Reliable Adaptable Network RAM

Tia Newhall, Daniel Amato, Alexandr Pshenichkin

Computer Science Department, Swarthmore College
Swarthmore, PA 19081, USA

***Abstract*—We present reliability solutions for adaptable Network RAM systems running on general-purpose clusters. Network RAM allows nodes with over-committed memory to swap pages over the network, storing them in the idle RAM of other nodes and avoiding swapping to slow, local disk. An adaptable Network RAM system adjusts the amount of RAM currently available for storing remotely swapped pages in response to changes in nodes' local RAM usage. It is important that Network RAM systems provide reliability for remotely swapped page data. Without reliability, a single node failure can result in failure of unrelated processes running on other nodes by losing their remotely swapped pages. Adaptable Network RAM systems pose extra difficulties in providing reliability because each node's capacity for storing remotely swapped pages changes over time, and because pages may move from node to node in response to these changes. Our novel dynamic RAID-based reliability solutions use idle RAM for storing page and reliability data, avoiding using slow disk for reliability. They are designed to work with the adaptive nature of our Network RAM system (Nswap), allowing page and reliability data to migrate from node to node and allowing pages to be added to or removed from different parity groups. Additionally, page recovery runs concurrently with cluster applications, so that cluster applications do not have to wait until all data from a failed node is recovered before resuming execution. We present results comparing Nswap to disk swapping for a set of benchmarks running on our gigabit cluster. Our results show that reliable Nswap is up to 32 times faster than swapping to disk, and that there is virtually no impact on the performance of applications as they run concurrently with page recovery.**

I. Introduction

In general purpose clusters and in networks of workstations there are likely to be imbalances in RAM usage across nodes as several parallel or parallel and sequential applications are simultaneously running on the system. Even among the processes of a single parallel application there can be imbalances in memory usage [7], and these memory imbalances can cause slowdowns in performance [7], [5]. Performance slowdowns will be significant if these imbalances result in swapping on some of the nodes. Swapping is more likely to occur when some nodes are running applications that process large amounts of data such as parallel scientific or multimedia applications.

Network RAM systems take advantage of imbalances in RAM usage and allow nodes with over-committed RAM to locate and use the idle RAM of remote nodes as backing store; pages are "swapped out" over a fast network and stored in the idle RAM of other nodes.

Using remote idle memory as a backing store for networked and cluster systems is motivated by the observation that network speeds are improving more quickly than disk speeds [11]. This disparity will likely continue to grow because disk speeds are limited by mechanical movement. As a result, swapping to local disk will be slower than using remote idle memory as a "swap partition" and transferring pages over the faster network. Further motivation for network RAM is supported by several studies [1], [3], [13] showing that large amounts of idle cluster memory are almost always available (even when some nodes are overloaded), and that large chunks are available for significant amounts of time. As a result, a Network RAM system should be able to find usable amounts of idle RAM to store swapped pages.

The amount and location of idle RAM in a cluster changes over time with changes in each node's workload. Thus, it is important that a Network RAM system adapt to these changes, releasing RAM back to the local paging system when needed by local processes, and finding and allocating RAM when it becomes idle. Without this type of adaptive behavior, a fixed amount of RAM would have to be permanently allocated on each node, which could lead to an increase in the amount of cluster-wide swapping.

As the number of nodes in a cluster increases, it becomes more likely that a node will fail or become unreachable. In a Network RAM system a single node failure not only results in losing processes running on that node, but can additionally affect unrelated processes running on non-failed nodes by losing their remotely swapped page data. It is therefore important that a Network RAM system provide reliability for remotely swapped data.

Any reliability support will add extra time and space overhead to network swapping. A good reliability solution will minimize these overheads. For example, a write-through scheme is easy to implement, but it can significantly slow down network swapping because every swapped page is also written to disk. In addition, it will interfere with local applications doing disk I/O [15]. A better reliability solution will use idle RAM for reliability data.

We present reliability solutions that we have implemented in Nswap [15], our network RAM system for Linux clusters and networked Linux machines. Nswap transparently provides network RAM to applications running on these systems. Nswap supports dynamic growing and shrinking of each node's Nswap Cache (the set of RAM pages currently allocated for storing remotely swapped page data) in response to the node's local memory use. When an Nswap node shrinks its Nswap Cache, it may migrate some of the remotely swapped pages it currently stores to other Nswap nodes with available Nswap Cache space. Nswap only reverts to swapping pages to disk

when there is no available idle RAM in the cluster.

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 storing remotely swapped page data, which would result in more swapping on a node when its local processes needed more RAM space. However, the dynamic growing and shrinking of Nswap Cache space complicates reliability schemes as pages and reliability data can migrate from node to node.

Our novel dynamic reliability schemes solve the problem of efficiently using idle remote RAM for reliability in an environment where capacity and page placement can change. We use an approach based on RAID [17] that stripes page data and reliability data across idle cluster RAM, and avoids using slow, local disk for reliability. Our solutions solve the problem of efficiently providing reliability for remotely swapped page data in an adaptable Network RAM system. Our solutions do not require fixed placement of page or reliability data, nor do they require fixed parity group assignment or fixed-size capacity of each node's available RAM for storing remotely swapped data. We use dynamic parity groups where parity group membership and parity and data page placement is not fixed; a page can move from one parity group to another, a parity group's pages can move from node to node, and the size of a parity group can change. Our solutions are designed to be time and space efficient, adding minimal overhead to normal page swapping with an emphasis on minimizing overhead on nodes that are actively swapping. In addition, our solutions are designed to scale to large-sized clusters.

In section II we discuss related work. In section III we present an overview of Nswap's implementation. In section IV we present our dynamic reliability solutions. In section V we present results comparing Nswap with no reliability, reliable Nswap, and disk swapping for several workloads running on our gigabit cluster. Finally, in section VI we conclude and discuss future directions for our work.

II. Related Work

There are several projects that examine using remote idle memory as backing store in clusters or networks of workstations [20], [16], [6], [2], [13], [14], [12], [8], [10]. Reliability issues are addressed in three of these projects. The first, Feeley et. al. [14], views remote memory as a cache of network swapped pages that are also written through to disk; only clean pages are remotely cached. Write-through is easy to implement, but it adds much more disk I/O. In our previous study [15] we found that disk swapping resulted in a significant slowdown over network swapping, and that workloads with file I/O were even more negatively affected by disk swapping. By adding disk swapping overhead to every swap-out, write-through will interfere with these types of workloads in a similar way as disk swapping does.

