Implementation and Evaluation of a Remote Memory Pager*

Evangelos P. Markatos George Dramitinos
Kosmas Papachristos

Technical Report FORTH/ICS 129

Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
P.O.Box 1385
Heraklio, Crete, GR-711-10 GREECE
markatos@csi.forth.gr

March 1995

Abstract

Traditional operating systems use magnetic disks as paging devices, even though the cost of each page-fault measured in processor cycles continues to increase.

In this paper we explore the use or 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. Our pager has been implemented as a block device driver linked to the DEC/OSF1 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 conclude that the increasing use of fast local area networks will improve the performance of remote memory paging even more.

1 Introduction

Applications like multimedia, windowing systems, scientific computations, engineering simulations, etc. running on workstation clusters (or networks of PCs) require an everincreasing amount of memory, usually, more than any *single* workstation has available. To make matters worse, the use of *multiprogramming* and *time-sharing* further reduces the amount of *physical* main memory which is available to each application. 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 everincreasing, the cost of paging to a magnetic disk becomes unacceptable. Our performance measurements explained in section 5 suggest that the completion time of an application rises sharply when its working set exceeds the physical memory of the workstation. Faster swap disks would only temporarily remedy the situation, because processor speeds are improving at a much higher rate than disk speeds [7]. 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

*The authors are also affiliated with the University of Crete, Department of Computer Science. This report is available via anonymous ftp from ftp.ics.forth.gr:tech-reports/1995/95.TR129.remote_memory_paging.ps.Z.

systems and it is not used most of the time. It is the collective memory of all the workstations, hereafter called *remote memory*.

Remote memory provides high transfer rates which are mainly dictated by the interconnection network. For example, ATM networks provide a data transfer rate of 155 Mbits/sec per link; a transfer rate higher than any single disk can provide. A collection of ATM links serving several sources and several destinations may easily exceed an aggregate transfer rate of 1 Gbit/sec, more than expensive disk arrays provide! Fortunately, most of the time remote main memory is unused. To verify this claim, we profiled the unused memory of the workstations in our lab* 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! Thus, even at business hours there is a significant amount of main memory that could be used by applications that need more memory than *a single* workstation provides.

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

- **Memory-to-memory transfer rates 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: for most disks, transfer rates are still in the neighborhood of a few (4-6) Mbytes. Moreover, disk accesses suffer from seek and rotation latency which is not expected to be reduced from advances in semiconductor technology. In contrast, memory-to-memory transfers do not involve any mechanical parts, and therefore, are expected to take advantage of all improvements in semiconductor technology.

- **Application's working sets have increased dramatically over the last few years:** Modern processors provide 64-bits 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 file systems and databases, sophisticated window interfaces, 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 [15]. Similar ability is provided by Telegraphos [8], and SHRIMP [2]. Fast remote memory accesses have also been implemented using Active Messages [18, 1], programmed network interfaces [9], and trap-based remote invocation [17]. All these sources report that a single remote memory access takes as low as a few μs. The ability to perform single remote memory accesses efficiently, will enhance the performance of a remote memory paging policy significantly. If, for example, an application needs to make just a few accesses to a page, then it is not worthwhile to bring the *entire* page from remote memory, replacing an already resident and potentially more useful page. If the network provides the ability of efficient single remote accesses, the application can use these to access infrequently used pages.

In this paper we show that it is both possible and beneficial to use remote memory as a a paging device, by building the systems software that transparently transfers operating system pages across

*Our measurements are pessimistic, because in our Lab workstations are heavily used running VERILOG simulations for most of the time.

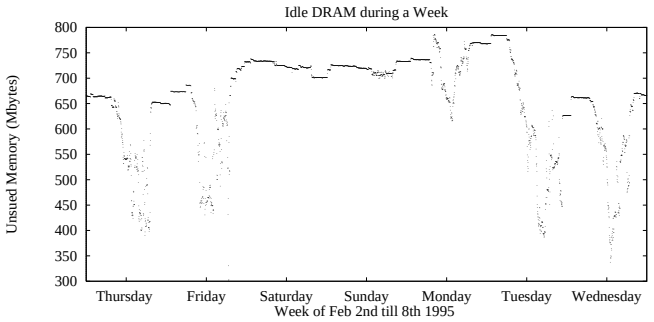

Figure 1: **Unused memory in a workstation cluster.** *The figure plots the idle memory during a typical week in the workstations of our lab: a total of 16 workstations with about 800 Mbytes of total memory. We see that memory usage was at each peak (and thus free memory was scarce) at noon and afternoon. Only exception were days 2-4 that were the weekend. In all times though, more than 300 Mbytes of main memory were unused.*

workstation memories within a workstation cluster. We describe a pager built as a device driver of the DEC/OSF1 operating system. Our pager is completely portable to any system that runs DEC/OSF, 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 slow 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 112% 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 networks like FDDI and ATM. Our extrapolated results suggest that paging over a 100 Mbits/sec interconnection network, reduces paging overhead to only 20% of the total execution time. Faster networks will reduce this overhead even more.

