Issues in Reliable Network Memory Paging

Evangelos P. Markatos[*]
Computer Architecture and VLSI Systems Group
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
Vassilika Vouton, P.O. Box 1385 GR 711 10 Heraklion, Crete, Greece

In Proceddings of MASCOTS 96, San Jose, CA, Feb 1995

Abstract

Recent distributed systems are connected with high-performance networks, that make possible the use of a new level of memory hierarchy: network memory. Network memory provides both high bandwidth and low latency, that make it attractive for uses like paging and file caching.

In this paper we explore the issues of building a reliable network memory system that is resilient to single workstation failures. We propose novel parity-based policies and evaluate their performance using trace-driven simulation of realistic applications. The presented performance results suggest that our proposed policies provide reliability at a surprisingly small run-time overhead.

1 Introduction

Most computing environments today consist of a number of workstations or personal computers[1] connected via a (high-speed) interconnection network. In a workstation cluster it is very probable that several resources are idle most of the time, even during business hours. Such resources include CPU cycles, main memory, etc. Although a significant amount of work has been done to exploit the unused processor cycles in a distributed system [12], very little work has been done to exploit the unused main memory in a workstation cluster. This is partly because, network memory reduces the reliability of the applications that use it. For example, if a workstation crashes (and there are several of them that may crash at any time), the applications that use the workstation's memory may not be able to complete their execution.

This situation may be remedied if network memory is used in a *reliable* way, where even if a workstation that holds an application's data crashes, the system will be able to recover the lost data, and allow the application to complete its execution.

In this paper we describe a novel way to build a reliable network memory system at a small performance cost. Our system is based on parity mechanisms: for each set of workstations that are used for the network memory (called memory servers) there is a parity workstation (called parity server), whose memory holds the parity memory contents of the memory servers. If a memory server crashes, its memory contents can be reconstructed by taking the XOR of the rest of the memory servers in the set (including the parity server).[2]

In this paper we focus on a specific application of network memory: remote memory paging [1, 3, 7, 8, 10]. In remote memory paging, the idle memories of all workstations in a workstation cluster are used as a paging device. Our reliability methods can be applied to other uses of network memory, like file system caching [5, 6, 9], database caching, etc. The contributions of this work are:

- We describe how to build a reliable network memory system that is resilient to individual workstation failures, that is, if a workstation crashes, its main memory contents can be recovered.
- We propose three novel reliability policies that impose low overhead and have low resource requirements.
- Using trace-driven simulation, we show that our policies provide reliable network memory at low cost.

The rest of the paper is organized as follows: In section 2 we describe the policies used to construct a reliable system. In section 3 we evaluate the performance of the proposed policies, in section 4 we present related work, and we conclude the paper in section 5.

2 Reliable Network Memory

2.1 The Environment

The network memory paging environment we assume is as follows: A sequential application that is executed on a workstation (client) may require more memory than the client actually has, thus may be forced to use paging.

All idle workstations that are willing to donate their memories for paging, act as memory servers that respond to page-in and page-out requests initiated by the client software. Every few memory servers, there is a *parity server* that holds the parity of the memory contents of the memory servers. Both memory servers and the parity server comprise a *parity set*. The assignment of pages to memory servers and to parity sets is simple: Pages are assigned to

[*]E.P. Markatos is also with the University of Crete. He can be reached at markatos@ics.forth.gr.

[1]In the rest of the paper we will use the term workstation to mean both workstation and high-speed personal computer.

[2]Parity-based methods have been extensively used in Redundant Arrays of Inexpensive Disks (RAIDs) [2], which recover from *disk* failures, but do not cope with main memory failures.

memory servers in a round-robin fashion[3]. Each server has only one page in each parity set, and each page participates in only one parity set. If the server crashes, its lost page can be reconstructed by XORing the rest of the pages in the parity set[4]. To improve performance, a parity server may be distributed over several workstations. For example, instead of having nine memory servers, and one parity server, we may have ten servers, where 90% of their memory is dedicated to store memory pages, and 10% is dedicated to store parity pages. This approach distributes the load more evenly across all servers. Although we advocate such an approach, in the rest of the paper we will refer to the parity server as a separate workstation, in order to make the presentation simpler. The performance results we present are the same in both approaches.