The second, Iftode et. al [10], presents a memory server for multicomputers. Network RAM is added as an extra layer in the memory hierarchy between RAM and disk. Their modified virtual memory system pages from RAM to Network RAM, and Network RAM to disk. They discuss several design alternatives for such a system, some of which take fault tolerance into consideration. In their design for shared virtual memory systems, duplicate copies of a page can be stored in network RAM, allowing for page data to be recovered when one copy is lost. Because the duplicate copy mechanism is under the control of the virtual shared memory system, it determines fault tolerance for pages stored on memory servers. However, it does so by duplicating full page data, resulting in using twice as much RAM to store pages. Additionally, they discuss a write-through scheme for fault tolerance.

The third, Markatos and Dramitinos [13], describes reliability schemes that use RAM for storing reliability data. They implement several RAID-based solutions and show that their parity logging scheme performs best. In parity logging, clients locally compute a parity page as pages are swapped out over the network. Complete parity pages are then sent to a parity server. By having the client compute the parity page, the number of page transfers to the parity server is significantly reduced. However, their solution uses a fixed placement of page and reliability data across servers, it requires that the client keep track of parity group information, and it 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. By requiring fixed-placement of remote page data on servers, remotely swapped page data can be swapped to servers' disks. We use a technique similar to their parity logging to reduce the number of additional messages on page swap-outs. However, our solution differs from theirs in several ways. First, our solution does not require that clients keep parity group state. Second, we do not require fixed parity group membership. Third, we do not require fixed page or parity page placement on servers. Fourth, we do not swap pages to a server's disk. Fifth, we do not require that pages from exited processes remain on remote servers until all pages in their parity group have been freed. And finally, our solution does not have the potential bottleneck of a single parity server.

III. Nswap Overview

Nswap [15] is our Network RAM system for Linux clusters and networks of Linux machines. It is implemented as a loadable kernel module that is easily added as a swap device to cluster nodes and runs entirely in kernel space on an unmodified [1] Linux 2.6 kernel; Nswap transparently provides network swapping to applications, requiring no special re-compiling or linking with special libraries. Nswap is designed to be efficient, to adapt to changes in nodes' RAM use, and to scale to large-sized clusters.

Each Nswap node is an equal peer running both the Nswap client and the Nswap server (shown in Figure 1). The client is active when a node is swapping. The server is active when a node has idle RAM space that is available for storing page

[1] Currently, we require a re-compile of the the kernel to export two kernel symbols so that our module can read the kernel's swap slot map for our "device", but no kernel code is modified

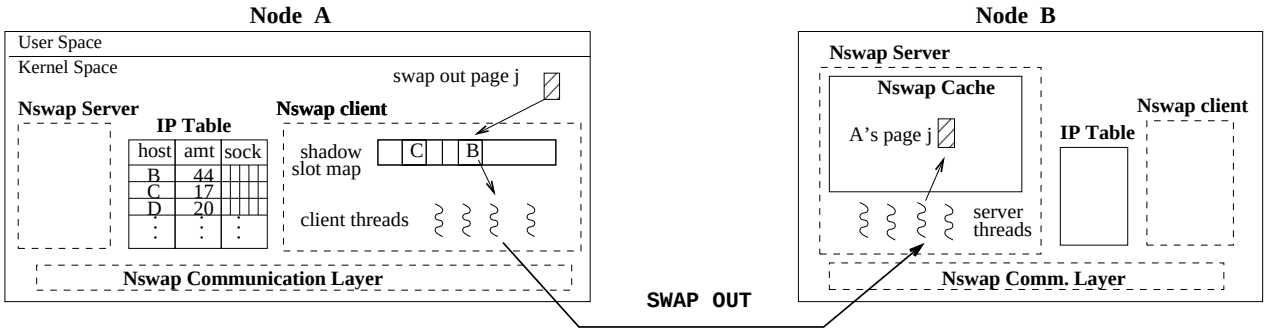

Fig. 1. Nswap System Architecture. *Node A shows the details of the client including the shadow slot map used to store information about which remote servers store A's pages. Node B shows the details of the server including the Nswap Cache of remotely swapped pages. In response to the kernel swapping out (in) a page to our Nswap device, a client thread issues a SWAPOUT (SWAPIN) request to write (read) the page to a remote server. The server storing the page is encoded in the shadow slot map entry.*

data swapped to it from remote nodes. At any point in time a node is acting either as a client or a server, but typically not as both simultaneously; its role changes based on its current local RAM usage. The client and server send messages using the Nswap Communication layer that is an interface on top different network implementations.

Nswap is designed to scale to large clusters using an approach similar to the Mosix [4] design for scalability. To find available remote swap space, each node uses only its own local information that does need to be complete nor completely accurate, and each node is responsible for managing just its portion of local RAM that it currently has available for storing pages swapped from other nodes. Because there is no central authority managing network RAM allocation, Nswap will easily scale to large sized clusters.

The multi-threaded Nswap client is implemented as a device driver for our pseudo-swap device. A client thread is activated when the kernel makes a swap-in or swap-out request to our swap "device" just as it would to a driver for a swap partition on disk. For any swap device, the kernel has a data structure called a slot map used to keep track of allocated swap space on the device. The Nswap client keeps additional information about each slot of swap space in a data structure called the shadow slot map. There is one shadow slot map entry per kernel slot map entry. Each entry stores the ID of the Nswap server storing the page and some additional data encoding usage of the swap slot that is used to detect and handle conflicting operations to the same slot.

When a client thread receives a swap-in request from the kernel, it looks-up the server ID from the corresponding shadow slot map entry and sends a SWAP-IN request to the Nswap server storing the page. When the client receives a swap-out request from the kernel, it finds a good Nswap server candidate using information that it keeps in a small data structure called the IPTable, and sends a SWAP-OUT request to the server (see Figure 1). Only when there is no available Nswap Cache space in the system is the page swapped to disk.

The IPTable is a small data structure that contains information about some (not necessarily all) nodes in the system. Each entry contains a node's IP, an estimate of its available Nswap Cache space, and a cache of open sockets to it that is used to avoid creating a new connection on every communication. [2] IPTable entries are modified when a node receives a periodic UDP broadcast from other nodes containing information about their available Nswap Cache space.

The multi-threaded Nswap server is responsible for managing the portion of its RAM currently allocated for storing remotely swapped page data (the Nswap Cache). Server threads receive swap-in and swap-out messages from Nswap client nodes. On a Swap-in request, a server thread does a fast look-up of the page in its Nswap Cache and sends a copy of the page to the requesting client. The page also remains in the server's Nswap Cache as this is still the valid backing store for the page. On a Swap-out request, a server thread allocates a free page in its Nswap Cache to store the remotely swapped page data. Multiple threads allow the Nswap server to concurrently handle multiple swapping requests.

The server is responsible for growing and shrinking the size of its Nswap Cache in response to local memory use: when local processes need more RAM space, the Nswap server releases pages from its Nswap Cache back to the local paging system; when idle RAM becomes available, the server allocates some of it, increasing the size of its Nswap Cache. Only when a node has idle RAM, does Nswap allocate some of it to use to store remotely swapped page data from other nodes. When an Nswap server gives RAM back to the paging system, remotely swapped page data that is stored in that RAM are migrated to other Nswap servers that currently have available Nswap Cache space. If no available Nswap Cache space exits, the pages are migrated back to their owner's node and written to swap space on disk. Servers periodically UDP broadcast their Nswap Cache sizes.

