Implementation Experiences in Transparently Harnessing Cluster-Wide Memory

Michael R. Hines, Mark Lewandowski 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. As commodity gigabit Ethernet LANs with support for jumbo frames become increasingly popular, it is time to re-examine an interesting solution that has not received the attention it deserves. Specifically, we investigate the benefits and tradeoffs in pooling together the collective memory resources of nodes across a high-speed LAN. 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 LAN, without requiring any modifications to the large memory applications. We 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 show that, when compared to disk based paging, unmodified single-process applications execute 2 to 3 times faster and multiple concurrent processes execute up to 7.7 times faster when accessing remote memory via Anemone. 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 technology have caused an unprecedented increase in memory capacity of standard off-the-shelf systems. At the same time, the growth in memory requirement of applications, such as multimedia and graphics processing, high resolution volumetric rendering, weather prediction, large-scale simulations, and large databases, continues to remain ahead of this growth in memory capacity. 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. 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. In the paging process, the system treats the much slower local disk as an extension to the DRAM by writing and reading excess contents of applications' virtual memory and application performance remains throttled by large disk access latencies.

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. Typical resource utilization levels in most commodity clusters remains 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 [28].*

An interesting question naturally follows from the above discussion: *Can we transparently pool together (or virtualize) 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?* While a number of prior research efforts [7, 13, 12, 19, 21, 14, 22, 26] have attempted to address this question, the interest level in exploiting memory resources over the network has not quite caught the momentum it deserves. This is primarily because most of the earlier attempts have either relied upon expensive fast-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 application itself or the end-host operating system (or even both). While some commercial vendors build servers with specialized hardware to hold large amounts of DRAM, these solutions are prohibitively expensive and the technology itself quickly becomes obsolete. Note that the above research question of transparent remote memory access is different from (though sharing many of the concerns with) the research on Distributed Shared Memory (DSM) [10] systems that permit nodes in a network to behave as if they were shared memory multiprocessors, often requiring use of customized application programming interfaces.

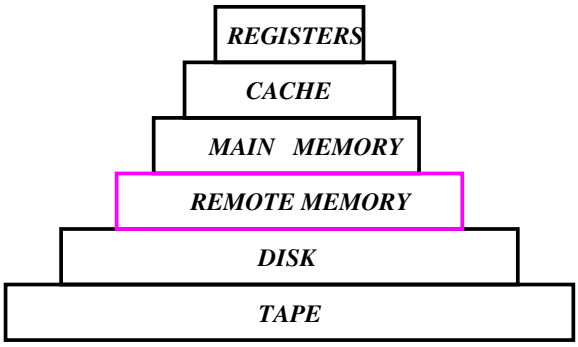


Figure 1: The position of remote memory in the memory hierarchy.

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. In this paper we re-examine the feasibility of transparent remote memory access using modern commodity gigabit Ethernet. More specifically, our goal is to *virtualize* the collective unused memory resources in a cluster to support memory-hungry applications without the need to modify the applications. *Virtualization* refers to the ability to subdivide a pool of physical resources, such as computation, memory, storage, or bandwidth, into higher level logical partitions. Just as the operating system seamlessly virtualizes various hardware resources within a single machine for executing processes, our goal is to virtualize the collective memory resources in a cluster in a transparent manner.

We present our experiences in the design, implementation, and evaluation of the ***Adaptive Network Memory Engine (Anemone)*** system. Anemone virtualizes the collective pool of memory resources across nodes in a gigabit Ethernet LAN and provides memory hungry applications with transparent and low-overhead access to potentially "unlimited" remote memory resources. The Anemone system makes the following specific contributions.

- To the best of our knowledge, Anemone system is the first attempt at evaluating the feasibility of remote memory access over commodity gigabit Ethernet hardware that exploits jumbo frame support.
- Anemone presents a unified *virtual interface* for each memory client to access an aggregated pool of potentially unlimited remote memory.
- Any application can benefit from Anemone without requiring any code changes, recompilation, or relinking thus eliminating painstaking and extensive end-system changes. 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.

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 – from an average of 9.8ms with disk based paging without caching to about $500\mu s$ with Anemone. Anemone speeds up single-process large-memory applications by a factor of 2 to 3, and and multiple concurrent large-memory applications by a factor of 2 to 6. Our implementation experience indicates that important factors in this performance improvement must include effective flow-control between the swap daemon and the Anemone client module, predictive page caching, and the design of a low-level lightweight reliable communication protocol. Our performance numbers for multiple-process benchmarks have slopes that indicate a trend towards an order of magnitude improvement when increasing the problem size.

The rest of this paper is organized as follows. Section 2 presents the architectural design of the Anemone system the rationale behind various design choices. Section 4 presents details of Anemone implementation. Section 5 provides performance evaluation of a working Anemone prototype. Section 6 compares Anemone to earlier efforts on remote memory access. Section 8 summarizes our contributions and outlines future directions.

2 Design Choices in Anemone

This section examines the complete design space of distributed remote memory access that we strive to encompass in the Anemone system. We examine the various design alternatives, tradeoffs, and practical rationale that have influenced our system. We finally summarize the subset of issues we specifically address in this paper on the first Anemone prototype.

