# An Exploration of Network RAM

Eric A. Anderson Jeanna M. Neefe

December 9, 1994

Computer Science Division
University of California at Berkeley
Berkeley, CA 94720
1-510-642-8284
eanders,neefe@cs.berkeley.edu

**Abstract**

The goal of network RAM is to improve the performance of memory intensive workloads by paging to idle memory over the network rather than to disk. In this paper, we reexamine common assumptions about network RAM, compare possible implementations, describe the structure and performance of our user-level implementation and investigate various methods for providing reliability.

**Keywords:** network RAM, remote memory paging, reliability, user-level implementation

## 1 Why Network RAM

Virtual memory was intended to let program size be limited only by disk size and to have the system transparently move code and data between the disks and memory without sacrificing speed. Although disks are much cheaper than memory, they are also much slower, so programs that paged to disk ran much slower than an entirely DRAM version. Therefore people stopped running programs which paged to disk, and either purchased enough DRAM to run their programs, or decided they didn't need to run the programs. Recently computer networks have become much faster, and studies have shown that most workstations are idle, so people realized that they could page to and from the memory on idle workstations. It was clear that paging to idle memory is faster than paging to disk because network ram eliminates the physical seek time and the bandwidth of network connections is increasing faster than the bandwidth to disk. The hope for network RAM is that it would be fast enough so that two machines each with 64 megabytes of memory would perform comparably to a single machine with 128 megabytes of memory.

There are four major assumptions that people make when evaluating network RAM:

1. Disks are slow.
2. Networks are fast.
3. Most machines have free memory.
4. Software overhead is low.

We would like to question these assumptions because they are somewhat inaccurate.

## 1.1 Disks are Slow

The performance of disks for paging is generally assumed to be the worst case latency. We measured the random access time for disk to be about 16 ms. For some of the algorithms in the LAPACK suite, we discovered that the latency for the disk was about 10 ms. However, for sequential access, we measured the performance to be only 2 ms, which we believe is an effect of operating system pre-fetching and track buffers on disks. If a program strides though an array and the operating system places the pages of the process intelligently on disk or the array is a memory mapped file, then we can expect the per page performance to be about 2 ms.

## 1.2 Networks are fast

Current networks such as ATM over OC-3 have a peak bandwidth of 155 Mbits/second, which means that at least 0.22 ms must be wasted to transfer a 4k page. Assuming an average time to access memory of 200 ns, and a TLB overhead of 1 $\mu$s, an entirely DRAM program could make $\frac{220{,}000_{\mathrm{ns}}-1000_{\mathrm{ns}}}{200_{\mathrm{ns}}} = 1095$ accesses to memory. Assuming that each access is 4 bytes, the program running only in DRAM would have the time to access every single byte on the page while network RAM is just transferring the page across the network. Since the DRAM times are rather pessimistic, and the network RAM time ignores software overhead, hardware overhead, and switch latency, programs which use network RAM will need to repeatedly use the data on a page between page-in and page-out to achieve a speed even close to DRAM speeds. Unfortunately, there is a class of problems, namely large matrix problems for scientific programs which will often stride though an array, causing memory accesses once or twice a page. These types of programs will run much slower using network RAM than using all DRAM. Until achievable network bandwidths get much faster, the time to bring pages into the network will limit the types of programs that can effectively use network RAM.

## 1.3 Most machines have free memory

One goal for network RAM is to use it in a NOW[1]. While right now most of the workstations on a network are idle, it is unclear that once a parallel workload, and a file system which caches on client nodes are added to the sequential workload that there will still be sufficient free workstations and therefore free memory. For the purpose of this paper, we must assume that there exist workstations with large amounts of free memory to act as network RAM servers, and so we will ignore this problem.

## 1.4 Software overhead is low.

We know that the cost for a TLB miss is at most 1 $\mu$s on current workstations. The equivalent cost for network RAM is the cost of going through the VM subsystem. Unfortunately, the cost of going through the VM subsystem, which we saw as calls to mmap, is hundreds of $\mu$s. This is probably because the VM subsystem has never been on the critical path in operating systems since page faults were either program errors, or were going to be disk accesses. This software overhead further limits the usability of network RAM.

## 1.5 Some good news.

Although the problems above show that network RAM will have a hard time achieving a speed comparable to DRAM, network RAM is still faster than disks. Our implementation achieves speeds of 2-5 times faster than average disk speed (10ms). Further improvement eliminating communication overhead will further lower this number.