2.2 The Policies

2.2.1 The Disk

The simplest policy that provides a reliable network memory writes all updated data both to a magnetic disk, and to the network memory at the same time. If a workstation crashes, its lost data can always be found on some disk. Unfortunately, the performance of this policy is limited by the amount of disk bandwidth available in the system. If the disk bandwidth is significantly smaller than the network bandwidth (as is the case with modern gigabit networks), the performance of the network memory system is bound to suffer.

2.2.2 Simple Parity

The memory contents of a memory server may change only when it receives a page out request along with the corresponding page from the client. In this case, the parity server should be notified of the modified page, in order to update its parity page to reflect the changes. In this policy, for each page out, a page (say A_{new}) is sent from the client to the memory server to replace the older version of the same page (A_{old}). In the memory server, the A_{new} is XOR'ed with A_{old}, and the resulting page is sent to the parity server to be XOR'ed with the parity page of the parity set that contains page A. This second XOR operation has the effect of *removing* A_{old} from the parity page and *adding* A_{new} to it.

2.2.3 Delayed Parity

Fortunately, there is some spatial locality[5] in the page-out sequence that we may take advantage of, and reduce the number of page transfers, and hence the cost of reliability: pages are frequently paged out in large chunks to reduce disk seek overhead.

Based on this locality observation we see that pages that belong to the same parity set will probably be paged out at about the same time. When the system pages out all the pages that belong to the same parity set, the parity server need not be notified for each and every one of them. Instead, the XOR of all these pages can be computed at the client node and sent to the parity server when the last page is paged out. Even when not all the pages of the same set are paged out consecutively, "Delayed Parity" may have a performance advantage: suppose that only 7 out of the 10 pages of the set are paged out consecutively. Then, the parity calculated represents only the 7 pages. Thus, the client tells the rest 3 servers to sent their pages to the parity server for calculation of the new parity resulting in a total of 11 page transfers (7 for the pages, 3 for the rest of the servers, and 1 for the incomplete parity page). The simple parity policy would send 2×7=14 pages, or 20% more.

2.2.4 Logging

It should be obvious that we get the best performance out of parity-based methods, only when we perform the least number of updates to the parity server, which happens when whole parity sets are paged out at a time. We propose a new parity-based method called *logging* which does exactly this: for each set of paged out pages, it constructs a new parity set, computes the parity page on the client side, sends the pages to their servers, and sends only one parity page to the parity server after the whole new parity set is sent out.

Lets see the example in figure 1. There we have four memory servers, and one parity server. Memory server 1 provides backing store for pages 1, and 3, memory server 2 provides backing store for pages 2, and 8, etc. Pages 1, 2, 4, and 6 belong to one parity set: their corresponding parity page P1 is held on the parity server. Similarly, pages 3, 8, 7 and 9 belong to the second parity set: their correspond parity page is P2. After paging out pages 2, 4, 6, and 8, the system *does not* place the new copies of the pages in their original positions, but forms a new parity set, sends the pages to servers 1 through 4, creates a new parity page (P3) and sends it to the parity server. Finally, the client updates its tables, marking the old positions of pages 2,4,6, and 8 as invalid. Note, that the servers that keep the invalid pages are not informed of the fact that the pages are invalid. To reduce communication overhead, the space of the invalid pages is not deallocated at this point. However, if the memory that stores older versions of pages is *never* deallocated (recycled), then the corresponding page frames will be wasted, and eventually the system will run out of memory. To avoid this memory shortage problem, a page frame cleaning method should be used. This method will reclaim space (periodically or on demand), update the affected parity frames, and use the reclaimed space to store pages from new page out requests.

We will describe two cleaning policies that try to achieve the best balance between page transfers and wasted memory.

Logging - Greedy: In the Greedy policy we reclaim space in the following two cases:

Rule 1: When we page out a page, its previous position is marked as invalid. If all pages in the same parity set are invalid, the set is declared free, and all its page frames can be reused. No communication is necessary, because both the free lists and the bitmaps of invalid pages are kept on the client side. The parity page needs no update, because the parity set is empty.

Rule 2: When the system runs out of space, it finds the parity set with the largest number of invalid pages. All

[3] More elaborate policies of assigning pages to parity sets can be defined but are beyond the scope of this paper.

[4] By having each page on several parity sets, we may be able to tolerate multiple server crashes. Fortunately, being able to tolerate one server's crash is usually more than enough, because the probability of two servers crashing within a small time interval is negligible [2].

