Distributed Anemone: Transparent Low-Latency Access to Remote Memory

Michael R. Hines, Jian Wang, and Kartik Gopalan
Computer Science Department, Binghamton University
{mhines,jianwang,kartik}@cs.binghamton.edu

Abstract: Performance of large memory applications degrades rapidly once the system hits the physical memory limit and starts paging to local disk. We present the design, implementation and evaluation of *Distributed Anemone* (Adaptive Network Memory Engine) – a lightweight and distributed system that pools together the collective memory resources of multiple machines across a gigabit Ethernet LAN. Anemone treats remote memory as another level in the memory hierarchy between very fast local memory and very slow local disks. Anemone enables applications to access potentially "unlimited" network memory without any application or operating system modifications. Our kernel-level prototype features fully distributed resource management, low-latency paging, resource discovery, load balancing, soft-state refresh, and support for 'jumbo' Ethernet frames. Anemone achieves low page-fault latencies of 160μs average, application speedups of up to 4 times for single process and up to 14 times for multiple concurrent processes, when compared against disk-based paging.

1 Introduction

Performance of large-memory applications (LMAs) can suffer from large disk access latencies when the system hits the physical memory limit and starts paging to local disk. At the same time, affordable, low-latency, gigabit Ethernet is becoming commonplace with support for jumbo frames (packets larger than 1500 bytes). Consequently, instead of paging to a slow local disk, one could page over a gigabit Ethernet to the unused memory of remote machines and use the disk only when remote memory is exhausted. Thus, remote memory can be viewed as another level in the traditional memory hierarchy, filling the widening performance gap between low-latency RAM and high-latency disk. In fact, remote memory paging latencies of about 160μs or less can be easily achieved whereas disk read latencies range anywhere between 6 to 13ms. A natural goal is to enable unmodified LMAs to transparently utilize the collective remote memory of nodes across a gigabit Ethernet LAN. Several prior efforts [1–8] have addressed this problem by relying upon expensive interconnect hardware (ATM or Myrinet switches), slow bandwidth limited LANs (10Mbps/100Mbps), or heavyweight software Distributed Shared Memory (DSM) [9, 10] systems that require intricate consistency/coherence techniques and, often, customized application programming interfaces. Additionally, extensive changes were often required to the LMAs or the OS kernel or both.

Our earlier work [11] addressed the above problem through an initial prototype, called the ***Adaptive Network Memory Engine (Anemone)*** – the first

attempt at demonstrating the feasibility of transparent remote memory access for LMAs over commodity gigabit Ethernet LAN. This was done without requiring any OS changes or recompilation, and relied upon a central node to map and exchange pages between nodes in the cluster. Here we describe the implementation and evaluation of a *fully distributed* Anemone architecture. Like the centralized version, distributed Anemone uses lightweight, pluggable Linux kernel modules and does not require any OS changes. Additionally, it achieves the following significant improvements over centralized Anemone. **(1)** Memory resource management is distributed across the whole cluster. There is no single control node. **(2)** Paging latency is reduced by over a factor of 3 – from around $500\mu s$ in the to less than $160\mu s$. **(3)** Clients can perform load-balancing across multiple memory servers, taking into account their memory usage and paging load. **(4)** A *distributed resource discovery* mechanism enables clients to discover newly available servers and track memory usage across the cluster. **(5)** A *soft-state refresh* mechanism enables memory servers to track the liveness of clients and their pages. **(6)** The distributed version incorporates the flexibility of whether or not 'jumbo' frames should be used, allowing Anemone to operate in networks with any MTU size. **(7)** We are currently incorporating reliability into the paging process, where clients can replicate pages to protect against server failures. We evaluated our prototype using unmodified LMAs such as ray-tracing, network simulations, in-memory sorting, and k-nearest neighbor search. Results show that average page-fault latencies reduce from 8.3ms to $160\mu s$, single-process applications speed up by up to a factor of 4, and multiple concurrent processes by up to a factor of 14, when compared against disk-based paging.