# 2 Possible Network RAM Implementations

Having decided that network RAM could benefit some programs, we next considered how to implement network RAM. The following is six different choices as well as some advantages and disadvantages of each.

- **Explicit Program Management.** This implementation would require user code to explicitly move data back and forth from either the disk or the network. This idea requires substantial code modification, but it is being explored as a possible technique for compilers. Carter [5] shows that substantial speedup can be achieved by considering all levels of the memory hierarchy from registers to disk. This solution is likely to have the best performance because it deals with the entire memory hierarchy and has the most high level knowledge about data use patterns. Combined with user-level access to the network, this solution should have a very small overhead.
- **User-Level.** This solution requires that the user modify their code to use a new malloc which allocates from network ram backed memory. This is an easier modification than explicit program management, but can still require source code. This method allows the user to limit the memory the application uses on their workstation, allowing other interactive tasks to run faster. This solution is more portable than any of the following solutions because it assumes less about the operating system, but it has the overhead of handling interrupts on page faults and managing the page tables through system calls.
- **User Level Pager.** This solution is possible in the operating system Mach [11], which allows a user level program to handle moving pages to and from second-level storage. The dependence on Mach makes this solution rather un-portable. This solution's overhead is probably similar to the overhead of the device driver and user-level process solution.
- **Device Driver.** This solution replaces the swap device for the operating system with a device which sends the pages to network RAM. One standard implementation for this is to have the device driver up-call to a process which will handle transmitting the page. Using a process requires that the OS lock down pages for that process, and correctly schedule that process while handling a page fault. This solution has the advantage that it requires no code modification or kernel source modification. It has the disadvantage that OS pages can be sent over the network, making reliability more important. This solution is less portable than user-level implementations, but more portable than kernel modifications. The overhead for this solution includes the cost of the VM subsystem plus if a process is used the cost of context switches and copying pages in and out of the kernel.
- **Modified Kernel.** This solution should give the highest performance without modifying user programs because it requires no extra interrupts or context switches. However, changes to kernels are not portable between architectures. As VM overhead drops, this solution will become more preferable because it eliminates many other overheads.
- **Network Interface.** This solution requires replacing the memory controller with some sort of custom memory/network chip, making this solution the most un-portable. However this approach is used on many multiprocessors such as Alewife, Flash and Shrimp to handle shared memory. This solution can have a lower overhead than a modified kernel because it can operate on cache lines rather than entire pages, making the data transferred smaller, and because this implementation can avoid VM overhead. The multiprocessors incur another overhead to make the memory coherent, but we can ignore that overhead for this comparison because it should not occur if only a single processor is using the data.

After examining these choices, we decided to do a user-level implementation. We believed that explicit program management would be to hard to convince people to use, and that modifying the kernel or building a network interface would not be practical during a single semester. Two other groups [10, 6] were already doing implementations as device drivers, so a user level implementation would allow for comparisons. We felt that the portability advantages, and the lower risk to the operating system made our solution attractive for both experimentation and use. As we discovered after talking to the other two groups, the overheads, and hence performance from these different approaches are quite similar, so users and implementors will have to examine the other factors in choosing between these two approaches.

# 3 A User Level Implementation

This section describes how to use our implementation, how it was implemented and how well it performs on different machines.

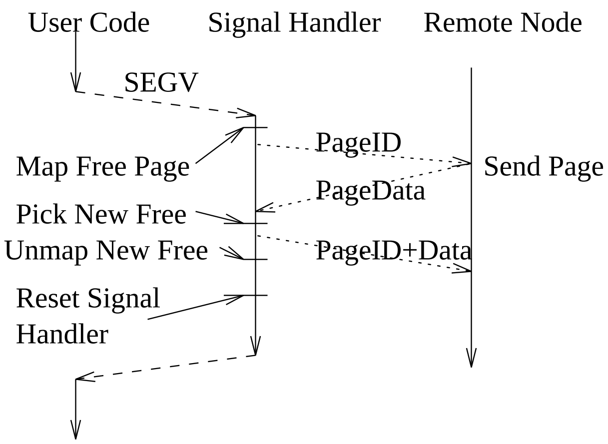


Figure 1: Steps taken to handle a page fault when sending pages to a remote paging server

