RELIABILITY FOR NETWORK SWAPPING SYSTEMS THAT SUPPORT MIGRATION OF REMOTELY SWAPPED PAGES

Tia Newhall, Benjamin Mitchell, Julian Rosse
Swarthmore College, Swarthmore, PA, USA

ABSTRACT

Network swapping systems allow individual cluster nodes with over-committed memory to use the idle memory of remote nodes as their backing store, and to swap their pages over the network. As the number of nodes in a cluster increases, it becomes more likely that a node will fail or become unreachable, making it important that such a system provide reliability support. Without reliability, a single node crash can affect programs running on other cluster nodes by losing remotely swapped page data that was stored on the crashed node. Our network swapping system, Nswap, has design features that complicate reliability: swapped pages can migrate from one node to another in response to changes in a node’s local memory needs. As a result, reliability schemes that rely on fixed placement of page and reliability data are not applicable to our system. Our reliability solutions solve the unique challenge of providing reliability to network swapping systems that both support dynamic changes to the size of remote RAM swap space and support migration of remotely swapped page data. Results show that even though our Mirroring reliability scheme adds time and space overhead to Nswap, it still outperforms swapping to disk by a factor of up to 8.2. Our dynamic Parity scheme will provide reliability with minimal time and space overhead.

KEY WORDS
Cluster Computing, Network RAM, Reliability.

1 Introduction

Nswap [10] is our network swapping system for Linux clusters. Network swapping, using remote idle cluster memory as backing store, is motivated by two observations: first, network speeds are getting faster more quickly than are disk speeds, and this disparity is likely to grow [8]; second, there is often a significant amount of idle memory space in the cluster that can be used for storing remotely swapped pages [1, 3, 9]. Thus swapping to local disk will be slower than transferring pages over a faster network and using remote idle memory as a “swap device”.

Network swapping systems must provide reliability support to remotely cached pages so data can be recovered when a cluster node fails or becomes unreachable. Without reliability, failure of a single node can affect other cluster nodes; if a node fails, pages cached from other nodes are lost. As the size of a cluster grows, the mean time to node failure decreases, making reliability more important.

Any reliability support will add extra time and space overhead to remote swapping; reliability information uses cluster idle RAM space that could be used for storing remote pages and typically requires some extra computational and message passing overhead. RAID-based[11] reliability schemes are likely to provide a good balance of reliability and cost in addition to providing some flexibility. However, there are two unique characteristics of the Nswap system that make implementing a strict RAID-like reliability scheme difficult. First, Nswap is designed to adapt to each node’s local memory needs. The amount of local RAM space made available for remote swapping (what we call the Nswap Cache) grows and shrinks in response to local processing needs; the amount of ”swap” space is not fixed in size, and an individual node’s Nswap Cache capacity changes over time. The second difficulty is caused by Nswap’s support for migrating remotely swapped pages between cluster nodes. Remote page migration occurs when a node needs to reclaim some of its Nswap Cache space for local processing, but this can further complicate reliability support. For example, two pages in the same parity group could end up on the same node, resulting in a loss of reliability for that parity group.

The dynamic nature of Nswap is an important and powerful feature. Without it, nodes would be forced to relinquish a fixed amount of their RAM for remote caching, resulting in an increase in the amount of cluster-wide swapping. However, this complicates reliability schemes. Our novel reliability solutions solve the problem of efficiently providing reliability in such an environment. We use dynamic parity group membership to better match the dynamic nature of the Nswap system, and our migration protocol is modified to ensure that reliability is maintained. Our solutions are designed to add minimal time and space overhead to normal page swap-ins and swap-outs, with a particular emphasis on reducing added costs to the nodes that are actively swapping.

In the Section 2 we present related work, in Section 3 we give a brief overview of Nswap, in Section 4 we present our reliability algorithms, in Section 5 we present results, and in Section 6 we conclude.
2 Related Work

There have been several previous projects examining the use of remote idle memory as backing store for nodes in networks of workstations [2, 5, 9, 7, 4, 12, 6]. Feeley et al. [5] implement a system that views remote memory as a cache of network swapped pages that are also written through to disk; this system only remotely caches clean pages. Their write-through reliability scheme is easy to implement, but it results in a significant amount of disk I/O overhead. In our previous study [10], we found that workloads with file I/O were negatively affected by disk swapping. By adding disk swapping overhead to every remote page out, write-through will interfere with these types of workloads in the same way as does swapping to disk. A reliability scheme that uses remote idle memory instead of disk will be more efficient.