2 Design and Implementation

Distributed Anemone has two major software components: the *client module* on low memory machines and the *server module* on machines with unused memory. The client module appears to the client system simply as a block device that can be configured as the primary swap device. Whenever an LMA needs more virtual memory, the pager (swap daemon) in the client swaps out pages from the client to other server machines. As far as the pager is concerned, the client module is just a block device not unlike a hard disk partition. Internally, however, the client module maps swapped out pages to remote memory servers.

The servers themselves are also commodity machines, but have unused memory to contribute, and can in fact switch between the roles of client and server at different times, depending on their memory requirements. Client machines discover available servers by using a simple distributed resource discovery mechanism. Servers provide regular feedback about their load information to clients, both as a part of the resource discovery process and as a part of regular paging process. Clients use this information to schedule page-out requests. For instance, a client can simply choose the least loaded server node to send a new page. Also, both the clients and servers use a soft-state refresh protocol to maintain the liveness of pages stored at the servers. The earlier Anemone prototype differed

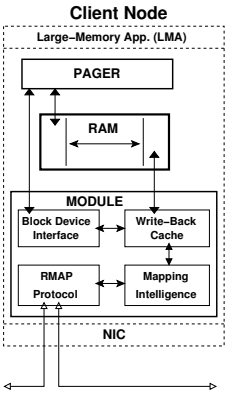

Fig. 1. The components of a client.

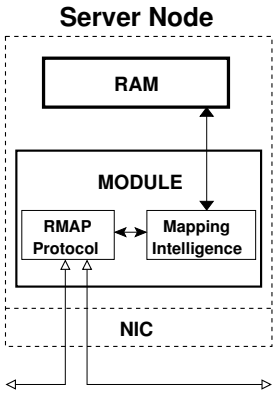

Fig. 2. The components of a server.

in that the page-to-server mapping logic was maintained at a central *Memory Engine*, instead of individual client nodes. Although simpler to implement, this centralized architecture incurred two extra round trip times on every request besides forcing *all traffic* to go through the central Memory Engine, which can become a single point of failure and a significant bottleneck.

2.1 Client and Server Modules

Figure 1 illustrates the client module that handles paging operations. It has four major components: (1) The Block Device Interface (BDI), (2) a basic LRU-based write-back cache, (3) mapping logic for server location of swapped-out pages, and (4) a Remote Memory Access Protocol (RMAP) layer. The pager issues read and write requests to the BDI in 4KB data blocks. The BDI, in turn, performs read and write operations to our write-back cache (for which pages do not get transmitted until eviction). When the cache is full, a page is evicted to a server using RMAP. Figure 2 illustrates the two major components of the server module: (1) a hash table that stores client pages along with client's identity (layer-2 MAC address) and (2) the RMAP layer. The server module can store/retrieve pages for any client machine. Once the server reaches capacity, it responds to the requesting client with a negative acknowledgment. It is then the client's responsibility to select another server, if available, or to page to disk if necessary. Page-to-server mappings are kept in small hashtables whose buckets are allocated using the `get_free_pages()` call. Linked-lists contained within each bucket hold 64-byte entries that are managed using the Linux slab allocator (which performs fine-grained management of small, equal-sized memory objects). Standard disk block devices interact with the kernel through a *request queue* mechanism, which permits the kernel to group spatially consecutive block I/Os (BIO) together into one "request" and schedule them using an elevator algorithm for seek-time minimization. Unlike disks, Anemone is essentially random access with a fixed read/write latency. Thus BDI does not need to group sequential BIOs. It can bypass request queues, perform out-of-order transmissions, and asynchronously handle un-acknowledged, outstanding RMAP messages.