The migration protocol is shown in Figure 2. Server C sends a MIGRATE request to server B. If B responds indicating that it can take the page, C sends B a copy of page and now both C and B store the page. After B receives the migrated page it sends the page's owner, A, an UPDATE message telling A that it now stores its page. Node A updates its shadow slot

[2] Additionally, the IPTable could contain other statistics about nodes, such as CPU load, that could be used to select a "good" server.

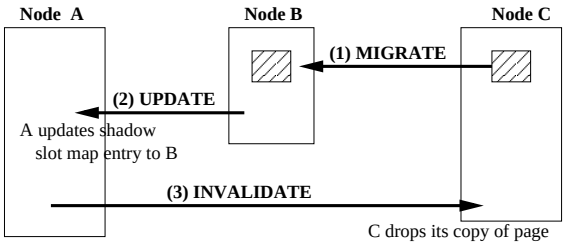

Fig. 2. Page Migration. *(1) a MIGRATE from C to B, (2) triggers an UPDATE from B to the page's owner A, (3) which in turn triggers an INVALIDATE from A to C telling C that it can drop the page.*

map entry for the page, indicating that B is the server for the page, and A sends C an INVALIDATE message. At this point C can safely drop its copy of the page. The migration protocol is optimized for the owner of the page, ensuring that it can always find its page quickly because its shadow slot map entry always records a valid server storing the page.

To make the most efficient use of Nswap Cache space, it is important to ensure that it not fill up with "dead pages" from exited processes. Because the kernel assumes that swap partitions are under local control, there is no call from the kernel to the swap device driver notifying it that a slot has been freed. When a node is actively swapping, dead pages are cleaned from the system as slots are reused. When a node is not actively swapping, Nswap periodically runs a garbage collector thread to scan through the kernel's slot map and the shadow slot map finding slots that the kernel has freed for which the shadow slot map still has valid mappings. The garbage collector sends messages to servers notifying them that they can drop the dead pages from their Nswap Cache.

For performance reasons, Nswap allows multiple page operations to happen concurrently. This concurrency, however, can result in conflicting operations on the same page that must be resolved. For example, a page may be being migrated while at the same time a new swap-out of the page is occurring. When these two events overlap, it is important that the client does not lose the location of the newly swapped out page, and it is important that the client not lose track of the location(s) of the old page before it is garbage collected. To solve these problems, usage count and migration-count values stored with each shadow slot map entry are also stored with each swapped out page and are included as part of the page meta-data sent with messages. These values are used to detect and correctly handle conflicting operations like the one above. For more details about the Nswap system see [15].

IV. Reliability Solutions

Making Nswap reliable involves modifying and extending the core system to generate and store redundant information that is necessary for recovering remotely swapped page data lost in a node crash. Our solutions apply a RAID-like approach to distributing page and reliability data across idle RAM in the cluster. The goals of our reliability solutions are: to minimize the amount of extra computation and state necessary for added reliability, with a particular emphasis on minimizing the impact on the swapping node; to scale to large-sized clusters; to avoid using (slow) disk for reliability; and to work with the dynamic adaptable nature of our system.

A. Dynamic Mirroring

Our dynamic mirroring solution requires an Nswap client to find two Nswap servers to store each swaped-out page. The client's shadow slot map contains entries for both servers. If one server fails, the page can be recovered from the other server, and copied to a new server to restore reliability for it. When a page is migrated, the new server first checks to see if it already has a copy of the page (it is the mirror for this page), and if so, it does not accept the page migration and the original server picks a different node to which to migrate the page.

Mirroring is a an easy way to add reliability. However, it is not very efficient as it requires twice as much idle RAM to store pages, since two copies of every page are stored in the system, and because it requires twice as much network bandwidth on each page swap-out as each page is sent to two servers. As a result, we expect that our parity-based reliability schemes will be more efficient.

B. Centralized Dynamic Parity

Our Centralized Dynamic Parity is based on RAID level 4. Swapped-out pages are organized into parity groups; each page in a parity group resides on a different Nswap server node, and parity pages reside on a dedicated parity server node. A parity page consists of the XOR of the group's page data and meta-data that identifies each page currently in the group. Both the number of pages in a parity group and the specific set of pages that make up a parity group can change over time as a result of page migrations, garbage collection, growing and shrinking of Nswap Caches, and parity group merging (creating one large parity group out of several small, non-overlapping ones).

A single cluster node serves as the designated parity server and the other nodes are regular Nswap client/server nodes. Unlike regular Nswap nodes, the parity server runs no cluster application processes. In large clusters, nodes are divided into parity partitions, each with a dedicated parity server. The size of a parity partition will vary from system to system [3]. The parity server is responsible for storing parity pages, implementing the recovery algorithm to restore lost page data, and managing and controlling parity group membership. Regular Nswap nodes keep no state about parity groups other than the IP of the parity server.

1) Creating Parity Groups: Parity groups are first created by Nswap clients as pages are swapped out. When a page is swapped to an unused swap slot (i.e. there is not a page associated with an older use of the swap slot stored at an Nswap server), the page becomes part of a new parity group. We use a technique similar to Markatos and Dramitinos' Parity Logging [13] to reduce the number of additional page sends

[3]Through a simulation study, we found that our gigabit cluster, up to 64 nodes can be in the same parity partition before communication with the parity server becomes a bottleneck.

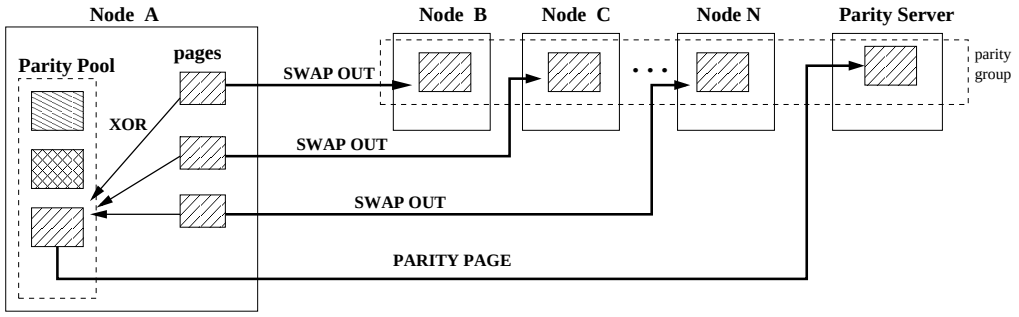

Fig. 3. A Page Swap-out to an Unused Slot. *The client's Parity Pool is used to add the page to a new parity group. The Parity Pool consists of some number of in-progress parity pages. Before the client (Node A) swaps out a page, the page data are added to one of the parity pages in the Parity Pool (page data are XOR'ed into a parity page and the page's meta-data is added). Only when a parity page in the Parity Pool is full is the parity page sent to the Parity Server (resulting in one page send to the parity server per N swap-outs). After sending the parity page, the client knows nothing about this parity group.*