(A) Transparent Virtualization: The first design considerations in Anemone are 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 achieve this goal is by modifying the virtual memory subsystem of the operating system kernel. When the system runs low on local memory, page-table entries of a process can first be mapped to remote memory pages, and when the remote memory runs low, the page-table mappings can be directed to local disk. Although transparent to the application itself, this approach requires modifications to the core operating system. The second approach is to virtualize the remote memory resources as one or more *pseudo block devices*. Each pseudo block device can then be used by the swap daemon as a primary swap partition, in which the pages are swapped to remote memory instead of a local disk. This approach is simpler to realize as a self-contained kernel module that does not require changing the core operating system, and permits the swap daemon to handle all paging related decisions to both remote memory, and the local disk. For these reasons, we adopt the second approach to remote memory access. In addition, the pseudo block device's interface can also be treated as a low-latency alternative to disk that can host temporary files for quick access or can be memory-mapped by applications that are aware of the remote-memory. Though Anemone supports these additional uses in its current form, this paper mainly focuses upon its use for swapping to 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. While one could technically implement RMAP on top of TCP, UDP, or even the IP protocol layers, this choice comes burdened with unwanted protocol processing. A lower-overhead approach is to implement a light-weight communication protocol that executes

directly on top of the network device driver, without any intervening protocol layers. The second design consideration relates to memory page size and the maximum packet size, also called the maximum transmission unit (MTU), of the LAN. While the normal page size in most operating systems is 4KB (and sometimes 8KB), the MTU in traditional Ethernet networks is limited to 1500 bytes. This implies that a 4KB memory page would have to be fragmented into three packets at the sender and re-assembled at the receiver, incurring substantial overheads. An alternative is to exploit the feature of *jumbo frames* in modern Gigabit LANs, in which packet sizes can be greater than 1500 bytes (typically between 8KB to 16KB). Jumbo frames enable the RMAP to transmit complete 4KB pages to remote memory using a single packet and without fragmentation costs. The third design consideration relates to reliability of RMAP. It is common for hardware components such as network cards and switches to drop packets under heavy loads. Thus RMAP needs to be reliable because dropping applications' memory pages while swapping over the network would lead to unacceptable disruption of application execution. 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 traditional features found in TCP 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: As in any distributed system, Anemone faces the design choice of whether to manage the allocation of the global memory pool in a centralized entity or in a distributed manner across all participant nodes. The centralized model is simpler to implement and has the advantage that a single entity has a global view of memory resources in the LAN enabling more efficient resource management. The obvious disadvantage is that the central entity can become a bottleneck and a single point of failure. Even in normal operations, the single entity introduces 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 resources. However, the extra coordination required may not be prohibitively expensive. We believe a distributed model is a better approach of the two when considering long-term scalability, performance and security. The immediate practical consideration in Anemone system was to quickly investigate whether virtualized access to remote memory was feasible using Gigabit Ethernet technology. Hence, for simplicity and speed of prototyping, an Anemone prototype was implemented to use a centralized Memory Engine that mediates remote memory requests between clients and servers. Additionally, have built in the flexibility to easily migrate to a distributed implementation 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 the kind of resource waste that would arise from dedicating a single memory server for every client.

(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 was 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 faces the following design requirement: (a) It should seamlessly absorb the increase/decrease in cluster-wide memory capacity, insulating the memory clients from resource fluctuations, (b) It should 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 the size of it's aggregated memory space dynamically and use disk as a backup resource in case the remote memory pool becomes over committed.

(G) Fault-tolerance: On one hand, the ultimate consequence of any failure in swapping to remote memory is no worse than failure in swapping to local disk, namely, the application execution is compromised. At the same time, the probability of failure is greater in a LAN environment because of multiple components involved in the process, such as network cards, connectors, cables, switches and the remote memory server itself. Hence the design requirements for reliability are greater in Anemone than swapping to local disk. There are two feasible alternatives for fault-tolerance, both of which involve storing redundant data. The simplest approach is 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. The second approach is to keep redundant copies of each page on multiple remote servers. This approach avoids disk activity and reduces recovery-time from server failures, but consumes precious network bandwidth, reduces effective global memory pool, and remains susceptible to network failures. In both alternatives, maintaining consistency among multiple copies is important. We prefer the first approach in Anemone due to its simplicity, better security against malicious/inadvertent data loss, and due to less frequent occurrence of disk failures compared to network failures.

(H) Security: Ensuring security and data integrity becomes an important concern when relying upon third party nodes in the LAN to store applications' memory contents. Data integrity itself can be ensured by using encryption and data checksums, although at the cost of additional client-side computation overhead. Loss of data due to maliciously compromised servers 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 security.

3 Architecture of Anemone

This section describes the detailed architecture of the Anemone system and how it incorporates the design choices discussed in Section 2. There are three principal components of the Anemone system: *Memory Engine*, *Memory Servers*, and *Memory Clients*. Figure 2 illustrates the components and their interaction.

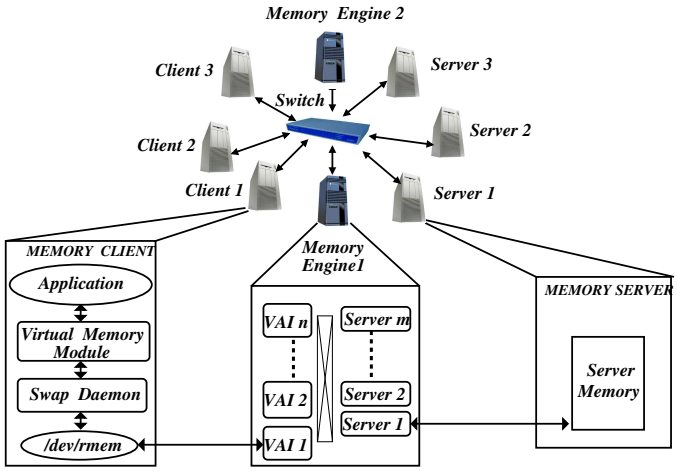


Figure 2: Architecture of Anemone: The Memory Engine multiplexes client paging requests among back-end memory servers through a Virtual Access Interface. Multiple engines may be used for scalability and reliability.

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 centralized architecture was chosen as a proof of feasibility and is a precursor to a fully distributed architecture where functions of the Memory Engine will be subsumed by individual clients and servers. 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 Memory Engine uses a lightweight window-based Reliable Memory Access Protocol (RMAP) to communicate with both the Memory Clients and Memory 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 can be used by memory clients to access their remote memory contents. Memory access requests from the client are sent to this remote VAI device in the form of read/write requests. The engine is capable of simultaneously supporting VAIs for multiple clients in the cluster. At the same time it also allows multiple Memory Servers to contribute unused memory space for use by clients. Each VAI can have attributes such as logical capacity, the set of Memory Servers storing VAI contents, reliability requirements. and priority relative to other VAIs. The Memory Engine stores these attributes in memory and returns a VAI handle that the client can use to interact with the VAI.

