Reliable Adaptable Network RAM

Tia Newhall, Daniel Amato, Alexandr Pshenichkin
Computer Science Department
Swarthmore College
Swarthmore, PA  USA
newhall@cs.swarthmore.edu
Network RAM

General purpose cluster nodes share each other’s idle RAM as a remote swap partition

When one node’s RAM is overcommitted, swap its pages out over the network to store in idle RAM of other nodes
+ Avoid swapping to slower local disk
+ Almost always some significant amt idle RAM even when some nodes overloaded
Goals of Network RAM system

- Scalable
  - No central authority

- Adaptable
  - Node’s RAM usage varies
  - Don’t want remotely swapped page data to cause more swapping on a node
    ⇒ relinquish RAM it’s using to local paging system when needed by local processes, & allocate when idle RAM

- Fault Tolerant: recover remotely swapped page data lost in node crash
  - A single node failure can lose pages from processes running on remote nodes
  - One node’s failure can affect unrelated processes on other nodes
Nswap

- Network swapping lkm for Linux clusters
  - Runs entirely in kernel space on unmodified Linux 2.6
- Completely Decentralized
  - Each node runs a multi-threaded client & server
  - Client is active when node swapping
    - Uses local information to find a “good” Server when it swaps-out
  - Server is active when node has idle RAM available
How Pages Move around system

**Swap out**: from client A to server B

**Swap in**: from server B to client A (B still is backing store)

**Migrate**: server B shrinks its Nswap Cache sends pages to server C
Reliability Algorithms

- Avoid reliability solutions that use disk for reliability
  - want a RAID-like solution that stripes page & reliability data across cluster-wide idle RAM

- Has to work with Nswap’s:
  1. Dynamic resizing of Nswap Cache
  2. Varying Nswap Cache capacity at each node
  3. Support for migrating remotely swapped page data between servers
  - Reliability solutions that require fixed placement of page and reliability data won’t work
Centralized Dynamic Parity

- RAID 4 like
- A single, dedicated, parity server node

  - In large clusters, nodes divided into Parity Partitions, each partition has its own dedicated Parity Server
  - Parity Server stores parity pages and implements page recovery

  + Client & server don’t need to know about parity grps

```
Parity Partition 1
- Parity Server
- Node 1
- Node 2
- Node m-1

Parity Partition 2
- Parity Server
- Node m+1
- Node 2m-1
```
Centralized Dynamic Parity (cont.)

- Like RAID 4
  - Parity group pages striped across cluster idle RAM
  - Parity pages all on single parity server

- with some differences
  - Parity group size and assignment is not fixed
  - Pages can leave and enter a given parity group (garbage collection, migration, merging parity grps)
Page Swap-out, case 1: new page swap

Parity Logging:

- client stores a set of in-progress parity pages
- As page swapped out it is added to a parity log
  - minor computation overhead on client (XOR of 4K pages)
- As parity logs fill, they are sent to the Parity Server
  - One extra page send to parity server every ~N swap-outs
Page Swap-out, case 2: overwrite

Server has old copy of swapped out page:

- Client sends new page to server
  - No extra overhead on client side vs. non-reliable Nswap
- Server computes the XOR of the old and new version of the page and sends it to the Parity Server before overwriting the old version with the new

![Diagram showing the process of overwriting a page with XOR and updating parity information.](Diagram.png)
Node Failure

Detecting node sends a RECOVER ALL message to Parity server

Parity Server

RECOVER ALL (Node 2)

Node

Node 2

Parity Server rebuilds all pages that were stored at the crashed node
As it recovers each page, it migrates it to a non-failed Nswap Server
page may stay in same parity group or be added to a new one
The server receiving the recovered page tells client of its new location
Soln 3: Decentralized Dynamic Parity

- Like RAID 5:
  - No dedicated parity server
  - Data pages and Parity pages striped across Nswap Servers
    - not limited by Parity Server’s RAM capacity nor Parity Partitioning
    - every node is now Client, Server, Parity Server

- Store with each data page its parity server & P-group ID
  - For each page, need to know its parity server and to which group it belongs
  - A page’s parity group ID and parity server can change due to migration or merging of two small parity groups
    - First set by client on swap-out when parity logging
    - Server can change when page is migrated or parity groups are merged

- Client still does parity logging
  - Finds a node to take the parity page as it starts a new parity group
  - One extra message per parity group to find a server for parity page

- Every Nswap server has to recover lost pages that belong to parity groups whose parity page it stores.
  - +/- Decentralized recovery
## Kernel Benchmark Results

<table>
<thead>
<tr>
<th>Workload</th>
<th>Swapping to Disk</th>
<th>Nswap (No Reliability)</th>
<th>Nswap (Centralized Parity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Sequential R&amp;W</td>
<td>220.31</td>
<td>116.28 (speedup 1.9)</td>
<td>117.10 (1.9)</td>
</tr>
<tr>
<td>(2) Random R&amp;W</td>
<td>2462.90</td>
<td>105.24 (23.4)</td>
<td>109.15 (22.6)</td>
</tr>
<tr>
<td>(3) Random R&amp;W &amp; File I/O</td>
<td>3561.66</td>
<td>105.50 (33.8)</td>
<td>110.19 (32.3)</td>
</tr>
</tbody>
</table>

8 node Linux 2.6 cluster (Pentium 4, 512 MB RAM, TCP/IP over 1 Gbit Ethernet, 80 GB IDE (100MB/s))

**Workloads:**

1. Sequential R & W to large chunk of memory (best case for disk swapping)
2. Random R & W to memory (more disk arm seeks w/in swap partition)
3. 1 large file I/O, 1 W2 (disk arm seeks between swap & file partitions)
# Parallel Benchmark Results