### 3.1 User Interface

Our user level implementation requires a code modification to allocate memory from network ram. Pseudo C code to use this implementation looks like:

```
ptr = ralloc_init(size,cache_sz,group_sz);
/* choose backup method */
do_netbackup();
/* or do_localmirror_netbackup() */
add_servers();
/* take block pointed to by ptr,
   and use it as memory */
```

The `size` parameter specifies the amount of memory to be allocated for use in network RAM. The `cache_sz` parameter controls the amount of memory that the implementation will use on the client node, i.e. where the program is running. Finally the `group_sz` parameter causes the implementation to act as if the page size was really `group_sz` times larger. Although we implement other choices for backing store, we have only extensively tested sending the pages to remote memory servers. Our implementation does not require that the remote memory servers have the same architecture as the client machine.

### 3.2 Implementation

When a process attempts to access a page which is not mapped or is mapped without the appropriate permissions (e.g. attempting to write to a read only page), the operating system delivers a signal to the process. Using the `signal` C library call we can handle the signal and fix the page instead of terminating the process. The system will automatically restart the process at the last memory access, which will now succeed, and the program will continue without realizing that anything special has happened.

The important steps in the signal handler are marked with small horizontal lines on Figure 1. Our code maintains a free page so that transfer time and computation can be overlapped. The step of resetting the signal handler may be eliminated for some machines which correctly implement POSIX signal handling. We handle both retrieving and sending pages in a single call to the signal handle to reduce overhead from handling the signal and to simplify the implementation.

The code to manage the page mappings through mmap and munmap are almost identical on the Alphas, HP's and Sun's. We use a file on the local disk as a cache file. We use mmap and munmap to place the different pieces of the cache file in different parts of the address space, making it appear as though all of the pages are mapped. Thus only the cache requires space on disk, and better support from the operating system could eliminate even this requirement. Unfortunately, a side effect of mapping a file into memory is that the OS will try to write the pages out to disk. Because the writes are asynchronous and because current network overhead is so high, this has a negligible effect. We also tried using a private mapping, but on Solaris 2.3, this seemed to halve the memory the process actually used.

The non-communication overhead for this implementation as shown in the figure is the cost of handling an interrupt, calling mmap and munmap, and possibly resetting the signal handler. We measured this time as well as the time to retrieve a page and send a page out, which we call a page replacement. In our measurements, we found the time over Ethernet was mostly because of the low bandwidth. The time for the Alpha on Ethernet is about twice as long as all the other machines on Ethernet because the Alpha page size is 8k rather than the 4k page sizes of the other machines. TCP and ATM overhead for these switches limited the performance of Solaris over ATM. The 2.1 ms page replacement time for the HP's over FDDI is very good because it is only 3 times higher than the link time for the data of 0.6 ms. We expect the time for the faster machines will drop once a more efficient protocol than TCP is used.

We attempted to compare our non-communication overhead with the measurements of other groups. Mainwaring, Wright and Yoshikawa [10] measured an overhead of about 1 ms for their implementation of a device driver plus process for Solaris 2.3. Chan and Hsu [6] measured a total time of about 3 ms to retrieve a page in their implementation on Solaris 2.3. Because they used active messages, their time was smaller than other groups. Unfortunately, they have not yet had a chance to measure their minimal overhead.

The following graph summarizes the overhead and total page time for the different workstations and networks that we have tested:

**PRETTY GRAPH GOES HERE**

# 4 Reliability Issues in Network RAM

**"The main disadvantages of remote memory paging lie in security and fault tolerance." Felten and Zahorjan, 1991.**

## 4.1 Reasons To Provide Reliable Network RAM

- *Increased Probability of Failure*
  Clearly, with each additional node we involve, we are multiplying the probability of a failure. This is especially worrisome if we envision network RAM in an environment where workstations sit right on people's desks. In this type of environment, accidental node failure is more likely (due to workstations being turned off and on, cables being kicked, etc). Its not the same as relying on dedicated servers which are handled only by special personnel.

  The network itself is also an inherent source of unreliability. In the absence of adaptive routing, a network failure on the path between a client and a node storing its pages is basically equivalent to the failure of the remote node.

- *Desire to Present A Clean Abstraction*
  Ideally, network RAM would provide the user with the logical model of running on her local node only faster. This abstraction is compromised if the failure of some remote node, which the user did not choose or even know about, causes the failure of the user's process. When a global resource manager chooses the idle workstations (and even migrates pages as nodes transition from idle to busy) without consulting the user, it should make the failure of these remote nodes transparent to the user.

