The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Implementation of a Reliable Remote Memory Pager

Evangelos P. Markatos and George Dramitinos
Institute of Computer Science, FORTH, Crete

Implementation of a Reliable Remote Memory Pager

Evangelos P. Markatos and George Dramitinos*
Computer Architecture and VLSI Systems Group
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH), Crete

Abstract

Traditional operating systems use magnetic disks as paging devices, even though the cost of a disk transfer measured in processor cycles continues to increase.

In this paper we explore the use of remote main memory for paging. We describe the design, implementation and evaluation of a pager that uses main memory of remote workstations as a faster-than-disk paging device and provides reliability in case of single workstation failures. Our pager has been implemented as a block device driver linked to the DEC OSF/1 operating system, without any modifications to the kernel code. Using several test applications we measure the performance of remote memory paging over an Ethernet interconnection network and find it to be faster than traditional disk paging. We evaluate the performance of various reliability policies and prove their feasibility even over low bandwidth networks, like Ethernet.

We conclude that the benefits of reliable remote memory paging in workstation clusters are significant today and will probably increase in the near future.

1 Introduction

Applications like multimedia, windowing systems, scientific computations, engineering simulations, etc. running on workstation clusters (or networks of PCs) require an ever increasing amount of memory, usually more than any *single* workstation has available. To alleviate the memory shortage problem, an application could use the virtual memory paging provided by the operating system, and have some of its data in main memory and the rest on the disk. Unfortunately, as the disparity between processor and disk speeds becomes ever increasing, the cost of paging to a magnetic disk becomes unacceptable. Faster swap disks would only temporarily remedy the situation, because processor speeds are improving at a much higher rate than disk speeds [14]. Clearly, if paging is going to have reasonable overhead, a new paging device is needed. This device should provide high bandwidth and low latency. Fortunately, a device with these characteristics exists in most distributed systems and it is not used most of the time. It is the collective memory of all computers in the distributed system, hereafter called *remote memory*.

Remote memory provides high transfer rates which are mainly dictated by the interconnection network. Fortunately, most of the time remote main memory is unused and thus can be exploited by remote memory paging systems. To verify this claim, we profiled the unused memory of the workstations in our lab[1] for the duration of one week: 16 workstations with a total of 800 MBytes of main memory. Figure 1 plots the free memory as a function of the day of the week. We see that for significant periods of time more than 700 Mbytes are unused, especially during the nights, and the weekend. Although during business hours the amount of free memory falls, it is rarely lower than 400 Mbytes!

Architecture and software developments suggest that the use of remote memory for paging purposes is desirable, possible and efficient:

- **Memory to memory transfer rates**

*The authors are also with the University of Crete.

[1] We expect that more main memory will be available in places that have lighter load. Our workstations are heavily used running VERILOG simulations for most of the time.

between workstations have increased sharply in the last few years: Local Area Networks (like ATM and FDDI) have a high throughput and (usually) low latency. This increase in communication bandwidth implies a dramatic decrease in network transfer time for large messages (like operating system pages). On the other hand, the disk technology has *not* shown a similar increase in transfer rates. Moreover, disk accesses suffer from seek and rotation latency which is not expected to be reduced from advances in semiconductor technology.

- **Application's working sets have increased dramatically over the last few years:** Modern processors provide 64-bit address spaces, which make it possible for the processor to address an enormous amount of memory. Thus, software that takes advantage of a large address space is being developed: memory-mapped files and databases, sophisticated window interfaces, and multimedia, are a few examples that require an enormous amount of main memory.

- **Modern architectures provide low latency remote memory accesses:** Modern distributed systems provide a variety of efficient access operations to remote memories. The SCI-to-SBUS interface provides SPARC workstations with the ability to access the memories of other workstations in a network using simple load and store operations [23]. Similar ability is provided by Telegraphos [19], Hamlyn [5], Memory Channel [13], and SHRIMP [4]. Fast remote memory accesses have also been implemented in software using Active Messages [26, 2], programmed network interfaces [16], and trap-based remote invocation [25]. The ability to perform single remote memory accesses efficiently will enhance the performance of a remote memory paging policy, since the application can use them to access infrequently used pages.

In this paper we show that it is both possible and beneficial to use remote memory as a reliable paging device by building the systems software that transparently transfers operating system pages across workstation memories within a workstation cluster. We describe a pager built as a device driver of the DEC OSF/1 operating system. Our pager is completely portable to any system

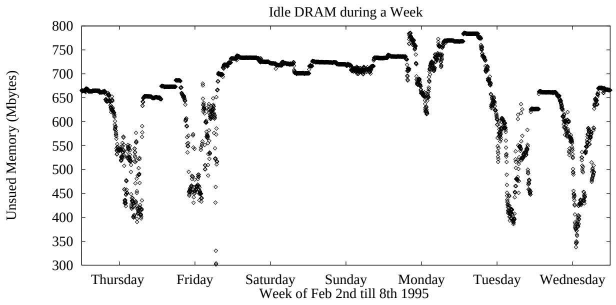

Figure 1: **Unused memory in a workstation cluster.** *The figure plots the idle memory during a typical week in the workstations of our lab: a total of 16 workstations with about 800 Mbytes of total memory. We see that memory usage was at each peak (and thus free memory was scarce) at noon and afternoon of working days. In all times though, more than 300 Mbytes of main memory were unused.*

that runs DEC OSF/1, because we didn't modify the operating system kernel. More important, by running real applications on top of our memory manager, we show that even on top of low bandwidth interconnection networks (like Ethernet), it is *efficient* to use remote memory as backing store. Our performance results suggest that paging to remote memory over Ethernet, rather than paging to a local disk of comparable bandwidth, results in up to 96% faster execution times for real applications. Moreover, we show that reliability and redundancy comes at no significant extra cost. We describe the implementation and evaluation of several reliability policies that keep some form of redundant information, which enables the application to recover its data in case a workstation in the distributed system crashes. Finally, we use extrapolation to find the performance of paging to remote memory over faster than Ethernet networks like FDDI and ATM. Our extrapolated results suggest that paging over a 100 Mbits/sec interconnection network, reduces paging overhead to less than 17% of the execution time of the application running over such a network. Faster networks will reduce this overhead even more.

