overhead of compressing and decompressing pages against the potential savings in communication latency and storage. We will also explore two novel features unique to Anemone: (1) To maintain a compressed page cache at the Memory Engine which stores frequently accessed remote pages and (2) To perform application-specific compression/decompression by observing the page content characteristics.

G. Reliability of Remote Memory Access: A degree of uncertainty arises from critical swap contents being remotely stored, possibly on multiple back-end servers. (1) First, we will explore strategies for efficient replication and striping of memory pages on multiple back-end servers. (2) Second we will explore the memory engine itself as a single point of failure. We propose to apply the *process-pair* concept [2]

in which the operations of a primary memory engine will be mirrored by a backup memory engine. The specific design issues in the process-pair approach include (1) how the backup engine will intercept NFS packets destined to the primary engine, (2) how the backup will maintain synchronization with the primary without impacting its normal operations, and (3) the mechanism of failure detection, such as using heartbeats, and recovery by redirecting the clients to the backup device.

H. Performance Testing Using Graphics Rendering and Database Applications: Graphics and image processing applications are one class of applications that need to process large amounts of data. For instance, volume rendering and ray tracing programs require enormous amounts of memory to perform rendering computations. Performance testing of Anemone will be conducted using a popular graphics rendering application called ***Povray***. Povray provides a meta-language to specify very large rendering scenarios. We will execute Povray on the Anemone system to demonstrate the improvement in rendering times. Similarly, a number of database applications need to sort large data-sets in order to satisfy queries to the database. Also, we will demonstrate the performance improvement of popular sorting algorithms such as QuickSort and HeapSort.

I. Preliminary Results: As a proof-of-concept Anemone prototype, we have implemented a user-level Memory Engine which exposes a set of in-memory VAIs to a client machine and intercepts/responds to NFS mount and read-write requests directly without depending upon the file-system stack. The Memory Engine also communicates with a back-end memory server to store page-in/page-out requests. We compared the latencies of page-in operations using the user-level implementation against disk-based paging when simultaneously executing multiple memory intensive applications. We observed that disk based page-in latencies varied from 6.3–7.1ms. On the other hand the page-in latencies using the Memory Engine on a Gigabit LAN varied between 1.8–2.1ms – an improvement of up to 3.5 times. The majority of the remote page-in overhead (about 1.3ms) lies in copying and context switching between user-space and kernel-space at the Memory Engine node. An efficient kernel-level Memory Engine is expected to reduce this overhead to within 500μs.

V. Conclusions

In this research we address the problem of harnessing the distributed memory resources in a cluster to support memory-intensive high-performance applications. We presented the architectural design and implementation aspects of Anemone - an Adaptive NEtwork MemOry Engine - which is designed to provide transparent, fault-tolerant, low-latency access distributed memory resources. The novel aspect of Anemone is that no modifications are required to either the memory intensive application or to the end-host system. Our preliminary results based on a user-level Anemone prototype promise up to 3.5 times improvement in page-in latencies

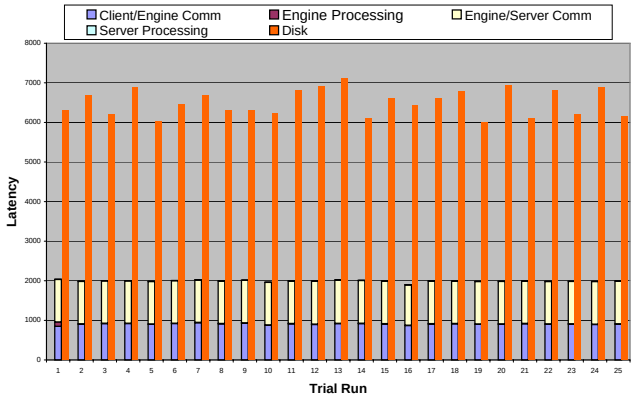

Fig. 3. Comparison of paging to local disk vs. paging to remote memory using user-level anemone implementation.

over disk-based paging mechanisms. Currently we are investigating kernel-level mechanisms to overcome data copying and context switching overheads, various latency reduction, reliability and scalability techniques, and finally benefits of using compression and caching.

References

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

[2] J. Bartlett. A nonstop kernel. In *Proc. 8th ACM SIGOPS SOSP*, 1981.

[3] F. Brasileiro, W. Cirne, E.B. Passos, and T.S. Stanchi. Using remote memory to stabilise data efficiently on an EXT2 linux file system. In *Proc. of the 20th Brazilian Symposium on Computer Networks*, May 2002.

[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] F.M. Cuenca-Acuna and T.D. Nguyen. Cooperative caching middleware for cluster-based servers. In *Proc. of 10th IEEE Intl. Symp. on High Performance Distributed Computing (HPDC-10)*, Aug 2001.

[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] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. Request for Comments - RFC 3530.

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

[13] L. Xiao, X. Zhang, and S.A. Kubricht. Incorporating job migration and network RAM to share cluster memory resources. In *Proc. of the 9th IEEE Intl. Symposium on High Performance Distributed Computing (HPDC-9)*, pages 71–78, August 2000.