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 stored on any of the Memory Servers. This provides Memory Engine with the flexibility to implement 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. In addition to the page mapping information for each VAI, the Memory Engine maintains a cache in which it stores frequently requested pages. Upon a read request from client on any of the VAIs, the Memory Engine first searches its cache for the requested page and asks the Memory Server only upon a cache miss. There are no disks accesses involved in the entire process and all operations occur completely in memory.

The Memory Engine thus resides at the center of operations in Anemone, hiding all the complexity of managing

distributed memory resources from the clients, and the complexity of managing client requirements from the Memory Servers. The VAI seen by the clients aids in transparently virtualizing the collective memory resources. Clients simply interact with their VAIs while the Memory Engine hides and efficiently manages the underlying complexity of virtualization.

3.2 The Memory Servers

The memory servers store client's data pages that are forwarded by the memory engine. It 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. Upon initialization, the memory server provides the Memory Engine with information about the contributed memory amount. The amount of memory multiplexed among all the VAIs at any time does not exceed the cumulative memory contributed by the all the servers. 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 request. We are currently incorporating dynamic adaptation such that, when utilization levels of servers change, the Memory Engine can dynamically migrate pages from one server to another in order to balance their load.

3.3 Memory Client

The primary consideration in supporting remote memory access on client side is maintaining transparency to large-memory applications. Towards this end, the memory client component of Anemone is designed to be 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 PDB is be 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. As far as the memory-intensive application is concerned, its need for additional memory is transparently satisfied by the underlying virtual memory subsystem on the client. The virtual memory subsystem in turn interacts 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 only operations required at the client side involves loading the kernel module to create a PBD and configuring it as the primary swap partition. On the network side, the PBD communicates with its VAI on the memory engine using a window-based reliable memory access protocol (RMAP). The PBD also includes two more components: a small cache to store often requested pages locally and a pending request queue that holds requests waiting to be sent out to the VAI. Both the cache and the pending request queue together have a small upper bound on their size (about 16MB) to avoid creating additional memory pressure on an already stretched client memory system.

Read and write requests from the swap daemon to the PBD are handled as follows. For write requests, if the page already resides in the cache or if there is space available in the cache, then the page is written to cache and

control returned to the swap daemon. If the number of pages in the cache exceeds a high water mark as a result of a write, victim pages are selected based on least recently used policy and scheduled on the pending requests queue to be sent to the VAI. For read requests, a page that already resides in the cache is retrieved and returned directly to the swap daemon. Upon a cache miss, the read request to the VAI is placed in the pending request queue. If both the cache and the pending request queue are full, this indicates back-pressure from the network interface, i.e. swap daemon is generating requests faster than can be sent out over the network reliably. In such a situation, the swap daemon needs to wait for free space to become available.

4 Anemone Prototype Implementation

The client, engine and server are implemented by way of Linux loadable kernel modules. First, we discuss the technology used in our system, present the fundamental parts of the implementation and then discuss some obstacles their solutions that make Anemone work well.

4.1 Operating Systems and Hardware

Our prototype system involves 8 machines: one client, one engine, and six memory servers. The client, engine and servers must all run Linux 2.4 or 2.6.

The client, servers, and engine all run 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 three through six have 1.0 GB of memory, and the engine machine has 1.0 GB. As stated earlier, the system runs on a gigabit network. Each of the three machines use Intel Pro/1000 MT Desktop Adapter cards. The switch, to which all 8 machines are connected, is an 8-port SMC Networks gigabit switch supporting Jumbo Frames. The machines and network card settings have been precisely tuned to operate at gigabit speeds and we have observed a peak UDP-packet stream between any two machines of 976 Mbits/second, disabling interrupt coalescing and using jumbo frames. Interrupt coalescing is a network card feature that allows the card to deliver interrupts for packets to the kernel only after a specified interval of time instead of immediately upon arrival of the packet. Large amounts of interrupts, which are prevalent at gigabit speeds, can severely hinder the performance of the CPU. However, even with interrupt coalescing disabled, we easily achieve speeds just below 900 Mbits/second for a PCI card, and around 944 Mbits/second for on-board cards. If we were to use interrupt coalescing, the delay imposed on a page-fault would be lower-bounded by the interval of time specified to the network card between interrupts. This is not desirable because Anemone's first priority is to optimize page-fault latency before bandwidth. Thus, Anemone currently operates without this hardware feature.

4.2 Memory Client Implementation

The memory client is designed to provide a transparent interface between the client machine's swap daemon and the Anemone memory cluster. It is loaded into the Linux kernel as a standard module, where it acts as a *pseudo block device* (PDB), in effect fooling the operating system into believing that it is a regular disk-type device.

When the module is loaded it registers with the kernel's default block device interface. Standard block devices interact with the kernel through a request queue. The request queue provides the means for the kernel to arrange I/O requests in byte offset order, thus arranging reads and writes with similar offsets to be grouped together into one "request." The request is then placed on the request queue using an elevator algorithm. As is mentioned in [24], kernel developers have taken excruciating pains to optimize the request queue to exploit any sequential properties

in a set of block I/Os (BIO) in an effort to speed up disk response times. Unlike disks, Anemone is not reliant on sequential properties to maintain maximum I/O transfer rates. Since it is built on solid state memory, access times remain constant for any I/O given to the block interface.

As a result the client module bypasses the request queue and instead deals with individual BIOs. The kernel gives block device driver authors the ability to register their own function with the block interface to replace the default "make_request" function. By intercepting BIOs before they are placed on the request queue, the Anemone client is able to skip the unnecessary processing that goes with placing the I/O on the queue.

