Solving Games with Distributed Computing: A System for Network RAM

Aaron Davidson - 263995
davidson@cs.ualberta.ca

December 12, 2000

Abstract

The problem of solving games is introduced as a memory-bound computation (requiring a great deal more memory than is physically available) and distributed network memory is introduced as a possible technique for providing large amounts of relatively fast memory (compared to disk-based virtual memory) to these applications. A specialized system is built and evaluated for managing large shared, read-only memory spaces for parallel access. A large amount of effort is focused on robustness and proper idle resource management. The final system provides significantly better access times than were previously possible.

1 Introduction

Many algorithms are memory-bound as well as, or instead of being time-bound. A particular set of problems of interest to the UofA GAMES group (`http://www.cs.ualberta.ca/~games/`) is that of solving games. Not only does this problem require a massive amount of computation, but in order to be tractable at all, a gigantic amount of fast memory is required.

The problem faced by game tree search programs that use giant endgame databases, is that the access pattern is often uniformly randomly distributed over the entire database file. Sequential, or nearby access to disk are much faster than accesses which are far apart from each other. With sequential reads, the read-head and disk platter movement is minimized. If the accesses are nearby to each other, good locality is rewarded by caching strategies.

With uniformly distributed access to very large files, the amount of movement of the disk platter and read-head is very large. Because this is so slow, huge speed gains can only be made by keeping as much of the file as possible in faster memory.

One way to meet these demands is with the growing pool of computational power and memory in our local area network (LAN). For the majority of each day, most of the machines sitting on our desks and in our labs are powered on and idle [7]. If this resource can be tapped without interfering with the users of the machines, problems such as solving large games could become tractable with existing resources. This is discussed in Section 3.

Section 4 introduces the architecture of a network memory system that was built with game solving in mind. The system provides a large capacity pool of memory from idle workstations on a local area network. The memory can be accessed in near constant time, providing a speed advantage an order of magnitude above random access to disk. In Section 5, the evaluation of the system is presented, including tests with a real games application.

2 Solving Games

In solving a game, it must be proven that the game is a win, loss, or draw for the players. In other words, one wishes to determine the game-theoretic value of a position, or in some cases, any arbitrary position. Schaeffer *et al.* give four classifications for game-theoretic solutions [9]. [1]

An *ultra-weak* solution is a proof of the theoretical outcome of a game, but offers no strategy for doing so. For instance the game of Hex has a simple mathematical proof that the game is a win for the first player, yet there is no known strategy to accomplish this in practice.

A *weak* solution gives a strategy for a position, that if followed, provably leads to the game-theoretic value.

A *strong* solution gives a strategy for *any* arbitrary position, not just one.

Ultra-strong solutions take into account that an opponent may not always play perfectly and gives a maximal strategy for situations where the opponent may make mistakes. For example, despite being in a drawn position, by making moves that are difficult to reply to, the opponent may stumble off of a drawing line and choose a losing move instead.

2.1 Solving Connect Four

in 1988, Victor Allis solved the game of Connect Four, proving that it was a win for the first player [1]. He constructed a proof that no matter what move the second player chooses, there is no move that can win against the first player (given that the first player plays perfectly). A variety of search techniques were

[1] The first three classifications were initially defined by Victor Allis [1]. Schaeffer *et al* added the fourth.

used, as well as a database of half a million positions. Connect Four has a search space on the order of 10^{13} positions.

2.2 Solving Checkers

Checkers is a much larger search space than Connect Four, so a solution will require a massive amount of computation. *Chinook*, the World Checkers Champion, is a program written by Jonathan Schaeffer at the University of Alberta. *Chinook* uses a meticulously hand-crafted evaluation function and sophisticated game tree search methods to search as many moves into the future as possible. One of the program's strengths is its large endgame database [6]. The database contains perfect information for all checkers games where there are eight or less pieces left on the board. When a search has reached a position with eight or less pieces on the board, it can simply see if the position is a win, loss, or draw by consulting the database, rather than searching further.

This program requires more than six gigabytes of memory in order to store its gigantic endgame database. Otherwise, the program must constantly make accesses to the database on disk. This means that without six gigabytes of memory, Chinook is often I/O bound doing database lookups during the search.

The University of Alberta possesses some parallel computers such as the Aurora & Borealis SGI-Origin 2000/2001 computers, that have more than six gigabytes of memory. However, solving the game of checkers will likely take many months or even years of CPU time. Since the SGI-Origin's must be shared with many other researchers on campus, they cannot be dedicated full-time to game solving.