during swap-outs. Each Nswap client keeps a pool of in-progress parity pages. As a page is swapped out, it is added into one of the pages in the pool (see Figure 3). When a parity page in the pool becomes full (determined by the max parity group size, *N*, for the cluster) it is flushed to the parity server resulting in only one additional page send per every *N* swap-outs. When the parity server receives a new parity page, it becomes the only entity that knows anything about the parity group; once the client sends a full parity page to the parity server, the client keeps no state about that parity page nor does it keep any state about to which parity groups its pages belong. The max parity group size *N*, can be almost as large as the parity partition size. However, because each node's Nswap Cache size varies, there is no guarantee that a parity group's pages can be stripped across all nodes in the partition. As a result, a newly formed parity group's size depends both on the number of nodes with free Nswap Cache space currently available in the system and on the max size N.

2) How Parity Groups Change: There are several ways in which a parity group can change. One way is that the parity page data and meta-data are updated as new versions of pages are swapped out or as dead pages are removed from parity groups. When a page is swapped-out to a slot that contains a valid mapping (i.e. a page associated with the previous swap-out to the slot is stored at an Nswap server), the client swaps-out the new copy of the page to the server storing the older copy (this is exactly how Nswap with no reliability behaves). The server will overwrite the old page with the new one. However, it first sends a message to the parity server with the XOR of the old and new page data and meta-data. The parity server applies the XOR to the parity page, taking the old page out and adding the new page into the parity page (see Figure 4). Immediately after sending the parity server the XOR, the old server sends the client an ACK to its SWAP_OUT request notifying the client that it can safely reuse the memory page. At this point if there is a node crash the parity server has received the XOR of the old and new data, so every page in the parity group can be recovered. The ACK to the client does not need to wait to be sent until the parity server applies the XOR, it just needs to wait until the

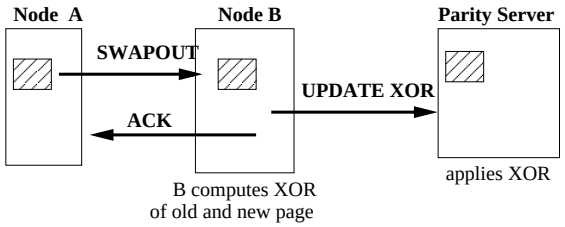

Fig. 4. A Page Swap-out to an In-Use Slot. *Only if a page is swapped out to a slot already in use (an older version of the page is stored at some Nswap Server) does the parity server need to be involved in the swap-out. When server B gets a new copy of the page, it computes the XOR of the old and new page data and send it to the parity server to update the parity page containing this data page. B also sends an ACK to the client when it is safe for the client to re-use the memory storing the swapped-out page data.*

parity server receives the message containing the XOR.

Pages can be removed from a parity group when a dead page is garbage collected. When the server receives a garbage collection message from the client, it sends a message containing the dead page's data and meta-data to the parity server before removing the dead page from its Nswap Cache. The parity server then removes the page data and meta-data from its parity group, and the resulting parity group size is one smaller.

Page migration also results in changes to the parity group. When a page is migrated from one server to another, the parity server must be notified of the page's new location. To the migration protocol shown in Figure 2, two additional messages are added (shown as (4) and (5) in Figure 5). When the old server (*C*) receives the INVALIDATE message, it sends to the parity server an UPDATE_PARITY message containing meta-data for the old and new page. The parity server then updates the meta-data associated with the page's parity page and sends a DROP_PAGE message to the old server telling it that it is now safe to drop its copy of the page.

Parity group membership can change as a result of a page migration that causes a conflict in parity. For example, in Figure 6, a page from parity group 2 is being migrated to server C that already has a page in parity group 2. This type of conflict is detected when the parity server receives the UPDATE_PARITY message from the old server. To resolve the

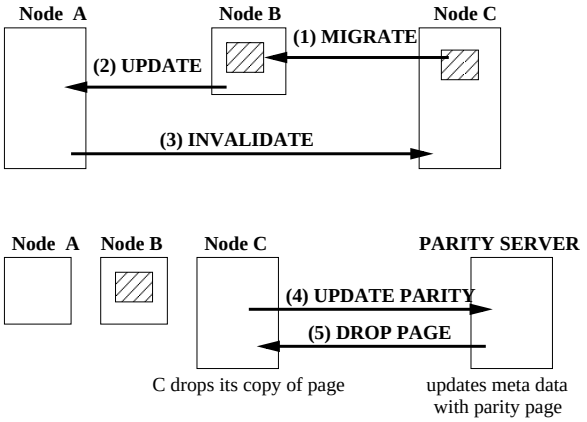

Fig. 5. Page Migration and the Parity Server. *To the basic page migration protocol, messages (4) and (5) are added: (4) the old server notifies the Parity Server of the new location of the page; and (5) the parity server tells the old server it is safe to drop its copy of the page.*

conflict, the parity server sends the old server a PAGE_DATA message requesting a copy of the page, which it removes from Group 2 and adds to another parity group (shown as additional messages A1 and A2 in Figure 6). By having the parity server resolve parity group conflicts that arise from page migration, we do not add extra overhead on regular Nswap nodes for handling this atypical case.

Parity group membership also changes when the parity server periodically merges several small, non-overlapping parity groups into a single larger one. This is necessary because page migration with conflicts and garbage collection can result in many small parity groups wasting RAM on the parity server. Because the regular Nswap nodes need know nothing about parity groups, parity group merging is an entirely local operation on the parity server.

The parity server uses page meta-data to resolve conflicts that can occur when a page migration overlaps with other operations on the page. For example, if a swap-out to the new server overlaps with the page's migration, the page's meta-data are used to delay applying the XOR from the swap-out to the new server until the PAGE_DATA request from the page migration has been handled and the page has been moved to a new parity group.