Dealing with BIOs directly comes with some other advantages as well. It allows the client to handle BIOs in the order they arrive, and with some clever queuing of incomplete BIOs allows them to complete out of order. Compare this to the request queue method where completing an I/O out of order involved the use of a secondary queue to store incomplete requests, and some method to look inside them for individual BIOs. The client modules also includes several tunable parameters that can drastically affect the performance of the client. In the current implementation the user is allowed to specify the upper bound on the transmission queue. The goal of the transmission queue is to receive BIOs from the block interface as quickly as possible. It is best to set this queue size to a value that is large enough where the swap daemon will not be able to keep it full, but caution needs to be taken to keep its memory footprint at a minimum, usually a value around two hundred has shown to be a acceptable. The size of this queue detracts from memory that is available to applications as they run, so it is important not to make it too large.

Further discussion on the communication between the client and engine, and the protocol's tunable parameters are detailed in section 4.6.

4.3 Memory Engine Implementation

The memory engine's most basic function is to map client pages to the server that the page is stored on. We use a hash-table to do this. The hash-table is keyed by the identity of a particular client page and the data attached to that key indicates the particular server that holds that page. A key (a page's identity) is composed of the triplet: {Client IP address, Offset, VAI id}. The VAI id within the triplet corresponds to a particular VAI or Virtual Access Interface, the engine's name for an exported directory made available for mounting by a client using the RMAP. Inside this VAI appears a simulated file from the client's point of view. Recall that the engine may offer many such VAIs. A VAI may only be used by at most one client, whereas a particular client may mount multiple VAIs. The VAI is not a real file, but simply an in-memory structure containing file-like meta-data. Requests to this VAI are scheduled onto any particular memory server that the engine sees fit. Thus, to identify the page, the VAI that the page came in on must be identified, in addition to the source address of the client.

The offset contained within the triplet is a file offset. When the client activates and issues RMAP READ and WRITE requests to the virtualized file contained within the VAI, those requests include the offset within that file that is being addressed. As the swap daemon fulfills page-faults or kicks out dirty virtual memory pages to the VAI, the pages get stored or retrieved from different offsets within the file. To complete the identification of a page, that offset is added to the triplet.

Finally, when indexing into the hash-table, a hash-function passes over each byte within this triplet, creating a hash-value that is unique enough to maximize usage of all the buckets within the hash-table and prevent collisions. At the moment, our system prototype only uses one client for performance testing, but this basic framework will allow us to easily implement scheduling algorithms as well as tailored hash-functions that minimize insertion/lookup times

for hash-table and bucket usage while decreasing collisions. Thus, we only use a simple modulus based hash-function. Pages that get swapped out for the first time are entered into the hash-table. Future page-faults to the same page cause the engine to re-construct a key, look into the hash-table, get the page's location, contact the server, retrieve the page, and complete the RMAP/memory request.

The Anemone engine implements a small set of network requests, including READ, WRITE, and STAT.

Many different levels of the virtual memory process do pre-fetching of data, (also called "read-ahead"), where an additional number of blocks of data 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 pre-fetching. As a result, RMAP 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 resultant of storing those pages across different servers in past requests. The queue will silently drop packets when the queue gets full or when a packet has been retransmitted X number of times. Additionally, Anemone clients have their own flow control mechanism, by which they retransmit a request upon a timeout event if the engine has dropped one and failed to answer it.

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. Future work that implements a system using multiple memory engines may require the servers to have a queue. This is the extent of the server's design - it is very simple and operates only in memory.

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. The protocol needs only handle retransmissions. This is done by timing out requests contained within the aforementioned request queue. After a maximum amount of retransmissions to the memory server to store or retrieve a page, the request is dropped and the client must retransmit the request from scratch.

4.5 Kernel-Space Specifics

The Memory Engine uses an extensive amount of memory management in the Linux kernel. The books in [24] and [5] provide a lot of the nitty-gritty details described here that are used in Anemone's kernel code.

Both the engine and server implementations sit directly on top of the network card - between the link layer and network layer, as shown in Figure 3. In Linux, when the network card sends the OS an interrupt after the arrival of a packet, a second software-based interrupt called a SOFTIRQ (or bottom-half) is scheduled to run a short time

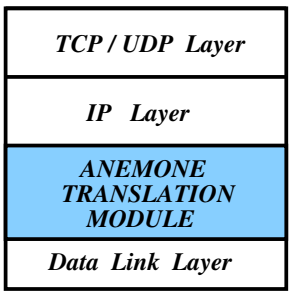


Figure 3: Placement of Anemone in the memory engine's network stack.

later. This softirq, which runs in "interrupt mode", is what actually takes the packet and hands it to the network layer for IP processing. Instead, the engine uses standard linux NetFilter hooks to pull those packets away from the path of IP and process them first after the softirq fires. From here, Anemone interprets the request, schedules communication with a server and returns control to the kernel all in one shot. On the server-side, the exact same procedure happens upon reception of a message to store or retrieve a page.

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 is 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. For every 64 GB of remote memory, the engine would need 1 GB, which is not an unreasonable demand on the engine. The slab allocator does fine-grained management of small, same-sized objects of memory by packing them into pages of memory side by side until a page is full. When full, a new page is allocated by the slab allocator and the process repeats as more entries are added to the table. kmalloc() is never 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. The slab allocator is also very appropriate for the other memory needs for the various structures and objects used in our kernel modules.

Linux, which is the only operating system that the prototype implementation of Anemone's engine and server is currently written for, has a specific way of addressing memory. Linux partitions memory into 3 zones: DMA (Direct Memory Access), Normal, and High Memory. The first 16 MB of memory is reserved for DMA, used for direct device-to-memory transfers. After that, the Normal zone occupies memory between the range of 16 MB and 896 MB. The kernel's critical memory usage starts here. Everything after the 896 MB range is called High Memory. In order to use any 4 KB page in the system, the kernel has a page table (PT) and each user process gets its own PT. The kernel's PT is big enough to statically address everything in the 'Normal' zone, which is referred to as a 1 GB / 3 GB 'split'. In order to "atomically" address 4 GB of memory in kernel space (i.e. pull from high memory), one must call alloc_pages() and *temporarily* map that page into the kernel's page table. These mappings are very short-lived because the number of PTEs in the kernel's PT is limited, so a particular thread should not expect to own that mapping in the PT the next time it executes - it must be re-mapped each time. This is sufficient for our design, but requires time to implement and get right. A temporary solution to this on our memory servers is to use a patch that changes the kernel/user split to 4 GB / 4 GB. To do this, the TLB (Translation look-aside buffer) must be flushed each and every time the switch between user mode and kernel mode is made, incurring a few microseconds overhead, which we found to be negligible in our performance tests. As a result, we can allocate all available memory in an server machine connected to the Anemone system.