<table>
<thead>
<tr>
<th>Workload</th>
<th>Swapping to Disk</th>
<th>Nswap (No Reliability)</th>
<th>Nswap (Centralized Parity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linpack</td>
<td>1745.05</td>
<td>418.26</td>
<td>415.02</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(speedup 4.2)</td>
<td>(4.2)</td>
</tr>
<tr>
<td>LU</td>
<td>33464.99</td>
<td>3940.12</td>
<td>109.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(8.5)</td>
<td>(8.2)</td>
</tr>
<tr>
<td>Radix</td>
<td>464.40</td>
<td>96.01</td>
<td>97.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(4.8)</td>
<td>(4.8)</td>
</tr>
<tr>
<td>FFT</td>
<td>156.58</td>
<td>94.81</td>
<td>95.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1.7)</td>
<td>(1.6)</td>
</tr>
</tbody>
</table>

8 node Linux 2.6 cluster (Pentium 4, 512 MB RAM, TCP/IP over 1 Gbit Ethernet, 80 GB IDE (100MB/s))

Application Processes running on half of the nodes (clients of Nswap), the other half are not running benchmark processes and are acting as Nswap servers.
Recovery Results

- Timed execution of applications with and without concurrent page recovery (simulated node failure and the recovery of pages it lost)
  - Concurrent recovery does not slow down application

- Measured the time it takes for the Parity Server to recover each page of lost data
  - ~7,000 pages recovered per second
  - When parity group size is ~5: 0.15 ms per page
  - When parity group size is ~6: 0.18 ms per page
Conclusions

- Nswap’s adaptable design makes adding reliability support difficult
- Our Dynamic Parity Solutions solve these difficulties, and should provide the best solutions in terms of time and space efficiency
- Results testing our Centralized Solution, support implementing the Decentralized Solution
  + more adaptable
  + no dedicated Parity Server or its fixed-size RAM limitations
  - more complicated protocols
  - more overlapping, potentially interfering operations
  - each node now a Client, Server, and Parity Server
Acknowledgments

Swarthmore Students:

Dan Amato’07           Alexandr Pshenishkin’07
Jenny Barry’07           Heather Jones’06
America Holloway’05      Ben Mitchell’05
Julian Rosse’04          Matti Klock ’03
Sean Finney ’03          Michael Spiegel ’03
Kuzman Ganchev ’03

More information:

http://www.cs.swarthmore.edu/~newhall/nswap.html
Nswap’s Design Goals

- **Transparent**
  - User should not have to do anything to enable swapping over NW

- **Adaptable**
  - A Network RAM system that constantly runs on cluster must adjust to changes in local node’s memory usage
  - Local processes should get local RAM before remote processes do

- **Efficient**
  - Should be fast swapping in and out
  - Should use a minimal amount of local memory state

- **Scalable**
  - System should scale to large sized clusters (or networked systems)

- **Reliable**
  - A crash of one node should not effect unrelated processes running on other nodes
Complications

- Simultaneous Conflicting Operations
  - Asynchrony and threads allows for fast, multiple ops at once, but some overlapping ops can conflict, ex. Migration and new swap-out for same page

- Garbage Pages in the System
  - When process terminates we need to remove its remotely swapped pages from servers
  - Swap interface doesn’t contain call to device to free slots since this isn’t a problem for disk swap

- Node failure
  - Can lose remotely swapped page data
How Pages Move Around the System

- **SWAP-OUT:**
  - Node A
  - SWAP_OUT?
    - Node B
      - Nswap Client
        - swap out page
          - shadow slot map
            - B
      - Nswap Server
        - Nswap Cache
          - OK

- **SWAP-IN:**
  - Node A
  - SWAP_IN
    - Node B
      - Nswap Client
        - swap in page i
          - B
      - Nswap Server
        - Nswap Cache
          - YES,
Nswap Client

- Implemented as device driver and added as a swap device on each node
  - Kernel swaps pages to it just like any other swap device

- Shadow slot map stores state about remote location of each swapped out page
  - Extra space overhead that must be minimized

Diagram:

1. Kernel finds free swap slot i
2. Kernel calls Nswap Client’s write function
3. Add server info. to shadow slot map
4. Send the page to server B
Nswap Server

- Manages local idle RAM currently allocated for storing remote pages

- Handles swapping requests
  - Swap-out: allocate page of RAM to store remote page
  - Swap-in: fast lookup of page it stores

- Grows and Shrinks the amount of local RAM available based on the node’s local memory usage
  - Acquire pages from paging system when there is idle RAM
  - Release pages to paging system when they are needed locally
  - Remotely swapped page data may be migrated to other servers
Finding a Server to take a Page

- Client uses local info. to pick “best” server
  - Local IP Table stores available RAM for each node
  - Servers periodically broadcast their size values
  - Clients update entries as they swap to servers
  - IP Table also caches open sockets to nodes

+ No centralized remote memory server

**IP Table**

<table>
<thead>
<tr>
<th>HOST</th>
<th>AMT</th>
<th>Open Socks</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

**Nswap Client**

- swap out page i
- look up a good candidate server and get an open socket to it
- shadow slot map

B
Soln 1: Mirroring

On Swap-outs: send page to primary & back-up servers

On Migrate: if new Server already has a copy of the page it will not accept the MIGRATE request and old server picks another candidate

+ Easy to Implement
- 2 pages being sent on every swap-out
- Requires 2x as much RAM space for pages
- Increases the size of the shadow slot map