[5] One of the arts in designing systems software and computer architecture is the ability to *discover* the locality in program behavior and to *design* systems that exploit it, neither of which is an easy task. In this paper we describe a form of locality that exists in the page out sequence of a pager, and design policies that exploit it.

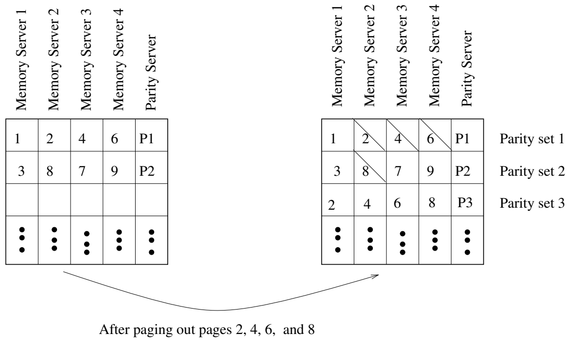

Figure 1: **Example of Logging:** after paging out pages 2,4,6, and 8, their previous positions are marked as invalid, and the pages form a new parity set and are stored in new servers.

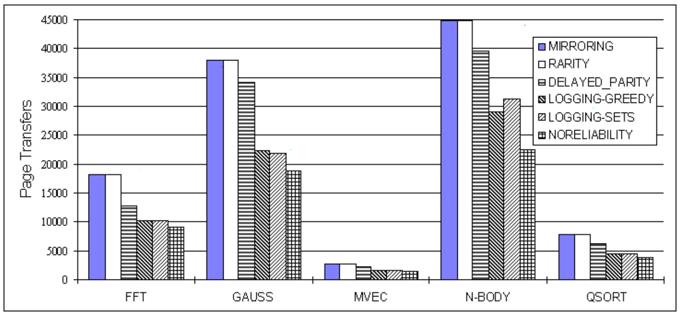

Figure 2: **Performance of various Reliability Policies** In this experiment, 8 memory servers and one parity servers were used. The Logging policies use an extra 18.9% of scratch space.

subsequent page out operations go to this set, until it fills up. In the meantime, a whole set may be freed (due to rule 1 before). If not, rule 2 is applied again.

This method works very well as long as the page out requests invalidate pages that will eventually lead to empty parity sets. If there are no empty parity sets, then almost-empty parity sets are selected, which are the next best thing. If there are neither empty, not almost-empty sets, half-empty, or almost-full sets are being used, which increases the cost of cleaning.

Logging - Sets: In this policy we reclaim space in the following two cases:

1. Rule 1: same as previously.
2. Rule 2: When we run out of space, we find the parity set with the largest number of invalid pages. The valid pages in the set are sent to other non-full sets, and a new empty parity set is created.

The main difference with the previous method is that the Greedy policy uses half-full sets (most of the time), while this method empties the parity sets (by transferring the valid pages to other half-empty sets) before using them.

Both logging policies work well when empty parity sets are created at no extra cost (by Rule 1.). To achieve this, both logging policies must have a little more space than the minimum necessary to store memory pages. This space will be referred to as *scratch* space. Logging polices will take advantage of scratch space to keep older version of the pages, in order to avoid creating empty sets by moving pages.

3 Performance Results

3.1 The environment

We use trace-driven simulation, to measure the performance of the proposed policies. The traces are page-in and page-out requests that we gather from realistic applications running on a real environment. We modified the kernel of the DEC/OSF 1 operating system to produce a trace record every time its pager performs a page in, or a page out operation.[6] Then, we feed these traces to a simulator that simulates a reliable network memory system.

The reliable network memory system consists of a set of memory and parity servers. Every eight memory servers there is one parity server. Each server has 16 MBytes of free memory. The page size is set to 8 KBytes.

The applications we use are:
`MVEC`: Matrix vector multiplication of a 2000×2000 matrix.
`GAUSS`: Gaussian elimination on a 1900×1900 matrix. [7]
`SORT`: Sorting of an array of 32 Mbytes, using the standard quicksort algorithm.
`NBODY`: N-body simulation of 140.000 particles.
`FFT`: Fast Fourier Transform on 900.000 elements.