- *Threat To Operating System Code*
  Also, in some implementations of network RAM, it is possible to page out operating system data. At the device driver level there is no notion of what pages are being sent out, so Mainwaring, et. al[10] had trouble with OS code being paged out. In such implementations, reliability is especially important because the failure of a remote node puts more than just an individual user process at risk.

- *Especially Bad for Long Running Processes*
  Another important point to consider is that long running jobs are especially at risk. Clearly, the longer a process runs the more opportunity there is for a failure to occur and the more time the process already has invested in its computation. It is likely that many of the processes wanting to take advantage of network RAM will be long running.

For these reasons, we believe that it is worthwhile to explore methods for providing reliable network RAM.

## 4.2 What we mean by reliability

In light of the needs of these long running jobs, it is worth clarifying our use of the word reliable. Network RAM does not have the same reliability requirements as a file system or a transaction processing system. Unlike these types of systems where lost data may not be reproducible, for network RAM all data can be recovered by re-running the program with the same input. Therefore, reliability in the sense of "safely on disk" is not nearly as important as reliability in the sense of "accessible in a timely manner". For network RAM, the worst case scenario is having to restart the process. However, for long running jobs, this solution may not be attractive nor acceptable.

| Comparison Factor | Original Cost |
|---|---|
| Messages in network (normal case) | |
| whole pages | $P_{out} + P_{in}$ |
| short messages | $P_{in}$ |
| CPU local | request needed page<br>wait for page to arrive (overlap with next 3)<br>choose a page to free<br>decide where to send page being replaced<br>send page |
| CPU remote | receive and store page<br>send page when requested |
| Remote memory | $P_{out}$ pages |
| Page-out latency | send page |
| Page-in latency | |
| best case | send request to remote node<br>wait for page to arrive |
| if single failure | detect failure<br>process dies! |

Figure 2: This table summarizes the costs associated with basic "unreliable" network RAM. We are assuming reliable message delivery to nodes that are alive. $P_{out}$ refers to the number of pages paged out. $P_{in}$ refers to the number of pages retrieved from a remote node.

## 4.3 Reliability Methods: Description and Comparison

In this section, we will be discussing ways to provide accessibility to the needed pages in a timely manner even in the face of remote node failures. Our goal is to enable the system to recover from the failure of some fixed number of nodes. If too many failures occur the process will have to be restarted. The reliability methods which we have considered fall into three general categories: sending asynchronously to disk, replicating pages, and computing parity over pages.

### 4.3.1 A Basis for Comparison

Before beginning our discussion of the various reliability methods, we should describe the basic model of "unreliable" network RAM that we are assuming. We assume the existence of a global resource manager that maintains idle/busy status information for each node in the system. When a client wishes to use network RAM, it must first contact the global resource manager to find idle nodes willing to receive its pages. When the client sends a page to an idle node, it must also send a key which can later be used to request the page. The remote nodes must record the mapping between the client's keys and their corresponding pages. For all pages sent out, the client must also record the location of each copy.

Figure 2 lists some costs incurred by basic network RAM. For each reliability method discussed, we will provide a similar table, containing the *additional* costs associated with the method. The categories listed in these tables are some of most important factors on which we based our comparison of the different reliability methods. Its important to point out that these tables do not discuss the transmission protocols at the level of acknowledgements and retransmissions. To simplify the protocols described, we are assuming reliable message delivery to any node which is alive and accessible. Of course, at a lower level these additional messages do exist and we assume that the client maintains a "being paged out" pool where pages sit until acknowledgements are received indicating that they have been successfully received by a remote node.

### 4.3.2 Sending Asynchronously to Disk

The simplest reliability scheme would be to asynchronously write the page to disk at the same time that it is paged out over the network. (Of course only dirty pages need to be sent to disk.) When the page is once again needed, in the normal case we are able to request the page from the remote node and avoid the

| Comparison Factor | Change in Cost |
|---|---|
| Messages in network (normal case) | |
| whole pages | no change |
| short messages | no change |
| CPU local | send to disk |
| CPU remote | no change |
| Remote memory | no change |
| Page-out latency | wait for room in write buffer |
| Page-in latency | |
| best case | no change |
| if single failure | time to read from disk always available! |

Figure 3: This table summarizes the costs of asynchronously sending the page to disk over and above the cost of base network RAM.

penalty of reading from disk. Even if the remote node is unable to respond to the request, the data is still safely accessible from the local disk. In the common case, we would have the benefit of faster access time without the threat of process failure due to remote node or network failure.