3) Page Recovery: The parity server is completely responsible for recovering lost page data. The recovery algorithm proceeds concurrently with regular Nswap swapping activity, and the parity server resolves conflicts that result from this concurrency. When a node discovers that another node is unreachable, it sends a RECOVER message containing the failed node's IP to the parity server. The parity server scans through its set of parity pages, identifying parity groups that contain pages on the failed server and wakes up recovery threads to handle page recovery. For each parity group containing a lost page, a parity thread requests the page data from the non-failed nodes in the parity group using a PAGES_DATA message to request a set of page data from Nswap nodes that will be used to recover lost pages. As Nswap servers send the parity server page data, the pages are XORed out of a copy of the parity page to recover the lost page. The parity server then finds a new Nswap server that can store the recovered page, possibly moving the recovered page to a different parity group. To accomplish this, the parity server uses a protocol similar to the regular migration protocol: the parity server sends a MIGRATE_RECOVER message and the page to a new server and drops its copy of the page; the new server sends an UPDATE_RECOVER message to the page's owner who updates its shadow slot map entry with the new server's ID. The client does not need to send an INVALIDATE message to the parity server, because the parity server has already dropped its copy of the recovered page data. We can do this because if the new server fails before it sends the client an UPDATE_RECOVER message, the client will detect it when it tries to swap the page and will send the parity server a RECOVER message. This is identical to the client's behavior if the client swaps to the old server before the page has been recovered.

During recovery the parity server and regular Nswap Nodes keep track of which nodes are currently being recovered so that the parity server does not get swamped with RECOVER messages as Nswap nodes keep detecting that the failed node has failed. If a failed node is rebooted, it gets a new unique identifier consisting of its IP and its new boot time, allowing the parity server to detect which data needs to be recovered and which data is newly stored on the rebooted node.

4) Parity Server Recovery: If the parity server fails, then all information about parity groups is lost, including all of the parity pages. Because no other node in the system keeps any state about parity groups, it is not at all important that the parity groups be reconstructed as they were before be the parity server failure; any set of N pages on N different Nswap servers can be put into a newly constructed parity group during the recovery phase. When the parity server comes back up, or a stand-by node becomes the new parity server, all Nswap servers with pages in their Nswap Cache send the new parity server their page data. The new parity server creates new parity groups to recover lost reliability data.

Having each Nswap server send the entire contents of its Nswap Cache can use a large amount of network bandwidth. However, the likelihood that the parity server is the node that fails is very low, so the high use of network bandwidth to handle this extremely uncommon case is not a performance concern. Also, data compression can be used to reduce the amount of data transfered. Another solution would be be to have a primary and back-up parity server. The primary and back-up parity servers would receive the same data from regular Nswap nodes and construct and manage parity groups for these data. It is possible that the primary and backup nodes would construct different parity groups for the same set of pages, which is fine, because the parity server is the only entity that keeps parity group information, so that if the back-up becomes the primary, its version is now the official version. The main problem with this approach is that it requires twice the bandwidth for regular communication with the parity

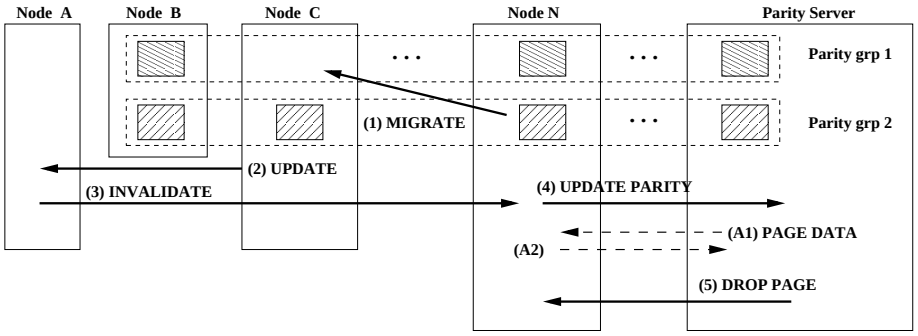

Fig. 6. Example Page Migration Requiring Parity Group Change. *If Node N migrates a page currently in parity group 2 to Node C, the page can no longer stay in parity group 2 (the parity server must take the migrated page out of group 2 and could add it into group 1). Two additional messages are added to the normal migration protocol: (A1) the parity server adds a PAGE DATA message to the old server N and (A2) the old server sends the page data to the parity server so that the page can be taken out of parity group 2 and added into a new parity group.*

server as now all data are sent to the primary and the back-up parity servers. As a result, we chose to minimize network bandwidth use for regular operation, and require full Nswap Cache flushes to recover the parity server data on the unlikely event that the parity server fails.

As the parity server recovers from failure, regular Nswap nodes continue to send normal communication to it. The parity server, however, cannot apply the actions (e.g. UPDATE_XOR) until the lost parity information for the associated page has been recovered. Thus, during recovery, all new actions on parity pages must be queued until the lost parity data for an associated action's pages has been recovered.

5) Problems with the Centralized Parity Approach: There are two main problems with the centralized parity solution. The first is that the parity server has a fixed-sized RAM and can thus only store a fixed number of parity pages. This can limit the amount of free RAM in the system that regular Nswap nodes can make available for network swapping. To scale to large-sized clusters, the centralized solution has to create parity partitions. Without partitioning nodes in this way, a single parity server could not store all the parity pages in the system in its RAM, and would have to swap some to its local disk, severely limiting performance. In parity partitioning, each partition consists of some fixed, disjoint subset of cluster nodes and a dedicated parity server for that subset; there are multiple parity server nodes, each serving as the central parity server for one partition. The problem with this approach is that memory usage between parity partitions is likely not balanced, and as a result, there can be available Nswap Cache space in the cluster that cannot be used by nodes that are currently swapping because they are not in the same parity partition as the nodes with available Nswap Cache space.

The second problem with the centralized approach is that communication with the parity server can become a bottleneck. However, our experiments show that the physical RAM limitations of the parity server become the limiting factor before communication does. A better solution would be to distribute parity server functionality across all Nswap nodes in the system. This would eliminate the potential bottleneck of the centralized solution, and it would remove parity partitions that limit the set of nodes to which an Nswap client can swap.

C. Decentralized Dynamic Parity

Our Decentralized Dynamic Parity solution is loosely based on RAID level 5. Much like the Centralized Dynamic Parity solution, parity group membership and parity group size can change over time as a result of page migration, invalidation of dead pages, and parity group merging. However, in the decentralized approach every cluster node acts as both a regular Nswap client/server node and as a parity server node; Nswap nodes store both remotely swapped page data and parity pages. As in regular Nswap, at any point in time a node is either acting as an Nswap client (a consumer of Nswap Cache space) or it is acting as a combined Nswap server and parity server (a provider of Nswap Cache space). The benefits of the distributed scheme are that recovery data and communication to the parity server is distributed across all nodes in the cluster, and that the number of parity pages that can be stored in the system is limited only by the amount of available Nswap Cache space in the cluster.

