Adding Flexibility to a Remote Memory Pager

Evangelos P. Markatos   George Dramitinos

Computer Architecture and VLSI Lab
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
P.O.Box 1385, Heraklio, Crete, GR-711-10 GREECE
{markatos, dramit}@ics.forth.gr

Proceedings of the Fourth International Workshop on Object Orientation in Operating Systems.

Abstract

Traditional operating systems use magnetic disks as paging devices, although the cost of each page fault measured in processor cycles continues to increase. In this paper we argue that applications should be given the opportunity to use as backing store either magnetic disk or the memory of idle workstations within the same LAN. We have implemented a pager that provides this flexibility, measured its performance over an Ethernet, and found it to be superior to traditional disk paging. We conclude that as the available network bandwidth increases, the use of network memory as backing store becomes a evenmore attractive alternative.

1 Introduction

Traditional operating systems use magnetic disks as paging devices. In such systems, pages that can't be placed in main memory are stored on the magnetic disk, and recalled to main memory when needed. Because magnetic disks speed does not keep up with processor speed, the cost of paging (measured in processor cycles) increases with time. For example, only five years ago, the cost of a page fault was 0.1 to 0.6 million processor cycles [3]. Currently, a page transfer from disk to main memory takes an average of 10 to 20 ms, which corresponds to an equivalent of 2-4 million processor cycles on a modern 200 MHz processor! Thus, the cost of a page fault over the last five years, has increased by almost an order of magnitude relative to processor speed, a cost most applications are not willing to pay. To make matters worse, disk-latency-hiding methods like *multiprogramming*, are not going to hide the latency of paging, because multiprogramming implies that more applications will contend for the same main memory and will probably produce an even higher number of page faults, leading to *thrashing*.

There are two ways that will make paging efficient and thus useful for applications that require large amounts of main memory:

- Provide *fast* paging devices.
- Give systems the *flexibility* to choose among the available paging devices, and tune the paging parameters to the needs of each application.

1.1 Fast Paging Devices

Traditional operating systems force all applications to page to the local disk of their host computer or to a remote disk. However, in most systems there is a variety of paging devices that an application may use for paging. For example, in workstation clusters, the unused main memory of all workstations can be collectively used for paging. Experimental results suggest that remote memory paging may have substantial performance improvements over local disk paging [4]. An another example, computer systems which are equipped with Redundant Arrays of Inexpensive Disks[2] provide higher throughput than a single disk, thereby reducing the cost of paging.

In this paper we argue that systems should use remote memory as an alternative paging device.This approach benefits from

- high data rates provided by modern networks.
- diverting some load from the disk that can now be used solely for file system I/O.

1.2 Flexibility

There exist several devices for paging that provide a wide range of performance, cost, and fault-tolerance levels. The best paging device for each application depends on several factors including (i) the performance requirements of the application, (ii) the application's fault-tolerance needs, and (iii) the current load of each paging device, etc.

Traditional operating systems do not provide any flexibility in their paging system. A noticeable exception are operating systems that support user-level memory management [1]. In these systems each object is managed by a user-level process, the memory manager, which is responsible for storing and retrieving the object from the paging device. Although flexible, user-level memory managers induce several system calls and protection domain crossings, which usually result in large amounts of overhead.

In this paper we describe the implementation of a pager that, although provides a high degree of flexibility, avoids traditional overheads. A prototype of our pager has been implemented as an external device driver linked to the DEC/OSF-1 operating system. Our pager is faster than user-level pagers, and can be used in any system that runs DEC/OSF-1, because not a single change was made to the operating system kernel. The rest of the paper is organized as follows: section 2 describes the implementation, section 3 presents some preliminary performance results and section 4 presents our conclusions.

2 The Implementation of a Flexible Pager

We have already implemented a pager that supports either the local disk, or another workstation's main memory as the paging device. 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 whether we use remote main memory or a 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 DEC/OSF-1 operating system kernel.