The fact that the pages are safely stored on the local disk has some definite advantages. For example, if a network failure causes the client to be disconnected from the network, it will appear as if all of the client's page servers have failed at once. This is the only method which we considered which would be able to continue in this case. However, in fairness to the other methods, if the machine is completely cut off from the network, it is likely to have other problems to prevent it from continuing (such as access to remote files, etc.). Another benefit of having the pages stored safely on disk is that remote machines can feel free to simply dump the pages if needed- greatly simplifying the idle to busy transitions for remote nodes.

The main problem with this method is that fast pageout times will cause disk write buffers to fill eliminating truely asynchronous writes. We will still get the benefit of the fast reads over the network, but slowing to disk speed on our writes will certainly limit the potential performance gains. For example, in our implementation, we have measured the time to replace a 4K page to be between 2.1 ms and 9.4 ms. (See the bar graph in Section 3.2.) At these current speeds, disk writes can truly be asynchronous if the disk can sustain between 0.42 and 1.9 MB/sec of write bandwidth which is a reasonable expectation. However, as the times for page replacement over the network go down (as they need to in order to make network RAM more reasonable) the disk may no longer be able to sustain sufficient write bandwidth. For example, with a page replacement time of 0.25 ms, the disk could have to provide 15.6 MB/sec of write bandwidth to keep the writes asynchronous. A single disk is probably unable to provide this and equipping all nodes with an array of disks seems a bit extreme.

Another problem with the sending asynchronously to disk is that we still limit the size of our processes to the size of the local disk. Although we currently live with this limitation, network RAM provides the opportunity to free us from this restriction and it would be a shame to choose a method which prevents us from taking advantage of it.

### 4.3.3 Replication

The next two reliability schemes we have considered is locally controlled replication and remotely controlled replication. In remotely controlled replication, the remote node must report back to the original node the location of all the replicas. Otherwise, the remote node as the information holder would become a single point of failure. The second method has the advantage that the client need not oversee the transfer to other hosts, thus shifting responsibility away from the actively processing node.

In both of these schemes, when the original node needs its page back, it can retrieve it from any one of the remote nodes holding a copy. If a failed node is detected, a remaining copy holder should send a duplicate page to replacement idle node so that the system can return to a stable, fault tolerant state.

| Comparison Factor | Change in Cost | |
|---|---|---|
| | Local Control | Remote Control |
| Messages in network (normal case) | | |
| whole pages | $P_{out} * k$ | $P_{out} * k$ |
| short messages | $P_{in} * k$ | $P_{in} * k + 1$ |
| CPU local | transmission of duplicates<br>free duplicate pages | record location information sent by server<br>free duplicate pages |
| CPU remote | no change | decide where to send duplicates<br>send duplicates<br>send location information to client |
| Remote memory | $P_{out} * k$ pages | $P_{out} * k$ pages |
| Page-out latency | send $k$ extra pages | no change |
| Page-in latency | | |
| best case | no change | no change |
| if single failure | detect failure<br>send request to alternate node<br>wait for page to arrive | detect failure<br>send request to alternate node<br>wait for page to arrive |

Figure 4: This table summarizes the costs of the two replication schemes over and above the cost of base network RAM. We are assuming reliable message delivery to nodes that are alive. $P_{out}$ refers to the number of pages paged out. $P_{in}$ refers to the number of pages retrieved from a remote node. $k$ is the degree of replication.

The main problem with replication schemes is that they use extra memory. One solution to this problem is combining replication with migration of secondary copies to remote disk. In this scheme, we make a distinction between the primary copy and the secondary copy of a page. The primary copy remains in memory on the remote node while the secondary copy can be migrated to remote disk. If the primary copy holder fails, the secondary copy can be retrieved from disk and made the new primary. A replacement idle node can be selected to become the new secondary copy holder.

In the common case, the primary copy holder will be able to quickly satisfy the client's request for the page. If the primary copy holder fails, the client must request the secondary copy- paying not only network transfer cost but also the cost of reading from the remote disk. Therefore, since accesses to secondary copies are extremely expensive, they must be kept rare to reduce their impact.

