Remote-Memory Based Network Swap for Performance Improvement

Nirbhay Chandorkar[1], Rajesh Kalmady[1], Phool Chand[1], Anup K. Bhattacharjee[2], and B.S. Jagadeesh[1]

[1] Computer Division, Bhabha Atomic Research Centre, Trombay, Mumbai
[2] Reactor Control Division, Bhabha Atomic Research Centre, Trombay, Mumbai
{nirbhayc,phool,rajesh,anup,jag}@barc.gov.in

Abstract. On High Performance Computing Clusters we run scientific application programs which are highly memory intensive. Such applications require large amount of primary memory, the execution of which causes swapping on to the secondary memory of the cluster nodes. As the swapped blocks reside on the secondary memory (disk), any access to these affects the throughput of the applications significantly. In this paper we discuss the design and development of a fast and efficient swap device. This device is a network based, pseudo block device, developed to improve the performance of memory intensive work loads by dynamically unifying free physical memory of various nodes of a cluster over the network. We also discuss the performance observed over different network interconnects.

Keywords: Block Device, Remote Memory, Swap Device, Cluster.

1 Introduction

With the increase in availability of greater processing power, scientists are attempting to develop and solve high-resolution models to make accurate simulations of problems that they are trying to investigate. Applications like these need, in addition to huge processing power, large primary memory to give results in a reasonable time frame. But, the primary memory on a commodity computer is always limited owing to the cost trade-offs governed by the number of memory slots and the density of the memory-modules that go into them.

When the over-all memory requirement of all the active processes in a system exceeds the primary memory present on the system, swapping is employed to free some of the memory blocks (as decided by a pre-determined replacement policy) on to the swap space on a secondary storage (disk) to make them available for an active process that has an immediate requirement of memory. Swap space is a space on the disk, which is used by the operating system as an extension to the primary memory. Because of the large disparity in the access times of blocks in primary memory and the blocks that are to be brought from the swap space on the secondary storage, the performance of the program gets reduced drastically when swapping is employed by the operating system. This assumes even more significance as most of the High Performance Computing systems are built by interconnecting commodity processors having

N. Meghanathan et al. (Eds.): CCSIT 2011, Part I, CCIS 131, pp. 125–134, 2011.

limited memories through commodity interconnects and execute multiple jobs at a given time. Even though the memory available on each of the processors is limited, the aggregation of the primary memory of all the processors in a cluster is indeed substantial.

It is worth noting that current-day clusters typically have at least 256 nodes with each node having about 16-32 GB of Memory and are interconnected using high speed, low latency networks. With such large number of nodes present in a cluster and the memory requirement of programs being executed in them being dynamic in nature, free, unused memory will be available on some of the cluster nodes. We have explored the idea of utilizing the free-memory of the nodes of a cluster for storing the swapped out pages of the other nodes.

2 Related Works

Availability of free memory on work stations have been explored in [1, 2]. The concept of utilizing remote memory started for network of workstations with the high bandwidth, low-latency networks becoming affordable. Several studies [3, 4, 5, 6] have been made to utilize the remote memory over the network. Feeley et. al. [3] have implemented a system in which remote memory is used as the cache for swapped pages that are also written through to disk. They modified the memory management system of the DEC OSF/1 kernel. Their servers only cache clean pages and arbitrarily drop a page when memory resources become scarce. Markatos and Dramitinos [4] proposed the reliability schemes for a remote memory pager for the DEC OSF/1 operating system. Their system is implemented as a client block device driver, and a user-level server for storing pages from remote nodes. For providing reliability they have considered mirroring and parity logging based schemes and further the requests can also be forwarded to disk or disk based file. Bernard and Hamma's work [5] focuses on policies for balancing a node's resource usage between remote paging servers and local processes in a network of workstations. Anderson and Neefe [6] describe the common assumptions about the network RAM and have discussed the possible ways of its implementation. They have further discussed their user level implementation which required modification in code of memory allocator to allocate remote memory.

With further increase in disparity between network and disk access times with introduction of network technologies based on InfiniBand or 10G Ethernet, the concept of utilization of remote physical memory is being increasingly investigated [7, 8, 9].

