Implementation Experiences in Transparently Harnessing Cluster-Wide Memory

Michael R. Hines, Mark Lewandowski, Jian Wang, and Kartik Gopalan
Computer Science Department, Florida State University

Abstract

There is a constant battle to break even between continuing improvements in DRAM capacities and the growing memory demands of large-memory high-performance applications. Performance of such applications degrades quickly once the system hits the physical memory limit and starts swapping to the local disk. In this paper, we investigate the benefits and tradeoffs in pooling together the collective memory resources of nodes across a high-speed LAN based cluster. We present the design, implementation and evaluation of *Anemone* – an Adaptive Network Memory Engine – that virtualizes the collective unused memory of multiple machines across a gigabit Ethernet LAN, without requiring any modifications to the large memory applications. We have implemented a working prototype of Anemone and evaluated it using real-world unmodified applications such as ray-tracing and large in-memory sorting. Our results with Anemone prototype show that unmodified single-process applications execute 2 to 3 times faster and multiple concurrent processes execute 6 to 7.7 times faster, when compared to disk based paging. The Anemone prototype reduces page-fault latencies by a factor of 19.6 – from an average of 9.8ms with disk based paging to 500μs with Anemone. Most importantly, Anemone provides a virtualized low-latency access to potentially "unlimited" memory resources across the network.

1 Introduction

Rapid improvements in DRAM capacities have been unable to keep up with the unprecedented the growth in memory demands of applications, such as multimedia/graphics processing, high resolution volumetric rendering, weather prediction, large-scale simulations, and large databases. The issue is not whether one can provide enough DRAM to satisfy these modern memory-hungry applications; rather, provide more memory and they'll use it all up and ask for even more. Simply buying more memory to plug into a single machine is neither sustainable nor economical for most users because (1) the price per byte of DRAM within a single node increases non-linearly and rapidly, (2) memory bank limitations within commodity nodes prevent unrestricted scaling and (3) investments in large memory servers with specialized hardware are prohibitively expensive and such technology itself quickly becomes obsolete.

In this constant battle to break-even, it does not take very long for large memory applications to hit the physical memory limit and start swapping (or paging) to physical disk, which in turn throttles their performance. At the same time, it is often the case that while memory resources in one machine might be heavily loaded, large amounts of memory in other machines in a high-speed LAN might remain idle or under-utilized. In typical commodity clusters, one often sees a mixed batch of applications of which some have very high memory demands, while most have only low or moderate demands and the cluster-wide resource utilization levels range around 5% to 20%. Consequently, instead of paging directly to a slow local disk, one could significantly reduce access latencies by first paging over a high-speed LAN to the unused memory of remote machines and then turn to disk-based paging only as the last resort after exhausting the available remote memory. As shown in Figure 1, remote memory access can be viewed as another level in the traditional memory hierarchy which fills the widening performance gap between very low latency access to main memory and high latency access to local disk. In fact, remote memory paging latencies of about 500 μs or less can be easily achieved whereas the latency of paging to slow local disk (especially while paging in) can be as much as 6 to 13ms depending upon seek and rotational overheads. *Thus remote memory paging could potentially be one to two orders of magnitude faster than paging to slow local disks.*

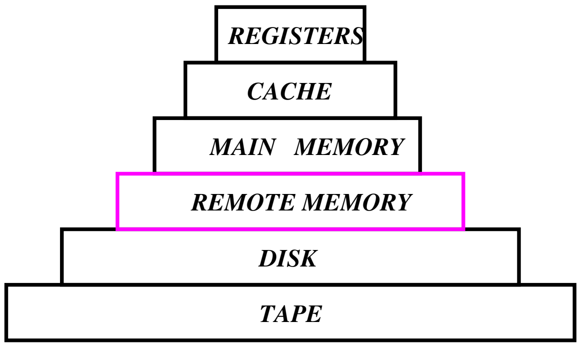

Figure 1: The position of remote memory in the memory hierarchy.

Recent years have also seen a phenomenal rise in affordable gigabit Ethernet LANs that provide low latency, support for jumbo frames (packet sizes greater than 1500 bytes), and offer attractive cost-to-performance ratios. An interesting question naturally follows from the above discussion: *Can we transparently virtualize (or pool together) the collective unused memory of commodity nodes across a high-speed LAN and enable unmodified large memory applications to avoid the disk access bottleneck by using this collective memory resource?* Prior efforts [6, 11, 10, 16, 18, 12, 19, 22] to address this problem have either relied upon expensive interconnect hardware (such as ATM or Myrinet switches) or used bandwidth limited 10Mbps or 100Mbps networks that are far too slow to provide meaningful application speedups. In addition, extensive changes were often required either to the large memory applications or the end-host operating system or even both. Note that the above research question of transparent remote memory access is different from the research on Distributed Shared Memory (DSM) [9] systems that permit nodes in a network to behave as if they were shared memory multiprocessors, often requiring use of customized application programming interfaces.

This paper presents our experiences in the design, implementation and evaluation of the ***Adaptive Network Memory Engine (Anemone)*** system. Anemone virtualizes the collective unused memory resources across nodes in a commodity gigabit Ethernet LAN and allows *unmodified* memory-hungry applications to transparently benefit from low-overhead access to potentially "unlimited" remote memory resources. The Anemone system makes the following specific contributions.

- Anemone presents a unified *virtual interface* for each memory client to access an aggregated pool of remote memory.
- Any application can benefit from Anemone without requiring any code changes, recompilation, or relinking. Anemone system is implemented in pluggable kernel modules and does not require any changes to the core operating system.
- Anemone is designed to automatically adapt as the applications' memory requirements change and/or the cluster-wide memory resource availability changes. It effectively isolates the clients from changes in the aggregate memory pool and isolates the memory contributors from changes in client memory requirements.
- To the best of our knowledge, Anemone system is the first attempt at evaluating the feasibility of remote memory access over commodity gigabit Ethernet with jumbo frame support.