1) Keeping Track of Parity Groups: Because there is no central parity server, additional information must be kept with each swapped-out page including its current parity server ID (the IP and last boot time of the node currently storing its parity page) and its current parity group ID. A page's parity group ID is first created by the client as it computes the parity page in its parity page pool. It consists of the client's IP and a unique count value incremented each time a node creates a new parity group. When a client starts a new parity group, it first makes sure that a server can be found to store the parity page for the group; the client sends a SWAP_OUT request to a server, and if the server responds with YES, the client allows a new parity page to be computed (keeping the server thread waiting for the parity page data until the parity page is full and sent to it). Once a parity page has been sent to the Nswap server, the client keeps no state about it nor does it keep state about to which parity groups its pages belong; the pages making up the parity group can change, and a page's parity group ID can change.

When a node enters a phase of shrinking its Nswap Cache, it

tries to migrate some proportion of its set of data pages and its set of parity pages. The idea is to maintain a balanced system by distributing parity and data pages somewhat evenly around the system. When a page is migrated, the old server sends the server storing the parity page the UPDATE_PARITY message much like in the centralized solution. When a parity page is migrated, additional small messages must be sent to all the pages in the parity group notifying them of the new location of their parity page. If modifications to the parity page are made during its migration, the migration protocol is extended to propagate all changes to the new location of the parity page.

Each node performs recovery, parity group conflict resolution, and parity group splitting and merging. If an Nswap node creates a new parity group as a result of one of these actions, it assigns a new unique parity group ID using its local IP and unique parity group counter.

2) Advantages of the Decentralized Approach: Our decentralized dynamic parity approach solves problems with our centralized approach. It allows parity group management and page recovery to be distributed among the regular Nswap nodes and it puts no restriction on the set of Nswap nodes to which a given node can swap or store its parity pages; any free Nswap Cache page can be used to store any remotely swapped page or parity page. The decentralized scheme also maintains the goals of the centralized scheme by ensuring that clients need not keep any state about which parity groups its pages belong, and need not be involved in the recovery of their pages.

3) Problems with the Decentralized Approach: There are some difficulties with the decentralized approach. First, because every node implements the functionality of both a regular Nswap client/server node and a Nswap parity server, each node is more complex and needs to maintain more state. We minimize the amount of extra state by sharing the data structures for finding and managing page data and parity pages on each node. To support this, we add extra fields to structs that define pages and parity pages in the system.

Second, page migrations are more complicated in this scheme due to parity page migration. For example, we must ensure that concurrent modifications to the migrated parity page (e.g. UPDATE_XOR) are not lost during migration. To handle these cases, we propagate changes made to the copy at the old server node to the new node as the final step in migration.

Finally, recovery is more complicated because when any node fails it loses both data and parity pages. Data page recovery works as it does in the centralized scheme, but parity page recovery is more complicated. Due to concurrent page migrations, it may not be possible to reconstruct the exact parity groups at the time of a node failure. Because our model is such that the node storing the parity page for a group decides if the group will be merged or split, it is fine the recovery algorithm reconstructs a different set of parity pages for a set of data pages (and possibly a different number of parity groups) for parity information lost in the crash. However, because every swapped out parity page keeps a guess as to which node stores its parity page, creating, merging, or splitting an existing parity group requires notifying all nodes storing pages in the effected group(s) with changes in their parity group ID or in changes to the server storing the group's parity page.

The decentralized approach has the advantages of distributing the parity server load and avoiding the need for a dedicated parity server(s). However, because the centralized scheme scales well by using parity partitioning, the extra complexity of the decentralized approach may not result in much performance improvement over the centralized scheme.

V. Results

As a first step in evaluating our dynamic parity solutions, we implemented and tested our Centralized Dynamic Parity solution. We present performance results of our recovery algorithm and results comparing swapping to disk, Nswap with no reliability, and Centralized Dynamic Parity Nswap for two sets of benchmark programs running on a cluster. The first are a set of kernel benchmarks designed to range from the best possible case for disk swapping to less favorable cases. These will show how well Nswap does for cases when swapping to disk is optimal and for cases where disk swapping is likely to not perform well. The second set consist of applications from the Splash2 [19] [9], and Linpack HPL [18] benchmark suites. This set is designed to evaluate how well Nswap does on a parallel workload. All experiments are run on a eight node cluster, each node running version 2.6.8 of the Linux kernel and connected to a 1 Gigabit Ethernet switch [4].

A. Implementation

The parity server runs on a dedicated cluster node; it is not also a regular Nswap client/server node, nor does it run cluster application processes. The parity server provides fast look-up of the parity pages it stores based on page meta-data that is sent to it with most messages, it has a data structure similar to the Nswap nodes' IPTable that is used to cache open sockets to Nswap nodes, and it is multi-thread so that it can simultaneously handle requests from multiple Nswap nodes. In addition, it has special recovery threads that handle page recovery and a memory management thread that performs parity group merging.

The centralized solution requires minimal changes to regular Nswap nodes. Nswap clients need only keep a pool of in-progress parity pages, which they send to the parity server when full. Nswap server migration and garbage collection protocols need to communicate with the parity server, and the client and server need to send RECOVERY messages to the parity server when they detect a failed node.

B. Kernel Workload Results

Table I shows the runtime, in seconds, of several workloads comparing swapping to disk, Nswap, and Reliable Nswap using the Centralized Dynamic Parity solution. We ran one

[4]Each node has a Pentium4 processor, 80GB Seagate Barracuda7200 IDE disk drive, and 512MB of RAM. The parity server has 1GB of RAM.

TABLE I

KERNEL WORKLOAD RESULTS. *Comparing Swapping to Disk, Nswap without Reliability, and Nswap with Centralized Dynamic Parity. The rows are 1 and 2 process runs of each workloads. Time, in seconds, is the average of 10 runs. Speed-up over disk swapping is in parentheses.*

Workload (#procs)	Disk	Nswap (speed-up)		Reliable Nswap (speed-up)	
WL1 (1)	**220.31 secs**	**116.28 secs**	**(1.9)**	**117.10 secs**	**(1.9)**
WL1 (2)	338.90 secs	113.61 secs	(2.9)	116.80 secs	(2.9)
WL2 (1)	**2462.90 secs**	**105.24 secs**	**(23.4)**	**109.15 secs**	**(22.6)**
WL2 (2)	1214.11 secs	76.50 secs	(15.9)	84.60 secs	(14.4)
WL3 (1)	**3561.66 secs**	**105.50 secs**	**(33.8)**	**110.19 secs**	**(32.3)**
WL3 (2)	2995.44 secs	95.90 secs	(31.2)	91.89 secs	(32.6)

and two process versions of each workload with one node acting as the client and five nodes acting as servers. We also disabled Nswap Cache growing and shrinking to reduce the amount of variation between timed runs.

Workload 1 consists of a process that performs a number of iterations of a large sequential write to memory followed by a large sequential read. It is designed to be the best case for swapping to disk; because of the way in which Linux allocates swap space, there will be a minimal amount of disk head movement in 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. Workload 3 consists of two processes. The first runs a Workload 2 application and the second performs a large sequential write to a file. Workload 3 further stresses disk head movement with concurrent file I/O and swap I/O to different disk partitions. The total number of pages swapped differs for each workload, so only results in the same row should be compared. The two process versions of each workload access the same total size of memory as the one process versions, but memory is divided between the two. They are designed to represent a slightly more realistic cluster workload.