3 System Design

In order to utilize primary memory of remote machines, we had to develop two basic components in our design. The first component, which runs on the node where the user application gets executed and requires remote memory, is the client part of the system and the other component which runs on the remote nodes serving their memory to the application is the server part of the system.

3.1 Design Issues

To design the system, we have considered the following design issues:

a) Kernel level vs. User level design. The implementation to utilize remote memory can be realized at different levels such as in user space or kernel space. In the user space design the client will have to act as an improvised dynamic memory allocator that allocates memory from remote nodes. In this design scheme we cannot achieve transparency in user level applications, because the applications will have to be written to access the functions used in the implementation. The user level design incurs more software overheads, because of the memory protection mechanism, in which user level programs cannot access the kernel memory directly; instead the data is copied from user to kernel space before transfer. Moreover, if the server were to be in user space, the memory allocated by it might as well get swapped on to the nodes' secondary memory. Hence, we chose to implement the system entirely in kernel space to achieve complete transparency to user applications.

b) Memory allocation scheme. The server module needs to allocate free memory on its node for clients on other nodes. Different policies can be adopted for this. One way is to make the server to allocate available memory pages whenever a swap-out request comes from a client and then free it when it is required no more. But there are several causes for concern present here such as, the additional memory allocation overhead for each request, the possibility of allocation failure and the like. Moreover, the Linux kernel never notifies the swap device that a particular page is no longer needed and can be freed. The other option is to make the server module reserve some fixed amount of memory for the client, when it asks for the memory to initialize the device. The drawback of this approach is that this reserved memory cannot be utilized by other programs on the server node. We chose to implement the second policy, because it does not have risk of memory allocation failure as weighed against the cost of free memory being blocked on the server side.

The block diagram of the system is depicted in Figure 1. The client module is implemented as a network based pseudo block device driver. The server has been developed as a loadable kernel module. The server allocates memory in multiples of fixed size memory chunk, called a "*Swap Block*". Hence, the size of the device can be in multiples of the *Swap Block* size only. At a given time a cluster node either acts as a server or a client. Here onwards the proposed system consisting of two modules, client and server will be referred as Network RAM Block Device (NRBD).

3.2 Client Module

The client module is implemented as the block device driver and Linux kernel's swapping mechanism communicates with this module exactly in the same manner as it would with any hard disk driver. Client module consists of two layers, layer one accepts swap-in swap-out request from the kernel Block I/O layer and forwards the request to second layer, which consists of multiple kernel threads to process the requests. The second layer also has a Main Thread which runs as the daemon; it initializes the pseudo device and acts as an interface between user space and Client Module.

Client's Main Thread initializes the pseudo block device, it negotiates with the servers in the cluster nodes and gets allocated the amount of memory required to create the device. Client keeps track of allocated *Swap Block*s distributed over the network using its *Swap Map* and *Server List*. *Swap Map* maps contiguous pseudo block device to the distributed Swap Block allocated on different nodes. *Server List* has one entry for each Server node which has allocated memory for the Client. Its each node has following fields: request queue, worker thread, pointer to socket.