2.2 Transparent Virtualization

To enable LMAs to transparently access remote memory (no relinking or recompilation), the client module exports a BDI to the pager. Additionally, no changes are required to the core OS kernel because the BDI is implemented as a self-contained kernel module. One can invoke the standard `open, read,` and `write` system calls on the BDI like any other block device. Although our paper does not focus on this aspect, the BDI can be used as a low-latency store for temporary files and can even be memory-mapped by applications aware of the remote memory. The system also performs two types of multiplexing in the presence of multiple clients and servers: (a) Any single client can transparently access memory from multiple servers as one pool via the BDI, and (b) Any single server can share its unused memory pool among multiple clients simultaneously. This provides the maximum flexibility in efficiently utilizing the global memory pool and avoids resource fragmentation.

2.3 Remote Memory Access Protocol (RMAP)

RMAP is a tailor-made, low-overhead communication protocol for remote memory access within the same subnet. It implements the following features: (1) Reliable Packet Delivery, (2) Flow-Control, and (3) Fragmentation and Reassembly. While one could technically communicate over TCP, UDP, or even the IP protocol layers, this choice comes burdened with unwanted protocol processing. Instead RMAP takes an integrated, faster approach by communicating directly with the network device driver, sending frames and handling reliability issues in a manner that suites the needs of the Anemone system. Every RMAP message is acknowledged except for soft-state and dynamic discovery messages. Timers trigger retransmissions when necessary (which is extremely rare) to guarantee reliable delivery. We cannot allow a paging request to be lost, or the application that depends on that page will fail altogether. RMAP also implements flow control to ensure that it does not overwhelm either the receiver or the intermediate network card and switches. However, RMAP does not require TCP's features such as byte-stream abstraction, in-order delivery, or congestion control. Hence we chose to implement RMAP as a light-weight window-based reliable datagram protocol. All client nodes keep a static-size window to control the transmission rate, which works very well for purely in-cluster communication.

The last design consideration in RMAP is that while the standard memory page size is 4KB (or sometimes 8KB), the maximum transmission unit (MTU) in traditional Ethernet networks is limited to 1500 bytes. RMAP implements dynamic fragmentation/reassembly for paging traffic. Additionally, RMAP also has the flexibility to use *Jumbo frames*, which are packets with sizes greater than 1500 bytes (typically between 8KB to 16KB). Jumbo frames enable RMAP to transmit complete 4KB pages to servers using a single packet, without fragmentation. Our testbed includes an 8-port switch that supports Jumbo Frames (9KB packet size). We observe a 6% speed up in RMAP throughput by using Jumbo Frames. In this paper, we conduct all experiments with 1500 byte MTU with fragmentation/reassembly performed by RMAP.

2.4 Distributed Resource Discovery

As servers constantly join or leave the network, Anemone can (a) seamlessly absorb the increase/decrease in cluster-wide memory capacity, insulating LMAs from resource fluctuations and (b) allow any server to reclaim part or all of its contributed memory. This objective is achieved through *distributed resource discovery* described below, and *soft-state refresh* described next in Section 2.5. Clients can discover newly available remote memory in the cluster and the servers can announce their memory availability. Each server periodically broadcasts a *Resource Announcement* (RA) message (1 message every 10 seconds in our prototype) to advertise its identity and the amount of memory it is willing to contribute. Besides RAs, servers also piggyback their memory availability information in their page-in/page-out replies to individual clients. This distributed mechanism permits any new server in the network to dynamically announce its presence and allows existing servers to announce their up-to-date memory availability information to clients.

2.5 Soft-State Refresh

Distributed Anemone also includes soft-state refresh mechanisms (keep-alives) to permit clients to track the liveness of servers and vice-versa. Firstly, the RA message serves an additional purpose of informing the client that the server is alive and accepting paging requests. In the absence of any paging activity, if a client does not receive the server's RA for three consecutive periods, it assumes that the server is offline and deletes the server's entries from its hashtables. If the client also had pages stored on that server that went offline, it needs to recover the corresponding pages from a copy stored either on the local disk on on another server's memory. Soft-state also permits servers to track the liveness of clients whose pages they store. Each client periodically transmits a *Session Refresh* message to each server that hosts its pages (1 message every 10 seconds in our prototype), which carries a client-specific session ID. The client module generates a different and unique ID each time the client restarts. If a server does not receive refresh messages with matching session IDs from a client for three consecutive periods, it concludes that the client has failed or rebooted and frees up any pages stored on that client's behalf.