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

2 Related Work

Li and Petersen [10] have implemented a related system where they add main memory module on the I/O bus (VME bus) of a computer system. This memory module can be used both as backing store, and as (slow) main memory accessed via simple load and store operations. Although this approach increases the amount of memory available to a single workstation, this memory module can not be accessed by the other workstations in the same cluster. Thus, only a single workstation benefits from the extra memory. Instead, our approach uses the *existing* main memory of workstations in the same cluster for storing an application's data. Thus, (i) we do not increase the cost of any workstation by adding main memory to its I/O bus, (ii) we use the otherwise unused memory in the workstation cluster, and (iii) we scale the

[†]We distribute the external pager along with the test programs freely using anonymous ftp from `ftp.ics.forth.gr:pub/pager`.

amount of memory available to an application with a factor proportional to the number of workstations that are part of the same LAN.

Felten and Zahorjian [6] have implemented a remote paging system on top of a traditional Ethernet-based system, and presented an analytical model to predict its performance. Unfortunately, they do not report any results regarding the benefits of remote memory paging on real applications.

Schilit and Duchamp [14] 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 page-in over an Ethernet, they quote, is about 45 ms for an 4Kbyte page, which we believe is rather high. Their implementation is dominated by various overheads induced by Mach, and the slow local buses of portable computers. Thus, their performance figures are somewhat discouraging with respect to the usefulness of remote memory paging. Our implementation instead, has eliminated all unnecessary overheads, reducing the remote memory page-in time over an Ethernet to as low as 8.7 msec for an 8Kbyte page.

Comer and Griffoen [3] 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. Our work differs from [3] in the following aspects: (i) we argue that *local* disk paging is slower than remote memory paging, (which we think is not at all obvious), while [3] argues that *remote* disk paging is slower than remote memory paging. (ii) Instead of using *dedicated* servers for remote memory paging, any workstation in the system can be a remote memory server.

Anderson *et. al.* have proposed the use of network memory as backing store [1] and as a file cache [4]. Their simulation results suggest that using remote memory over a 155Mbits/s ATM network "is 5 to 10 times faster than thrashing to disk" [1]. In their subsequent work [12], they outline the implementation of a remote memory pager on top of an ATM-based network. Our work differs from [1] in that (i) we base our results on executing real applications on top of our implemented pager, instead of simulating them, (ii) we show that even in the case where the interconnection network has as low bandwidth as the disk, remote memory paging results in significant performance improvements over the disk, and (iii) we present and evaluate a novel mechanism that provides fault-tolerance in case of a memory server crash, while it requires only an insignificant additional amount of memory.

Our work bares some similarity with distributed-shared-memory systems [11, 5] 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.

3 The Design of a Remote Memory Pager

3.1 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 which use remote memory for swapping are called clients. Depending on its workload, a workstation may act as a server, or as a client, or as both, but only during a short transition interval. Along with the names, the load of the servers is also provided, so that prospective clients can locate the least loaded server.

All server workstations run a remote memory server that handles requests for page ins, page outs, as well as 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, 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 move its pages to another server, or its local disk.

3.2 Reliability

In an distributed system, a workstation may crash at any time. In the case of a remote memory server crash, we would like to be able to complete the execution of the application, and recover its lost pages. To provide this level of reliability, some form of redundancy must be used. We explore three different policies: mirroring, parity, and parity caching.

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. On a broadcast network like Ethernet or FDDI, each page is automatically broadcasted to all remote memory servers, thus eliminating the need to explicitly send each page both to the main and the mirror server. Nevertheless, 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 [13]. 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:

- The client sends the swapped out page to the server, which computes the XOR of the old and the new page.
- 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 $(1 + 1/S)$.