Fortunately, the search can be parallelized easily, with decent granularity. Idle computers can participate in sharing the workload of searching positions of the game tree. The only problem is that any idle computers that are to work on searches also need access to the giant endgame database. For most workstations, it is infeasible to use up six gigabytes of disk space (just for someone else's experiment!). Even if this was done, the disk access bottleneck remains. A past solution was to have a dedicated server that sent blocks of the databases out to clients on the network. The server quickly became I/O bound servicing requests. Individual searches have decent locality in the database, but several searches in parallel end up creating a fairly uniform distribution of database block requests.

2.3 Solving Awari

Another game studied by the GAMES group is Awari, an ancient African game played with pebbles and pits. Like *Chinook*, it also uses a large endgame database [10]. It is believed that solving Awari may also be tractable provided the program is not perpetually I/O bound. Awari has a search space of roughly 10^{12} positions, while checkers has 10^{20} [9].

Awari may be the first game the GAMES group will attempt to solve, before moving on to the more challenging checkers problem. Awari starts with 48 stones on the board, and stones are gradually removed from the board as the game progresses. The player to capture the most stones wins the game. The UofA GAMES group has constructed endgame databases for all positions with 35 stones or less on the board. Thomas Lincke at ETH Zürich, Switzerland, has independently constructed a 36 stone database. The cumulative size of the awari databases up to 35 stones is 48.7 gigabytes!

3 LAN Memory

This research project will explore the concept of network memory as a faster alternative to disk-based virtual memory. With modern switched 100-base-T networks (or better) and anywhere from a dozen to several hundred idle machines, there is a vast amount of memory available for use that should be much faster to access than hitting a local disk [5]. A disk access is a glacial event in computer time. A physical read head must move across the disk platter, and then we must wait until the sector on the platter spins around to the read head. A network does not have moving mechanical parts. Provided latency can be kept low, it should be several times faster to access memory from the network than from a disk [4].

3.1 Parallel Memory Access

The distributed nature of network memory gives more advantages than just being able to have very large memory spaces. An immediate advantage of a distributed memory space is that it makes much better use of the network. Memory requests do not all go to the same machine, or even the same part of the network, so a single machine is not bottlenecked servicing requests. This was one of the drawbacks in the database server used in the past chinook projects.

3.2 Scalability & Fault Tolerance

The biggest technical difficulty in attempting to use the networked workstation environment, is that resources can never be considered fixed. Machines may crash, networks may go down, and depending on the time of day certain machines will or will not be idle and available. Any system attempting to work well in such a dynamic, uncontrollable environment, needs to scale to the currently available resources. This project uses a tracking program as an abstraction of the dynamic network resources. Resources are allowed to aggressively drop from the pool without harm. Great care is taken to ensure that no matter what happens, be it a lightning strike or network congestion, that the worst case is simply a slowdown in performance [3].

For instance, if there is not enough idle memory available to service the needs of a program, a safety net can always be available. When a network read fails, the program can always recover by going to disk for the missing data, or if the disk is unavailable, a dedicated program can service disk requests, as was done in the old parallel chinook system.

3.3 Being Friendly in the Network Environment

One of the most daunting tasks in getting a distributed software system working is finding machines to run your distributed software. Finding the machines is not so hard, but convincing the users of those machines to allow the software on their computer can be next to impossible. Users will undoubtedly fear that such a program will interfere with their usage of the machine. The software might gobble up resources which they require, cause noticeable slow downs, or even crash the machine.

As an absolute rule, a distributed program should *never* bother the user. If this rule is ever violated, your overall access to useful machines will decline. It will be much more difficult for future distributed projects to gain a foothold if users have had negative experiences in the past. It is always better to suffer a minor slowdown to convenience the user, than to lose access to a machine indefinitely.

The social behaviour of a distributed application cannot be implemented as an afterthought. It must be considered in the high level design of the distributed architecture. If a user suddenly starts using an active node, then that node must immediately shut down or sleep, and the rest of the system must be able to handle the sudden loss.

The only way to deal with this is to have a good idle detection mechanism, which will be dependent on the specific operating system and platform. By checking load averages, key presses, the number of users logged into the machine, daily access patterns, and so on, a very accurate idle detection mechanism can be created.