We conducted performance evaluations of the Anemone system using two real-world unmodified applications – ray-tracing and large in-memory sorting. Anemone reduces page-fault latencies by a factor of 19.6 – from an average of 9.8ms with disk based paging to about 500μs with Anemone. Anemone speeds up single-process large-memory applications by a factor of 2 to 3 and multiple concurrent large-memory applications by a factor of 6 to 7.7. Our implementation experience indicates that important factors in this performance improvement include the design of a low-level lightweight reliable communication protocol, effective flow-control and caching.

2 Design Space of Anemone

This section examines the design alternatives, tradeoffs and practical rationale behind Anemone.

(A) Transparent Virtualization: The first design consideration in Anemone is to enable memory-intensive applications to transparently benefit from remote memory resources without requiring any modifications, recompilation or relinking of the application. The first alternative to is to modify the virtual memory subsystem of the operating system (OS) such that, when free local memory is low, page-table entries of a process can first be mapped to remote memory pages, and then to swap space on local disk. Although transparent to the application, this requires modifications to the OS kernel. The second approach is to virtualize the remote memory resources using a *pseudo block device*. The pseudo block device can be used by the swap daemon as a primary swap partition to which the pages are swapped instead of the local disk. This approach is simpler to realize as a self-contained kernel module that does not require changing the core OS and permits the swap daemon to handle all paging related decisions to both the remote memory and the local disk. For these reasons, we adopt the second approach. In addition, the pseudo block device can also be treated as a low-latency store for temporary files and can even be memory-mapped by applications aware of the remote memory.

(B) Remote Memory Access Protocol (RMAP): A number of design requirements need to be addressed by the communication protocol for remote memory access. The first requirement is to ensure that the latency in accessing a remote page of memory is not dominated by protocol processing overheads. A low-overhead approach is to implement a light-weight RMAP that executes directly on top of the network device driver, without the overhead of intervening protocol layers such as TCP, UDP or IP. The second design consideration is that while the standard memory page size is 4KB (or sometimes 8KB), the maximum packet size (also called the maximum transmission unit or MTU) in traditional Ethernet networks is limited to 1500 bytes. To avoid the overhead of fragmentation/reassembly of a memory page to/from multiple packets, we exploit the feature of *jumbo frames* in modern gigabit LANs. Jumbo frames are packets with sizes greater than 1500 bytes (typically between 8KB to 16KB) and enable the RMAP to transmit complete 4KB pages to remote memory using a single packet. The third design consideration is that RMAP needs to ensure reliability against memory pages being dropped by network cards and switches under heavy loads. RMAP also requires 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 choose to implement RMAP as a light-weight window-based reliable datagram protocol.

(C) Centralized vs. Distributed Resource Management: Another central design choice is whether to manage the global memory pool in a centralized or in a distributed manner. A central entity has a global view of memory resources in the LAN enabling efficient resource management. However, it can also become a bottleneck and a single point of failure besides introducing an additional level of indirection in remote memory access. A carefully designed distributed management model does not have these disadvantages, but requires greater coordination among participant nodes to efficiently utilize global memory. While we believe that a distributed approach is better when considering long-term scalability and performance, our immediate practical consideration was to quickly investigate whether virtualized remote memory access was feasible in the first place. Hence, our first Anemone prototype uses a central memory engine, with flexibility to migrate to a distributed design as our next step.

(D) Multiplexing: In a network of multiple remote memory consumers (clients) and remote memory contributors (servers), it is important to support two types of multiplexing: (a) any single client must be able to harness memory from multiple servers and access it transparently as one pool via the pseudo block device interface and (b) any single server should be capable of sharing its memory pool among multiple clients simultaneously. This provides the maximum flexibility in efficiently utilizing the global memory pool and avoids resource fragmentation.

(E) Load Balancing: Memory contributors themselves are other commodity nodes in the network that may have their own processing and memory requirements. Hence another design goal of the Anemone system is to a avoid overloading any one memory server as far as possible by transparently distributing client loads among back-end servers. In the centralized model, this function is performed by the memory engine which keeps track of server utilization levels. The distributed model requires additional coordination among servers and clients to exchange accurate load information.

(F) Adaptation to Resource Variation: As memory contributors constantly join or leave the network, the Anemone system should: (a) seamlessly absorb the increase/decrease in cluster-wide memory capacity, insulating the memory clients from resource fluctuations, (b) allow any server to reclaim part or whole of its contributed memory by transparently migrating pages to other less loaded servers. Thus Anemone needs to scale by growing and shrinking the size of it's aggregated memory space dynamically and fall back to disk in case the remote memory pool is over committed.

(G) 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. There are two feasible alternatives for fault-tolerance. (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 global memory pool and is susceptible to network failures. We prefer the first approach due to its simplicity, and better protection against data loss.

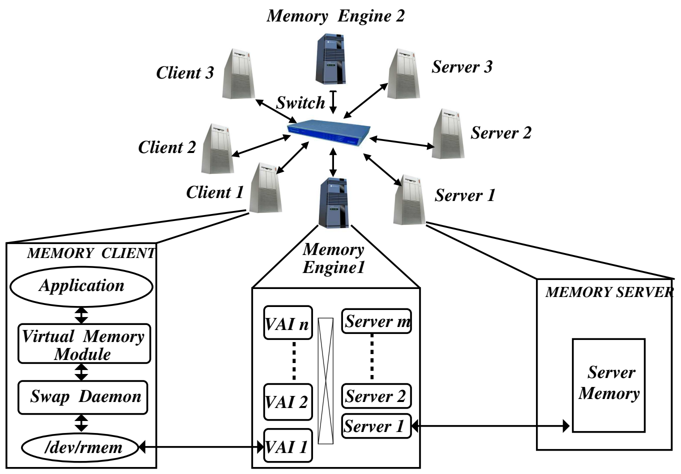

Figure 2: Architecture of Anemone: The Memory Engine multiplexes client paging requests among back-end memory servers through a Virtual Access Interface.