The rest of the paper is organized as follows: Section 2 presents the design of a remote memory pager and the issues involved. Section 3 presents the implementation of the pager as a device driver. Section 4 presents our performance results which are very encouraging. Section 5 presents some aspects that we plan to explore as part of our future work. Section 6 presents related work. Finally, section 7 presents our conclusions.

2 The Design of a Remote Memory Pager

2.1 Selection of Workstations

All workstations, that participate in remote memory paging are registered in a common file. These workstations are known as remote memory servers, while the workstations that run applications that use remote memory for swapping are called clients. Depending on its workload, a workstation may act either as a server, or as a client.

All server workstations run a remote memory server that handles requests for pageins, pageouts, as well as for swap space allocation. When a client wants to swap out memory it picks the most promising server, asks for a number of page frames and starts sending requests to it. When a server runs out of memory, it denies further swap space allocation requests. When native memory-demanding processes start on a server workstation, part of the server's memory is swapped out to disk. Future requests will be serviced from the disk, and a note will be sent to the client, advising it to send no more pages to this server. On reception of this message, the client will try to find another server having enough free memory and migrate the pages that were stored by the loaded server to the new one. If no server having enough free memory can be found the client's local disk will be used to house these pages. Whenever the client's local disk is used to store some of it's paged out pages, the client periodically checks the memory load of all possible remote memory servers. If a server having enough free memory is found, some of the pages stored at the local disk are replicated to this server. Future requests concerning these pages will be served by the remote memory server rather than the disk.

2.2 Reliability

In a distributed system, a workstation may crash at any time. If the crashed workstation acts as a server, it will lose the pages of several clients. Clearly, it is not acceptable for applications running on the client workstation to crash due to remote server crash. Instead, we would like to be able to recover their pages. Otherwise a remote server crash will cause a client crash as well, since all programs that have some of their pages swapped out (including programs like init and system daemons) will not be able to continue execution.

There are many types of crashes. First of all there may be machine crashes due to a black out. This situation is not addressed by this paper, since most computer buildings are equipped with UPSs. Another cause of failure may be a network problem (e.g. network partitioning due to a bridge failure). In this case, the client can not retrieve its pages from the servers. As a result it remains blocked waiting for the network to recover. The most frequent cause of crash is a software crash, followed by a hardware error. To avoid loss of data due to a server crash, some systems write all network memory pages to the disk as well ([1, 11]). Instead we implement a *reliable* remote memory paging system that is able to reconstruct the lost pages.

To provide this level of reliability, some form of redundancy must be used. The main issues that must be taken into account regarding the form of redundancy used are:

- The runtime overhead introduced must be minimal since it is a cost paid even when no server crashes.
- The memory overhead introduced must be as low as possible because the memory reserved for reliability could be used in order to store memory pages of other workstations.
- The crash recovery overhead, that is the time it takes to recover from a server crash. This overhead is not as important as the previous two, since it is affordable to devote a few more seconds whenever a server crashes, which happens rather rarely.

We explore three different policies: mirroring, parity, and parity logging.

Mirroring: The simplest form of redundancy is *mirroring*. In mirroring, there exist two copies of each page. When the client swaps out a page, the page is sent to two different servers. Even when one of the servers crashes, the application is able to complete its execution, because all pages of the crashed server exist on the mirror servers. Obviously the crash recovery overhead, in case of mirroring, is minimal. However, the runtime overhead is rather high, since each pageout requires two page transfers. To make matters worse, mirroring wastes half of the remote memory used.

Parity: To reduce the main memory waste caused by mirroring, we can use parity-based redundancy schemes much like the ones used in RAIDS [6]. Suppose, for example, that we have S servers, each having P pages. Page (i,j) is the j_{th} page that resides on server i. Assume, that we have P parity pages, where parity page j is formed by taking the XOR of all the j_{th} pages in all servers. We say that all these j_{th} pages belong to the same parity group. If a server crashes, all its pages can be restored by XORing all pages within each parity group.

When the client swaps out a page it has to update the parity to reflect the change. This update is done in two steps:

1. The client sends the swapped out page to the server, which computes the XOR of the old and the new page.
2. The server sends the just computed XOR to the parity server, which XORs it with the old parity, forming the new parity.

Unfortunately, this method involves two page transfers: one from client to server, and one from server to parity. Moreover, the client should not discard the page just swapped out, because the server may crash before the new parity is computed, thus, making it impossible to restore the swapped out page. This parity method increases the amount of remote main memory only by a factor of $(1+1/S)$ minimizing the memory overhead, but it still imposes a significant runtime overhead.

Parity Logging: To avoid the additional page transfers induced by the basic parity method, we have developed a parity logging scheme. The key idea is that a given page need not be bound to a particular server or parity group. Instead, every time a page is paged out, a new server and a new parity group may be used to host the page.

