Anemone: Adaptive Network Memory Engine

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

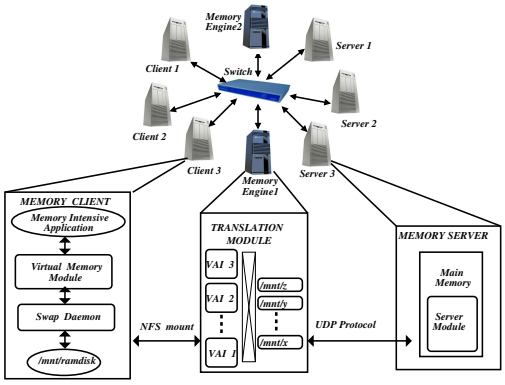

Fig. 1. Architecture of Anemone: Memory-hungry Clients on the left send their paging requests over NFS to the Memory Engine in the center, which in turn get serviced by Memory Servers on the right.

I. Introduction

There is a constant battle to break-even between continuing improvements in DRAM capacities and the demands for even more memory by modern memory-intensive high-performance applications. Such applications do not take long to hit the physical memory limit and start paging to disk, which in turn considerably slows down their performance. We tackle this problem in the Adaptive Network Memory Engine (Anemone) project by pooling together the distributed memory resources of multiple machines across a gigabit network based cluster. Anemone is a distributed memory virtualization system that can dramatically improve application performance by paging over the gigabit network to the unused memory of remote clients. Anemone provides clients with completely transparent access to a potentially unlimited amount of collective memory pool. Earlier research efforts in this area advocate significant modifications to client systems, either in terms of a specific programming interface for applications, or in terms of extensive changes to the operating systems and device drivers. In contrast, Anemone requires no modifications to either the client system or the memory-intensive application. For more detailed information, go to:

http://www.cs.fsu.edu/˜ mhines/papers/nsdi05.pdf

II. Design

A basic remote memory paging mechanism can be readily created where a single client machine mounts a RamDisk (in-memory storage device) located on a second machine over NFS. The mounted RamDisk hosts a file that can be used as swap-space across the network. Anemone expands on this static single-client/single-server approach as shown in Figure 1. While the client stays the same, the server hosting the Ramdisk is replaced by what we call a ”memory engine”. The engine serves two purposes: (1) It is a fully-functional, pseudo-NFS server that accepts NFS requests from client machines, and (2) It multiplexes over one or more backend “memory servers” contributing unused memory space. When a client mounts a file from the engine, the engine allocates a Virtual Access Interface, or VAI. The VAI is an NFS mountable virtual filesystem interface that contains all the metadata associated with it that a real filesystem on disk. Upon mounting, an NFS file handle corresponding to the VAI is returned to the client. The backend memory server stores data in its RAM and offers fixed amounts (or all) of its unused memory to the engine. A memory server does all communication with the engine using a lightweight, reliable, stop-and-wait style protocol. The engine may serve any number of clients in the cluster and may also allow any number of servers to connect and offer memory space to the clients. Upon a page-fault, the page-in request is sent to the engine over NFS. The engine internally looks up the location of the server at which the page was previously stored from a page-out and retrieves the page. The page is wrapped in an NFS reply header (without additional copying) and retransmitted to the client, fulfilling the page-fault. There are no disk operations involved in the entire process and the engine and server(s) both operate completely in memory. The engine and backend memory servers are implemented as loadable kernel modules.

III. Performance and Current Work

Disk-based page-fault latencies average 6.5 milliseconds. In contrast, individual latencies for a 4 kilobyte page-fault using Anemone are an average 650-700.2 microseconds, about 8.9 times faster than disk-based block requests. For single memory-bound processes, we observed that real-world unmodified applications like the graphics ray-tracer POV-Ray, the quicksort algorithm, and the ns2 network simulator had their execution times reduced by a factor of 2 and 3. Next, we performed an experiment that stress-tested the clients’ system by running multiple simultaneous ray-tracers to measure the speedup Anemone provides when multiple processes compete for RAM and cause lots of context switching, using about 1 GB of remote memory. We observed a speedup of a factor 13! Hence, Anemone significantly reduces the execution times of real applications in addition to providing access to potentially unlimited remote memory resources. Ideas that are currently being developed for improving and optimizing the already well-performing prototype include latency reduction by compressing pages, sending pages directly back to the client from the server instead of going through the engine, and the caching of frequently used pages at the engine itself. To guarantee scalability and fault tolerance we plan to replicate pages among multiple servers and implement backup memory engines that work together.