2.6 Server Load Balancing

Memory servers themselves are commodity nodes in the network that have their own processing and memory requirements. Hence another design goal of Anemone is to a avoid overloading any one server node as far as possible by transparently distributing the paging load evenly. In the earlier centralized architecture, this function was performed by the memory engine which kept track of server utilization levels. Distributed Anemone implements additional coordination among servers and clients to exchange accurate load information. Section 2.4 described the mechanism to perform resource discovery. Clients utilize

the server load information gathered from resource discovery to decide the server to which they should send new page-out requests. This decision process is based upon one of two different criteria: (1) The number of pages stored at each active server and (2) The number of paging requests serviced by each active server. While (1) attempts to balance the memory usage at each server, (2) attempts to balance the request processing overhead.

2.7 Fault-tolerance

The ultimate consequence of failure in swapping to remote memory is no worse than failure in swapping to local disk. However, the probability of failure is greater in a LAN environment because of multiple components involved in the process, such as network cards, connectors, switches etc. Although RMAP provides reliable packet delivery as described in Section 2.3 at the protocol level, we are currently implementing two alternatives for tolerating server failures: (1) To maintain a local disk-based copy of every memory page swapped out over the network. This provides same level of reliability as disk-based paging, but risks performance interference from local disk activity. (2) To keep redundant copies of each page on multiple remote servers. This approach avoids disk activity and reduces recovery-time, but consumes bandwidth, reduces the global memory pool and is susceptible to network failures.

3 Performance

The Anemone testbed consists of one 64-bit low-memory AMD 2.0 GHz client machine containing 256 MB of main memory and nine remote-memory servers. The DRAM on these servers consist of: four 512 MB machines, three 1 GB machines, one 2 GB machine, one 3 GB machine, totaling to almost 9 gigabytes of remote memory. The 512 MB servers range from 1.7 GHz to 800 MHz Intel processors. The other 5 machines are all 2.7 GHz and above Intel Xeons, with mixed PCI and PCI express motherboards. For disk based tests, we used a Western Digital WD800JD 80 GB SATA disk, with 7200 RPM speed, 8 MB of cache and $8.9ms$ average seek time, (which is consistent with our results). This disk has a 10 GB swap partition reserved on it to match the equivalent amount of remote memory available in the cluster, which we use exclusively when comparing our system against the disk. Each machine is equipped with an Intel PRO/1000 gigabit Ethernet card connected to one of two 8-port gigabit switches, one from Netgear and one from SMC. The performance results presented below can be summarized as follows. Distributed Anemone reduces read latencies to an average $160\mu s$ compared to $8.3ms$ average for disk and $500\mu s$ average for centralized Anemone. For writes, both disk and Anemone deliver similar latencies due to write caching. In our experiments, Anemone delivers a factor of 1.5 to 4 speedup for single process LMAs, and delivers up to a factor of 14 speedup for multiple concurrent LMAs. Our system can successfully operate with both multiple clients and multiple servers. Due to space constraints, we omit the speedup results for multiple clients, which emulates the single-client speedups.

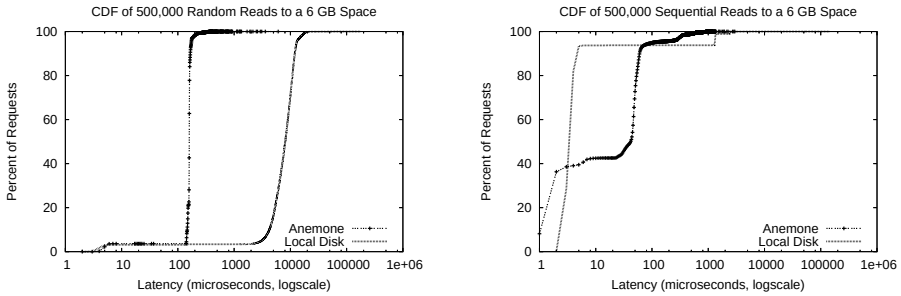

Fig. 3. Comparison of latency distributions for random and sequential reads.