(H) Data Integrity: Ensuring data integrity becomes an important concern using remote nodes in the cluster to store applications' memory contents. Data integrity itself can be ensured by using checksums and/or encryption, although at the cost of additional client-side computation overhead. Loss of data can be handled by keeping a local copy of every page on the disk (as described above).

Summary and Focus of This Paper: Our description of the initial Anemone prototype in this paper focuses specifically upon design aspects (A)–(E) above. Having demonstrated the feasibility and usefulness of highly efficient remote memory access, our ongoing work on the system is to address the remaining aspects of distributed resource management, adaptation, fault-tolerance and data integrity.

3 Architecture of Anemone

Anemone contains three main components: *Memory Engine*, *Memory Servers* and *Memory Clients*.

3.1 The Memory Engine

The memory engine is a dedicated entity for global memory resource management that coordinates the interaction between the clients executing large-memory applications and servers hosting remote memory. The memory engine itself is not the primary source of remote memory; rather it helps to pool together and present a unified access interface for the remote memory resources across the gigabit network. The memory engine transparently maps the client memory requirements to available remote memory and hides the complexity of memory resource management from both memory clients and back-end servers.

The client-side face of the memory engine consists of a number of *Virtual Access Interfaces* (VAI), which are logical devices (not physical devices) that memory clients use to access remote memory. Memory access requests from the client are sent using RMAP to this remote VAI device in the form of read/write requests. The engine can simultaneously support VAIs for multiple clients and also support multiple memory servers to contribute unused memory. Each VAI can have attributes such as logical capacity, the set of memory servers storing VAI contents, and reliability requirements.

Upon a write, the Client Module sends the page to the engine which forwards the page to be stored on one of the back-end memory servers. Successive pages, including pages from the same client, can be can be sent by the memory engine to any of the memory servers based upon different types of server selection algorithms. Upon a read, the client sends a read request to the memory engine which then looks up its internal mapping tables to locate the server holding the requested page, retrieves the page from that server, and transmits it to the client. The memory engine also maintains a cache of frequently accessed pages that it searches first before contacting a memory server.

3.2 The Memory Servers

The memory servers store client's data pages that are forwarded by the memory engine. Servers may or may not be a general purpose machine in the cluster. Each memory server can choose to set aside a portion of its unused memory for use by remote clients. Depending upon each server's own memory requirements, the amount of contributed memory may change dynamically in coordination with the memory engine. The memory engine can balance the load among multiple memory servers based upon different utilization criteria such as ratio of number of pages stored at each server to its contributed capacity, the frequency of read requests being made to each server, or the CPU utilization at each server. The server with the lowest utilization value is selected to store a fresh write request. We are incorporating dynamic migration of pages between servers for load balancing.

3.3 Memory Client

To maintain transparency for large-memory applications, the memory client is designed as a *pseudo block device* (PBD) that appears to be a regular block device to the client system, while in reality it is a convenient front-end to transparently manage remote memory access. The PBD is configured as a primary swap partition to which the swap daemon can send read/write requests at runtime in response to page-faults from large memory applications. The virtual memory subsystem interacts with the swap daemon to page-out or page-in memory contents to/from the PBD. The fact that the primary swap device is in fact remote memory of one or more memory servers is transparent to both the application and the swap daemon. A major advantage of this approach is that the PBD is a self-contained pluggable kernel module that transparently sits above the network device driver and does not require any changes either to the memory-intensive application or to the client's operating system. The PBD communicates with its VAI on the memory engine using a window-based reliable memory access protocol (RMAP).

The PBD includes a small cache to store often requested pages locally and a pending queue of requests to be sent out to the VAI. The size of the cache and the queue together is bounded to about 16MB to avoid creating additional memory pressure. For write requests, if the page already resides in the cache or if there is space available, then the page is written to cache. If the number of pages in the cache exceeds a high water mark, victim pages are evicted based on LRU policy. For read requests, a page that already resides in the cache is returned directly to the swap daemon. Upon a cache miss, the read request is forwarded to the VAI. If the cache and the pending request queue are full, the swap daemon needs to wait because requests are being generated faster than can be transmitted reliably.

4 Anemone Prototype Implementation

4.1 Operating Systems and Hardware

Our prototype system involves 8 machines: one client, one engine and six memory servers, with more modes being added as we write this paper. Our initial Anemone prototype provides proofof-concept that virtualization of remote memory is indeed feasible and promising on a larger scale. The client, engine and servers components are implemented in Linux 2.6 using loadable kernel modules. All machines contain 2.6-3.0 Ghz Pentium Xeon processors. The client machine has 256 MB of memory, Server one has 2.0 GB, Server two has 3.0 GB, Servers 3 through 6 have 1.0 GB of memory, and the engine machine has 1.0 GB. All machines use Intel Pro/1000 MT network cards and are connected by an 8-port SMC Networks gigabit Ethernet switch supporting jumbo frames.

4.2 Memory Client Implementation

The memory client is implemented as a Linux kernel module, where it acts as a *pseudo block device* (PBD) that provides the illusion of a regular disk-type block device to the operating system. 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 which minimizes disk seek latency. Unlike disks, Anemone does not suffer from seek latency overhead and hence does not need to group sequential I/O requests together. As a result the client module bypasses the request queue and directly intercepts the BIOs before they are placed on the request queue. Hence, the Anemone client is able to skip the unnecessary overheads, dispatch the BIOs in their arrival order and allow their completion out of order.

4.3 Memory Engine Implementation

The memory engine uses a hash table to map clients' pages to servers. The hash key (a page's identity) is composed of the triplet: {Client IP address, Offset, VAI_id}. The engine can host many VAIs, each having one VAI_id. A VAI can only be used by at most one client, whereas each client can mount multiple VAIs. The VAI is not a real file, but simply an in-memory structure containing file-like meta-data. Write requests to VAI can be stored in any memory server that the engine sees fit. The offset contained within the triplet is a file offset accessed by the client's READ/WRITE requests.