Parity Caching To avoid the additional page transfers induced by the basic parity method, we have developed a parity caching scheme which computes the parity on the client side, instead of sending the pages to a parity server. Our policy assumes that a small amount (e.g. 8) of memory frames on the client side act as a software cache for parity pages. Parity is updated in two stages:

1. When a page is swapped in, its parity is fetched in (if not already in the client's cache) and the XOR of the page and the parity is computed and stored into the local parity frame. This operation "removes" the newly swapped in page from the contents of the parity block.
2. When a page is swapped out, its parity is fetched in (if not already in the client' cache) and the XOR of the page and the parity is computed and stored into the local parity frame. This operation "adds" the swapped out page to the parity block.

When a server crashes, all of its pages that do not reside in the client's memory can be restored by XORing the pages in its group (that do not reside in the client's memory) with the parity page.

Compared to its naive ancestor, parity caching results in significantly fewer page transfers, and does not need to keep pages around waiting for the parity server to complete computing the new parity. Our performance measurements reported in section 5.6 show that even when only a small number of frames (8) is used from the client's memory as a cache for parity frames, parity caching results in at most 5% more page transfers than the case where no reliability policy is used.

4 Implementation

A subset of the proposed system has been built and is currently in use. It is robust enough to be used during the development and compilation of the system itself. It consists of a client issuing paging requests and a server satisfying these requests. The client side has been integrated in 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 page-in and page-out requests. In order to service these requests, it may forward them either to a user level client running on another host, or to the local disk. The DEC-OSF/1 kernel is not even aware that we 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 systems kernel.

4.1 The Remote Memory Pager

Normally the Remote Memory Pager (RMP for short) is a multi-threaded client which forwards the paging requests to a remote server, via sockets over an Ethernet. Our first prototype implements a single remote memory server system used to evaluate the performance of remote paging. The RPM connects to the remote memory server using a socket over TCP/IP. One client thread issues page-in and page-out requests to the server, while a second client thread accepts the data sent by the server.

RPM 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 the client's requests, paging to local disk is used.

Although the current implementation runs on top of a slow 10 Mbps Ethernet, remote paging is up to 2 times faster than using a local disk of the same bandwidth. It takes about 8.7 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 [18]. For the time being our performance results suggest the communication protocol latency for page transfers rarely exceeds 5%, even for 100 Mbits/sec networks.

Although bus-based networks will eventually become a performance bottleneck, the single link ensures that every request can reach every interested host. This means that the implementation of reliability using mirroring of parity, can be rather simple. A second prototype, still under development, uses mirroring in order to ensure the integrity of the pages transferred even when a server crashes. It also scatters the paged out pages to many servers in order to give the operating system the illusion of a swap device of high capacity. A dedicated RPM thread uses a socket to broadcast each request. A second thread waits until some server replies to the pending requests. A server in this case may be *primary* or *backup*. In case of a primary server crash, the second thread will soon be informed, based on a timeout mechanism, and will ask the backup server responsible for the pages maintained by the crashed server to fulfill its requests.

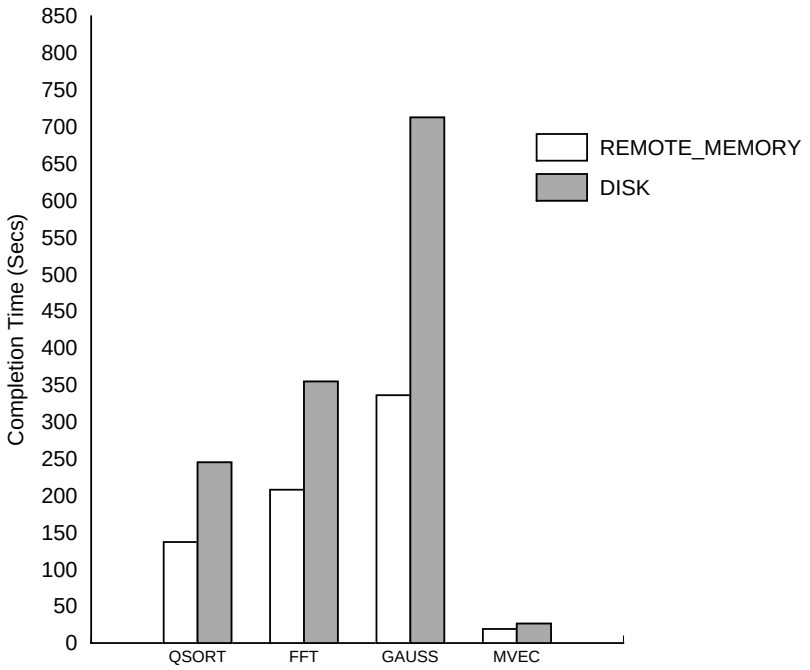

Figure 2: **Performance of applications using either the `DISK`, or the `REMOTE_MEMORY` as paging device.** We see that for all applications, the use of `REMOTE_MEMORY` results in significantly faster execution. All applications were run on a DEC Alpha 3000 model 300 workstation. The input sizes for `QSORT` was 4000 records, for `GAUSS`, a 2000×2000 matrix, for MVEC, a 2100×2100 matrix, and for `FFT` an array with 900000 elements.