Since a lot of network activity will also be generated by a LAN-memory system, the saturation of the network is also a potential problem. We must take care to not significantly impact the network performance for other active users.

3.3.1 Detecting Idle Machines

In modern UNIX operating systems, there is usually a way to determine both the number of processors online, and the current load average. Unfortunately, each OS has a different way of obtaining this information so some messy code is needed to gather it. A fairly decent test for an idle, or at least underutilized machine, is to compare the load average to the number of processors. If the load is less than the number of processors, then the machine still has idle cycles available. The listing in Appendix A gives the code that handles idle detection.

3.3.2 Determining Idle Memory

Before a page is allocated, a call to `sysconf()` is made to obtain the system page size and number of available free physical pages. If there is enough free memory to allocate the new page, then we can proceed.

Another way to determine if memory is idle or not is an intrusive detection mechanism. While, not ideal, it works quite well in practice. The memory node creates a second thread that periodically monitors the status of memory using the system call `getrusage()` which returns numerous statistics about the calling program. One of these statistics is `ru_nswap`, which is the number of times a page from your memory space has been swapped since program startup. Each time this value is obtained, it is compared to the number of swaps from the last checkpoint. If this number has gone up it means that the machine is out of free memory. Either the node is swapping its own pages out in order to allocate new pages, or other programs on the machine are causing our memory to swap. In either case, if this occurs, the node selects a victim page that has a low access count, and drops it. Alternatively, a node could attempt to migrate the page elsewhere instead of just dropping it. Since pages in our test domain are read-only copies of a large file on disk, it is safe to simply drop the pages. Safely migrating a page is left for future work.

4 Architecture

The design of this network memory system has been optimized for the specific problem of speeding up random access to memory spaces larger than physical memory, as well as allowing this large memory space to be accessed by multiple clients in parallel.

Other systems have been developed in the past that are tailored to much more general use. For arbitrary parallel applications these general systems will likely outperform my system, but would not work well at all with the game solving applications. One such system, Treadmarks [2] uses the common practice of page swapping, in order to share memory. It is optimized for sharing data, and particularly, in handling read/write consistency between processes sharing memory. There are no optimizations for abnormally large shared memory spaces, nor would it handle the random access patterns well. Since the entire shared memory space is much larger than the physical memory at any machine, and the access pattern is random, Treadmarks would need to constantly swap pages in and out of memory.

This project will not move pages from node to node. Due to the nature of random access patterns, memory will stay in only one location at a time, and reads and writes will be individual requests over the network. For any program with a predictable access pattern (i.e. good locality), moving frequently accessed pages to local memory would be far more efficient than sending out thousands of individual messages. Since our applications do not have this common quality, the latter approach is actually more efficient than moving pages.

With random access over a very large memory space, a paging based system would perform horribly. In order to read a single byte, an entire page has to be sent over the network. The probability that the page will be needed again before it must be swapped out is very low (since the amount of local memory is far smaller than the shared memory space). This was empirically verified in an early experiment that implemented local page caches. For random access patterns, the system performed extremely badly, but performed well when access patterns were more localized. The current library does not implement local page caching, but could be easily extended to do so.

Memory is still kept in pages, for management, but since these pages are relatively stationary, they are much larger than in traditional systems. One nice bonus of not having local page caches is that memory consistency problems do not arise. For any shared memory address, there is only one actual location for that value, and reads and writes are sequentially consistent.

4.1 Memory Nodes

Memory nodes are the individual machines running the `memnode` program. A UDP message protocol allows other programs to ask the node to create a page, write to a page, read from a page, and some other operations that will be discussed later.

A node will allow pages to be created until it has sensed swapping or the machine has become active by other users. For idle machines, particularly at night, nearly all of the machine's memory can be allocated.

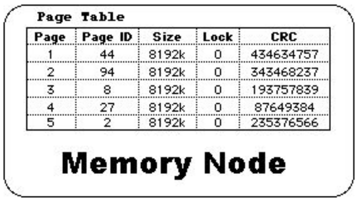
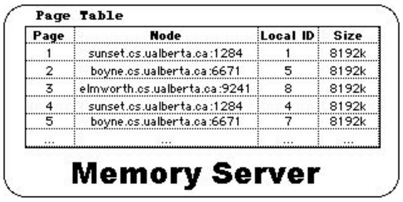


Figure 1: A Memory Node with Sample Page Table



Page	Node	Local ID	Size
1	sunset.cs.ualberta.ca:1284	1	8192k
2	boyne.cs.ualberta.ca:6671	5	8192k
3	elmworth.cs.ualberta.ca:9241	8	8192k
4	sunset.cs.ualberta.ca:1284	4	8192k
5	boyne.cs.ualberta.ca:6671	7	8192k
...	...	...	...