Many different levels of the virtual memory process perform prefetching of data, (also called "read-ahead"), where an additional number of data blocks are fetched during an I/O request in anticipation that the next few blocks will also be read soon. The VFS (Virtual File System), and the swap daemon all do their own types of prefetching. As a result, requests to the engine tend to have lots of requests grouped together which are spatially local to each other. To handle this the engine creates a small request queue for each client. Since the engine can handle any number of servers, acknowledgment pages coming from different servers destined for a particular client may come out of order, so the queue is searched upon receipt of any server-ACK (which may contain a page, or simply an ACK to a page-out). This allows for a pipelined stream of page data, from multiple servers, to be returned to the client during a page-fault. This is the result of storing those pages across different servers in past requests.

4.4 Server Implementation

Just as the engine uses a hash-table, keyed by the identity of a page, the server does as well. The difference is that the data attached to a unique key in the engine is the location of a page, whereas the data attached to that same key in the server is the page itself. This is why the key is transmitted to the server through the messaging protocol on all requests. Additionally, the server does not maintain a request queue like the engine does. The engine and server speak to each other using a lightweight, stop-and-wait messaging protocol. A message consists of the type of operation (Page-IN or Page-OUT), the pre-constructed key that identifies the page and may or may not contain the page itself. Upon the server's connection to the engine, the server continuously blocks for page requests from the engine. By implementing a lightweight flow control at the client level, flow control between the engine and the server is also taken care of and the engine-server protocol only needs to handle retransmissions.

4.5 Kernel-Space Specifics

Anemone implementation sits directly on top of the network card - between the link layer and network layer, as shown in Figure 3. In Linux, a software-based interrupt mechanism, called the *softirq* (or bottom-half) handles the demultiplexing of incoming packet higher network-level protocols, such as IP processing. The Anemone module uses a Linux hook, called NetFilter, to pull the Anemone-specific packets after the softirq fires. From here, Anemone processes the request, schedules communication and returns control to the kernel all in one shot. The hash-tables used in the engine and server consume very little memory. The number of buckets in the hash-table remains static upon startup and are allocated using the get_free_pages() call. Linked-lists contained within each bucket hold 64-byte entry structures that are managed using the Linux slab allocator. The slab allocator performs fine-grained management of small, same-sized objects of memory by packing them into pages of memory side by side. The kmalloc() mechanism is not used as it is not necessary and causes fragmentation. All memory is allocated atomically and statically with the GFP_ATOMIC flag, a requirement of code that operates in interrupt mode.

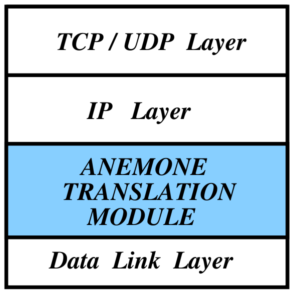

Figure 3: Placement of Anemone in the memory engine's network stack.

4.6 RMAP: Reliable Memory Access Protocol

The translation module, shown in Figure 2 needs to be fast so that page-fault latencies are kept small. Use of TCP for reliable communication turns out to be too expensive because it incurs an extra layer of processing in the networking stack. In addition Anemone does not need TCP's features of congestion control and in-order byte-stream delivery. Additionally, Anemone does not need network-layer IP routing functionality because it operates within a LAN. The need for packet fragmentation is eliminated because gigabit networks support the use of *Jumbo* frames which allow MTU (Maximum Transmission Unit) sizes larger than 1500 bytes (typically between 9KB and 16KB). Thus, an entire 4KB or 8KB memory page, including headers, can fit inside a single jumbo frame. Consequently, *Anemone uses its own reliable window-based datagram protocol for communication, but without the requirement of in-order delivery.* All the communication modules (in the clients, engine and the servers) operate right above the network device driver, bypassing all other protocol stack layers.

The RMAP implementation includes five network functions. REG{ister} and UNREG{ister} connect and disconnect a client with the memory engine. When the engine receives a REG request it allocates a VAI and returns a file handle for it. When the engine receives an UNREG request it deallocates the VAI and all meta-data associated with that client and frees remote pages belonging to the clients. READ, WRITE and their associated ACK messages provide basic reliability for reading and writing between the clients, engine and servers. If a READ/WRITE communication does not receive its associated acknowledgment after a given amount of time, the request is retransmitted. Finally a STAT function is made available to allow the client to gather information about its corresponding VPI(s) in the engine, such as available memory and average workload.

A lightweight flow control strategy has also been implemented in RMAP. Upon loading the client module, a fixed size FIFO transmission queue is allocated within the module itself. As requests are sent to the module, from the client's swap daemon (or direct I/O), the transmission queue is allowed to fill. To empty the transmission queue the client module uses a window to limit the number of outstanding network memory requests. When the queue is full, the client stalls any upper layer paging activity until ACKs come in and shrink the queue. These ACKs are allowed to arrive out-of-order. The RMAP implementation also provides several other tunable parameters. The first parameter is the *window size* which controls the number of packets waiting for acknowledgment in the transmission queue. It can be tuned so as not to overload either the network interface card or receiver. Increasing the window size has a tendency to increase the number of page retransmissions between the client and engine. The current experiments have found the ideal window size to lie somewhere between 10 and 12 page-requests. The second parameter is a *retransmission timer* which triggers the RMAP to check if there are any packets awaiting retransmission. Setting this timer to a very low value can degrade the performance as the system will get bogged down frequently checking queued packets. However setting the parameter too high increases the retransmission latency. The third parameter is the *per packet retransmission latency*, i.e. the amount of time between successive retries of the same request. Finally, the fourth parameter specifies the *latency wait between successive packet transmissions*. Again, depending on the workload, this can place a lower bound on the paging-rate of the client, but can perform effective flow control.

5 Performance

In this section we evaluate the Anemone prototype implementation. We focus on answering the following key questions addressing the performance of Anemone: (1) What reduction in paging latency does Anemone deliver when compared to disk-based paging? How do the latency characteristics vary across sequential/random reads and writes? (2) What application speedups can be obtained with Anemone for real-world unmodified applications? (3) How does the number of concurrently executing processes impact application speedups with Anemone? (4) How much processing overhead does Anemone introduce? (5) How does the communication protocol effect the performance of remote memory access? (6) How do the application memory access patterns impact the remote paging performance?