Markatos and Dramitinos [9] describe reliability schemes that use RAM for storing reliability data, avoiding the overhead of a write-through scheme. They implement several RAID-based solutions and show that their parity logging scheme performs best. In parity logging, clients compute parity pages as they swap out pages and then send complete parity pages to a parity server. By having the client compute the parity page, the number of page transfers to the parity server is reduced significantly. However, parity logging requires that clients find distinct servers to take pages in the same parity group, which could slow down swap-outs. Their solution also requires that old page data stay on servers until all pages in a parity group are replaced, wasting RAM space that could be better used for storing active pages. In addition, their solution requires fixed-placement of remote page data on servers, and can result in remotely swapped page data being swapped to servers’ disks.

3 Nswap

Nswap is a network swapping system for Linux clusters. It is implemented as a loadable kernel module that transparently provides network swapping on cluster nodes. Nswap is designed to be efficient in both time and space, to scale to large clusters, and to adapt to changes in a node’s local memory needs. Individual cluster nodes run both a multi-threaded Nswap client and a multi-threaded Nswap server (Figure 1). The Nswap client is active when a node is actively swapping its pages of memory. The Nswap server is active when a node is accepting remote pages that it caches in its idle RAM space (the Nswap Cache). The size of the Nswap Cache grows and shrinks in response to a node’s local memory needs. The client is written as a device driver for our pseudo swap device. Client code is triggered when the kernel swaps a page in from or out to our device. The server is responsible for managing Nswap Cache space and for adjusting the size of the Nswap Cache in response to local processes’ memory needs. Nswap allows nodes to change their roles dynamically based on their current local processing needs; when a node needs more memory space than it has available locally, it swaps its pages to other nodes with idle RAM space. At any time an individual node is acting as an Nswap server or as a client, but typically not as both simultaneously.

The Nswap communication protocol defines how pages move between cluster nodes. The communication protocol has five types of requests: PUTPAGE, GETPAGE, PUNTPAGE, UPDATE and INVALIDATE. When Nswap is used as a swap device, the kernel swaps pages to it. The client receives a swap-out request and initiates a PUTPAGE to send the page to a remote server. The client has a data structure called the shadow slot map that shadows the kernel’s slot map for our swap device. In it we store extra information about which remote servers store which swapped pages. When the kernel issues a swap-in request for a page, we use the information in the shadow slot map to issue a GETPAGE request to the remote server storing the page (Figure 1). If the workload distribution changes, it may be necessary for a server to reduce its Nswap Cache size using PUNTPAGE to migrate pages to other servers. Moving a page from one server to another involves an UPDATE request to alert the client to the new location of the page, and an INVALIDATE request from the client to the old server to inform the old server that it can drop its copy of the page (see Figure 2). Because a swap-in leaves the page data on the remote server, old page data can accumulate on servers. We handle this by trying to re-use the old server when a swap slot is re-used (the old server just overwrites the old page data with the new), or by issuing an INVALIDATE to the old server before issuing a PUTPAGE to the new server. In addition, we have a garbage collector thread that periodically cleans old page data out of the system.

GETPAGE and PUTPAGE are designed to be as fast as possible for the client who is currently swapping. If a client makes a bad server choice for a PUTPAGE, the Nswap servers handle it through page migration rather than forcing the client to make a better server choice, which
Figure 2. Page Migration in Nswap. Node A acts as an Nswap client, and Nodes B and C act as servers. A PUNTPAGE from server B to server C (1), triggers an UPDATE from server C to client A (2), which in turn triggers an INVALIDATE from client A to server B (3).

4 Reliability Algorithms

The most efficient reliability schemes for a network swapping system will use remote idle memory and avoid using cluster disks for reliability. We present two RAID-based schemes for implementing reliability in Nswap that solve the reliability problem in the presence of page migration and changes in remote RAM storage capacities.

4.1 Mirroring

A mirroring scheme places two identical copies of a page at two remote servers. If one copy becomes unrecoverable, the other should be available. Mirroring provides reliability against the failure of a single server node.

To implement mirroring, we add additional information to the shadow slot map to keep track of the back-up server for each remotely swapped page, and we modify the Nswap protocols. On a swap-out request, the client attempts to locate two different remote servers to accept the page. If two such servers are found, the client uses the normal PUTPAGE protocol to transfer a copy of the page to each and stores the ID’s of the primary and back-up servers in the shadow slot map entry corresponding to the page. If only one accepting remote server can be found, the client does a PUTPAGE to this server and either writes a copy of the page through to local disk, setting the shadow slot map entry to reflect a write-through, or doesn’t provide reliability support for this page—Nswap is configurable at runtime to select either behavior. If no remote server can be found (an unlikely event), the page is written through to local disk and the shadow slot map entry is set to reflect this condition.

If nothing goes wrong, GETPAGE proceeds exactly as it does without reliability when using mirroring; the client simply fetches the requested page from the primary server using the normal GETPAGE protocol. If the primary server is unreachable, the client attempts to retrieve the back-up copy of the page.