Figure 2: A Memory Server with Sample Page Table

The program is not threaded, and handles requests one at a time. When actively serving memory, `memnode` is still I/O bound on the network, and uses very little CPU.

Each page is stored in a local page table, and has both a global and local page index. The global index specifies the logical page number in the memory space. The local page id is an index into the local page table array.

4.2 Memory Server

The server can be thought of as the master memory node. It provides an abstraction away from the fluctuating number of nodes and free memory in the system. Essentially, the server is just a page/node tracker. It coordinates page creation, and acts as a central tracker to let several clients share the memory in parallel. At startup, a memory node will send a registration message to the server, and at termination, it will send an unregister message. The server keeps all nodes in a rotating queue, dishing out page creation requests to each node in a cyclic fashion. If a node rejects a page creation request (because it has no more free memory), then the next node in the list is tried. If all nodes refuse, the server returns an error to the client that requested the page creation.

When a client needs to access a page for the first time, or has lost track of a pages current location, it can ask the server for the latest information on a page's location. Page locations are cached in the client, so the server is only needed when the cached location of a page is missing or invalid.

4.3 Handling Correctness

UDP is a connectionless, unreliable, but fast way to send data over the network. Care has been taken in the protocol design to ensure that the out-of-order packet delivery is not a problem. All messages are within a maximum length which ensures the data will fit in one packet. On our current network the Maximum Transfer Unit (MTU) is 1500 bytes, which must store packet information as well as the actual data. A maximum size of 1024 bytes was used, which is comfortably within this limit.

Most calls to `recvfrom()` are set to non-blocking mode with timeouts, to prevent any of the programs from accidentally blocking indefinitely. In the event of a timeout, the library calls do not attempt to retry. Instead, they return to the caller with an error. The library calls do handle cases where acknowledgments are received but are error conditions. Unknown errors such as timeouts are passed up to the user of the library where a better decision can be made [8]. For instance, in our test programs, any read error can be handled by simply reading from disk. In this way, any network problems may cause a slowdown in the program, but correctness will not be sacrificed.

All writes are verified with the liberal use of CRCs (32-bit). For block transfers of a maximum of 1024 bytes each, the bytes are accompanied with a CRC. In writing an entire page, several block transfer messages may be needed. After an entire page has been initialized, CRCs for the page are compared at both ends as well.

```
/* look in local table to see if we know where this page is */
node = local.findNode(pageID);

/* if we don't have a location for this page, ask the server */
if (node == null) {
    node = server.findNode(pageID);
    /* update our local table to reflect new page location */
    local.setPageToNode(pageID, node);
}

/* try and obtain the page from the node */
data = node.readByte(pageID, address, size);

/* if the node did not respond, or didn't have the page */
/* then our local page table has an invalid entry */
if (data == NULL) {
    /* invalidate our cached page location */
    local.setPageToNode(pageID, NULL);
    /* retry from beginning, or alternatively, return an error */
}
```

Figure 3: The following pseudo-code illustrates a simplified example of how a page can be accessed in the library, for reading. We first check the local page table. If the page location is not in the table, then it can be fetched from the server. Once the page location is known, a read request can be sent to the node. If this fails, we can either retry the entire procedure, or fail with an error.

4.4 Client Library

The client library is a user's interface to the network memory. Only a handful of simple routines are needed. Many more are available, but very few are needed to get started. A program first must initialize the library with a call to <int NMS_init(char *srvr, int port, int page_size)> which will attempt to connect to the given server. The page size specifies how large pages will be in the system. The handy function <unsigned int NMS_translate_address(unsigned long long *addr)> will map from the logical address space to the appropriate page and offset. Note the use of unsigned long long as an address type. This allows for memory spaces up to 2^{64} bytes.