The policies we simulate are:

- `MIRRORING`: For each page there are two copies of it in two different memory servers. Each page out must be sent to both servers. If one of the server crashes, the other can still serve requests for the page.
- `PARITY`, `DELAYED_PARITY`, `LOGGING-GREEDY`, and `LOGGING-SETS`: These are the policies described in sections 2.2.2 through 2.2.4.
- `NO-RELIABILITY`: For comparison purposes, we have included the policy that provides no reliability at all: for each page out only the page is sent to the memory server, no extra pages are transferred, no parity is computed, and no extra memory is needed.

The performance metric we use for our comparison is the *number of page transfers*: we count only the number of page transfers associated with page outs, because none of the above policies makes any extra page transfers per page in. When it is important, we normalize the page transfers count by dividing it with the number of page outs, and we get the *page transfers per page out* metric.

3.2 Policy Comparison

In the first experiment we do a comparison of all the policies on all applications. We assume that for every eight

[6] The interested reader may find the modifications to the kernel, the application code and the traces from ftp://ftp.ics.forth.gr/pub/pager.

[7] Because the completion times of the simulation were too long, we simulated only the first 10 minutes of real application execution time.

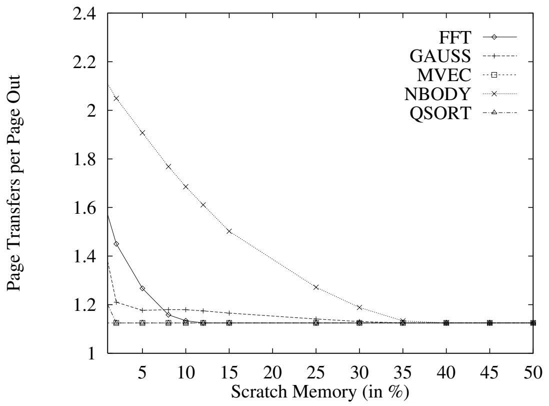

Figure 3: **Performance of LOGGING-SETS as a function of the amount of the scratch memory available.** Eight memory servers - one parity server

memory servers there is one parity server. Thus, the memory utilization of the PARITY and the DELAYED_PARITY policies is $8/9 = 88.8\%$ (the rest 11.1% holds parity frames). The memory utilization of the LOGGING policies were set to 70% which implies that 70% of the page frames are used to store pages, 11.1% are used to store parity pages, and the rest 18.9% is used as scratch memory to store invalid pages, and increase the flexibility and effectiveness of the LOGGING polices. The number of page transfers (both page outs, mirror and parity transfers, but excluding page ins) for each application running under each policy is shown in figure 2. The first thing we notice is that MIRRORING and PARITY induce the same number of page transfers, which is expected. In all cases, both MIRRORING and PARITY have the worst performance. The next best policy is DELAYED_PARITY which manages to gather several pages that belong to the same parity set at a time, and thus, it reduces the number of page transfers to the parity server. Its performance improvements over simple PARITY range as high as 30% for the FFT application.

Both LOGGING policies represent a significant improvement over simple PARITY . The reason is simple: Both LOGGING policies page out (almost) entire parity sets at a time, in which case only only one (or a few) pages per set need to be sent to the parity server.

The interested reader should notice that the LOGGING policies perform very close to the NO-RELIABILITY: only 11% worse for most applications (30% for NBODY).[8] This implies that the cost of providing a reliable network memory system is only 11%-30% higher than the cost of providing an unreliable network memory. Finally, we notice that the two LOGGING policies are comparable for all applications. They perform the same for FFT, MVEC and QSORT, and within a 5% of each other for the N-BODY and the GAUSS applications.

3.3 The Effect of Scratch Memory

[8] Note that all policies based on parity can not perform better than 11% from the NO-RELIABILITY policy, because 11% of the pages are parity pages.

Application	Scratch Memory Percentage
FFT	11.7
GAUSS	33.2
MVEC	0.07
NBODY	38.3
QSORT	1.2

Table 1: **Percentage of scratch memory needed by the applications to avoid cleaning**. The same percentage holds for both LOGGING policies.