4.6 RMAP: Reliable Memory Access Protocol

The translation module, shown in Figure 2, handles the client request processing at the engine. It needs to be fast so that page-fault latencies are kept short. The Memory Engine is dedicated solely to the task of handling memory requests. 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 the congestion control and in-order byte-stream services of TCP. Because Anemone operates within a LAN, there is no need for any network-layer routing functionality. Thus Anemone can avoid the extensive IP layer processing as well. The need for packet fragmentation is eliminated because gigabit networking technology provides support for the use of *Jumbo* frames which allows 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 datagram based reliable communication protocol modeled upon a modified sliding window protocol for communication with both clients and servers, but without the requirement of in-order delivery.* To accomplish this, all the communication modules of Anemone (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} establish a connection with the memory engine. When the engine reviews a REG request it allocates a VAI and returns a file handle on it. The client can use this handle to communicate with the VAI for the duration of the connection with the server. As one would expect, when the engine receives an UREG request it deallocates the VAI and all meta-data associated with that client, frees remote pages belonging to those clients, and returns success to the client. READ, WRITE, and their associated ACKs provide basic reliability for reading and writing between the client and engine. A simple per-request timeout is used for retransmissions of READ/WRITEs. If a READ/WRITE communication does not receive its associated acknowledgment after a given amount of time, the request is retransmitted. Retransmissions at the engine are also bounded, and if a request is retransmitted more than the retransmission bound it is silently dropped, leaving it up to the client to retry. Finally a STAT function is made available to allow the client to gather information on its corresponding VPI(s) in the engine, such as available memory and average workload.

A lightweight flow control strategy has also been implemented in the protocol. 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 staticly sized window technique to limit the number of outstanding network memory requests. When the queue is full, the client stalls any upper layer paging activity until it has received until ACKs come in and shrink the queue. These ACKs are allowed to come in out-of-order. Future implementations of the client module may include dynamic resizing of the queue, but are not currently realized in the RMAP protocol.

The RMAP implementation also provides several other tunable parameters. Of these one allows adjustment of the transmission window size, and three are timers, providing different options to the user. All four of these parameters interact with each other intricately, but below is quick summary on how each parameter can affect the overall client module.

Window size, which is overlayed onto of the transmission queue, allows the network transmission window size to be set, so as not to overflow the link in addition to not overflowing the engine(s). 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 optimal window size to lie somewhere between 10 and 12 page-requests. This appears to mainly

be due to swap daemon prefetching. This can be observed during the quick-sort experiment, and will be discussed further in Section 5.

There are also three timers available for optimization. The first is a timeout value, counted in jiffies (ms), where upon each trigger of the timeout RMAP checks to see if there are any packets that need to be retransmitted. Setting this timer to a very low value can degrade performance on the system, as it will be bogged down checking queued packets and exorbitant number of times, however setting the parameter too high will increase the retransmission latency. The second timer is the per packet retransmission latency, a maximum amount of time before a particular request is retried. The larger this is, the longer it takes to retransmit, but the longer the swap-daemon must wait for this page. Depending on the network and application workload, this must be chosen carefully, or be based on the current request RTT throughout the anemone system. Finally, the third timer is a spacing parameter that specifies how long to wait between any individual transmissions. Again, depending on the workload, this can put a lower bound on the paging-rate of the client, as it restrict transmissions to a maximum amount of bandwidth, but can do good flow control. Our future work involves making all these parameters dynamically change depending on system workload and engine workload.

4.7 Caching

We have made some initial attempts at implementing a small cache (similar to that of a disk-cache) into the Client module and a large cache into the Engine core. When the client module takes in page-requests, a branch of our prototype module employs an LRU-based, write-back cache to store the most recently used (MRU) pages of the swap daemon. This type of caching is hard to do because it sits directly under the cache that the swap daemon employs on its own. Because of Anemone's solid-state characteristics, the swap daemon is not aware that prefetching is unnecessary since all remote paging has the same latency. Typically this cache will occupy anywhere between 4 MB - 16 MB of preallocated memory. All three components of the system use hash-table and fifo queue structures in various places, including the caches. When a page is finally ejected from the cache it is written to the memory engine, since this is a write-back cache. This is a calculated communication cost, so that use of the cache can reduce the total client generated network traffic by the cache's hit rate.We have observed small hit rates between 5 and 10 percent, which we think can be improved significantly by learning more about a particular application's behavior. This includes reverse-prefetching blocks for the swap daemon instead of pure forward prefetching (which only a disk would benefit from) and possible investigation to more advanced access prediction algorithms. The Engine's cache can be much larger – on the order of several hundred megabytes, as it must potentially cache pages from multiple clients. Although we do not have performance numbers on this, the effects of both kinds of caches seem promising.

5 Performance

In this section we evaluate the Anemone prototype implementation. We focus on answering the following key questions addressing the performance of Anemone:

- 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?
- What application speedups can be obtained with Anemone for real-world unmodified applications?
- How does the number of concurrently executing processes impact application speedups with Anemone?
- How much processing overhead does Anemone introduce?

- How does the communication protocol effect the performance of remote memory access?

Our results can be summarized as follows. Anemone reduces all read latencies to less than $500\mu s$ compared disk read latencies up to $10ms$. 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 introduced by Anemone modules is negligible compared to round trip communication latency. Further, we observe that the window size parameter in the reliable communication protocol has a significant impact 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 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 the Linux slab allocator.