We make a slight modification to the PUNTPAGE protocol to support mirroring. We add a low-priority version of the PUNTPAGE request that adds a check by the new server to see if it already stores a copy of the page before it accepts the migrated page. Without this check, reliability can be lost if a server accepts a migrated page when it already stores the other copy of the page. We use a low priority version because moving a page off the server due to reliability constraints is less time critical than a regular PUNTPAGE, so care can be taken to ensure that both copies of a page don’t end up on the same server. If a server cannot be found, then the page is no longer mirrored in our system. When a client receives an UPDATE message for a page, it checks if it is for the primary or back up copy and updates the appropriate entry in the shadow slot map before sending the old server an INVALIDATE message.

The mirroring scheme’s biggest advantage is its simplicity. Servers are unaware that they are holding back-up rather than primary copies of pages. The additional time to send the back-up page is much smaller than the time to write-through to local disk. However, the additional space requirements for mirroring are less desirable. Since we send out two pages for each page swapped, the amount of Nswap Cache space for storing pages is reduced by half.

4.2 Parity

While write-through and mirroring based schemes offer good reliability, they are also very expensive in time or space. A better option is to use a parity-based reliability scheme that, like mirroring, uses remote memory rather than disk for storing reliability data. However, parity uses much less remote memory space for storing reliability data than does mirroring.

Parity has been used before in several contexts [9][11] with great success. However, since Nswap supports dynamic resizing of remote cache space and supports remote page migration, we cannot directly apply previous parity solutions to Nswap. Our solution uses dynamic parity group membership to better fit the dynamic nature of the Nswap system. In our scheme, a parity group consists of some number of pages and one parity page. The parity page is created by computing the exclusive or (xor) of the other n pages. Pages are added to and removed from the parity group by xor’ing them with the current parity page. At any time a parity page could represent parity information for any number of data pages; the size of a parity group is dynamic.

Parity Architecture

For small to medium sized clusters of m nodes, m − 1 of those nodes function as normal Nswap servers and clients.
The remaining node, the parity server, is used exclusively to store parity pages and state necessary for the recovery algorithm. This organization allows normal nodes to behave in a way that is altered only slightly from the normal Nswap protocols, and it avoids having to maintain distributed global state, which would be necessary if any node could hold parity data.

If \( m \) is very large, and a single parity server does not provide enough reliability, we partition the cluster into smaller parity sets, where each set is a group of nodes and their corresponding parity server (Figure 3). This limits the flexibility of the system slightly, but partitioning is needed only when the total capacity of the system is huge, in which case partitioning that capacity should not be a big problem.

Within each parity set, the number of parity groups is fixed\(^1\) to ensure that the memory capacity on the parity server is not exceeded, which would cause the parity server to swap. Having a fixed number of parity groups also allows us to have a global method for determining to which parity group a page belongs. We use a simple hashing function based on the IP address of the client who owns the page and the slot number of that page to determine a page’s parity group. The advantage of this method is that the parity group to which a given page belongs can be calculated on-the-fly, avoiding storing parity group information and avoiding adding it to the message passing overhead; since the parity server must keep track of the client IP and slot number anyway, there is no need to send, nor to store, the parity group assignment. The only additional information stored on each Nswap server is a list of parity groups of the server’s currently cached pages. The list is used to ensure that a single server does not cache multiple pages from the same parity group. No additional state need be kept by Nswap clients to implement parity.

While little state is added to the Nswap server or clients beyond what exists in the default Nswap scheme, the parity server must keep track of several things in order to perform its role. First, the parity server requires a set of parity pages. Additionally, for each parity page a list of \( \{\text{clientIP, serverIP, slot num, timestamp}\} \) tuples is kept, one tuple per page currently XORed into the parity page. This Page Data List (PDL) starts out empty, and the parity page starts zeroed; as pages are sent to the parity server, each is XORed into the parity page, and a tuple for the page is added to the PDL. Each entry in the PDL has a dirty bit, and there is a dirty count kept with each parity page. These are used to detect if the parity page is in an inconsistent state during recovery. For example, if a new version of a page has been XOR-ed with the parity page, but its old version has not yet been XOR-ed out, the parity page is inconsistent and the recovery algorithm must wait until the parity page is consistent before the lost page data can be recovered.

\(^1\)The number of parity groups is configurable off-line to meet the reliability needs and to match the size of the parity server’s RAM. However, once the Nswap module is loaded, the number of parity groups remains fixed.

### Nswap Protocols with Parity

The parity server must receive copies of pages that are remotely swapped. When an Nswap server receives a PUTPAGE from a client, it sends a copy of the page to the parity server. Upon receiving the page, the parity server calculates the parity group, XORs the page into the correct parity page and adds a tuple to the PDL for the page. Successive PUTPAGE requests that map to the same parity group result in the addition of page information to the PDL, and the XORing of the page data with the parity page (Figure 3).