4.2 The Remote Memory Server

The server is a user level program listening to a socket and accepting connections from clients. It uses portion of the local workstation's main memory to store the client's pages. When the client requests a page-in, the server transfers the requested page(s) over the socket. When the client requests a page-out, the server reads the incoming pages from the socket, and stores them in its main memory. [‡]

A backup server responds to page-out requests much like a primary server, but it does not respond to page-in requests unless the client has explicitly declared the primary server as crashed.

5 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 and `MVEC`, a matrix-vector multiplication. 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. The local disk that was used for paging is a DEC RZ55, providing 10Mbits/sec bandwidth, and average seek time of 16 msec.

[‡]The client is allowed to page-out at most a preset number of page frames to each server.

5.1 Performance of Remote Memory Paging Over the Ethernet

In our first experiment we evaluate two methods for paging:

- `REMOTE_MEMORY`, which uses only main memory of other workstations as a paging device. In this experiment only one remote memory server was used, because it was enough to provide the additional amount of memory our applications needed. The measurements were done on an (almost) idle Ethernet to ensure repeatability.
- `DISK`, which uses the local DEC RZ55 disk for paging.

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 `REMOTE_MEMORY` results in 112% faster execution time than `DISK`. Even for the `MVEC` application which performed very little paging, `REMOTE_MEMORY` results in 39% faster execution time.

The reason for the somewhat surprising performance improvements is 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. The *average* page-in/page-out service time was measured to be close to 9 ms for `REMOTE_MEMORY`, and close to 17 ms for `DISK`.

Our experimental results verify than 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`. 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.

5.2 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 `REMOTE_MEMORY` and under `DISK` is plotted in figure 3. We see that as soon as the working set size exceeds 20 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 by more than a factor of two.

5.3 Scaling the Network Bandwidth

Although figure 3 suggests that the performance of `REMOTE_MEMORY` 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 with a faster one (e.g. FDDI, ATM, FCS, etc.). To evaluate the performance of the applications on top of faster networks, or faster disks we make detailed performance measurements that separate the completion time of the application into three factors: (i) bandwidth-dependent blocking time, (ii) useful user time, and (iii) protocol-dependent systems overhead. Using the provided `time` command we measure the elapsed time for each application which is the sum of factors (i)-(iii). The same command also provides the user-time (factor (ii)), and the system time (factor (iii)). If from the elapsed time, we subtract the user plus the system time, we get the time the system was idle waiting for pages to go through the interconnection network (factor (i)). By dividing this idle time with the number of page ins plus page outs, we get the average time the application waits for each page to go through the interconnection network. Assuming that an X times faster interconnection network will

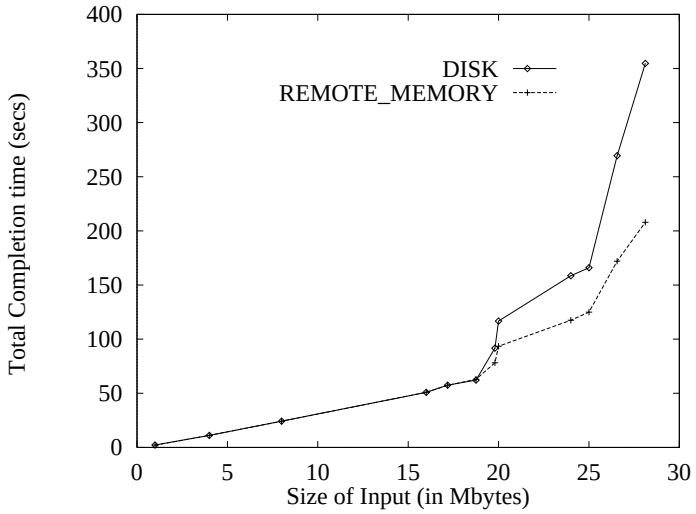

Figure 3: **Performance of FFT as a function of input size when either `DISK`, or `REMOTE_MEMORY` are used as backing store.**

reduce this waiting time by a factor of X, we can predict the completion time of the application on the faster network [§] by adding the measures user and system times, with the predicted blocking time.