Data can be read with either <int NMS_readByte(unsigned long long addr)> or <int NMS_readBytes(unsigned long long addr, char *buf, int blen)>. Several errors can be returned by these functions. Errors include situations such as where the page is not in the memory pool, the node is not responding, or has dropped the page, or the page exists, but is locked. As with the reading functions, there are equivalent functions to write data. A node can be queried for the CRC of any of its pages with <int NMS_getPageCRC(int page, unsigned long *crc)>, so pages can be verified to be holding the correct information.

<int NMS_ensurePage(int page)> ensures a page is in the memory pool. If it cannot find the page, it creates one, provided free memory is available. It then locks the page. This is useful for initializing pages in the memory pool.

```
#define TRIALS 100000

void test_disk(char *file) {
   int            fd;
   int            i;
   unsigned long  fsize;
   char           *buf;

   if ((fd = open(file, O_RDONLY) < 0)) return;
   fsize = lseek(fd, 0, SEEK_END);

   printf("Testing Disk with %ldkb file...\n", (fsize/1024));
   buf = (char *)malloc(TRIALS);
   srand48(42);
   for (i=0;i<TRIALS;i++) {
      lseek(fd,(drand48()*fsize),0);
      read(fd, &buf[i], 1);
   }
   printf("\tCRC = %lu\n", crc32_mem(buf, TRIALS));
   free(buf);
   close(fd);
}
```

Figure 4: Source code for timing the disk

5 Evaluation & Testing

In testing this system, we are interested in the fault tolerance and correctness of results under many trying situations. Once correctness has been ensured, performance must be evaluated. A test program was built to test both speed and correctness.

The baseline with which to compare against is the disk access times. The disk test routine, shown in figure 4 opens a file, seeds a random number generator, and does one hundred-thousand reads uniformly distributed over the length of the file. The reads are all buffered and afterwards a CRC for the buffer is calculated. This gives us a baseline time for comparing speed, and a CRC for checking correctness of the network memory system.

The test program then initializes the library with a call to `NMS_init()`, seeds the random number generator to the same value as used in the disk test, then executes one hundred-thousand reads to the same addresses, but this time calling `NMS_readByte()` instead. The first time this is executed, none of the file is in the network memory, so `NMS_readByte()` returns a page missing error. When this happens, the program calls a routine that will load the page segment from the disk, create and lock the page, write the page data with `NMS_writeBytes()`, verify the CRC, and unlock the page. If the page is loaded successfully, the program attempts `NMS_readByte()` once again. If it fails yet again, then the byte is fetched directly from the file on disk. After running this program once, assuming the memory nodes do not drop any pages, the subsequent runs find all of the pages already on the network, and the overall run time is much faster.

Figure 5 shows the times taken on various file sizes (Awari endgame databases for 18-27 stones), for one hundred-thousand random disk accesses, the network loading time and reads, and the network time for just reads on a pre-loaded file. Benchmarks were taken on `lousana.cs.ualberta.ca`, a Dual P-II 400, with 128mb of DRAM. A page size of 8mb, and five memory nodes were used to distribute the memory.

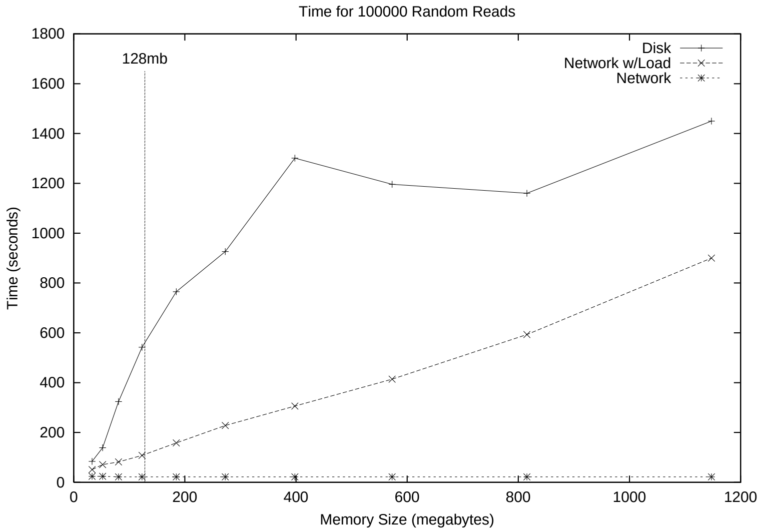


Figure 5: File Size vs Access Time for 100000 Random Reads

The disk access time spikes strangely at 400mb, and was always reproduced. This may be due to file fragmentation, or possibly some artifact of the operating system's file buffering.