Our results can be summarized as follows. Anemone reduces all read latencies to around $500\mu s$ compared disk read latencies up to $9.6ms$. For writes, both disk and Anemone deliver similar latencies due to caching affect. Anemone delivers up to a factor of 3 speedup for single process real-world applications, and delivers a up to a factor of 7.7 speedups for multiple concurrent applications. We show that the processing overhead of Anemone is negligible compared to round trip communication latency and that RMAP's window size parameter plays a significant role of Anemone's performance.

5.1 Anemone Testbed

Our experimental testbed consists of one low memory client machine containing 256 MB of main memory, six memory servers consists of four 1 GB machines, one 2 GB machine and one 3 GB machine. Of the 8 GB of remote memory available to use, the server machines themselves consume a significant part for kernel memory, processes and drivers. Including the client and engine, the eight machines utilize about 1.2 GB in total, leaving 7.8 GB of unused memory. For disk based tests, we used a Western Digital WD400BB disk with 40GB capacity and 7200 RPM speed. There is another additional source of memory overhead in Linux which reduces the effective collective memory pool available for paging, but does not adversely affect the performance speedups obtained from Anemone. Specifically, a header is attached to each 4KB page received and stored by the servers. Memory allocator of Linux assigns a complete 4KB page to this extra header in addition to the 4KB page of data, thus reducing the effective remote memory available to 3.6 GB. Solutions to this problem include either introducing an additional memory copy while receiving each packet at the server or using the "scatter" operations when receiving pages over the network to separate the header from the page data. The header space could instead be allocated from Linux slab allocator.

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

Table 1: Page-fault service times at intermediate stages of Anemone. Values are in microseconds.

5.2 Components of Page-in Latency

A page-in operation is more time-critical than page-out operation because application cannot proceed without the page. Here we examine the various components of the latency involved in processing a page-in request using Anemone. A request for a 4KB page involves the following steps: (1) A request is transmitted from the client to the engine (2) The engine requests the page from a server (3) The server returns the requested page to the engine (4) The engine returns the page back to the client. The entire sequence requires 4 network transmissions. Table 1 shows the average amount of time consumed at each of these 4 stages over 1000 different read requests to the Anemone system providing 2 GB of remote memory. It turns out that the actual computation overhead at the client, the engine and the server is negligibly small: less than 10 microseconds. The majority of the latency involved is spent in the client-engine and engine-server communication .

The client latency includes making a read/write request either via the swap daemon or via a system call, invoking the client module and sending off the request to the engine. 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 computation times are also small and clearly indicate the bottleneck the network creates.

The time spent in completing a ping on our network between two machines takes an average of 150 microseconds – a good lower-bound on how fast Anemone can get using our RMAP protocol.

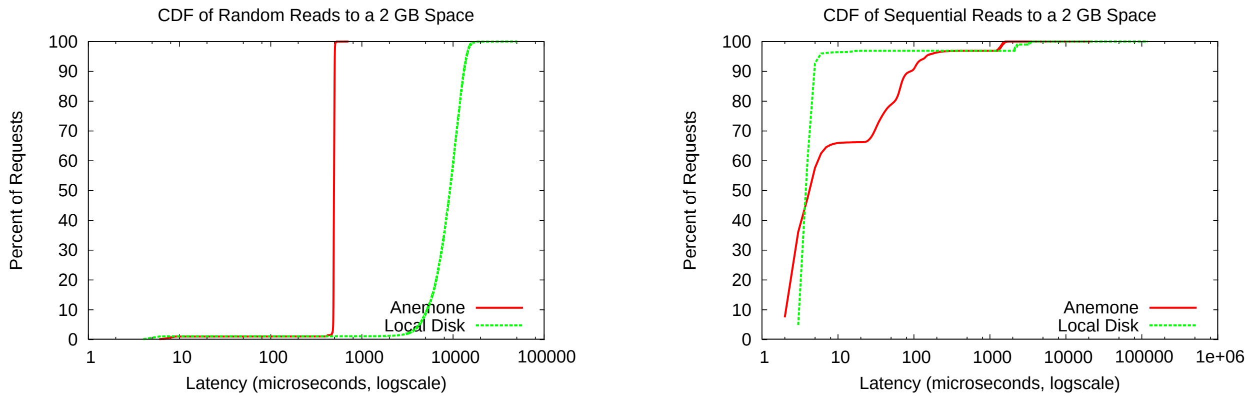

Figure 4: Comparison of latency distributions for random and sequential reads for Anemone and disk.

Notice how the communication latency has been dramatically reduced and that the total round trip latency is a consistently average 500 microseconds. As expected these two numbers were constant, around 250 microseconds between each hop (round trip). There is very low overhead, but there is still about 100 microseconds of room to improve on anemone at each round trip hop, according to a baseline ping. Furthermore, the time to place 4k bits on a gigabit link is not much more than 4 microseconds, which means that a great deal of the transmission time is spent in sending transmission/receiving interrupts, allocating ring buffers and sending the data over the bus to the NIC. Optimizing these pieces are not easy to do without significant modifications. Conclusively, the kernel-based version of Anemone is 19.6 times faster than disk-based block requests that require seek/rotation.