The internal hash-tables in the engine and server are configured to have 262144 or (2^{18}) buckets. The 3.6 GB or remote memory translates to 900,000 pages. So, if used in full, translates to between 3 and 4 collisions per bucket. Since our tests do not exceed 1.8 GB (450,000 pages), we get no more than 2 page keys per bucket. Also since we're only dealing with one client at the moment, our hash-function simply performs a modulus of the offset of the page and the number of buckets in the hash-table. Allocating a larger sequential array to hold more buckets requires better usage of the "High Memory" zone, which we plan to do in the future. The hash function will be enhanced in future versions to handle multiple clients that either use different VAIs or single clients that mount multiple VAIs.

5.2 Components of Page-in Latency

A page-in operation is the 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. An 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.

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.

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 .

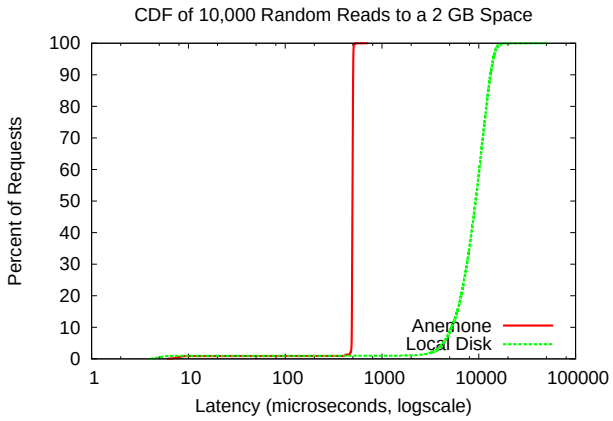
It turns out that the actual computation done at the engine and server is negligibly small: less than 7 microseconds. The majority of the latency involved is spent in communication time (which includes overhead of the network device driver) and when context switching or going between kernel/user space,

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 so 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. 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 roun 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 experience a latency between 5 to 10 microseconds. On the other hand most requests in Anemone experience close to



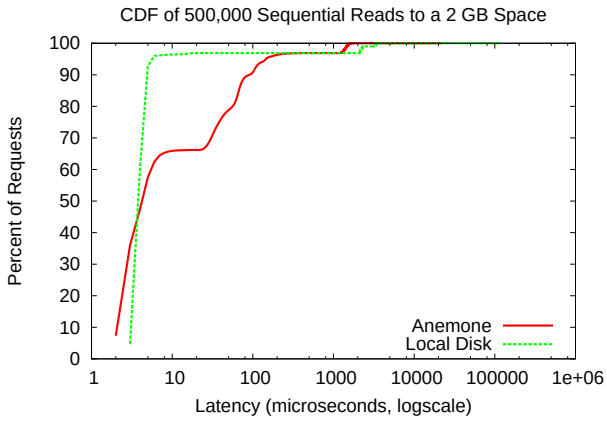
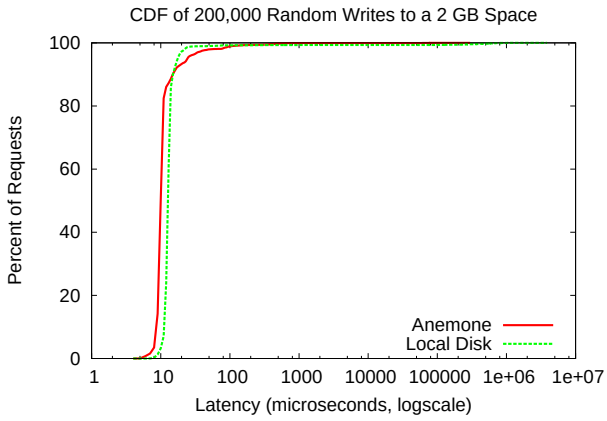


Figure 4: Comparison of latency distributions for random and sequential reads for Anemone and disk.



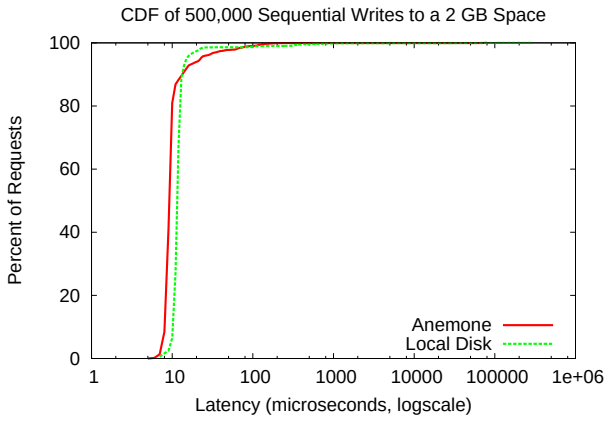


Figure 5: Comparison of latency distributions for random and sequential writes for Anemone and disk.

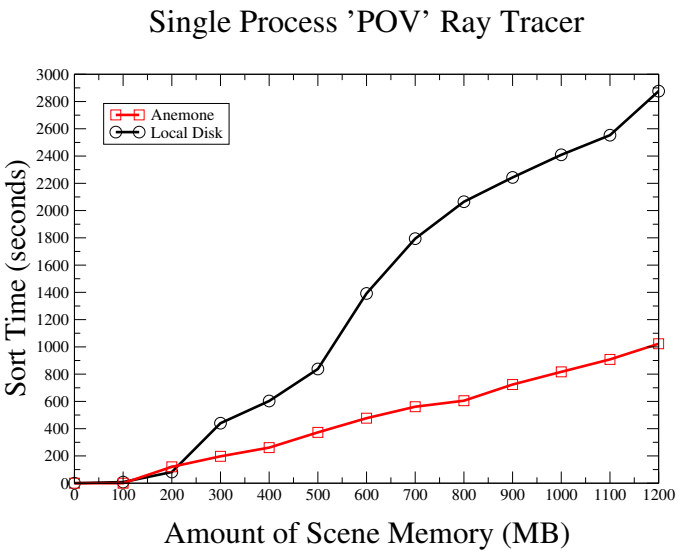


Figure 6: Comparison of execution times of POV-ray for increasing problem sizes.