Suppose the client uses S servers. Each paged out page is XORed with a page size buffer maintained by the client (which is initially filled with zeros) and then is transfered to a server following a round robin policy. Whenever S pages have been transfered, the buffer is also transfered to a parity server. Using this technique, the runtime overhead is minimal, since for each paged out page $1+1/S$ page transfers are required. When a server crashes, all of its pages can be restored by XORing the pages in their group with the corresponding parity page. [2]

Every time a page is repaged out, it is marked in the old parity group containing it as inactive. [3] When all the pages of a parity group are marked as inactive, all the memory server pages and the corresponding parity page can be reused. It is obvious that each memory server must have some extra overflow memory to support parity logging since many versions of a given page may be present simultaneously at the servers' memory. Also, due to this situation, it is possible that some server runs out of memory. In this case, one has to perform garbage collection freeing parity sets by combining their active pages to new ones. In our experiments, 4 servers were used devoting 10% more memory to support parity logging and we never had to perform garbage collection.

3 Implementation

The proposed system has been built and is in everyday use. It consists of a client issuing paging requests and servers satisfying these requests. It is also able to use the local disk for paging and may support either mirroring or parity logging. The client side has been linked with the DEC OSF/1 kernel of a DEC-Alpha 3000 model 300 with 32 MB main memory as a block device driver that handles all pagein and pageout requests. In order to service these requests, it may forward them either to user level servers running on other hosts, or to the local disk. The DEC OSF/1 kernel is not even aware that we use remote main memory instead of magnetic disk as a paging device. It just performs ordinary paging activities using a block device. This design minimizes the modifications needed in order to port the system to another operating system and avoids modifications to the operating system kernel.

3.1 The Remote Memory Pager

Normally the Remote Memory Pager (RMP for short) is a client which forwards the paging requests to a remote server using sockets over an Ethernet. The RMP connects to the remote memory servers using sockets over TCP/IP. One dedicated paging daemon issues pagein and pageout requests to the server and receives the data sent by them. When mirroring is used, it is responsible for selecting two servers for each paged out page

[2] Note that since the parity page is computed by the client, it is not necessary to wait for acknowledgments from the servers before transfering the parity page in order to be able to recover from a single server crash.

[3] However, the old version of the page is *not* deleted from the server's memory, because if it were, the *old* parity page should be updated, leading to more page transfers.

and transfer the data to them. When parity logging is used, it maintains all the data structures related to page and parity group management and computes the parity pages. Security is ensured by allowing access to our device only to the superuser and by using privileged ports for the communication among the client and the servers.

RMP is also capable of forwarding the requests to the local disk using either a specified partition or a file. In the former case, it invokes a routine that places the request in the disk queue. In the later case it issues a read or write operation through the VFS layer routines. When no server can be found in order to satisfy the client's requests, paging to local disk is used.

Although the current implementation runs on top of a low bandwidth 10 Mbps Ethernet, remote paging is up to 2 times faster than using a local disk of the same bandwidth. It takes about 8.4 ms to transfer an 8KB page through the network, while transferring a page to/from the local disk takes about 17 ms. Faster networks such as ATM, or FDDI should offer even more promising performance, especially when faster communication protocols are used [26].

3.2 The Remote Memory Server

The server is a user level program listening to a socket and accepting connections from clients. Each client is served by a new instance of the server which uses portion of the local workstation's main memory to store the client's pages. When the client requests a pagein, the server transfers the requested page(s) over the socket. When the client requests a pageout, the server reads the incoming pages from the socket, and stores them in its main memory. The server is also responsible for swap space allocation and for providing periodically information to the client concerning the memory load of its host. A parity server is by no means different than a memory server. It just performs pageins and pageouts responding to client requests without knowing whether it stores memory pages or parity pages.

4 Performance Results

To evaluate the performance of our remote memory pager, and compare it to traditional disk paging, we conducted a series of performance measurements using a number of representative applications that require a large amount of memory. Our applications include `GAUSS`, a gaussian elimination, `QSORT`, a quicksort program, `FFT`, a Fast-Fourier Transform, `MVEC`, a matrix-vector multiplication, `FILTER`, a two pass separable image sharpening filter described in [20] and `CC`, a kernel build after modifying the code of our device driver. All applications were executed on the DEC-Alpha 3000 model 300, and were compiled with the standard C compiler with the optimization enabled. All workstations that contributed their main memory for paging purposes were DEC-Alpha 3000 model 300, connected via a standard 10Mbits/sec Ethernet. In all experiments the amount of idle memory was larger than the amount of memory needed for paging and was equally distributed among all workstations. The local disk that was used for paging is a DEC RZ55, providing 10Mbits/sec bandwidth, and average seek time of 16 msec.

4.1 Performance of Remote Memory Paging Over the Ethernet

In our first experiment we evaluate four methods for paging:

- `NO_RELIABILITY`, which uses only main memory of other workstations as a paging device. In this experiment two remote memory servers were used. The measurements were done on an (almost) idle Ethernet to ensure repeatability.
- `PARITY_LOGGING`, which uses 4 servers plus a parity server, all devoting 10% overflow memory.
- `MIRRORING`, which uses one primary memory server and one mirror memory server.
- `DISK`, which uses the local DEC RZ55 disk for paging. In this case the page transfer requests go directly from the DEC OSF/1 kernel to the disk driver without the intervention of our pager.