5.3 Latency Distribution

In order to compare read/write latencies obtained with Anemone against those obtained with disk, we next plot the distribution of observed read and write latencies for sequential and random access patterns. Figure 4 compares the cumulative distributions of latencies with disk and Anemone for random and sequential read requests. Similarly, Figure 5 compare the two for random and sequential write requests. 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. For random read requests in Figure 4, most requests

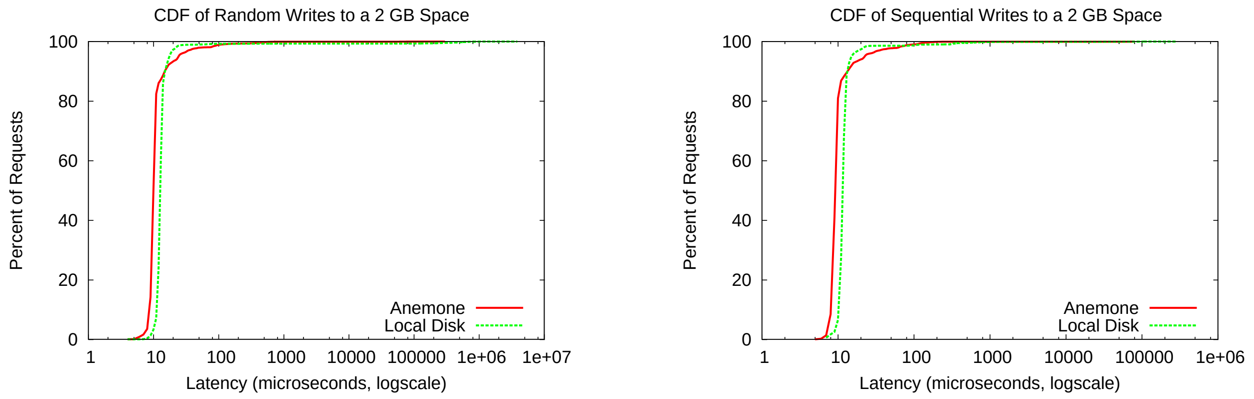

Figure 5: Comparison of latency distributions for random and sequential writes for Anemone and disk.

to disk experience a latency between 5 to 10 milliseconds. On the other hand most requests in Anemone experience only around 400 microsecond latency. In our most recent experiments with distributed Anemone architecture, we have measured even lower latencies of 210 microseconds due to absence of the memory engine. For sequential read requests in Figure 4, disk shows a slightly superior latency distribution than Anemone. Most sequential requests are serviced by disk within 3 to 5 microseconds because sequential read accesses fit well with the motion of disk head, eliminating seek and rotational overheads. In contrast, Anemone still delivers a range of latency values (most of them still less than 400 microseconds), mainly because network communication latency dominates, though it is masked to some extent by the prefetching performed by swap daemon or file-system. The write latency distributions in for both disk and Anemone in Figure 5 are comparable and most latencies are close to 10 microseconds because writes typically return after writing to the buffer cache.

5.4 Single Large Memory Processes

This section evaluates the performance improvements seen by two unmodified single process large memory applications using the Anemone system. The first application is a **graphics rendering program called POV-Ray** [20]. The POV-Ray application was used to render 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 6 shows the completion times of these

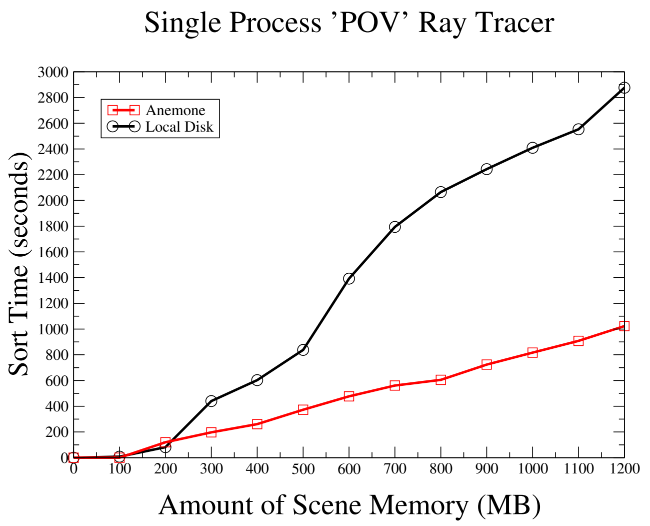

Figure 6: Comparison of execution times of POV-ray for increasing problem sizes.

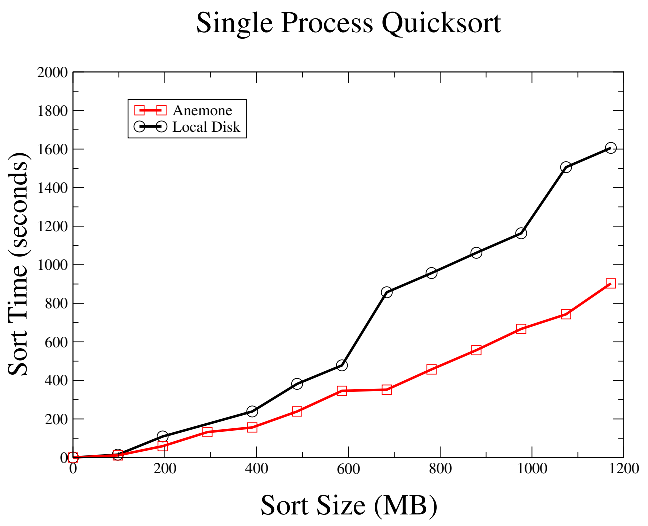

Figure 7: Comparison of execution times of STL Quick-sort for increasing problem sizes.

increasingly large renderings up to 1.2 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 application by a factor of up to 2.9 for 1.2 GB memory usage. The second application is a **large in-memory Quick-sort program** that uses an STL-based implementation from SGI [21], with a complexity of O($N \log N$) comparisons. We sorted randomly populated large in-memory arrays of integers. Figure 7 clearly shows that Anemone again delivers application speedups by a factor of up to 1.8 for single-process quick-sort application having 1.2 GB memory usage.