#### 4.3.4 Parity

Parity schemes were the third basic type of reliability method that we considered. As with the replication schemes, it is easy to conceive of both locally and remotely controlled parity schemes. However, we chose to consider only remotely controlled parity schemes because they free the actively processing client from the time required by the parity calculations. In a simple parity scheme, the client sends one copy of its page to a remote node, the remote node splits the page into pieces and computes a parity piece. The remote node then distributes these pieces to other idle nodes and informs the original client of the location of the pieces. Again, this is necessary to prevent the node which holds the location information from becoming a single point of failure. The problem with this scheme is that the client must reassemble or possibly even reconstruct the page on every single page request.

Another parity scheme solves the reassembly problem. by using a parity group (a group of remote nodes which together handle the client's page requests). Unlike in the simple parity scheme, when the remote node receives the client's page, it does not split it into pieces. Instead, it places the entire page into a block. Parity is computed across the corresponding blocks on each member of the parity group. For each block, one node in the group acts as the parity node. Therefore, the remote node, which receives the page, chooses a free block in which to place the page, reads the corresponding parity block (locking it), recomputes a new parity block, stores its updated block and sends the new parity block back. Thus only two remote nodes are affected when a page is received. When a node requests its page to be returned, in the normal case, the remote node which has the complete page is able to service the request directly with no reassembly required. If that node is inaccessible, the node can contact the rest of its paging group, ask for their corresponding

| Comparison Factor | Change in Cost | |
|---|---|---|
| | Simple Parity | Parity Group |
| Messages in network (normal case) | | |
| whole pages | $-P_{in}$ (fewer!) | $2 * P_{out}$ |
| partial pages | $P_{out} * n + P_{in} * (n+1)$ | 0 |
| short messages | $1 + P_{in} * n$ | no change |
| CPU local | record location info from remote<br>request all the pieces<br>reassemble | no change |
| CPU remote | break pages into pieces<br>compute a parity piece<br>distribute pieces<br>notify client of location | request corresponding parity page<br>compute new parity page<br>send back the updated parity page<br>store the new page |
| Remote memory | $1/n * P_{out}$ pages | $1/n * P_{out}$ pages |
| Page-out latency | no change | no change |
| Page-in latency | | |
| best case | request all pieces<br>reassemble | no change |
| if single failure | reconstruct | contact members of parity group<br>reconstruct |

Figure 5: This table summarizes the costs of the two parity schemes over and above the cost of base network RAM. We are assuming reliable message delivery to nodes that are alive. $P_{out}$ refers to the number of pages paged out. $P_{in}$ refers to the number of pages retrieved from a remote node. $n$ is the number of pieces over which parity is computed.

blocks and reconstruct the missing page. This method does incur the overhead of managing the page group, but we believe that this is a reasonable job for the same service which manages the idle resources. This management would involve handling membership changes in the parity group. If a group member fails, the resource manager must instruct the remaining members to cooperate in reconstructing the lost blocks.

### 4.4 Reliability Conclusions

In light of the previous discussion, we believe that the replication/remote disk combination and the parity group scheme are the most promising. They both allow for fast access in the normal case and do not use much extra memory. As we discussed sending asynchronously to disk has some very nice properties, but its fundamental limitations make us hesitant to recommend it.

## 5 Summary

We have shown that some of the common assumptions underlying network RAM are inaccurate, limiting the general applicability of network RAM. We have demonstrated that a user-level implementation can achieve speeds of 2-5 times faster than disk speeds and has a performance comparible to device driver implementations. We have examined ways to provide reliable network RAM, which should now be tested in an implementation.

## References


[1] Anderson, T.E.; Culler, D.E.; Patterson, D.A et al.; “A Case for NOW (Networks of Workstations)”, November 1994.

[2] Asami, S., “Evaluating Network RAM via Paging”, NOW Retreat Presentation slides, June 1994.

[3] Asami, S., “Design Issues for Network RAM”, 294-5 Presentation Slides, November 1992.

[4] Brewer, E.; Kuszmaul B., "How to Get Good Performance from the CM-5 Data Network" Proceedings of the 1994 International Perallel Processing Symposium.

[5] Carter, L.; Ferrante, J.; Hummel S. "Hierarchical Tiling: A Framework for Multi-level parallelism and Locality."

[6] Chan, T. and Hsu, W., "PRIME: Paging to Remote Idle MEmory", SCAMD, November 1994.