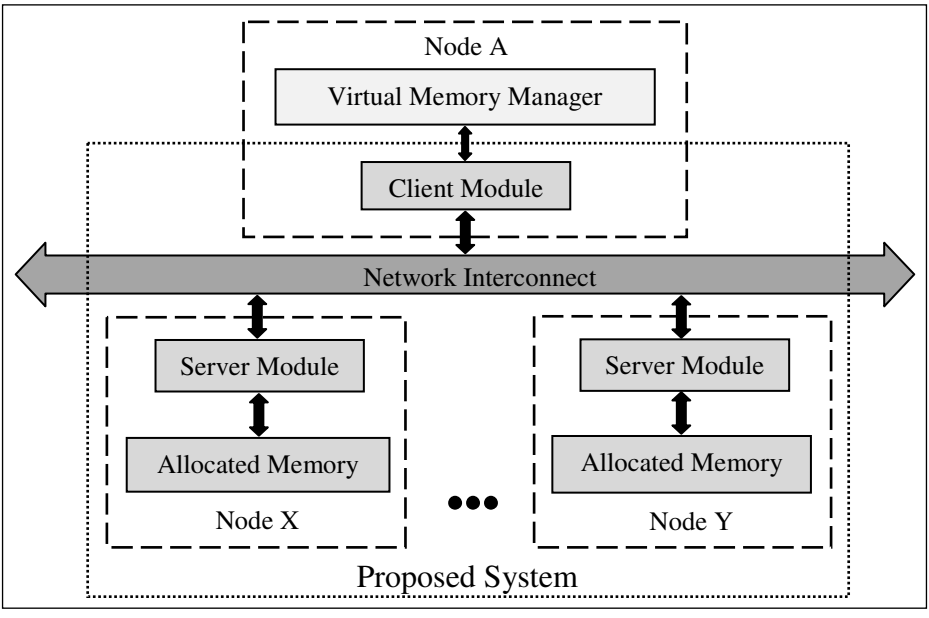

Fig. 1. Block Diagram of the System

Request queue is the linked list of I/O requests pending with the client. Field worker thread is a kernel thread to process the pending requests in the request queue.

Client Layer 1 is implemented as the *device_make_request*(*DMR*) module. Kernel invokes this module to submit the block I/O structure, named as *bio*, to the device driver (client). When kernel calls *DMR* module, it maps the *bio* (page) to the appropriate server (among the servers in the *Server List*) using *Swap Map*, and computes the offset within the *Swap Block* to have exact memory location on the remote server. Further, it encapsulates the *bio* in its custom request structure and then appends the request structure in the request queue of the server, and in this way forwarding I/O request to Layer 2. Figure 2 shows the block diagram of the client module.

Client Layer 2 contains the *Server List,* its each node has a worker thread to process the pending requests in queue corresponding to the server. When swapping occurs there are multiple requests pending in the queue. Worker thread extracts the request from the queue and processes the request. It sends the suitable command (read or write) to the server module and transfers the data.

Mapping of pages is done in such a way that, load is evenly distributed among the worker threads in Layer 2. If device consists of *n* Swap Blocks, then any set of *n* consecutive pages are mapped onto different Swap Blocks. Further, the layered structure of the client allows the *DMR* module to return, just after submitting the *bio* to Layer 2, rather being blocked till the I/O request is completed. Thus it takes very little time to submit *bio* as compared to the time taken by Layer 2 to complete the I/O over network, and hence because of this, large number of requests are appended in the request queue of each server, which in turn allows the client to merge many consecutive requests and process multiple of them simultaneously with its worker threads.

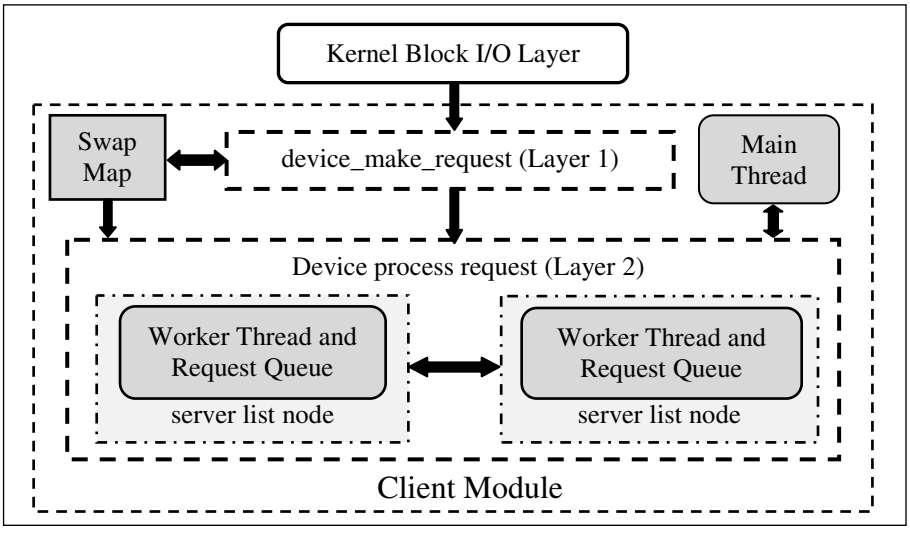

Fig. 2. Block Diagram of Client Module

In the implementation, we have bypassed the I/O scheduler layer and directly registered the *DMR* module to the Block I/O layer. Schedulers reorder and merge the requests to enhance the disk performance. But in our case, it may happen that the merged request may have the pages that map to two different Servers, as our device is based on distributed memory. In order to enhance the performance, we merge the requests that are pending in the request queue of the servers at Layer 2 of our client, as all the requests in the queue belong to the same server. This reduces the overheads of communication protocol.