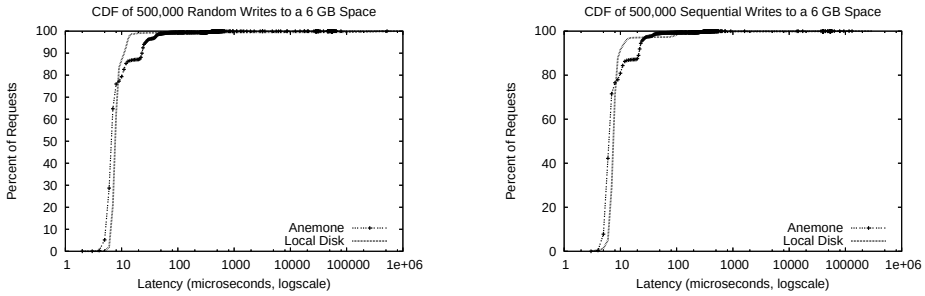

Fig. 4. Comparison of latency distributions for random and sequential writes.

3.1 Paging Latency

Figures 3 and 4 show the distribution of observed read and write latencies for sequential and random access patterns with both Anemone and disk. Though real-world applications rarely generate purely sequential or completely random memory access patterns, these graphs provide a useful measure to understand the underlying factors that impact application execution times. Most random read requests to disk experience a latency between 5 to 10 milliseconds. On the other hand most requests in Anemone experience only around $160\mu s$ latency. Most sequential read requests to disk are serviced by the on-board disk cache within 3 to $5\mu s$ because sequential read accesses fit well with the motion of disk head. In contrast, Anemone delivers a range of latency values, most below $100\mu s$. This is because network communication latency dominates in Anemone even for sequential requests, though it is masked to some extent by the prefetching performed by the pager and the file-system. The write latency distributions for both disk and Anemone are comparable, with most latencies being close to $9\mu s$ because writes typically return after writing to the buffer cache.

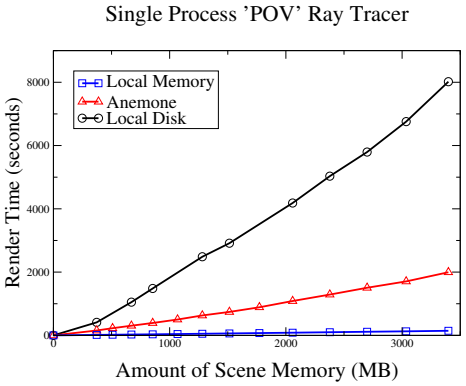

Fig. 5. Execution times of POV-ray for increasing problem sizes.

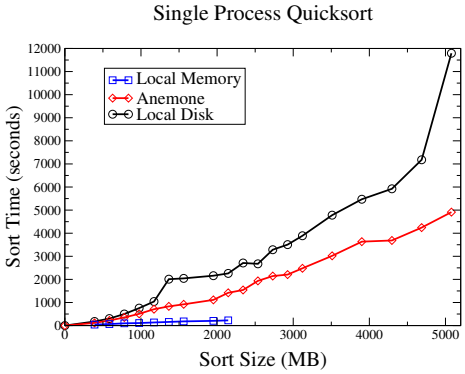

Fig. 6. Execution times of STL Quicksort for increasing problem sizes.

	Size (GB)	Local Mem	Distr. Anemone	Disk	Speedup $\frac{Disk}{Anemone}$
Povray	3.4	145	1996	8018	4.02
Quicksort	5	N/A	4913	11793	2.40
NS2	1	102	846	3962	4.08
KNN	1.5	62	721	2667	3.7

Table 1. Average application execution times and speedups for local memory, Distributed Anemone, and Disk. N/A indicates insufficient local memory.