Our next set of experiments investigates the performance of LOGGING policies as a function of the scratch memory space they are given. We expect that the more scratch space the LOGGING policies are given, the better their performance will be, because the system will have to clean parity sets and regain space rather infrequently. The performance results for the LOGGING-SETS policy are shown in figure 3. The x-axis represents the amount of scratch memory space the policy is given. To simplify the presentation of the results, the x-axis does not include the amount of memory used to store the parity pages. The y axis represents the number of pages transferred for each page out that the system performs. This number should be at least one (the page itself must be transferred to its server). We see that the performance improves (the number of page transfers per page out decreases) with the available scratch memory, as expected. However, it is surprising to see that for the MVEC and QSORT applications almost no scratch space is needed. Both applications have a regular page out pattern which results in emptying entire parity sets at a time, thus, needing little extra scratch space. In figure 3 we see that when the scratch memory is very small, applications NBODY, FFT, and GAUSS suffer some performance overhead, which is more pronounced for the NBODY application. A simple look at the application's page out patterns explains why. All three applications do not have a repeatable pattern of page outs (unlike QSORT). This phenomenon is especially pronounced in NBODY because its accesses are almost random, forcing the pages that are sent to backing store to follow an irregular pattern. Thus, page outs of the NBODY application result in several half-full parity sets, which need to be cleaned frequently, resulting in a large number of page transfers. However, we were surprised to see that even for an application like NBODY, the percentage of scratch memory it needs to achieve its maximum performance is only 35%. Thus, if we give an extra 35% of scratch memory to the LOGGING-SETS policy, all our applications (including those with the worst memory access pattern) perform only 1.12 page transfers per page out, or a mere 11% worse than the policy that provides no reliability at all. It is surprising to see how inexpensive (both in memory utilization and in page transfers) it is to achieve a reliable network memory system using the LOGGING policies. Table 1 presents the amount of scratch memory needed by all applications in order to avoid cleaning all-together. MVEC and GAUSS need almost no extra space (<1.2%), FFT needs little extra space (<12%), and GAUSS and NBODY need a moderate amount of extra space (33% and 38% respectively).

4 Related Work

Recent research that studies the application of network memory for paging in workstation clusters includes [1, 3, 7, 10], and [8] for large scale multiprocessors. Other research groups focus on applications of network memory for file system caching [1, 4, 5, 6]. In all previous work, however, reliability is either ignored, or achieved by storing dirty pages on magnetic disks, thus limiting the performance of the system by the available disk bandwidth. Our approach instead, is to use remote main (volatile) memory to reliably store through redundancy an application's data. Our experiments suggest that reliability (through redundancy) can be achieved at a small increase in memory size and the number of page transfers required.

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) [2].

Our work bares some similarity with logging and cleaning methods used in log-based file systems, but the mechanisms and policies in the two cases are different [11].

5 Conclusions

Workstation clusters give rise to a new level of memory hierarchy that consists of all the main memories of all the workstations in the cluster: the network memory. In this paper we present a log-based approach to construct a reliable network memory system that is resilient to individual workstation failures. Our approach can be used to implement reliable file systems, paging systems, and database caching systems. In this paper we propose three novel policies for network memory: the `DELAYED-PARITY` and two log-based policies. We have done several trace-driven simulation experiments based on traces of real applications running on a real environment. Based on our experimental results we conclude:

- *Reliable network memory systems impose little run-time overhead*: their performance is (most of them time) within 11% of the systems that provide no reliability at all (see figure 2).
- *Reliable network memory systems are inexpensive*: they consume a small percentage of the total memory of the system. Given that a significant portion of memory lies idle most of the time in a workstation cluster, dedicating a small percentage of (an otherwise idle) memory to reliability is a small price to pay.
- *Log-based parity policies outperform all other approaches*: our performance experiments suggest that the log-based approach to a reliable network memory outperforms both simple parity-based and mirroring policies.

Acknowledgments

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

We would like to thank Catherine Chronaki for useful comments in earlier drafts of this paper, and G. Dramitinos for providing the traces, as well as useful feedback in the design of the `LOGGING` policies.

References

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[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] 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] 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.

[5] 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.

[6] 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.

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

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

[9] A. Leff, J.L. Wolf, and P.S. Yu. Replication Algorithms in a Remote Caching Architecture. *IEEE Transactions on Parallel and Distributed Processing*, 4(11):1185–1204, November 1993.

[10] E.P. Markatos and G. Dramitinos. Implementation of a Reliable Remote Memory Pager. In *Proceedings of the Usenix Technical Conference*, January 1996.

[11] 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.

[12] J.M. Smith. A Survey of Process Migration Mechanisms. *Operating Systems Review*, 22(3):28–40, July 1988.