We made all these measurements on our FFT application, and predict its performance on a system with an interconnection network which is two and ten times as fast as the Ethernet. We also predict its completion time on a system with twice as fast disk (`DISK*2`), and on a system that has enough memory to hold all the working set of the application (`ALL_MAIN_MEMORY`). The predicted execution times, along with the measured execution times of `DISK` and `REMOTE_MEMORY` are plotted in figure 4. We see that `ETHERNET*10` performs very close to `ALL_MAIN_MEMORY`, and significantly better than both `REMOTE_MEMORY` and `DISK`.

To understand the results shown in figure 4, we analyze the execution time of FFT with 28Mbytes of input. The measured elapsed time is 208 seconds, consisting of 78.5 sec of useful user time, 5 sec of system time, and 124 sec of network blocking time, spent waiting for pages to go through the Ethernet. During the same run, the application suffered 6520 page-outs and 7791 page-ins. The average waiting time for a page transfer (both for page ins and page outs) on top of the Ethernet is $124/(6520+7791)$, or about 8.6 ms. Using a ten times faster interconnection network, the average waiting time will be reduced at least to 0.86 ms. Thus, the total completion time of FFT would be at most $78.5 + 5 + 124/10 = 95.9$ sec, divided as follows: 82% in user time, 5% in system time, and 13% in network blocking time. We see that a 100 Mbit/sec interconnection network reduces the total paging overhead to a mere 17% of the total applications 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.

5.4 The Latency of Remote Memory Paging

Based on our measurements above we can compute the paging latency. For example, the elapsed time of `FFT` on 28 Mbytes of input is 208 seconds, while the user time is 78.5 seconds. The rest 129.5 seconds should be attributed to paging overhead induced by 6520 page-in requests and 7791 page-out requests.

[§] Actually, the prediction we make is pessimistic because an X times faster network will not only reduce the page transfer time by a factor of X, but itit will also allow more overlap between computation and network transfer, thus reducing the average page transfer time by a factor larger than X! Thus, future faster networks will result in even better completion times than the ones we predict here, thereby making our case even stronger.

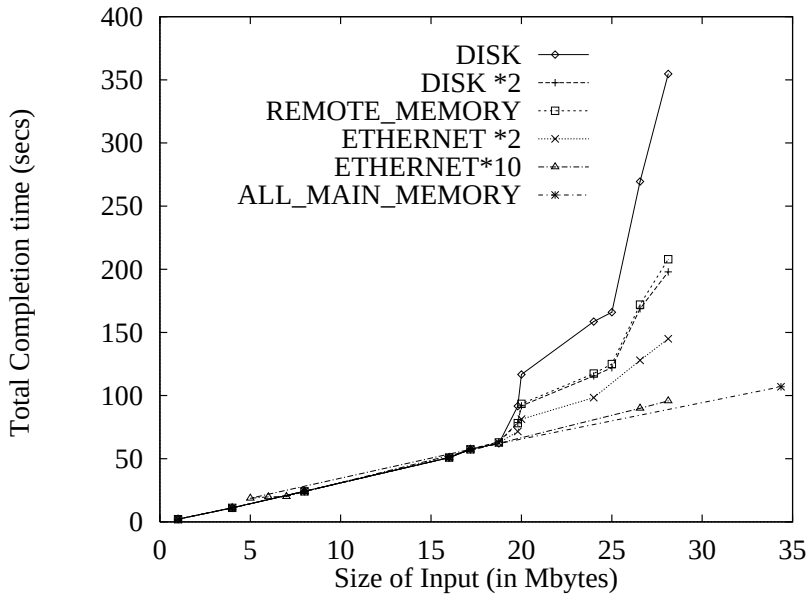

Figure 4: **Performance of FFT for various Architecture Alternatives.** `DISK` is the measured completion time when paging to a local disk. `REMOTE_MEMORY` is the measured completion time when paging to remote memory on top of the Ethernet. `ETHERNET*2` and `ETHERNET*10` is the predicted completion time when using remote memory as a paging device, on top of a network that is twice and ten times as fast as the Ethernet interconnection network. `DISK*2` is the predicted completion time when using a twice as fast disk for paging. `ALL_MAIN_MEMORY` is the predicted completion time of FFT when we use the same workstation but with enough memory to hold its entire working set.