3.2 Application Speedup

Single-Process LMAs: Table 1 summarizes the performance improvements seen by unmodified single-process LMAs using the Anemone system. The first application is a **ray-tracing program called POV-Ray**. The memory consumption of POV-Ray was varied by rendering different scenes with increasing number of colored spheres. Figure 5 shows the completion times of these increasingly large renderings up to 3.4 GB of memory versus the disk using an equal amount of local swap space. The figure clearly shows that Anemone delivers increasing application speedups with increasing memory usage and is able to improve the execution time of a single-process POV-ray by a factor of 4 for 3.4 GB memory usage. The second application is a **large in-memory Quicksort program** that uses a C++ STL-based implementation, with a complexity of O($N \log N$) comparisons. We sorted randomly populated large in-memory arrays of integers. Figure 6 shows that Anemone delivers a factor 2.4 speedup for a single-process Quicksort using 5 GB of memory. The third application is the popular **NS2 network simulator**. We simulated a delay partitioning algorithm [12] on a 6-hop wide-area network path using voice-over-IP traffic traces. Table 1 shows that, with NS2 requiring 1GB memory, Anemone speeds up the simulation by a factor of 4 compared to disk based paging. The fourth application is the **k-nearest neighbor (KNN)** search algorithm on large 3D datasets, which are useful in applications such as medical imaging, molecular biology, CAD/CAM, and mul-

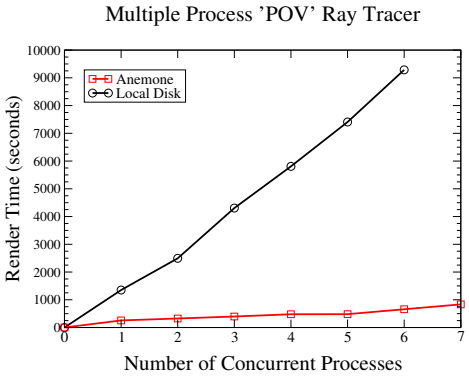

Fig. 7. Execution times of multiple concurrent processes executing POV-ray.

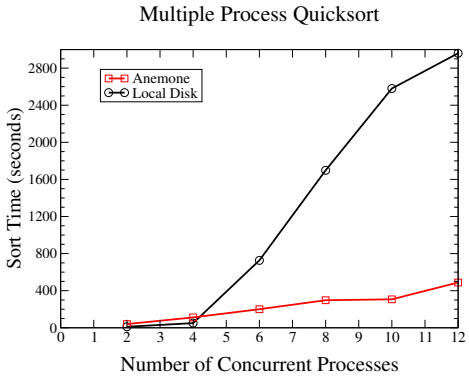

Fig. 8. Execution times of multiple concurrent processes executing STL Quicksort.

timedia databases. Table 1 shows that, when executing KNN search algorithm over a dataset of 2 million points consuming 1.5GB memory, Anemone speeds up the simulation by a factor of 3.7 over disk based paging.

Multiple Concurrent LMAs: In this section, we test the performance of Anemone under varying levels of concurrent application execution. Multiple concurrently executing LMAs tend to stress the system by competing for computation, memory and I/O resources and by disrupting any sequentiality in paging activity. Figures 7 and 8 show the execution time comparison of Anemone and disk as the number of POV-ray and Quicksort processes increases. The execution time measures the time interval between the start of the execution and the completion of last process in the set. We try to keep each process at around 100 MB of memory. The figures show that the execution times using disk-based swap increases steeply with number of processes. Paging activity loses sequentiality with an increasing number of processes, making the disk seek and rotational overheads dominant. On the other hand, Anemone reacts very well as execution time increases very slowly, due to the fact that network latencies are mostly constant, regardless of sequentiality. With 12–18 concurrent LMAs, Anemone achieves speedups of a factor of 14 for POV-ray and a factor of 6.0 for Quicksort.