3.3 Server Module

The server module monitors local memory load and allocates free memory for the clients and frees the allocated memory when client requests for it. The server module comprises of two components. The first component is Worker Component, which consists of allocated *Swap Block*s and a set of worker threads to fulfill the client's read or write request for the associated *Swap Block*s. The second component is Manager Component, it manages the first component and receives the control messages (discussed in section 3.4 Communication Protocol) from the clients and takes the suitable action. Figure 3 shows the block diagram of the server module and the way components are interconnected with each other.

Worker Component has *Client Space List* and *Swap Block Map*. Each node of *Client Space* List has following fields: Client IP Address, worker thread, pointer to connected socket, and pointer to Swap Block Map. Swap Block Map maps the swap block number to the starting logical address of the Swap Block. Server Manager Component allocates memory and initializes node for each client to which space has been allocated and adds it in the Client Space List.

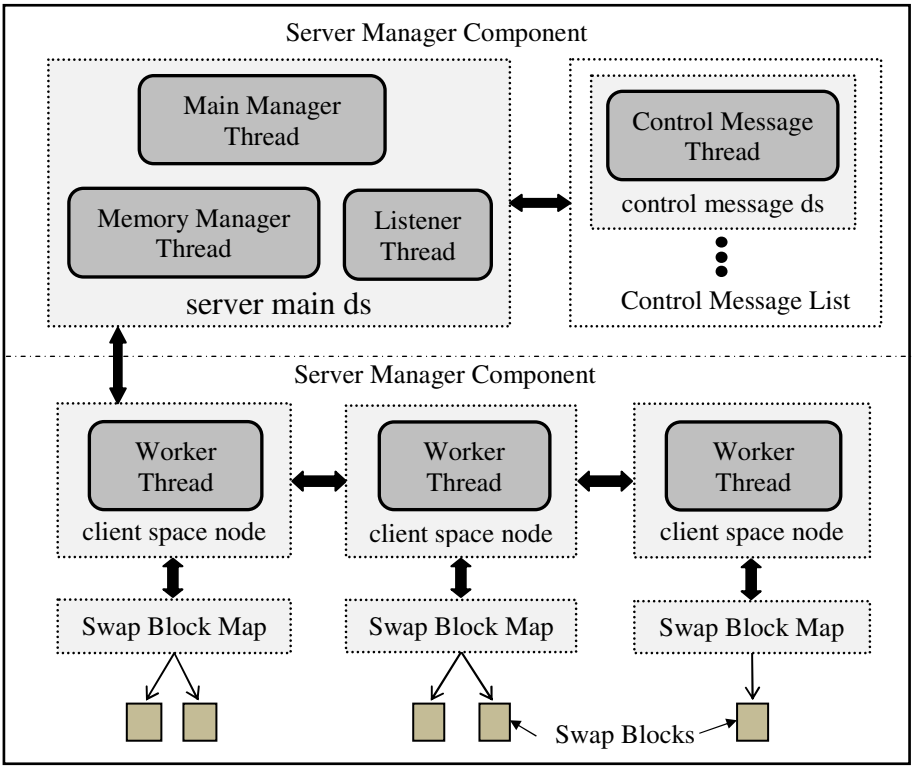

Fig. 3. Block Diagram of Server Module

Manager component consists of following: a Main Manager thread, a Listener thread, a Memory Manager thread, Control Message list, a memory status flag. The control message is request from client to allocate or free the memory.

Main Manager thread is invoked when Server Module is loaded, Main Manager initializes the data structures of the server module, and it starts the Listener and Memory Manager threads. The memory manager thread monitors the local memory and updates the memory status flag, for the main manager thread, at regular intervals. The listener thread listens on the well known port for any connection from the clients. When listener thread accepts a new connection, it initializes new *control message ds* structure and starts the control message thread. Control message thread communicates with the Client and receives the control message, it then updates the main manager thread that message has been received. When there is control message in the list the main manager thread processes the message, it takes the suitable action depending on the control command type (discussed in next section, 3.4 Communication Protocol).