The completion time of the applications is plotted in figure 2. We see that in all cases the use of remote memory results in significant performance improvements. For example, for the `GAUSS` application, the `NO_RELIABILITY` results in 96% faster execution time than `DISK`. Even for the `MVEC` application which performed very little paging, `NO_RELIABILITY` results in 32% faster execution time. The reliability methods induce some runtime overhead as expected but still perform

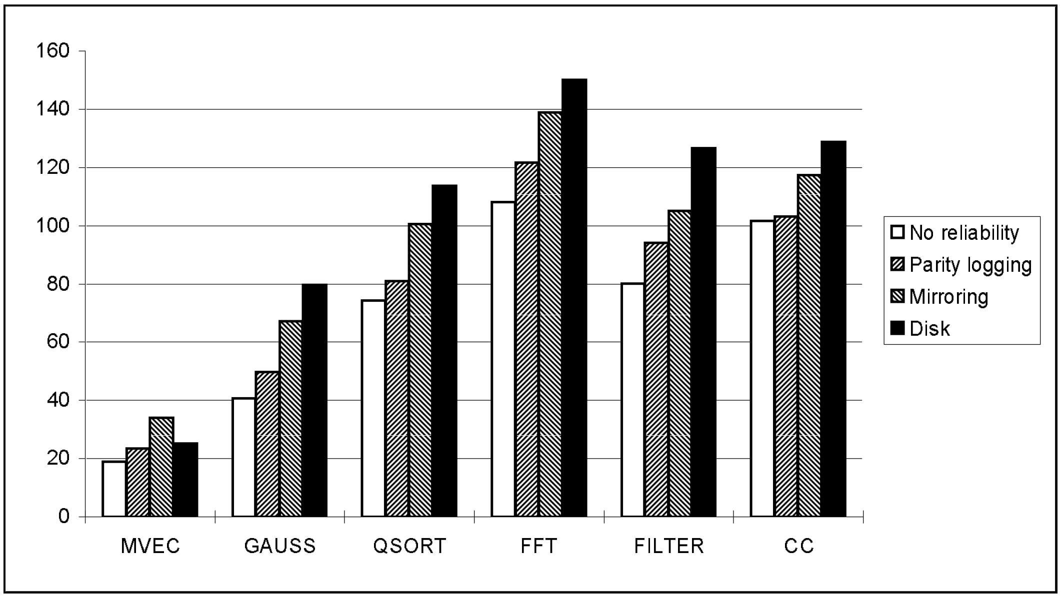

Figure 2: **Performance of applications using either the disk, or the remote memory as paging device.** *We see that for all applications, the use of remote memory results in significantly faster execution. All applications were run on a DEC-Alpha 3000 model 300 workstation. The input sizes for* `QSORT` *was 3000 records, for* `GAUSS`*, a 1700×1700 matrix, for MVEC, a 2100×2100 matrix, for* `FFT` *an array with 700 K elements, for* `FILTER` *a 12 MB image, and the whole DEC OSF/1 V3.2 kernel for* `CC`.

much better than `DISK`. `PARITY_LOGGING` results in 40.4% faster execution time for `QSORT` and in 59.86% faster time for `GAUSS`. `MIRRORING` also performs better than `DISK` for all applications except `MVEC`, since `MVEC` performs many pageouts and almost no pageins.

In order to evaluate the use of remote memory for a more realistic application, we measured the completion time of a kernel build after modifying the code of our device driver. As can be seen in figure 2, `NO_RELIABILITY` performs 26.56% better than disk, `PARITY_LOGGING` performs 24.65% better and `MIRRORING` performs just 9.7% better. We see that `PARITY_LOGGING` performs very close to `NO_RELIABILITY`. As the number of the remote memory servers used increases, the difference in performance between `NO_RELIABILITY` and `PARITY_LOGGING` becomes lower.

Our performance results suggest that paging to remote memory over an Ethernet interconnection network is simply faster than paging to the disk. Even though both the disk and the Ethernet have similar data transfer rates, remote memory does not suffer from seek and rotational latency as `DISK` does.

Our experimental results verify that even when the network data transfer rate is as low as the disk transfer rate, the performance of remote memory is significantly higher than the performance of disk. Moreover the performance requirements of reliability are surprisingly small. Since architecture trends suggest that modern high speed networks provide much higher data transfer rates than modern disks, the performance improvements of remote memory over disk are bound to increase.

4.2 Scaling the Input

To understand the impact of the working set size on the paging policy, we measure the execution time of one of our applications (`FFT`), as a function of its input size. The completion time of `FFT` both under `PARITY_LOGGING` and under `DISK` is plotted in figure 3. We see that as soon as the working set size exceeds 18 MBytes, the paging starts, and the completion time of the application rises sharply. Most users would not be willing to tolerate such a high overhead in order to run an application that does not fit in main memory. Fortunately, remote memory reduces this overhead substantially.

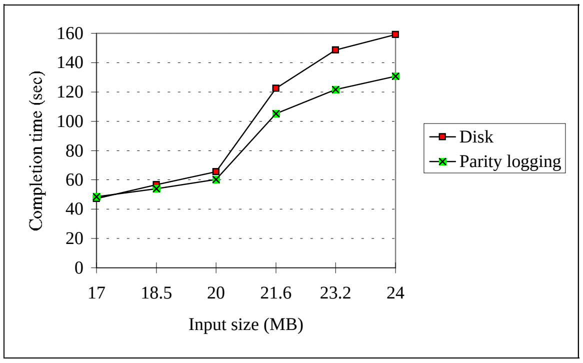

Figure 3: **Performance of FFT as a function of input size when either the disk, or remote memory are used as backing store.**

4.3 Scaling the Network Bandwidth

Although figure 3 suggests that the performance of remote memory (parity logging) is significantly better than the performance of disk, the completion time of an application even under remote memory may be unacceptably high. Hopefully, the performance of remote memory will be improved as soon as the Ethernet interconnection network is substituted by a faster one (e.g. FDDI, ATM, FCS, etc.). To evaluate the performance of the applications on top of faster networks we make detailed performance measurements that separate the completion time of the application into the following factors: (i) user time ($utime$), (ii) system time ($systime$) (iii) initialization time ($inittime$) (iv) page transfer time ($ptime$). Using the provided `time` command we measure the $utime$, $systime$, and elapsed time ($etime$) for each application. Subtracting the $utime$ and $systime$ from the $etime$ for instances of the applications that perform no paging we calculate the $inittime$, that is the time it takes the operating system to load and start executing the application. The $ptime$ consists of the protocol processing time ($pptime$) and the bandwidth dependent blocking time ($btime$). We measured the $pptime$ and found it to be equal to 1.6 ms per page for TCP/IP. We calculate the $btime$ using the formula : $btime = (etime - utime - systime - inittime - no\_of\_page\_transfers * pptime)$. Assuming that a network with X times higher bandwidth will decrease $btime$ by a factor of X, we can predict the etime of the application over this high bandwidth network. Thus, the formula used is : $Expected\_elapsed\_time = utime + systime + inittime + number\_of\_page\_transfers * pptime + btime/X$.