2.1 Fault-Tolerance

Workstations may crash at any time. If their memory is used as a paging device by some applications, then the data will be lost and the applications will not be able to complete their execution. To avoid this problem, the pager may implement a fault-tolerance policy, which will recover the lost data. Currently we are experimenting with two policies taken from the domain of disk arrays [2]: *mirroring*, and *parity*. In mirroring, each swapped out page is sent to two paging servers; if one of them crashes, the data can still be found in the other. In parity, all pages are partitioned into sets. In each set of pages, only one page of each server can participate. For each set of pages there is a parity page, that contains the parity of all the other pages in the set. When a server crashes, its pages can be recovered by XORing the rest of the pages in each set.

Both the described policies can tolerate the crash of only one server. Policies that tolerate the crash of several servers can be added as well. The choice of the suitable policy for each application depends on several factors including the application's requirements, and the type of hardware used. For example, on top of a broadcast network, like Ethernet, or FDDI, mirroring can be simply and efficiently implemented, because each page need to be sent (broadcasted) only once, for all the paging servers to receive it. On the other hand, on top of an interconnection based network, like ATM, the parity policy may be more appropriate because it needs less memory than mirroring to keep the redundant information.

A detailed study of fault-tolerance policies is outside the scope of this paper. The interested reader is refered to [4].

2.2 Performance Monitoring Tools

To make accurate decisions, the pager needs information about the load and response of each paging device, so as to be able to make the best choice. We are currently working on developing a tool that will monitor several dynamic performance metrics and make them available to pagers. The metrics monitored include disk traffic, networks traffic, main memory unused, etc. Based on these metrics, a pager can choose the best available paging device each time.

3 Sample Performance Results

In this section, we evaluate the use of remote memory for paging and predict the performance of our pager on top of high bandwidth networks.

3.1 Experimental Environment

We run a number of representative applications on top of our pager[1]. The applications can use either the disk, or the remote memory for paging. The local disk used is a DEC RZ55 providing 10Mbits/sec bandwidth, and an average seek time of 16 ms. Remote memory paging was done on top of standard Ethernet that provides the same bandwidth as the disk. All workstations used were DEC Alpha 3000 model 300. Our applications include `GAUSS`, a gaussian elimination, `QSORT`, a quicksort program, `FFT`, a Fast-Fourier Transform, `MVEC`, a matrix-vector multiplication and `NBODY`, a program calculating the gravity forces among particles. To ensure repeatability of our results, all experiments were run after system reboot.

3.2 Remote Memory vs Disk

We measured the completion time of the applications using either paging device (`DISK`, or `REMOTE_MEMORY`) and plotted the results in figure 1. We see that the use of remote memory paging even over a low bandwidth interconnection network like Ethernet yields significant improvement in the completion time of all the applications tested. The performance gain is proportional to the number of pageins and pageouts performed during the execution of each application and ranges from 40.45% for `NBODY` to 108.72% for `GAUSS`.

In table 1 we measure the cost of a pagein or pageout operation using either the `DISK` or the `REMOTE_MEMORY`. The detailed description of the way this measurement is performed can be found in [4]. We see that the time to service a pagein or pageout request using the `DISK` is more than twice the time to service the request using the `REMOTE_MEMORY`.

[1] We distribute the source code of our pager along with the test programs freely using anonymous ftp from `ftp.ics.forth.gr:pub/pager`