For the single process version of Workload 1 (first row in Table I), swapping to disk has the potential to outperform Nswap because the application accesses its swapped pages in sequential order on the disk swap partition, minimizing disk arm movement. However, even for this unlikely best case scenario, both Nswap and Reliable Nswap outperform disk by a factor of 1.9. For the other workloads, Nswap and Reliable Nswap perform significantly better than swapping to disk; for example, Workload 3 runs 34 times faster when using Nswap vs. swapping to disk. These results illustrate the performance penalty of disk arm movement when swapping to disk.

Adding reliability to Nswap results in a small amount of additional overhead. The largest slowdown from Nswap to reliable Nswap is 4% for Workload 3 (105.24 seconds vs. 110.19 seconds). This is significantly better than overheads of between 18% and 100% that we found in our Dynamic Mirroring solution. In addition, the Dynamic Centralized Parity scheme uses much less idle cluster RAM for reliability data then Mirroring does.

C. Parallel Benchmark Results

The second set of results evaluate Nswap's performance for parallel workloads. We selected applications from the Linpack HPL and SPLASH2 benchmarks that would compile and run on our small system and that processed a large amount of data resulting in some swapping. Each application was run on two or four cluster nodes, leaving the remaining nodes to act as Nswap server nodes. This configuration was designed to simulate the types of imbalances in RAM usage that can occur in general purpose clusters that run multiple parallel or parallel and sequential applications at any one time. In our experiments, the nodes running the application processes are currently acting as Nswap clients (users of remote RAM for swap space) and are not currently acting as Nswap servers (i.e. none of their RAM is allocated for Nswap Cache space, it is all allocated to the application processes running on these nodes). The nodes that are not running the parallel benchmarks, are acting as Nswap servers (i.e. part of their RAM is allocated for Nswap Cache space to be used to store remotely swapped page data from the nodes running the parallel applications). Additionally, we turned off growing and shrinking of Nswap Cache sizes so that there would be less variation from run to run of each benchmark program.

Each row in Table II shows the time in seconds and the amount of swap space used on each node to run the application. The columns show results for swapping to disk, swapping to Nswap with no reliability, and swapping to Nswap with Centralized Dynamic Parity. For the Nswap runs, the speedup values over disk are listed in parentheses, and the number of swap-outs and swap-ins are listed (these were obtained from performance counters in the Nswap module).

The data in Table II were run on Nswap nodes whose swap partitions are the same size as the disk partition, however the memory footprint of the kernel is slightly larger for Nswap because it includes the Nswap loadable kernel module, and Reliable Nswap is larger than Nswap due to the parity pool that consists of a few additional pages of RAM.

Overall, the results show both Nswap and Reliable Nswap outperform swapping to disk by a factor of between 1.6 and 8.5. The application with the lowest speed-up (FFT at 1.6 and 1.7) is the shortest running application. The longer running applications with more swapping benefit more from Nswap (e.g. LU with speed-ups of 8.2 and 8.5). The speed-up are less than those for the kernel benchmarks because these applications spend a smaller fraction of their total runtime swapping.

Reliable Nswap adds a small amount of overhead compared to Nswap with no reliability (the largest being 3% for LU).

TABLE II

PARALLEL WORKLOAD RESULTS. *Total Time, Speedup over disk swapping, and amount of the swap partition used in Mbytes for Swapping to Disk (column 2), Nswap without reliability (column 3), and Dynamic Centralized Parity Nswap (column 4). Times are shown in seconds, speedup over disk swapping in parentheses, and approximate amount of swap used on each node (rounded to the nearest MB). For the Nswap runs, the total number of swap-ins and swap-outs is also listed. Rows are times for the Linpack HPL benchmark, and SPLASH2 benchmarks.*

Workload	**Disk**		**Nswap**		**Reliable Nswap**	
	time	swap used	time (speed-up)	swap used	time (speed-up)	swap used
Linpack	**1745.05 secs**	**493MB**	**418.26 secs (4.2)**	**450MB**	**415.02 secs (4.2)**	**441MB**
			swapped in: 307K	swapped out: 294K	swapped in: 324K	swapped out:311K
LU	**33464.99 secs**	**519MB**	**3940.12 secs (8.5)**	**519MB**	**4082.19 secs (8.2)**	**519MB**
			swapped in: 3875K	swapped out: 4140K	swapped in: 4243K	swapped out: 4267K
Radix	**464.40 secs**	**518MB**	**96.01 secs (4.8)**	**518MB**	**97.65 secs (4.8)**	**518MB**
			swapped in: 234K	swapped out: 204K	swapped in: 235K	swapped out: 204K
FFT	**156.58 secs**	**390MB**	**94.81 secs (1.7)**	**390 MB**	**95.95 secs (1.6)**	**390MB**
			swapped in: 357K	swapped out: 371K	swapped in: 351K	swapped out: 362K

Reliable Nswap should result in slightly more swapping than Nswap without reliability because the memory footprint of Reliable Nswap is slightly larger than Nswap without reliability. However, for SPLASH FFT, Nswap with no reliability has more swapping than Reliable Nswap (362K vs. 371K swap-outs). Because there are two processes running on each client node in the FFT benchmark, there was much more variation between runs; the amount of swapping can differ between runs as a result of the interaction between process scheduling and the applications' memory reference patterns in combination with Linux page replacement policy. We suspect that this is the cause of the anomaly between reliable Nswap and Nswap in FFT.

Another anomaly occurs in the Linpack results; reliable Nswap does more swapping than Nswap with no reliability (311K swapped-out vs. 294K) but it is also slightly faster (415.02 vs. 418.26). This is caused by differences in the distribution of swapping activity between Nswap and Reliable Nswap runs, which are likely caused by the Linux page replacement policy for this application. Both versions of Nswap use the same number of client threads, but in the Reliable Nswap runs, more of the threads are active simultaneously than in the Nswap runs. This implies that in Reliable Nswap swapping occurs in larger bursts than it does in Nswap. The result is that Reliable Nswap is able to achieve more concurrency when swapping by keeping more client threads busy at the same time. Thus, even though the Nswap run has less swapping, there are fewer concurrent swaps than in reliable Nswap, resulting in slightly faster Reliable Nswap runs.

D. Recovery Results

We present results from experiments measuring the time to recover pages lost on a failed node, and results from experiments measuring application slowdown caused by concurrent page recovery.