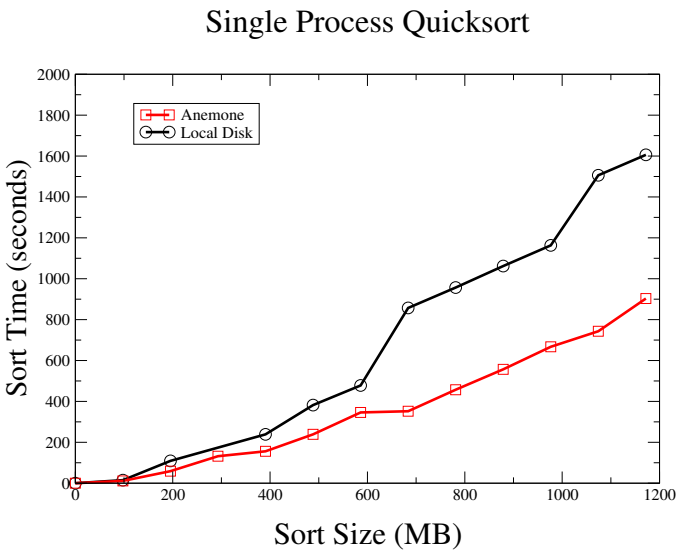


Figure 7: Comparison of execution times of STL Quick-sort for increasing problem sizes.

400 microsecond delay. 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 [23]. 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 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 [25], 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

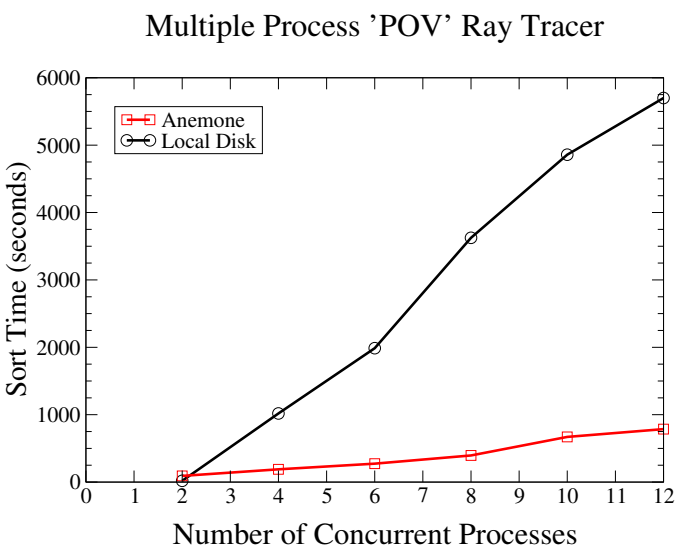


Figure 8: Comparison of execution times of multiple concurrent processes executing POV-ray for increasing number of processes.

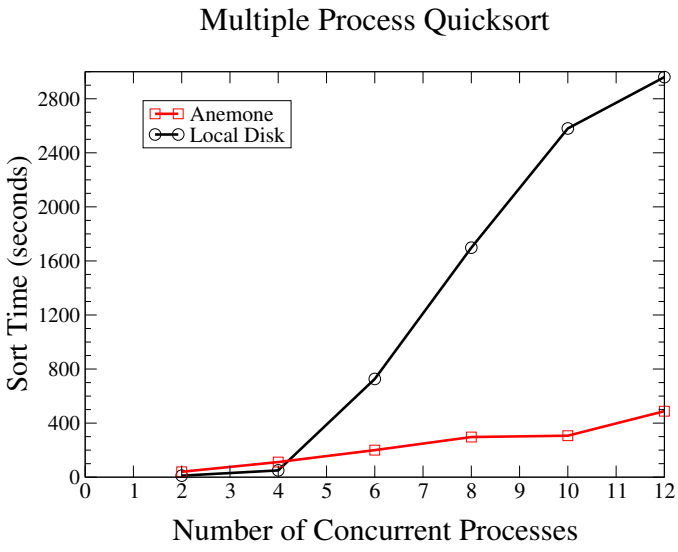


Figure 9: Comparison of execution times of multiple concurrent processes executing STL Quick-sort for increasing number of processes.

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 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: a write (or page-out), read (page-in) hits in the cache, and read misses. 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 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. Upon close examination however, we find that there is significant reverse memory traversal pattern in POV-ray, even though it is 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

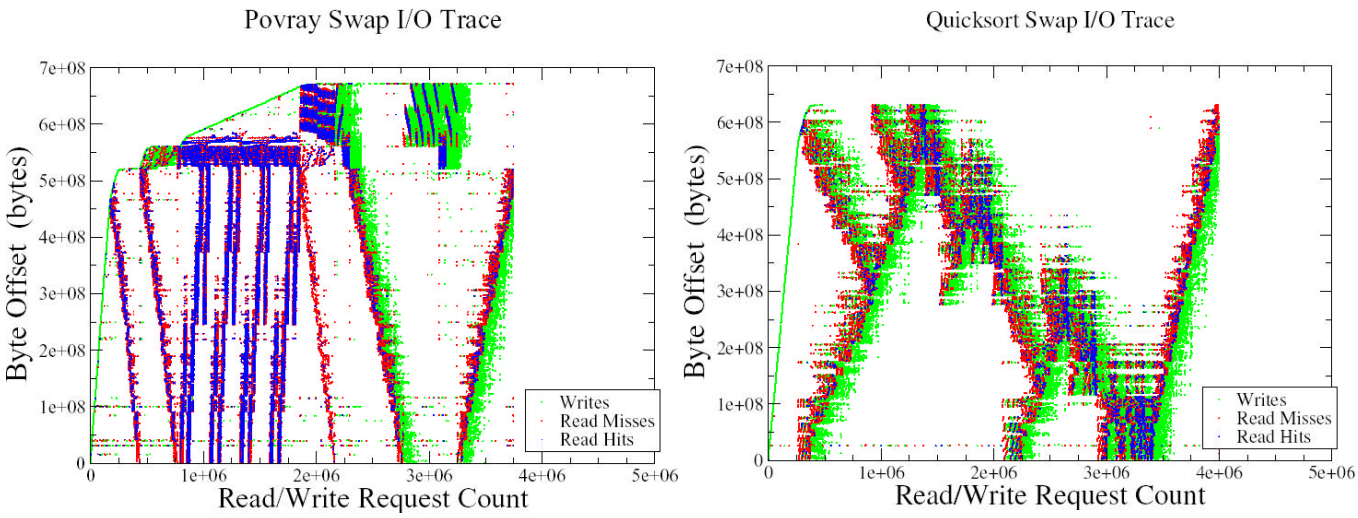


Figure 10: Trace of I/O activity for POV-ray and Quick-sort with progress of application execution. The green data points represent writes, red data points represent cache misses upon reads in the cache of pseudo block device and blue points represent cache hits upon reads.