5.5 Multiple Concurrent Processes

In this section, we test the performance of Anemone under varying levels of concurrent application execution. Multiple concurrently executing memory-intensive processes tend to stress the system by competing for computation, memory and I/O resources and by disrupting any sequentiality in the paging activity. Figures 8 and 9 show the execution time comparison of Anemone and disk as the number of POV-ray and Quick-sort 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 memory usage also increases. The figures show that the execution times using disk-based

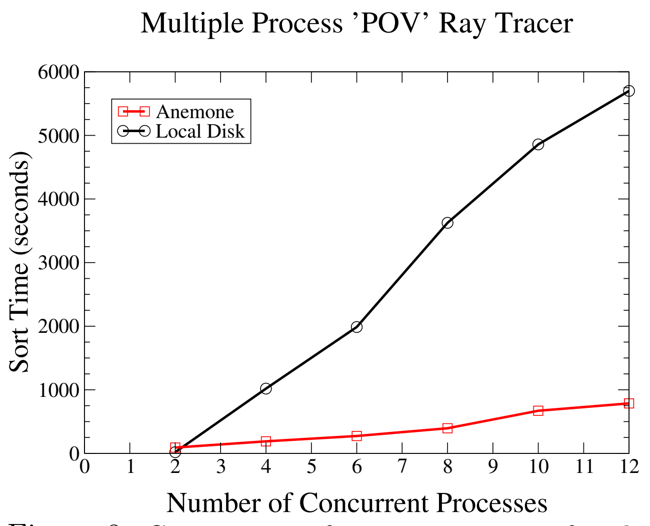

Figure 8: Comparison of execution times of multiple concurrent processes executing POV-ray for increasing number of processes.

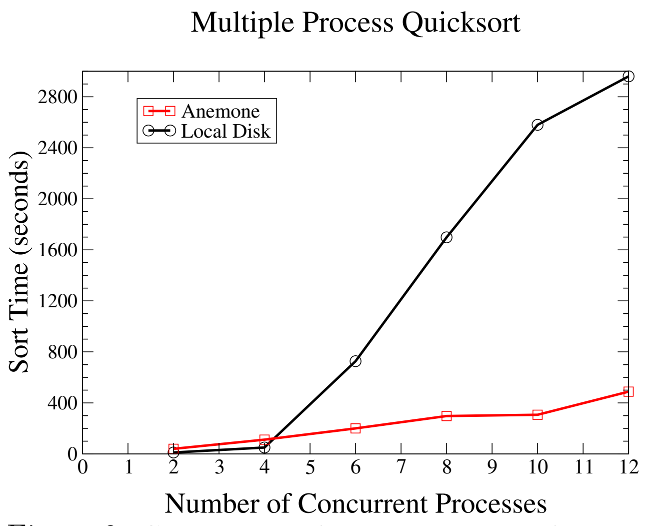

Figure 9: Comparison of execution times of multiple concurrent processes executing STL Quick-sort for increasing number of processes.

swap increases steeply with increasing number of concurrent processes. This is because the paging activity gradually loses sequential access property as the number of process increases. This makes the disk seek and rotational overheads a dominant factor in disk I/O latency. On the other hand, Anemone reacts very well to concurrent system activity and the total execution time increases very slowly. This is because, unlike disk based paging, Anemone 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 for POV-ray and a factor of 6.0 for Quick-sort.

5.6 Application Paging Patterns

In order to better understand application memory access behavior, we recorded the traces of paging activity of the swap daemon for one execution each of POV-ray and Quick-sort. Figure 10 plots a highly instructive plot of the paging activity count versus the byte offset accessed in the pseudo block device as the application execution progresses. The graphs plot three types of paging events: write or page-out (green data points), read or page-in (blue data points) hits in the cache, and read misses (red data points). First thing to notice is that read hits constitute only about 13% of all reads to the pseudo block device for POV-ray and about 10% of all reads for Quick-sort. This implies

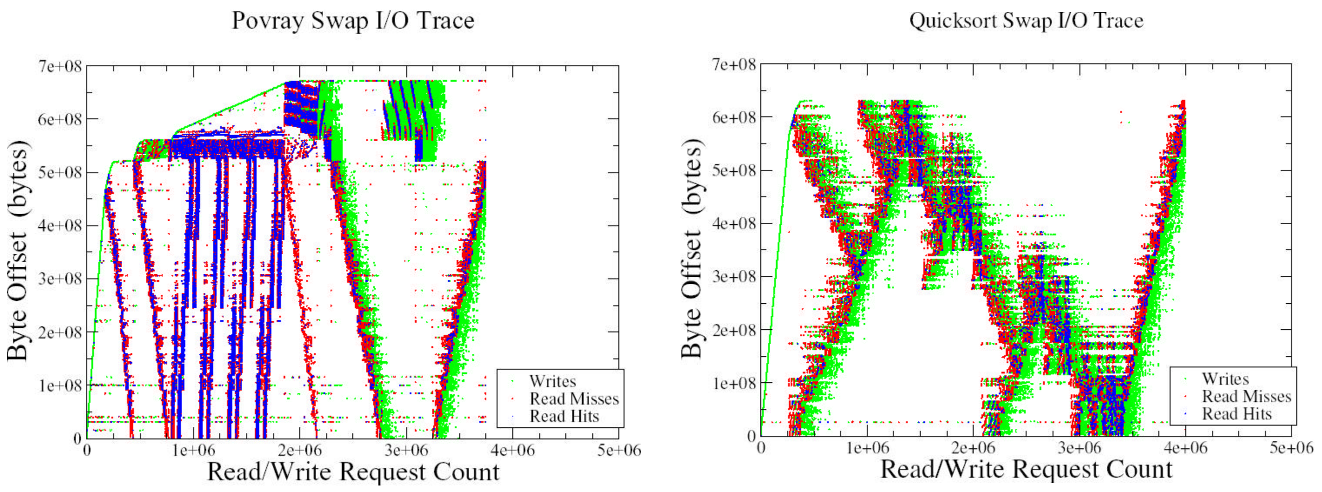

Figure 10: Trace of I/O activity for POV-ray and Quick-sort with progress of application execution. The green data points represent writes, red data points represent cache misses upon reads in the cache of pseudo block device and blue points represent cache hits upon reads.

that the LRU caching policy of the pseudo block device is not effective in preventing majority of read requests from being fetched over the network. Secondly, both traces show signs of regular page access patterns. The level of sequential access pattern seems significantly higher for POV-ray than it is for Quick-sort. However, upon close up examination, we found that there was significant reverse memory traversal patterns in POV-ray, even though it was sequential. On the other hand, Quick-sort has a tendency to access non-local regions of memory due to the manner in which it chooses pivots during the sorting process. This seems to indicate that both applications tend to defeat any read-ahead prefetching performed by the swap-daemon. This seems to indicate the need for more intelligent caching and prefetching strategy, beyond the conventional LRU scheme, for both the swap daemon and the pseudo block device. This is one of the focus of our ongoing research.