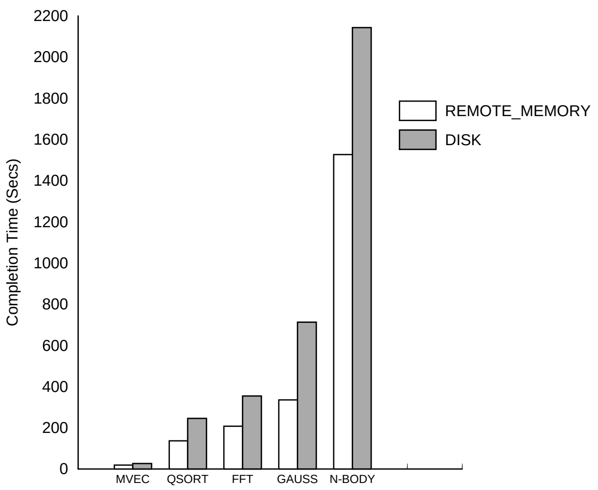

Figure 1: **Performance of applications using either the DISK, or the REMOTE_MEMORY as paging device.** We see that for all applications, the use of REMOTE_MEMORY results in significantly faster execution. All applications were run on a DEC Alpha 3000 model 300 workstation. The input sizes for QSORT was 4000 records, for GAUSS, a 2000×2000 matrix, for MVEC, a 2100×2100 matrix, for FFT an array with 900000 elements, and for NBODY an array with 140000 elements.

Application	DISK (ms)	REMOTE_MEMORY (ms)
MVEC	11.36	7.02
QSORT	17.89	9.63
FFT	18.77	8.7
GAUSS	18.74	8.42
NBODY	22.14	8.21

Table 1: **Cost of a page transfer using either the DISK or the REMOTE_MEMORY as a paging device.**

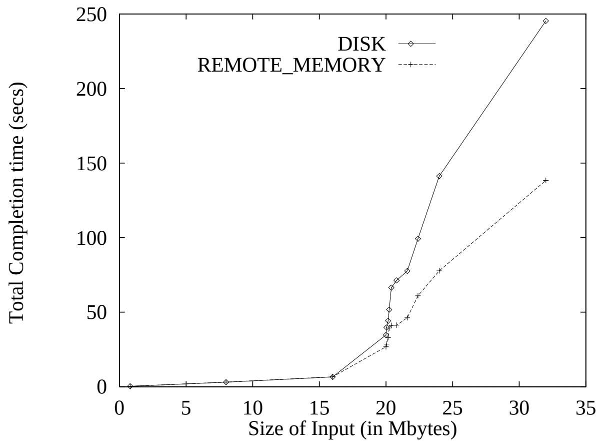

Figure 2: **Performance of QSORT using either the DISK, or the REMOTE_MEMORY as paging device.**

The cost of a page transfer to/from DISK varies from 11.36ms to 22.14 ms. Since the disk bandwidth is 10 Mb/s, it takes 6.25 ms to read/write an 8 KB page, thus the average seek time ranges from 5.11 ms to 15.89 ms depending on the application's memory access pattern. For an application that performs only pageouts, like MVEC, the average seek time is minimal since most paged out pages are written consecutively to the disk. On the other hand, for applications like NBODY that have an almost random access pattern leading to pageins of pages scattered over the disk, the average seek time increases considerably.

We also see that the cost per page transfer for REMOTE_MEMORY ranges from 7.02 ms for MVEC to 9.63 ms for QSORT. MVEC is an application that performs only pageouts, many of them not causing it to block. That's why it exhibits the lowest cost per page transfer. For the rest of the applications, the cost of the page transfer depends on the interleaving of pageins and pageouts and on the amount of computation performed per page, that determine the amount of overlapping between computation and paging. That's why QSORT that performs much lower amount of computation per page than all other applications exhibits the highest cost per page transfer.

To understand how the completion time scales with input size, we chose one application (QSORT) and plotted its completion time versus its input size in figure 2. We see that as soon as the input size exceeds the available main memory (16 MB), paging starts, which leads to unacceptable performance when the local disk is used for paging. The improvement in completion time using REMOTE_MEMORY is as high as 82% for input size equal to 24 MB.

3.3 Performance Prediction over High Bandwidth Networks

We see that the cost of disk paging makes it essentially useless. REMOTE_MEMORY over Ethernet improves the situation, but it still suffers from significant overhead. Fortunately, Ethernet is a slow old technology that will eventually be replaced by faster networks like FDDI and ATM.

To evaluate the performance of the applications on top of modern (faster) networks, we extrapolate from the measurements of QSORT we made on top of the Ethernet. Assuming that an X times faster interconnection network will reduce the bandwidth-dependent blocking time by a factor of X, and leave the rest of the user and system time the same, we can predict the completion time of the application on the faster network.