Successful calls to `NMS_readByte()` typically take 0.22ms to complete. However, last-minute benchmarks on newer machines, such as `elmworth.cs.ualberta.ca`, a Dual P-III 700 w/ 512mb of DRAM, showed even better performance with single-byte reads taking just 0.15ms. This is much faster than the disk accesses which took roughly 15ms per read on a 1.2gb file.

As well as reading single bytes, a second type of test did fifty-thousand, 1kb block reads (Figure 6). These are the same kind of reads that chinook does. It reads 1kb blocks from the database into a local cache, so this was a good test to prepare for working with chinook. A 1kb block read takes 0.46ms on `lousana`, and 0.38ms on `elmworth`. The equivalent disk access time was 15ms.

Not surprisingly, it is faster to load the file in sequential chunks, distribute them over the network, and do the random reads, than it is to just do the reads from a file. When the file has been loaded entirely into memory, it is an order of magnitude faster to read from the network than from the disk. However, as file sizes approach the size of available local memory, the operating system is able to cache a large percentage of the file in memory, and the speed advantage dissapears.

To test correctness, particularly fault-tolerance, the test program was run many times while memory nodes were killed, memory was taken by other processes, causing nodes to drop pages, and so on. I was even fortunate to have a machine crash while running a test. After each run, the CRC was compared to the CRC generated by the disk test. No matter what situation presented itself, the program managed to handle the errors, the loss of nodes and pages, without sacrificing correctness.

5.1 Using Network Memory in Chinook

For a more substantial test of the system, a real application, Chinook, was modified to use network memory instead of disk accesses.

A program called `endgamedb` was created, which simply loads each of the database files into the network memory space. Each of the files is mapped to a specific location in the global address space. The files are

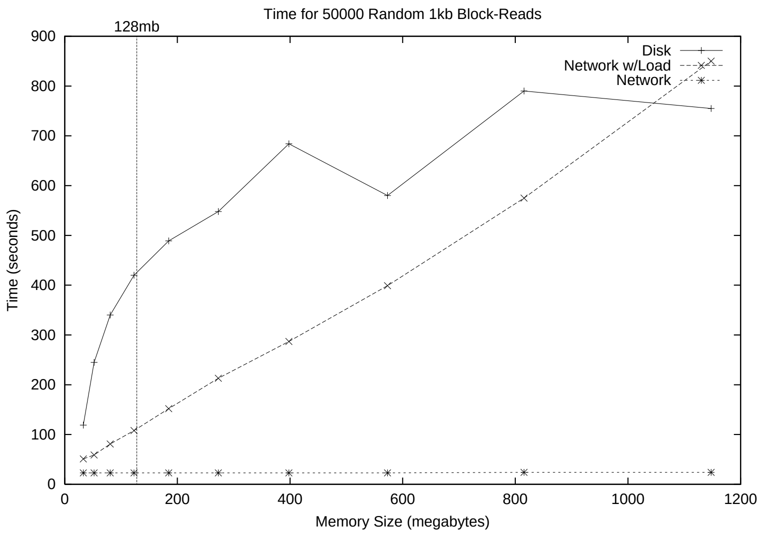


Figure 6: File Size vs Access Time for 50000 Random 1kb Block-Reads

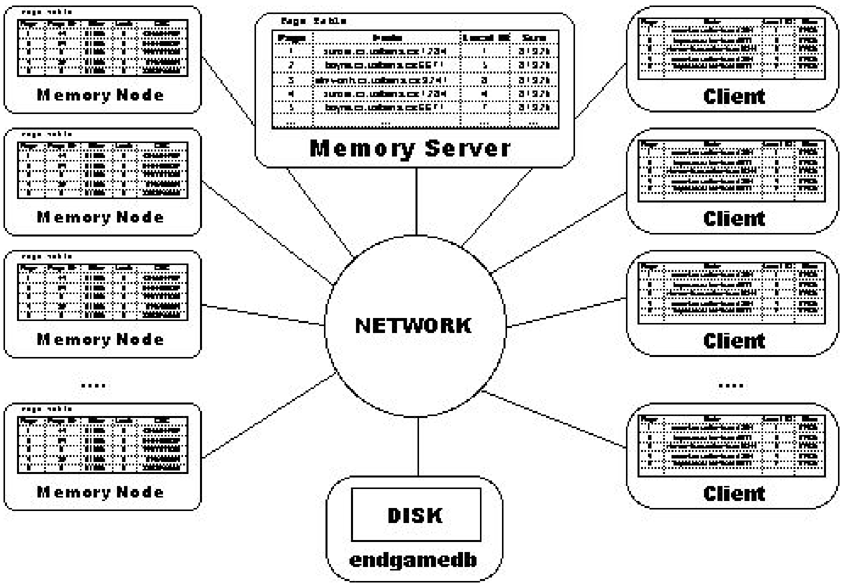


Figure 7: A high-level diagram of the system architecture used for Chinook.