We simulate node failure by writing a "failed" node's IP to a file in /proc that triggers an Nswap node to send a RECOVERY message to the parity server to recover all pages that are stored in the Nswap Cache of the "failed" node. Our recovery algorithm is complete except that because a node has not really failed, an Nswap client can always find a copy of all its pages; before recovery the page is still available from the "failed" node, and after recovery it is available from another Nswap server node. We made this simplification to reduce the amount of variation from run to run of an application. Without this simplification there is too much variation between runs of applications with concurrent recovery (i.e. there is no way to ensure that the application waits for the exact same set of its pages to be recovered at exactly the same point in its execution from run to run).

TABLE III

EXECUTION TIMES FOR APPLICATIONS RUN WITH NO CONCURRENT PAGE RECOVERY AND DURING CONCURRENT PAGE RECOVER. *Time is in seconds, and the average and standard deviation values are of 5 runs.*

Application	time	std dev	ave recovery time
appl w/ no recovery	22.41 secs	0.12	NA
appl w/concurrent page recovery	22.48 secs	0.20	2.46 secs

Table III shows results of a sequential application from our first set of experiments during which we triggered page recovery of one "failed" node. Because we did not actually cause node failure, the application can always find a copy of its page (either by grabbing it from the "failed" node or the new server it was migrated to after recovery). Thus, the results in Table III measure the amount of slowdown concurrent page recovery has on applications that do not try to access lost page data before those data have been recovered. The results show no slowdown in the application due to concurrent page recovery on the Parity server: 22.41 seconds for runs with no concurrent page recovery vs. 22.48 seconds for runs with concurrent page recovery (with standard deviations of 0.12 and 0.2).

Table IV shows the amount of time it takes to recover lost page data for different numbers of total pages lost in a node crash. Not surprisingly, the total time increases as the total number of pages to recover increase. However, the per page recovery time (shown in the second to last column) stays relatively constant as the total number of pages recovered increases. The differences in per page recovery times can be attributed to the average size of the parity groups. When the parity group size is close to five, four pages need to be fetched

TABLE IV

PAGE RECOVERY TIMES. *Each row lists the total recovery time in seconds (column 2), the per-page recovery time in milliseconds (column 3), and the average parity group size (column 4) for recovering some number of pages (column 1). The time values are in seconds.*

Total Number of Pages Recovered	Total Recovery Time	Per Page Recovery Time	Ave Parity Group Size
5,196	0.75 secs	0.144 ms	4.99
7,910	1.19 secs	0.151 ms	4.99
9,182	1.81 secs	0.197 ms	5.97
14,477	2.15 secs	0.148 ms	4.99
15,629	2.88 secs	0.184 ms	5.98
25,802	4.69 secs	0.181 ms	5.94
71,792	13.02 secs	0.181 ms	5.97

from Nswap servers to recover each lost page, resulting in per page recovery times of about 0.15 ms. When the average parity group size is closer to six, five data pages need to be fetched from Nswap servers to recover each lost page, resulting in per page recovery times of about 0.18 ms.

VI. CONCLUSIONS AND FUTURE WORK

Our Dynamic Parity reliability solutions solve the problem of efficiently providing reliability to remotely swapped page data in an adaptable Network RAM system. Our results show that our Centralized Dynamic Parity solution adds minimal overhead to Nswap without reliability, and that applications run up to 32 times faster on reliable Nswap than they do with disk swapping. Additionally, our results show that page recovery has no impact on cluster applications running currently with page recovery, and that our per-page recovery times are independent of the number of pages being recovered. As networking technology continues to improve, Nswap will increasingly perform better than systems that use only disk swapping. In fact, a study that examined the performance of Network RAM over Infiniband [12] suggests that reliable Nswap over Infiniband would result in even better performance than we saw with reliable Nswap over 1Gb Ethernet.

Because our results support our dynamic parity approach, we plan to implement and test our Decentralized Dynamic Parity solution as part of our future work. Additionally, we plan to investigate predictive schemes for determining when to grow or shrink Nswap Cache sizes, and we plan to run Nswap on larger clusters to evaluate our scalable design.

VII. ACKNOWLEDGMENTS

We thank Hongzhang Shan for providing us with MPI ports of the SPLASH2 benchmarks. And we thank Jenny Barry, America Holloway and Heather Jones for their participation in Reliable Nswap.

REFERENCES

[1] A. Acharya and S. Setia. Availability and Utility of Idle Memory on Workstation Clusters. In *Proc. ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 35–46, May 1999.

[2] E. Anderson and J. Neefe. An exploration of network RAM. In *Technical Report CSD-98-1000, UC Berkeley*, 1998.

[3] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proc.ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 267–278, 1995.

[4] A. Barak, O. La'adan, and A. Shiloh. Scalable Cluster Computing with MOSIX for Linux. In *Proc. Linux Expo '99*, pages 95–100, Raleigh, N.C., May 1999.

[5] A. Batat and D. G. Feitelson. Gang scheduling with memory considerations. In *Proc. 14th Intl. Parallel and Distributed Processing Symp.*, May 2000.

[6] G. Bernard and S. Hamma. Remote Memory Paging in Networks of Workstations. In *Proc. SUUG'94 Conference*, April 1994.

[7] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Proc. 1994 conference on Supercomputing*, pages 590–599, 1994.

[8] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Operating Systems Design and Implementation*, pages 267–280, 1994.

[9] Hongzhang Shan. MPI port of SPLASH2 benchmarks.

[10] Liviu Iftode, Karin Petersen, and Kai Li. Memory Servers for Multicomputers. In *Proc. IEEE COMPCON'93 Conference*, Febuary 1993.

[11] John L. Hennessy and David A. Patterson. *Computer Architectures A Quantitative Approach, 3rd Edition*. Morgan Kaufman, 2002.

[12] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. Swapping to remote memory over infiniband: an approach using a high performance network block device. In *IEEE Cluster Computing*, 2005.

[13] Evangelos P. Markatos and George Dramitinos. Implementation of a Reliable Remote Memory Pager. In *Proc. USENIX 1996 Annual Technical Conference*, 1996.

[14] Michael J. Feeley and William E. Morgan and Frederic H. Pighinand Anna R. Karlin and Henry M. Levy and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proc. 15th ACM Symposium on Operating Systems Principles*, December 1995.

[15] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: a network swapping module for linux clusters. 2003. Proc. Euro-Par'03 International Conference on Parallel and Disributed Computing.

[16] John Oleszkiewicz, Li Ziao, and Yunhao Liu. Parallel network RAM: Effectively utilizing global cluster memory for large data-intensive parallel programs. In *Proc. IEEE 2004 International Conference on Parallel Processing (ICPP'04)*, 2004.

[17] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. SIGMOD'88 the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM Press.

[18] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl/, January 2004.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. Proc. the 22nd International Symposium on Computer Architecture, June 1995.

[20] Li Xiao, Xiaodong Zhang, and Stefan A. Kubricht. Incorporating Job Migration and Network RAM to Share Cluster Memory Resources. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, 2000.