We made all these measurements on our QSORT application, and predict its performance on a system with an interconnection network which is ten times as fast as the Ethernet (100 Mbps ETHERNET). We also predict its completion time on a system that has enough memory to hold all the working set of the application (ALL_DRAM). The predicted execution times, along with the measured execution times of DISK and REMOTE_MEMORY (shown as 10 Mbps ETHERNET) are plotted in figure 3. We see that 100 Mbps ETHERNET

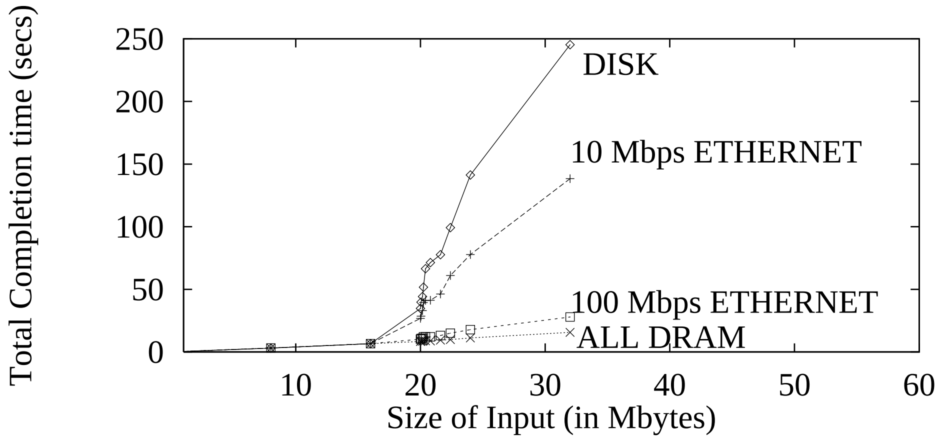

Figure 3: **Performance of QSORT for various Architecture Alternatives.** Measured performance for DISK and REMOTE_MEMORY. Predicted performance for ETHERNET*10. ALL_MAIN_MEMORY is the predicted completion time of QSORT when we use the same workstation but with enough memory to hold its entire working set.

performs very close to ALL_DRAM, and significantly better than both 10 Mbps ETHERNET and DISK. Thus, our approach to remote memory paging, on top of fast interconnection networks can effectively use the collective memory of a workstation cluster, and give applications the illusion that the physical memory they have available locally is sufficiently large.

4 Conclusions

In this paper we argue that the cost of paging to magnetic disks continues to increase with time because magnetic disks speed does not keep up with processor speed. Fortunately, the remote memory of idle workstations within the same LAN can be used as backing store. We describe the implementation of a pager that uses either remote idle memories or magnetic disks as backing store, and measure the performance of a number of applications using this pager. We see that even over low bandwidth networks, like the Ethernet, paging to remote memory is twice as fast as paging to traditional disk. Evermore, our pager avoids the traditional overhead associated with user-level memory managers, because it is being implemented as a device-driver linked to the DEC/OSF-1 kernel, resulting in fewer system calls, protection domain crossings, and data copying. We predict the performance of QSORT over high bandwidth networks and find that the performance of the application over a 100 Mbps network is very close to the performance of a system that has enough main memory to hold the entire working set of the application. We conclude that as the network bandwidth increases, the use of remote memory for paging provides an attractive alternative to disk paging at no extra hardware cost.

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 Catherine Chronaki and Manolis Katevenis for useful comments in earlier drafts of this paper. A. Alexandrakis is involved in the implementation of performance monitoring tools.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, pages 93–112, Pittsburgh, PA, June 1986.

[2] 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.

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

[4] E.P. Markatos, G. Dramitinos, and K. Papachristos. Implementation of a Remote Memory Pager. Technical Report TR129, Institute of Computer Science, FORTH, March 1995. available via anonymous ftp from ftp.ics.forth.gr:/tech-reports/1995.