We made all these measurements on our FFT application, and predict its performance on a system with an interconnection network which provides ten times more bandwidth than the Ethernet. We also predict its completion time on a system that has enough memory to hold all the working set of the application (`ALL_MEMORY`) by adding the utime, systime and inittime. The predicted execution times, along with the measured execution times of `DISK` and `PARITY_LOGGING` are plotted in figure 4. We see that `ETHERNET*10` performs very close to `ALL_MEMORY`, and significantly better than both `ETHERNET` and `DISK`.

To understand the results shown in figure 4, we analyze the execution time of FFT with 24MBytes of input when `PARITY_LOGGING` is used. The measured elapsed time is 130.76 seconds, consisting of 66.138 sec of useful user time, 3.133 sec of system time, 0.21 sec of initialization time and 61.279 sec of page transfer time. During the same run, the application suffered 2718 pageouts and 2055 pageins. Since 4 servers were used plus a parity server the number of page transfers was equal to $3397 + 2055 = 5452$. Thus the protocol overhead was equal to $5452 * 0.0016$, or about 8.723 sec. The bandwidth dependent blocking time was equal to $61.279 - 8.723$, or about 52.556 sec. Using a ten times faster interconnection network, the bandwidth dependent waiting time will be reduced to 5.255 sec. Thus, the total completion time of FFT would be $66.138+3.133+0.21+8.723+5.255$ sec, or 83.459 sec, divided as follows: 79.246% in user time, 3.754% in system time, 0.252% in initialization time and 16.748% in page transfer time. We see that a 100 Mbit/sec interconnection network reduces the total paging overhead to less than 17% of the total application execution time. We believe that most users would be willing to pay such an overhead in order to run an application that does not fit in main memory. After all, the only other option they have is to suffer from disk thrashing.

4.4 The Latency of Remote Memory Paging

As explained previously, the paging latency for `FFT` with input size equal to 24 MB is 61.279 sec, or 11.24 ms per page transfer. From these, 1.6 ms were spent during protocol processing and 9.64 ms were spent transferring each page on the Ethernet.

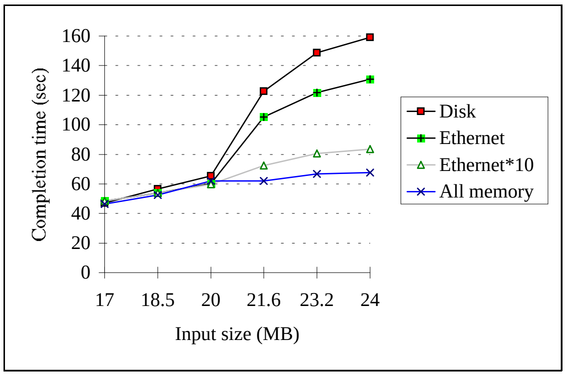

Figure 4: **Performance of FFT for various Architecture Alternatives.** `DISK` *is the measured completion time when paging to a local disk.* `ETHERNET` *is the measured completion time of parity logging to remote memory on top of the Ethernet.* `ETHERNET*10` *is the predicted completion time when using remote memory as a paging device, on top of a network that provides ten times more bandwidth than the Ethernet interconnection network.* `ALL_MEMORY` *is the predicted completion time of FFT when we use the same workstation but with enough memory to hold its entire working set.*

Previous measurements have reported that a 4 KByte page takes about 45 ms over an Ethernet for each pagein [22]. Of those 45 ms, 19 ms were spent on TCP overhead, 4 ms were spent on Mach IPC overhead, 7.2 ms were spend on the Ethernet, and the rest were spent on the computer's I/O bus. The total software latency of our implementation, is only 1.6 ms. The reason for this significant difference in performance is threefold:

- The I/O bus of the DEC-Alpha 3000 model 300 we use is significantly faster and does not pose a problem in performance.
- We use a DEC-Alpha processor, which is 3-4 times faster than the 386 processor used in [22].
- Finally, our pager is implemented as a block device driver, while in [22] it was implemented as a user-level memory manager on top of Mach. Although user-level memory management gives increasing flexibility it induces large overhead.

In general, although our approach may have less flexibility than a full-fledged user-level pager, it has much better performance. Moreover, our device-driver implementation provides better performance than traditional (local) disk paging, while user-level implementations have not reported performance results to support similar claims [22].

4.5 Using Busy Workstations as Servers

In all our experiments so far, the remote memory servers run on idle workstations. However, workstations that are able to donate their memory for paging purposes may not be completely idle, as they may run interactive applications. Thus, we would like to investigate how our performance figures change when a non-idle workstation is used as a memory server. So, we conducted the following experiment:

> On each server workstation we started an X-window environment, and an instance of the vi editor which was continuously used for editing. Then, we run the applications of the experiment in figure 2. The same inputs, and the same clients were used. The only difference was that the remote memory server processes were run on busy instead of idle workstations.[4]

We were surprised to see that for the `FFT`, `GAUSS`, and `MVEC` applications, their completion times were within 1 sec of their completion times when the server ran on an idle workstation. Only `QSORT` suffered a 7% overhead in its completion time: probably the kernel swapped out some of the remote memory server's pages on the disk. However, in order to find out how the completion time of our applications changes with server load, we ran `FFT` and `QSORT` under `NO_RELIABILITY` using two remote memory servers. On one of them a cpu bound program (performing a "while(1);" loop) was initiated. To our surprise, even then the completion times of our applications were within 7% of their completion times when the server ran on an idle workstation.