3.3 Tuning the Client RMAP Protocol

One of the important knobs in RMAP's flow control mechanism is the client's transmission window size. Using a 1 GB Quicksort, Figure 9 shows the effect of changing this window size on three characteristics of the Anemone's performance: (1) the number of retransmissions, (2) paging bandwidth, which is represented in terms of "goodput", i.e. the amount of bandwidth obtained after excluding retransmitted bytes and header bytes, and (3) completion time. As the window size increases, the number of retransmissions increases because the number of

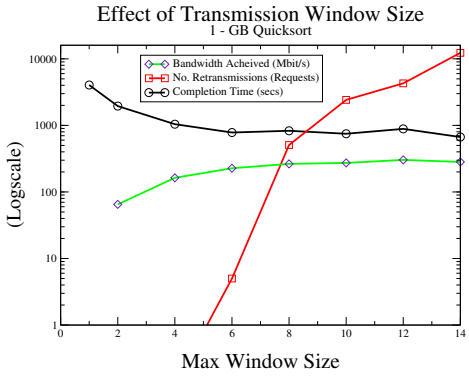

Fig. 9. Effects of varying the transmission window using Quicksort.

packets that can potentially be delivered back-to-back also increases. For larger window sizes, the paging bandwidth is also seen to increase and saturates because the transmission link remains busy more often, delivering higher goodput in spite of an initial increase in the number of retransmissions. However, if driven too high, the window size will cause the paging bandwidth to decline considerably due to increasing number packet drops and retransmissions. The application completion times depend upon the paging bandwidth. Initially, an increase in window size increases the paging bandwidth and lowers the completion times. Similarly, if driven too high, the window size causes more packet drops, more retransmissions, lower paging bandwidth and higher completion times.

3.4 Control Message Overhead

To measure the control traffic overhead due to RMAP, we measured the percentage of control bytes generated by RMAP compared to the amount of data bytes transferred while executing a 1GB POVRay application. Control traffic refers to the page headers, acknowledgments, resource announcement messages, and soft-state refresh messages. We first varied the number of servers from 1 to 6, with a single client executing the POV-Ray application. Next, we varied the number of clients from 1 to 4 (each executing one instance of POV-Ray), with 3 memory servers. The percentage control traffic overhead was consistently measured at 1.74% – a very small percentage of the total paging traffic.

4 Related Work

To the best of our knowledge, Anemone is the first system that provides unmodified LMAs with a completely transparent and virtualized access to cluster-wide remote memory over commodity gigabit Ethernet LANs. The earliest efforts [13, 14] in using remote memory aimed to improve memory management, recovery, concurrency control, and read/write performance for in-memory database and

transaction processing systems. The first two remote paging systems [1, 2] incorporated extensive OS changes to both the client and the memory servers and operated upon 10Mbps Ethernet. The Global Memory System (GMS) [3] 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 and operated upon a 155Mbps DEC Alpha ATM Network. The Dodo project [4, 15] provides a user-level library based interface that a programmer can use to coordinate all data transfers to and from a remote memory cache, requiring legacy applications to be modified. Work in [5] implements a remote memory paging system in the DEC OSF/1 operating system as a customized device driver over 10Mbps Ethernet. A remote paging mechanism [7] specific to the Nemesis operating system was designed to permit application-specific remote memory access and paging. The Network RamDisk [6] offers remote paging with data replication and adaptive parity caching by means of a device driver based implementation. Other remote memory efforts include software distributed shared memory (DSM) systems [9, 10], which allow a set of independent nodes to behave as a large shared memory multi-processor, often requiring customized programming to share common data across nodes. This goal is different from that of Anemone which allows unmodified application binaries to execute and use remote memory transparently. Samson [8] is a dedicated memory server with a highly modified OS over a Myrinet interconnect that actively attempts to predict client page requirements and delivers the pages just-in-time to hide the paging latencies. The NOW project [16] performs cooperative caching via a global file cache in the xFS file system, while [17] attempts to avoid inclusiveness within the cache hierarchy.