5.7 Tuning the Client RMAP Protocol

Handling flow control in any new situation often has challenges and, as described earlier, 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. Using a 1 GB Quick-sort as a benchmark, Figure 11 shows the effect of changing this window size on three characteristics of the application performance using Anemone. These characteristics include (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 (3) Application completion time. All the three characteristics are graphed together on a logarithmic scale. As the window size increases, the number of retransmissions increases because the number of packets that can potentially be delivered back-to-back also increases. For the range of window sizes shown in the figure, 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 number of retransmissions. However, for much larger window sizes (not shown in figure), the paging bandwidth is seen 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. For larger values of window sizes beyond the range shown in figure, we observe more packet drops, more retransmissions, lower paging bandwidth and higher completion times.

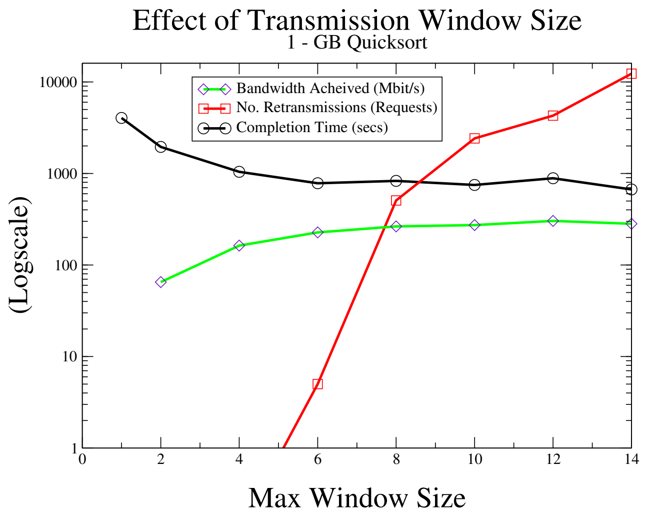

Figure 11: Effects of varying the transmission window using quick-sort on bandwidth, retransmissions and execution time.

6 Related Work

To the best of our knowledge, Anemone is the first system that provides unmodified large memory applications with a completely transparent and virtualized access to cluster-wide remote memory over commodity gigabit Ethernet LANs. The earliest efforts at harvesting the idle remote memory resources aimed to improve memory management, recovery, concurrency control and read/write performance for in-memory database and transaction processing systems [14, 13, 4, 15]. The first two remote paging mechanisms [6, 11] incorporated extensive OS changes to both the client and the memory servers and operated upon 10Mbps Ethernet. The Global Memory System (GMS) [10] 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 [16, 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. Legacy applications must be modified to use this library, leading to a lack of application transparency.

Work in [18] 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 [19] specific to the Nemesis [17] operating system was designed to permit application-specific remote memory access and paging. The Network RamDisk [12] 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]. DSM systems allow a set of independent nodes to behave as a large shared memory multi-processor, often requiring customized programming to share common objects across nodes. This is much different from the Anemone system which allows pre-compiled high-performance applications to execute unmodified and use large amounts of remote memory provided by the cluster. Samson [22] is a dedicated memory server over Myrinet interconnect 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 also extensively modified.

Simulation studies for a load sharing scheme that combines job migrations with the use of network RAM are presented in [24]. The NOW project [2] performs cooperative caching via a global file cache [8] in the xFS file system [3] while [23] attempts to avoid inclusiveness within the cache hierarchy. Remote memory based caching and replacement/replication strategies have been proposed in [5, 7], but these do not address remote memory paging in particular.

7 Conclusions

We presented the design, implementation and evaluation of the *Adaptive Network Memory Engine* (Anemone) system that enables unmodified large memory applications to transparently access the collective unused memory resources of nodes across a gigabit Ethernet LAN. We implemented a working Anemone prototype and conducted performance evaluations using two unmodified real-world applications, namely, ray-tracing and large in-memory sorting. When compared to disk-based paging, Anemone speeds up single process applications by a factor of 2 to 3 and multiple concurrent processes by a factor of 6 to 7.7. There are several exciting avenues for further research in Anemone. We are improving the scalability and fault tolerance of Anemone using a peer-to-peer memory sharing architecture which does not include a memory engine. We are also improving the reliability of Anemone using page replication for higher availability and load distribution. Another promising direction is to use compression of pages to reduce communication and increase the effective storage capacity.

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] P. Bohannon, R. Rastogi, A. Silberschatz, and S. Sudarshan. The architecture of the dali main memory storage manager. *Bell Labs Technical Journal*, 2(1):36–47, 1997.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] H. Garcia-Molina, R. Abbott, C. Clifton, C. Staelin, and K. Salem. Data management with massive memory: a summary. *Parallel Database Systems. PRISMA Workshop*, pages 63–70, 1991.

[14] H. Garcia-Molina, R. Lipton, and J. Valdes. A massive memory machine. *IEEE Transactions on Computers*, C-33 (5):391–399, 1984.

[15] S. Ioannidis, E.P. Markatos, and J. Sevaslidou. On using network memory to improve the performance of transaction-based systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, 1998.

[16] 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.

[17] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

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

[19] 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.

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

[21] Inc. Silicon Graphics. *STL Quicksort.*

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

[23] T.M. Wong and J. Wilkes. My cache or yours? Making storage more exclusive. In *Proc. of the USENIX Annual Technical Conference*, pages 161–175, 2002.

[24] 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.