Our performance figures suggest that most of the time the remote memory servers were able to satisfy the client's requests immediately, even on busy workstations. Our results agree with the

[4] One could argue that an X-window environment and an editor, induce almost no load on the workstation. But, this is exactly the point: a typical workstation, even when it is used, it is very lightly loaded. The rest of the workstations that are heavily loaded do not donate their main memory for remote paging.

measurements in figure 1 which report that a significant portion of all workstation's memory is unused even at business hours, thus no overhead is expected to be seen when some other server process uses the extra pages.

In the same course of experiments, we would like to see what is the overhead that remote paging induces on the server workstation. Thus, we measured the CPU utilization of the (otherwise idle) remote memory server for all our experiments, and found it always to be less than 15%. Thus, the computational overhead imposed on the remote workstation is so low that will not be noticed by the workstation's owner.

4.6 Using Remote Memory Paging over a Loaded Ethernet

All the experiments presented so far were done over an almost idle Ethernet to ensure repeatability of our results. However, we would like to find out how the performance of remote memory paging is affected by the load of the network. That is why we repeated our experiments using an already loaded Ethernet. The results showed a performance degradation even when the Ethernet was lightly loaded. This situation is by no means surprising since the paging itself uses all the bandwidth it can get. Adding more sources of traffic leads to an enourmous demand for bandwidth causing repeated collisions and lowering the effective bandwidth of the network, leading to throughput collapse.

Fortunately, this inefficiency is not inherent to remote memory paging but rather to the CSMA/CD protocol employed by the Ethernet [24]. This means that it is still beneficial to use remote memory paging over networks that employ other technologies (e.g. token ring), as long as they are able to provide to remote memory paging an effective bandwidth of 10 or more Mbps.

4.7 Using the Local Disk to Increase Reliability

In our work we use remote main memory to store redundant information that will be used to recover from workstation crashes. Another approach would be to store all remote pages to the local disk as well [11], effectively treating remote memory as a write-through cache of the disk. We will now compare the two approaches to find out the circumstances under which the one approach is preferable to the other.

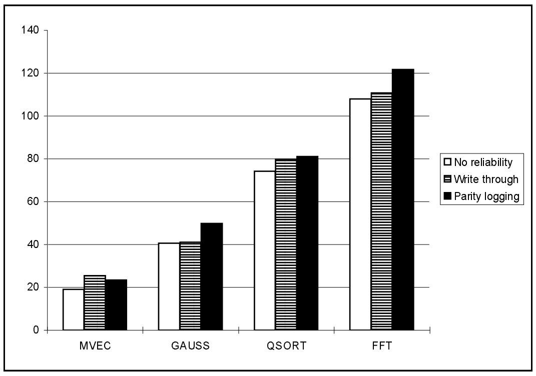

Figure 5: **Performance of parity logging and write through for various applications.** *The input sizes for* `QSORT` *was 3000 records, for* `GAUSS`*, a 1700×1700 matrix, for MVEC a 2100×2100 matrix, and for* `FFT` *an array with 700 K elements.*

Both approaches use the remote memory to satisfy the read requests. This means that both approaches perform reads at the same speed and avoid disk head movements due to reads, thus outperforming the local disk. *Parity logging* transfers $1 + 1/N$ pages per paged out page, due to the parity computation (in our experiments N was equal to 4). On the other hand, *write through* transfers each paged out page both to disk to the remote memory. These two page transfers are executed in parallel. This means that the choice of the right approach depends on the effective bandwidth offered by the disk and the network. If the network bandwidth is much higher than the disk bandwidth, then the disk will be the bottleneck for *write through* making it an unwise choice. If however the effective bandwidth offered by the disk is comparable to the bandwidth offered by the network and the system can overlap disk transfers with network transfers then it is unclear which method is best to use. In our experimental environment the disk and network bandwidth are both equal to 10 Mbps. When *write through* is used the efective disk bandwidth is close to 10 Mbps, since there are no head movements for reads and writes are performed in large chunks. In this environment *write through* performs better than *parity logging* and slightly worse than our no-reliability implementation in most cases, as shown in figure 5 . However, when a modern high bandwidth network is used, *parity logging* will probably be the best approach, since *write through* will eventually be limited by the local disk bandwidth.

5 Discussion - Future work

Our implementation suggests that it is possible to build a reliable *efficient* remote memory pager without making any modifications to the operating system kernel. Although our system contains all necessary mechanisms to support remote memory paging, there are a few more issues concerning the overall *policy* that deserve further investigation. Some of these issues are discussed below.

Network load: Although remote paging is faster than using the local disk, sometimes the network traffic may be so high that the bandwidth used by RMP will be limited. In this case the cost of using the network, especially in the case of old low bandwidth networks like Ethernet, may become higher than the cost of using the local disk. Such a situation could be handled by the RMP by measuring the time it takes to satisfy a request and using a threshold to determine whether it should continue to use the network to route pageout requests or it would be better to switch to the local disk.