5 Conclusions

In this paper, we presented Distributed Anemone – a system that enables unmodified large memory applications to transparently utilize the unused memory of nodes across a gigabit Ethernet LAN. Unlike its centralized predecessor, Distributed Anemone features fully distributed memory resource management, low-latency remote memory paging, distributed resource discovery, load balancing, soft-state refresh to track liveness of nodes, and the flexibility to use Jumbo Ethernet frames. We presented the architectural design and implementation details of a fully operational Anemone prototype. Evaluations using multiple real-world applications, include ray-tracing, large in-memory sorting, network simulations, and nearest neighbor search, show that Anemone speeds up single process application by up to a factor of 4 and multiple concurrent processes by up to a factor of 14, compared to disk-based paging. Average page-fault latencies are reduced from 8.3ms with disk based paging to 160μs with Anemone. There are several exciting avenues for further research in Anemone. We are incorporating fault-tolerance mechanisms into Anemone using page replication across servers as well as local disk. Additionally, compression of pages holds the potential to further reduce communication overhead and increase the effective storage capacity.

References

1. Comer, D., Griffoen, J.: A new design for distributed systems: the remote memory model. In Proc. of the USENIX 1991 Summer Technical Conference (1991) 127–135
2. Felten, E., Zahorjan, J.: Issues in the implementation of a remote paging system. Tech. Report 91-03-09, Comp. Science Dept., University of Washington (1991)
3. Feeley, M., Morgan, W., Pighin, F., Karlin, A., Levy, H.: Implementing global memory management in a workstation cluster. Operating Systems Review, 15th ACM Symposium on Operating Systems Principles **29**(5) (1995) 201–212
4. Koussih, S., Acharya, A., Setia, S.: 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)
5. Markatos, E., Dramitinos, G.: Implementation of a reliable remote memory pager. In: USENIX Annual Technical Conference. (1996) 177–190
6. Flouris, M., Markatos, E.: The network RamDisk: Using remote memory on heterogeneous NOWs. Cluster Computing **2**(4) (1999) 281–293
7. McDonald, I.: Remote paging in a single address space operating system supporting quality of service. Tech. Report, Dept. of Comp. Science, Univ. of Glasgow (1999)
8. Stark, E.: SAMSON: A scalable active memory server on a network (Aug. 2003)
9. Dwarkadas, S., Hardavellas, N., Kontothanassis, L., Nikhil, R., Stets, R.: Cashmere-VLM: Remote memory paging for software distributed shared memory. In: Proc. of Intl. Parallel Processing Symposium, San Juan, Puerto Rico. (April 1999) 153–159
10. Amza, C., et. al., A.C.: Treadmarks: Shared memory computing on networks of workstations. IEEE Computer **29**(2) (Feb. 1996) 18–28
11. Hines, M., Lewandowski, M., Gopalan, K.: Implementation experiences in transparently harnessing cluster-wide memory. In: Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), Calgary, Canada (Aug. 2006)
12. Gopalan, K., Chiueh, T.: Delay budget partitioning to maximize network resource usage efficiency. In: Proc. IEEE INFOCOM'04, Hong Kong, China. (March 2004)
13. Garcia-Molina, H., Lipton, R., Valdes, J.: A massive memory machine. IEEE Transactions on Computers **C-33 (5)** (1984) 391–399
14. Bohannon, P., Rastogi, R., Silberschatz, A., Sudarshan, S.: The architecture of the Dali main memory storage manager. Bell Labs Technical Journal **2**(1) (1997) 36–47
15. Acharya, A., Setia, S.: Availability and utility of idle memory in workstation clusters. In: Measurement and Modeling of Computer Systems. (1999) 35–46
16. Anderson, T., Culler, D., Patterson, D.: A case for NOW (Networks of Workstations). IEEE Micro **15**(1) (1995) 54–64
17. Wong, T., Wilkes, J.: My cache or yours? Making storage more exclusive. In: Proc. of the USENIX Annual Technical Conference. (2002) 161–175