Thus, the average latency per request is $129.5/(6520+7791)$ or 9.05 ms. From these, 7.2 ms were spend transferring each page on the Ethernet, and the rest 1.85 ms were the average software latency per paging request. ¶

Previous measurements have reported that an 8 KByte page takes about 45 ms over an Ethernet for each page-in [14]. 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 1.85 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 fast and does not pose a problem in performance.
- The processor we use is a DEC Alpha, which is 3-4 times faster than the 386 processor used in [14].
- Finally, our pager is implemented as a block device driver, while in [14] 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 disk paging, while user-level implementations have not reported performance results to support similar claims [14].

¶ We believe that this latency will be reduced if a faster than TCP/IP protocol is used.

5.5 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 editor. 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.

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 run on an idle workstation. Only `QSORT` suffered a 7% overhead in its completion time: probably the kernel swapped out some the remote memory server's pages on the disk. 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 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 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.

5.6 Reliability

Disks provide a stable storage for data, because their mean time between failures can be several years [13]. Unfortunately, main memories do not have such long times to failure. The mean time between crashes/reboots for a workstation may be from a few weeks to a few months. Thus, some form of redundancy is necessary to provide applications that page to remote memory with high reliability.

5.6.1 Parity Caching

To reduce the main memory requirements of mirroring, and the long latency of parity, we developed the method of *parity caching* described in section 3. Summarizing, each client reserves a small number of local pages to hold parity frames. When a page is swapped in or out, its parity frame is swapped in as well (if not already at the client's parity frames), and the new parity is computed. This method increases the number of page ins and page outs, because besides prorram pages, parity frames are swapped in and out as well. To measure the additional overhead of parity caching, and compare it to mirroring, we use execution driven simulation on top of the same DEC Alpha 3000 model 300 workstation. We use `ATOM` [16], an object file rewriting tool, that executes each application, while at the same time simulates the reliability policy we want to evaluate. The policies we evaluate are:

- `NO_RELIABILITY`: No redundant information is kept. When a server crashes, the application will not be able to continue its execution.
- `MIRRORING`: When a page is swapped out, it is sent to two servers instead of one, so that when one of them fails, the other will still have all the information the application needs.
- `PARITY_CACHING`: When a page is swapped in or out, its parity frame is swapped in (if not already there) and is XOR'ed with the page. When a server crashes, its pages which are not in the client's

main memory can be reconstructed by XORing the relevant pages of the other servers and the parity frames. In our experiments, we simulated 50 memory servers, and a client that caches as many as 8 of the parity frames locally. Pages are distributed round robin among the available servers.

The applications we simulate are:

- `MVEC`: Matrix vector multiplication of a 2000×2000 matrix.
- `GAUSS`: Gaussian elimination on a 2000×2000 matrix. ‖
- `SORT`: Sorting of an array of 32 Mbytes, using the standard quicksort algorithm.

The architecture simulated is a DEC Alpha 3000 model 300 workstation with 16 Mbytes of main memory available to applications. Only eight pages of the main memory were used to hold only parity frames. A total of 50 servers were simulated for each client.

If all workstations are connected via a broadcast interconnection network, no extra page transfers are needed to implement a reliable policy. For example, in `MIRRORING`, each swapped-out page needs to be broadcasted only once over the interconnection network to reach all servers. Similarly, `PARITY_CACHING` does nor need extra parity frame transfers. If the workstations that keep the parity frames snoop in the interconnection network, they can intercept all swapped-in and swapped-out pages, and update their parity records. If, however, the interconnection network is not broadcast-based, then extra page transfers are needed for the reliable policies. For example, `MIRRORING` doubles the number of page transfers for all swapped-out pages, while `PARITY_CACHING` increases the number of page transfers by a factor that depends on the effectiveness of caching. The exact magnitude of this factor is studied in our simulations, where we measure the number of pages swapped-in (including parity pages), and swapped-out (including parity and mirror pages) by each policy. The results are plotted in graphs 5 and 6. We see that the number of pages swapped in for `MIRRORING` and `NO_RELIABILITY` are the same, but the number of pages swapped out for `MIRRORING` are twice that of the `NO_RELIABILITY`. For `PARITY_CACHING`, both the number of pages swapped in and swapped out, are within a 5% of those for `NO_RELIABILITY`. The reason is that all applications have some locality of reference. Thus, pages swapped-out within a short time interval using some LRU policy, will probably be swapped-in also within a short time interval. Pages who were initially swapped out close to each other, belong to the same parity frame. Thus, as long as these pages are swapped close in time, their parity frame will reside in the client's cache, and no extra page transfers to move the parity will be needed.