Heterogeneous networks: The current implementation assumes a network of workstations that all have the same order of magnitude of physical memory and are interconnected by a local area network. It would be interesting to explore the requirements that heterogeneous networks pose to the design of the remote pager. For example, on a wider area network the time it takes to transfer a page may not be identical for each server. In this case there may be more than three levels in the memory hierarchy (local memory, remote memory, disk), depending on the variance of the cost of communication among the hosts of the network. Connecting machines that have an enormous amount of memory (e.g. a supercomputer) to a network of workstations also poses some problems. When the supercomputer memory is idle, it may not always be easy to find enough free remote workstation memory in order to be able to use reliability policies. In this case, a no reliability policy can be used, since all remote memory will be provided by a single host (the supercomputer).

6 Related Work

Several research groups have studied the issues in using remote memory in a workstation cluster to improve paging performance [2, 12, 7, 15, 22, 3].

Felten and Zahorjian [12] have implemented a remote paging system on top of a traditional Ethernet based system, and presented an analytical model to predict its performance. Their performance results, although preliminary, are encouraging towards using remote memory paging systems. Schilit and Duchamp [22] have implemented a remote memory paging system on top of Mach 2.5 for portable computers. Their remote memory paging system has performance similar to local disk paging. The cost of a single remote memory pagein over an Ethernet, they quote, is about 45 ms for a 4Kbyte page, which is rather high. According to their measurements, a significant percentage of this time (close to 16 ms) is spend executing Mach IPC and TCP code. Comer and Griffoen [7] have implemented and compared remote memory paging vs. remote disk paging, over NFS, on an environment with diskless workstations. Their results suggest that remote memory paging can be 20% to 100% faster than remote disk paging, depending on the disk access pattern. Anderson *et. al.* have proposed the use of network memory as backing store [2]. Their simulation results suggest that using remote memory over a 155Mbits/s ATM network "is 5 to 10 times faster than thrashing to disk" [2]. In their subsequent work [18], they outline the implementation of a remote memory pager on top of an ATM based network.

Our work differs from previous approaches to remote memory paging in the following aspects: (i) we use a variety of real applications to evaluate and demonstrate the feasibility of remote memory paging, and (ii) we explore the issues in building a *reliable* remote memory system that is resilient to individual workstation failures. Previous approaches either ignore workstation failures, or write dirty pages both to the disk and the remote memory, limiting their performance by the available disk throughput.

Recently, research groups start to explore the issue of using remote memory to improve file system performance [11, 1, 8]. Feeley *et. al.* have implemented a global memory management system in a workstation cluster, using the idle memory in the cluster to store clean pages of memory loaded workstations [11]. Anderson *et. al.* have implemented xFS, a serverless network file system [1, 9]. Both network memory systems have been incorporated inside the kernel of existing operating systems and their performance has been demonstrated. Although improvements in file system performance may ultimately lead to paging performance improvements, solutions developed for file systems may be cumbersome, or too general for remote memory paging systems: (i) in file systems, client processes may share file data, leading to *cooperative* remote memory management policies. In paging instead, clients *never* share their swap spaces. Thus, policies developed to optimize a client-server approach to file I/O, and facilitate cooperation among client processes that share data, do not necessarily apply to a paging system where no single paging server is used, and no sharing (of swap spaces) between client processes takes place. Finally, we use the network memory for storing both clean and dirty pages using our novel parity-based approach. Thus, page out (write) operations can be acknowledged at the speed of remote memory, while in [11, 1] page out operations are acknowledged at the speed of disk.

Although the area of reliability in network memory systems is new, it shares several of the ideas developed for other areas of reliable memory management. For example, parity based methods have been extensively used for Redundant Arrays of Inexpensive Disks (RAIDs) [6].

Log based methods have been used for Log based file systems, that send all updates to a file to be written in sequential blocks of the disk [21]. Thus, the head of the disk does not make random seek movements, and the effective data transfer rate of the disk increases. Log based file systems, alike our `LOGGING` methods, create a fragmented space that needs to be cleaned. Although the general ideas may be similar, there are substantial differences between a log based file system and the log based reliable network memory we propose. For example, (i) Fragmentation in log based file systems occurs in large chunks (several Mbytes), while fragmentation in log based reliable network memory occurs in small parity groups, and (ii) Log based reliable network memory systems may use parity groups as soon as they are emptied, but log based file systems may not used emptied disk blocks, because this would require a head movement. (iii) Cleaning in log based file system is much more infrequent than it is in network memory, thus it must be made more efficient, and (iv) the objective of log based network memory systems is to reduce page transfers, while the objective of log based file systems is to reduce disk head movements. For the above reasons, methods developed for log based file systems do not necessarily apply "as is" to network memory systems.

Our work bears some similarity with distributed shared memory systems [17, 10] in that both approaches use remote memory to store an application's data. Our main difference is that we focus on *sequential* applications where pages are not (or rarely) shared, while distributed-shared-memory projects deal with parallel applications, where the main focus is to reduce the cost of page sharing.

7 Conclusions

In this paper we explore the use of remote main memory for paging. We describe our prototype implementation of a remote memory pager implemented on top of the DEC OSF/1 operating system as a device driver. No modifications were made to the kernel of the (monolithic) DEC OSF/1 operating system. We run several applications that use our pager on top of a DEC-Alpha-based workstation cluster to measure the performance of the system. The contributions of this paper are:

- We describe how to build a reliable remote memory paging system; we propose a novel parity-based policy that is resilient to single workstation failures.
- We show that reliable paging to remote memory results in substantial performance improvements over local disk paging.

Based on our implementation and our performance results we conclude:

- *Paging to remote memory results in significant performance improvement over paging to disk.* Applications that use our pager even when running on top of *traditional* Ethernet technology show performance improvements of up to 96% (see figure 2). Extrapolating from our results, we show that on top of a faster interconnection network even higher performance improvements are realizable!
- *Paging to remote memory is an inexpensive way to let applications use more main memory than a single workstation provides.* Remote memory paging provides good performance with almost no extra hardware support. The only way for magnetic disks to provide comparable performance is to use expensive disk arrays.