When a new PUTPAGE request uses the same slot as an old cached page, the old page no longer has to be remotely cached since it has been overwritten in local memory. In this case, the default behavior is to send the page to the same server which holds the older copy; the server then knows that the old page can be overwritten with the new one. The parity scheme requires that before the old page is overwritten, the server must first send a copy of the old page to the parity server with an INVALIDATE tag. Upon receipt, the parity server removes the appropriate entry from the PDL and XORs the old page out of the parity page. The dirty bit will be set if the parity server receives a PUTPAGE for a page before it receives an INVALIDATE for the old version of the page. Before setting the dirty bit, the parity server can check if the old server is reachable, and if not, it can invoke the recovery algorithm to restore the old page data and XOR it out of the parity page.

A PUNTPAGE request begins normally through the point when the client sends the standard INVALIDATE to the old server telling it to drop the page. At this point, the parity server must also be sent an UPDATE message so that it knows where to get the page for recovery purposes (Figure 4).

One of the goals of Nswap is to be as fast as possible to the node that is currently swapping. For this reason, we never want to reject a PUTPAGE or PUNTPAGE. However, after accepting a page, a server checks the parity group of that page against a list it keeps of all parity groups of pages it currently stores. If it already has a page from the same parity group as the incoming page, the server initiates a LOW-PRIORITY-PUNTPAGE. As in the Mir-
The protocols must also detect and correctly handle simultaneous, conflicting operations (e.g., a new use of the slot with a PUNTPAGE and a PUNTPAGE for the old use of the slot). Time stamp information that is sent with messages and kept in the shadow slot map and PDL helps clients and servers detect and correctly handle these rare but complicated cases.

Evaluation of the Parity Scheme

Our parity solution is different from previous parity schemes because the members of a parity group are not fixed, and the storage location of pages in a parity group can change. These differences affect performance by requiring a small amount of extra state to be kept at each Nswap server, and by requiring one node to act as the parity server. Compared with mirroring, this extra space is small. By offloading the work of XORing pages and tracking page locations to the parity server, we minimize the computational impact on the Nswap clients and servers. By having the Nswap server send pages to the parity server, the Nswap client has no extra time overhead added to swap-outs.

Because any page can be part of a parity group, the scheme scales well. There is no limit to the number of clients and servers, other than the fact that too high a ratio of servers to parity groups will result in multiple pages from the same group ending up on the same machine. The number of parity groups in the system, in turn, is limited only by the amount of memory available on the parity server. Our solution scales to large clusters by partitioning cluster nodes into parity sets. It is possible, however, that the parity server could be a bottleneck to normal PUTPAGES. If this is the case, then our scheme can be modified by having Nswap servers perform some buffering of parity information before sending it to the parity server. Obviously, this can result in a loss of reliability if a server fails before sending its buffered reliability data to the parity server. However, a reasonable balance between maintaining reliability and reducing messaging to the parity server can be achieved by periodically flushing the buffer at each server.

5 Results

We measured total execution time of three workloads comparing swapping to disk, Nswap with no reliability support, and Nswap with our Mirroring reliability scheme. Workload 1 consists of a process that performs large sequential writes and reads. It is designed to be the best case for swapping to disk because it minimizes total disk head movement within the swap partition. Workload 2 consists of a process that performs random writes followed by random reads to a large chunk of memory. It stresses disk head movement within the swap partition only. Workload 3 adds a process that performs file I/O to the Workload 2 program. It further stress disk head movement between the swap and file partitions. Table 1 shows total execution times of each workload for each of the three swapping configurations.
Our results show that even though Mirroring adds 18% to 101% overhead to Nswap, it still outperforms swapping to disk for workloads 2 and 3 (by a factor of 4.1 and 8.2). On clusters running more than a single process per node, Workloads 2 and 3 are more representative of a typical cluster node’s swapping behavior. As a result, clusters running Nswap with Mirroring will perform better than clusters that use disk for swap. However, because Mirroring uses twice as much idle RAM space as Nswap with no reliability, our Parity scheme will be a more efficient solution for providing reliable network swapping.

6 Conclusions

Reliability is an important feature of a network swapping system, particularly as the cluster size increases making node failure more frequent. Without reliability, a single node failure can affect programs running on other nodes. Our novel RAID-based reliability schemes solve the problem of efficiently providing reliability to network swapping systems that both support dynamic changes to the size of remote RAM swap space, and support migration of remotely swapped page data. Preliminary results measuring Nswap with Mirroring show that it performs significantly better than swapping to disk for most workloads, and adds little time overhead to Nswap without reliability support. Our current effort involves more testing of the Mirroring scheme, and implementing and testing the Parity scheme.

References