[7] Iftode, L., Li, K., and Petersen, K., "Memory Servers for Multicomputers", 1993.

[8] Felten, E.W. and Zahorjan, J., "Issues in the Implementation of a Remote Memory Paging System", March 1991.

[9] Fox, A.; Gauthier, P.; Schuett, A., "Software Techniques for Improving MPP Bulk Transfer Performance", November 1994.

[10] Mainwaring, A., Wright, K., Yoshikawa, C., "The Design and Implementation of a Network RAM Prototype", November 1994.

[11] Mc Namee, D.; Armstrong, K. "Extending the Mach external pager interface to accomodate user-level page replaceemnt policies." USENIX Workshop Proceedings: Mach. 1990.

[12] Nguyen, G. and Oza N., "On the Use of Network DRAM in LAPACK Programs", November 1994.

[13] Nitzberg, B. and Lo, V., "Distributed Shared Memory: A Survey of Issues and Algorithms", August 1991.

[14] Shi, J.; Xi, J. "Paging across network: a study of queuing effect", November 1994.

# 6 Appendix

[3] and [14] describe other current work on network RAM. [9] examines methods to improve bulk transfer performance. In this appendix, we will be discussing the relationship of these works to our own.

We were unable to obtain a copy of Asami's final paper. His slides [3] discuss some previous work related to network RAM, including some references to his own simulations [2]. Our experience with network RAM has led us to believe that many of the asumptions underlying his original simulations were in error. He presented some of these same assumptions in his slide presentation. For example, the 10-20 $\mu$s latencies for network RAM are unachievable with current ATM switch and host interface overheads. Also, in our implementation, we saw achievable bandwidth of 2-4 MB/sec rather than 10-20 MB/sec. In his original simulation, he used these assumptions to conclude that network RAM could perform only slightly slower than all DRAM. As we have shown in our paper, VM overheads alone make this currently impossible, even disregarding network overheads.

Asami addresses many of the same reliability schemes that we do. However, we have noted several differences. In his parity scheme, he considers only client controlled parity which has the disadvantage that the client must handle the parity calculation for each page. Remotely controlled parity solves this problem. In addition, we considered a parity group scheme which does not require the client to reassemble or reconstuct each page.

We appreciated his references to other related work which we had not considered - inlcuding Schilit and Duchamp 91, Leff, You and Wolf 93, and Comer and Griffoen 90. He also considered other important policy questions such as load balancing. We had been assuming the existence a global resource manager such as Glunix, but he also considered random or cyclic distribution of pages.

Shi and Jiang [14] study the policy for allocating pages to different servers and the effects of having multiple clients in the network. Their work is very important because when an implemention of network RAM is finally deployed there will be multiple clients accessing multiple servers instead of the simpler single client testing that we did. Their work demonstrates a substantial flaw with our implementation. Currently it gets a list of servers at initialization, and cycles through the servers to send out pages. This usage pattern is similar to cyclic shifts as described in Brewer and Kuszmaul [4]. Cyclic use of servers results in slower participants falling farther and farther behind because of contention for a single server and a resulting piling-up effect.

Achieving reliability requires more machines because either duplicate pages, or parity pages need to be placed on extra machines. Therefore we need to consider methods of evenly distributing both page sending and page requests from the client across the servers to reduce contention at the network RAM servers. We believe that the most promising technique is to randomly choose a server among the possible choices. Shi and Jiang show that the random policy performs very well in comparison to both the greedy policy and a policy using global information. Furthermore, it is easier to implement the random policy than either of the other two choices.

We believe that the results that Shi and Jiang describe from their simulator are partially accurate. The steps in their simulator are similar to the steps taken in our implementation. However, they make much more optimistic assumptions about the performance of the machines and networks than we have, they assume overheads around 50$\mu$s, and network transit times of 300$\mu$s, as opposed to the overhead we measured of 200-1000$\mu$s and transit times of 1-2 ms. We believe that network RAM will not be very useful until overheads drop down to what Shi and Jiang use in their paper.

In our paper, we showed that without increases in actual network performance, network RAM will have limited applicability. Fox, Gauthier, and Schuett [9] show that on the Paragon, they achieve actual performance of 40 MB/s for 4k pages. This shows that we will be able to reduce the time on the network to 200 $\mu$s. Therefore, the VM system overhead becomes the bottleneck for network RAM performance.