- *Reliability in remote memory paging comes at low cost.* Parity logging based paging provides reliability at low runtime and memory overhead, performs very close to `NO_RELIABILITY` and much faster than disk paging.
- *The benefits of paging to remote memory will only increase with time.* Current architecture trends suggest that the gap between processor and disk speed continues to widen. Disks are not expected to provide the bandwidth needed by paging unless a breakthrough in disk technology occurs. On the other hand, interconnection network bandwidth keeps increasing at a much higher rate than (single) disk bandwidth, thereby increasing the performance benefits of paging to remote memory.

Based on our performance measurements we believe that remote memory paging is a cost-effective and performance-effective way to execute memory-limited applications on a network of workstations.

Acknowledgments

This work was developed in the ESPRIT/HPCN project "SHIPS", and will form a test application for the ACTS project "ASICCOM", funded by the European Union (DG III and DG XIII). We deeply appreciate this financial support, without which this work would have not existed.

We would like to thank Manolis Katevenis, Sotiris Ioannidis and Kosmas Papachristos for useful comments in earlier drafts of this paper. Michael J. Feeley and Henry M. Levy pointed out useful references. Finally, we deeply appreciate the thoughtful comments of the anonymous referees.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proc. 15-th Symposium on Operating Systems Principles*, December 1995.

[2] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[3] G. Bernard and S. Hamma. Remote Memory Paging in Networks of Workstations. In *Proceedings of the SUUG International Conference on Open Systems: Solutions for Open Word*, April 1994.

[4] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the Twenty-First Int. Symposium on Computer Architecture*, pages 142–153, Chicago, IL, April 1994.

[5] Greg Buzzard, David Jacobson, Scott Marovich, and John Wilkes. Hamlyn: a high-performance network interface with sender-based memory management. In *Proceedings of the Hot Interconnects III Symposium*, August 1995.

[6] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[7] D. Comer and J. Griffoen. A new design for Distributed Systems: the Remote Memory Model. In *Proceedings of the USENIX Summer Conference*, pages 127–135, 1990.

[8] T. Cortes, S. Girona, and J. Labarta. PACA: A Distributed File System Cache for Parallel MAchines. Performance under Unix-like workload. Technical Report UPC-DAC-1995-20, Departament d'Arquitectura de computadors, Universitat Politecnica de Catalunya (UPC), June 15 1995.

[9] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson. Cooperative Cahing: Using Remote Client Memory to Improve File System Performance. In *First USENIX Symposium on Operating System Design and Implementation*, pages 267–280, 1994.

[10] G. Delp. *The Architecture and implementation of Memnet: A High-Speed Shared Memory Computer Communication Network*. PhD thesis, University of Delaware, 1988.

[11] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proc. 15-th Symposium on Operating Systems Principles*, December 1995.

[12] E. W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report TR 91-03-09, University of Washington, November 1991.

[13] R. Gillet. Memory Channel. In *Proceedings of the Hot Interconnects III Symposium*, August 1995.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[15] L. Iftode, K. Li, and K. Petersen. Memory Servers for Multicomputers. In *Proceedings of COMPCON 93*, 1993.

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The FLASH Multiprocessor. In *Proc. 21-th International Symposium on Comp. Arch.*, pages 302–313, Chicago, IL, April 1994.

[17] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[18] A. Mainwaring, C. Yoshikawa, and K. Wright. NOW White Paper: Network RAM Prototype, 1994. http://now.cs.berkeley.edu/Nram/network-ram.html.

[19] Evangelos P. Markatos and Manolis G.H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture, (HPCA)*, San Jose, CA, USA, February 1996.

[20] Gary Newman. Organizing Arrays for Paged Memory Systems. *Communications of the ACM*, 38(7):93–110, July 1995.

[21] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proc. 13-th Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[22] B.N. Schilit and D. Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report CUCS-004-91, University of Columbia, 1991.

[23] Dolphin Interconnect Solutions. DIS301 SBus-to-SCI Adapter User's Guide.

[24] A. S. Tanenbaum. *Computer Networks*, chapter 3, page 128. Prentice Hall International, 1989.

[25] C.A. Thekkath, H.M. Levy, and E.D. Lazowska. Efficient Support for Multicomputing on ATM Networks. Technical Report 93-04-03, Department of Computer Science and Engineering, University of Washington, April 12 1993.

[26] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19-th International Symposium on Comp. Arch.*, pages 256–266, Gold Coast, Australia, May 1992.

Biographical information

Evangelos P. Markatos is an Assistant professor at ICS-FORTH and at the University of Crete. He received his diploma in Computer Engineering from the University of Patras in 1988, and the MS and Ph.D. degrees from the University of Rochester in 1990 and 1993 respectively. His interestes include parallel and distributed systems, operating systems and computer architecture.

George Dramitinos is a graduate student in Computer Science at the University of Crete, where he received a B.Sc. degree. He has worked at A.C.R.I. in Lyon, France, participating in the design and implementation of an OSF/1 based operating system for the company's supercomputer. He joined ICS-FORTH in 1993. His interests include operating systems, parallel and distributed programming and computer architecture.

The authors can be contacted at {markatos, dramit}@ics.forth.gr. or at their postal address at `Institute of Computer Science (ICS), FORTH, Science and Technology Park of Crete, Vassilika Vouton, P.O. Box 1385, GR 711 10 Heraklion, Crete, Greece.`

Availability

The most recent version of the pager along with the test programs are freely distributed using ftp from ftp.ics.forth.gr:pub/pager. More information about the project can be found at `http://www.ics.forth.gr/proj/arch-vlsi/os`.