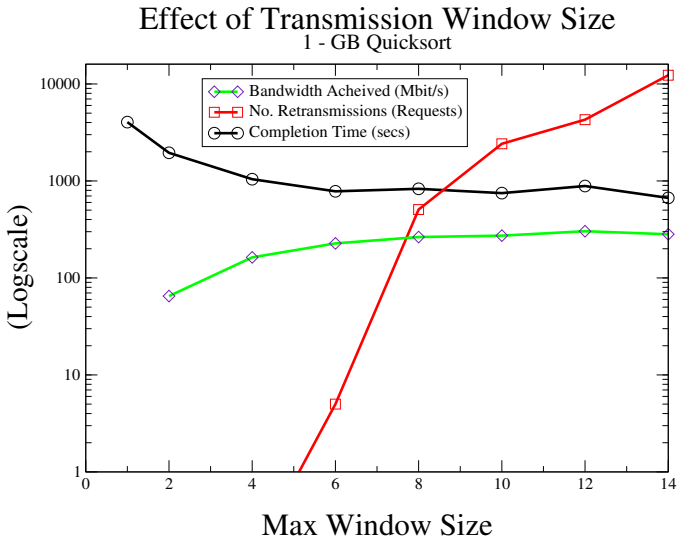


Figure 11: Effects of varying the transmission window using quick-sort on bandwidth, retransmissions and execution time.

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 out-of-order execution. 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. Paging Bandwidth through Anemone, 2. Retransmission rate, and 3. Completion Time. Paging bandwidth is represented in terms of "goodput", which is the amount of bandwidth gotten that does not include retransmitted bytes. The retransmission rate can significantly slow down the application, because the swap daemon must provide guarantees to the application that it will get its when needed. Finally, the completion times

thus depend on the previous two, as all three characteristics are graphed together on a logarithmic scale. Ideally, our future work will included the ability to change the window size in response to Anemone Round-Trip-Time and/or engine workloads using the RMAP protocol.

6 Related Work

The idea of using remote memory for paging activity is quite old and has received extensive attention during the 1990s. In recent years however, this area has not received the attention it deserves in spite of increasing LAN speeds and plummeting access latencies. In large measure, our work revisits many of the interesting old ideas in this area and examines how practical it is today to achieve transparent remote memory paging with modern off-the-shelf components. To the best of our knowledge, our Anemone system is the first attempt to evaluate the feasibility of remote memory paging over commodity Gigabit Ethernet LANs that support jumbo frames.

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 [16, 15, 4, 18]. Some of the related efforts [17, 29] did touch upon the performance benefits of paging to remote memory instead of local disks, though not as extensively as other efforts.

In the early 1990's the first two remote paging mechanisms were proposed [7, 13], both of which incorporated extensive OS changes to both the client and the memory servers and operated upon 10Mbps Ethernet. The Global Memory System (GMS) [12] 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. [11] discusses remote paging which is controlled at the application programming level and is non-transparent. Similarly, the Dodo project [19, 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 [21] 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 [22] specific to the Nemesis [20] operating system was designed to permit application-specific remote memory access and paging. The Network RamDisk [14] 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 [10]. 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 run unmodified and use large amounts of remote memory provided by the cluster. Samson [26] is a dedicated memory 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 also extensively modified.

Simulation studies for a load sharing scheme that combines job migrations with the use of network RAM are presented in [28]. The NOW project [2] performs cooperative caching via a global file cache [9] in the xFS file system [3] while [27] attempts to avoid inclusiveness within the cache hierarchy. Remote memory based caching and replacement/replication strategies have been proposed in [6, 8], but these do not address remote memory paging in particular.

7 Future Work

There are several exciting avenues for further research in Anemone. We are incorporating caching and compression of pages to reduce communication and increase the effective storage capacity. We are also improving the scalability and fault tolerance of Anemone using page replication for higher availability and using multiple memory engines for load distribution and SPOF (Single Point of Failure) recovery. Future plans for Anemone also include plans to evaluate the system under a more dynamic set of scenarios such as an larger number of clients and servers.

Developing Anemone into a peer-to-peer memory sharing cluster is also an appealing direction that will allow users across a LAN, to combine memory resources.

8 Conclusions

This paper 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 LAN. Although the idea of using remote memory for paging itself is not new, Anemone is the first attempt at evaluating the feasibility of remote memory paging over commodity Gigabit Ethernet LANs that support jumbo frames (frames of sizes greater than 1500 bytes).

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 provides a speeds up single process applications by a factor of 2 to 3 and multiple concurrent processes by up to a factor of 6. Thus Anemone provides significant performance improvements even when competing against modern disks that are optimized with high RPM, and have large local caches to improve spatial locality.

References

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

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

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

[4] 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] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] D. Engler, S. K. Gupta, and F. Kaashoek. AVM: Application-level virtual memory. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] L. Iftode, K. Li, and K. Petersen. Memory servers for multicomputers. *Digest of Papers. COMPCON Spring*, pages 538–547, 1993.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] Corbet J Rubini A. *Linux device drivers*. O'Reilly & Associates, Inc., 2nd edition, 2001.

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

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

[27] 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.

[28] 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.

[29] Y. Zhou, L. Wang, D.W. Clark, and K. Li. Thread scheduling for out-of-core applications with memory server on multicomputers. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems, Atlanta, GA*, pages 57–67, 1999.