Database	Depth	Reads	Time	CPU Time	I/O Time	Nodes/Sec
7 (disk)	17	5069	76 sec	49 sec	27 sec	78908
7 (net)	17	5069	53 sec	49 sec	4 sec	113151
7 (disk)	19	8805	142 sec	97 sec	51 sec	60425
7 (net)	19	8805	113 sec	100 sec	13 sec	103203
8 (disk)	17	5532	83 sec	38 sec	45 sec	54717
8 (net)	17	5532	47 sec	39 sec	8 sec	96629
8 (disk)	19	20823	304 sec	137 sec	167 sec	54418
8 (net)	19	20823	155 sec	139 sec	16 sec	106730

Table 1: Performance results for searches (conducted on `lousana`) with and without the network memory. The 8-piece database was held by a pool of 35 nodes (Linux & Solaris workstations) from the AI Lab, Systems Research Lab, and Undergraduate Open Access Lab.

padded to align with the page size. All the `endgamedb` program does is load the files onto the network, and then periodically verify the CRCs of each page. If a page is not found (because it has been dropped) it will attempt to reload it. For any pages not in the network memory, `endgamedb` acts as a classical file server, going directly to disk for each request. `endgamedb` is the only program that must have direct access to the database files. All other programs can be running on computers that have no direct access.

Chinook itself was modified to use our library instead of the database, for I/O requests. When chinook needs to load a 1kb block, the file and block address are mapped into the global address space, and `NMS_readBytes()` is called. If the read fails, a routine requests the data directly from `endgamedb`. If this fails, it is considered a fatal error, since it is now assumed that chinook does not have direct access to the database file.

Table 1 shows an experiment comparing search speeds with and without the memory system. An arbitrary position 30 ply into a checkers game was chosen (so that searches will hit the database more often), and all tests were conducted from this position with both 17 and 19 ply searches. Actual time measurements are broken down into CPU time and estimated I/O time, which is the difference between actual time and CPU time.

Performance during regular search improved by a fair amount. For example, In the 19-ply searches using the 8-piece database, the original version spent 55% of the total time I/O bound, while the modified version spent only 10% I/O bound. The I/0 time was roughly six times faster on average when using the network memory.

A single search has decent locality in the databases and chinook's caching strategy eliminates as much as 90% of the potential I/O. The performance benefits will be much more evident in the future, when many chinooks are accessing the database in parallel. The bottlenecks experienced in the past should be greatly reduced.

Correctness was verified by playing several identical games with both the regular and modified versions of chinook. Both versions played identically and gave identical results in evaluations.

6 Future Work

The memory pool system has plenty of room for future work and development. It is perfectly suited for the test applications mentioned, however many other features will be needed to expand the scope of usefulness to other domains.

One of the major flaws of the system is the lack of security. Any process can connect to the server or a memory node and read or modify data. Some sort of key should be required to gain access. This could also be extended to let the server track multiple memory spaces instead of only one.

The locking abilities are currently very primitive and not useful for much other than page initialization. A more sophisticated locking and memory consistence protocol would be required should page caching be implemented. Related to this would be support for page policies other than the default (just drop the page), such as migration, or disk caching.

Another flaw is that the library adds an extra layer of memory-to-memory copying of data, which potentially creates a significant amount of overhead [11]. There was no immediately clean and easy way around this, so this issue was not dealt with as of yet. It is possible that eliminating this extra overhead can improve the speedups even further. Continuing along these lines, full utilization of user-level networking techniques could make a huge difference in performance.

The library is currently not thread safe, which means only one I/O can be taking place at a time. A thread safe version would be a nice extension, allowing asynchronous reads and writes.