We see that reliability comes at little extra cost, actually, from 0% to 5%, depending on the nature of the interconnection network, the application, and the policy used. We believe that the little extra overhead is a small cost to pay for the benefit provided.

6 Discussion - Future work

Our prototype implementation suggests that it is possible to built an *efficient* remote memory pager without making any modifications to the operating systems 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.

Choosing servers In the current implementation the choice of the servers used by each client is hardwired. Clearly this solution is unacceptable for a real system. Workstations should be able to

‖Because the completion times of the simulation were too long, we simulated only the first 100 million references.

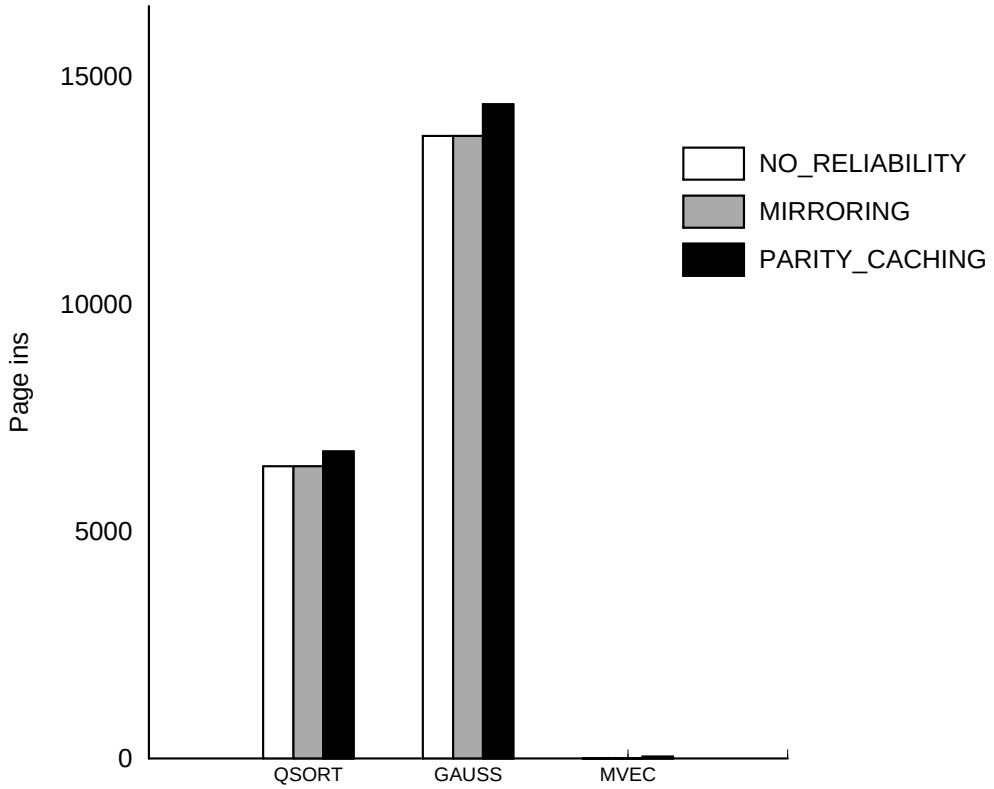

Figure 5: **Number of Pages swapped in.**

become active or inactive servers based on their load. Prospective clients should be able to dynamically choose the most appropriate server.

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 RPM will be limited. In this case the cost of using the network, especially in the case of old slow networks like Ethernet, may become higher than the cost of using the local disk. Such a situation could be handled by the RPM 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.

Thrashing In the case of multiple servers and clients, where clients and servers coexist at the same machines, there is the possibility of thrashing. This means that it is possible for a number of workstations to form a chain satisfying each others paging requests leading to network bandwidth waste and to unacceptable latency. This situation can be avoided if each server refuses to accept future pageout requests when the memory load of its workstation exceeds some threshold.

Compression Compression may be used in order to reduce the size of the transferred data and increase network utilization. For slow networks, such as Ethernet, compression would be especially useful, because the time to compress the page is a small percentage of the time saved due to reduced network traffic. However, depending on the network throughput and the processor speed the exact benefits of compression may vary.