Yet another nice feature would be the removal of the need for the `memsrvr` program. Instead of having a server, which is a single point of failure, the entire pool of nodes could keep coordinated using UDP broadcasts to locate each other.

Last, but not least, the system needs a better acronym! For the library I simply prefixed everything with the very boring and unoriginal 'NMS' which stands for 'Network Memory System'. Work simply cannot continue on a project without a clever acronym. Some considered, but rejected names included 'WAM' (Wicked-Ass-Memory), 'VM' (VastMem), and finally, 'AARON' (Aaron's Awesome Recollection Over Networks).

7 Conclusion

The main goal of this project was to develop a system that gives real performance advantages over disk-based virtual memory. The game solving problems are great applications for empirically testing this claim. Using games as a test-bed enables one to gather results for non-trivial, real-world problems. Accomplishing this goal must not come at the expense of users on the network. In fact, the whole philosophy is to accomplish the goal at *no additional expense*. It uses idle resources that are already paid for. A side-effect of being friendly is the need to have a robust system – one that is fault-tolerant and dynamically scalable to the current idle resources available. And finally, it would be nice if the resulting system design is easy to program with, enabling the range of applications that can use it to expand.

This project, for the most part, accomplished these goals. Users did not notice the `memnode` software interfering with their work (although there is still room for improvement). The system scales nicely with the available resources, and can recover from a variety of traumatic experiences. Speed advantages were realized, for real applications, and the library was extremely easy to use. The main area that still needs work is generality. The usefulness of the final system is limited to a fairly narrow group of applications, and it would be nice to extend its abilities to a larger scope.

References

[1] L. Victor Allis. A knowledge-base approach to connect-four. the game is solved: White wins. Master's thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1988.

[2] C. Amza and A. Cox. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[3] Eric Anderson and Jeanna M. Neefe. An exploration of network ram. *unpublished*, 1994.

[4] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for now (networks of workstations). *IEEE Micro*, Feb, 1995.

[5] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. Symp. on Operating Systems Design and Implementation*, pages 267–280, Monterey CA (USA), 1994.

[6] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. In *Advances in Computer Chess VII, Maastricht, Netherlands*, pages 135–162, 1994.

[7] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(44):269–284, 1991.

[8] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, pages 277–288, 1984.

[9] Jonathan Schaeffer and Robert Lake. Solving the game of checkers. In Richard J. Nowakowski, editor, *Games of No Chance*, pages 119–133. Cambridge University Press, 1996.

[10] Jack van Rijswijck. Learning from perfection (a data mining approach to evaluation function learning in awari). In *CG2000: 2nd International Conference on Computers and Games*, 2000.

[11] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface of parallel and distributed computing. In *The 15th ACM Symposium of Operating Systems Principles*, volume 29, pages 40–53, 1995.

8 Appendix

A test.c – Idle Detection

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>

#ifdef __sun__
 #include <sys/loadavg.h>
#endif

int sys_np = 1;  /* number of processors set to 1 by default */

/****************************************************************/
/* Examines system, determines the number of processors online. */
/****************************************************************/
void systemTest() {
   #ifdef _SC_NPROCESSORS_ONLN
      sys_np = sysconf (_SC_NPROCESSORS_ONLN);
      printf("processors online = %d\n", sys_np);
   #endif
}

/****************************************************************/
/* check the system load average, and if it is less than the    */
/* number of processors, the machine is considered idle.        */
/****************************************************************/
int systemIdle() {
   #ifdef __linux__
      FILE *fp;
      if ((fp = fopen("/proc/loadavg", "r") != NULL) {
         float n1,n2,n3;
         fscanf(fp, "%f %f %f", &n1,&n2,&n3);
         fclose(fp);
         return (((n1+n2+n3)/3) <= sys_np);
      }
   #else    /* BSD & SUN */
      double la;
      getloadavg(&la, 1);
      return (la < sys_np);
   #endif
   return 1;
}

/*********************************************/
/* Get the number of swaps the program has had */
/*********************************************/
int getSwapCount() {
   struct rusage usage;
   getrusage(RUSAGE_SELF, &usage);
   return usage.ru_nswap;
}

/*******************************************************************/
/*  Simple function to determine if there is idle memory available */
/*******************************************************************/
int memoryAvailable(int size) {
   long page_size = sysconf(_SC_PAGE_SIZE);
   long pages = sysconf(_SC_AVPHYS_PAGES);
   return (size < pages*page_size);
}
```