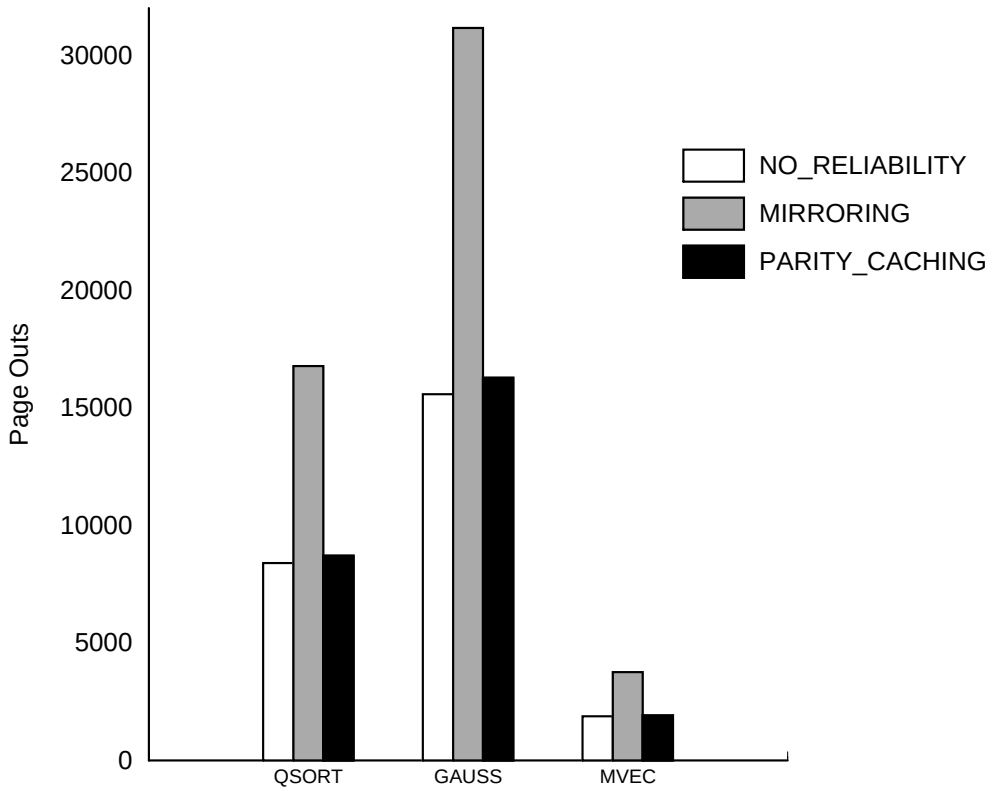

Figure 6: **Number of Pages swapped out.**

Reliability Clearly, it is not acceptable for an application to fail due to a crash of another workstation used as a remote memory server. Mirroring is the simplest solution that may be applied but it has a high memory overhead. Parity caching may be used instead, in which case the latency would increase in case of a crash. In both cases the network topology should be taken advantage of in order to reduce the messages needed to achieve reliability. We believe that all three options should be provided (no reliability, mirroring and parity).

7 Conclusions

In this paper we explore the use of remote main memory for paging. We describe our prototype implementation of a pager on top of the DEC OSF1 operating system as a device driver. No modifications were made to the kernel of the (monolithic) DEC OSF1 operating system. We run several applications to measure the performance of the system. 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 112% (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 that 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.

- *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.

- *Reliability comes at little extra cost.* Our parity caching method introduces at 5% more page transfers than the simple remote memory paging methods that provide no reliability: a small cost to pay for such a large benefit.

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

Acknowledgments

This work is being done within ESPRIT project 6253 "Supercomputer Highly Parallel System" **SHIPS**, funded by the European Union, through DG III of its Commission, HPCN Unit. We deeply appreciate this financial support, without which this work would have not existed.

We would like to thank Catherine Chronaki and Manolis Katevenis for useful comments in earlier drafts of this paper. A. Labrinidis, and A. Zaras provided useful feedback.

References

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

[2] 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.

[3] 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.

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

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

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

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

[8] Manolis Katevenis. Telegraphos: High-Speed Communication Architecture for Parallel and Distributed Computer Systems. Technical Report 123, ICS-FORTH, May 1994.

[9] 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.

[10] K. Li and K. Petersen. Evaluation of Memory System Extensions. In *Proc. 18-th International Symposium on Comp. Arch.*, pages 84–93, 1991.

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

[12] A. Mainwaring, C. Yoshikawa, and K. Wright. NOW White Paper: Network RAM Prototype, 1994.

[13] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

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

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

[16] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *PROC of the SIGPLAN '94 PLDI*, Orlando, FL, June 1994.

[17] 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 1992.

[18] 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.