3.4 Communication Protocol

Client and server modules communicate using TCP/IP protocol. Client and Server communicate by sending Request and Reply control messages. Request message

mainly consists of control command type and other information specific to control command. Reply message mainly consists of error code, which describes the status of the control command. The following are the types of control commands:

- *Allock Block:* This is sent by client to server, requesting it to allocate the *Swap Block.*
- *Disconnect:* This control command is sent by client to the server, requesting it to free all the allocated *Swap Blocks* and close the connection with it.
- *Read:* This control command is sent by the client to the server, to send *n* bytes of data from specified swap block, starting from given offset.
- *Write:* This control command is sent by the client to the server, to receive *n* bytes of data in the specified swap block, starting from given offset.

When main manager thread receives *Alloc_Block* command, it first checks the memory status flag. If the free memory is available, it allocates Swap Block. It initializes new *client space node* structure for the client. It starts the worker thread and updates other fields. It initializes the Swap Block Map (*swap block map* structure) with the swap block number and its logical address. Finally, it adds the *client space node* in the Client Space List.

When main manager thread receives *Disconnect* message, it frees all the Swap Blocks allocated to the client, stops the worker thread and closes the socket. It removes the client's *client space node* from the Client Space list.

Read and *Write* control commands are not processed by main manager thread, they are used for data transfer after the pseudo block device is being registered. Client side worker thread sends *Read* or *Write* control command to Server side worker thread to receive or send the data respectively.

4 Experimentation and Results

The execution times of the test programs needing very large memory have been used to compare the performance of the proposed network swap-device with the disk based swap. Section 4.1 describes the experimental setup used to perform the tests.

4.1 Experimental Setup

The experiments to compare swapping to disk and NRBD are conducted on a cluster of 20 nodes. A node has two quad core 3.0 GHz processors with 6 MB L2 cache and 32GB physical memory, InfiniBand and Gigabit Ethernet networks and a 7200 rpm 500 GB hard disk with a maximum transfer rate of 3 Gb/s. The Operating System is Linux and version 2.6.18.

We have setup a network RAM based swap device on two nodes, that is, both having NRBD client module loaded on them and each one them connected to the set of four nodes, running the NRBD server module (total eight server nodes). Each server allocates 4 GB of memory for the client, creating a network ram block device (NRBD) of 16 GB. We have 24 GB Disk swap space on each node.

4.2 Job Description and Performance Results

In order to evaluate the performance of the device in different situations, we selected three different types of workloads. We have written two of them, to test the devices under different memory access patterns, and the third workload is HPLinpack benchmark. The description of the workloads is as follows:

a) Job1 performs sequential writes to and reads from a large chunk of memory; it dynamically allocates large chunk of memory (which exceeds the free memory available) and then sequentially writes into it randomly generated integer values and then read them back in temporary variable. The idea was to observe the performance of contiguous access in swap device. In our implementation the logically contiguous swap device is spread across memory from different nodes. In this, network bandwidth and latency put a definitive penalty.

b) Job2 performs random writes/reads on a large chunk of memory, which in turn causes random access to swap device.

c) HPLinpack benchmark solves dense system of linear equations and is used to rank Top 500 supercomputer sites. It well known benchmark and simulates the memory intensive parallel application.

To compare the NRBD with disk based swap the average of execution time of 10 runs is taken into consideration. Workloads were run using the NRBD as the swap device under two interconnects – InfiniBand (IPoIB) and Gigabit Ethernet (GigE). We have recorded the baseline timing for the Job1 and Job2 with sufficient memory available.

For the HPLinpack runs, we used full 32 GB of RAM. We kept its problem size sufficiently large to cause swapping on the node with of 32 GB RAM. We ran the HPLinpack benchmark on two clients simultaneously.

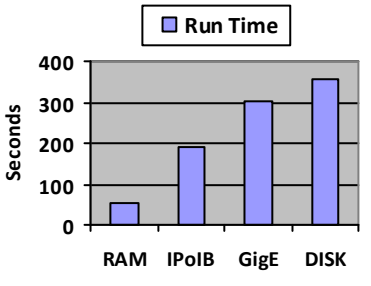

Fig. 4. Job1 Performance Results

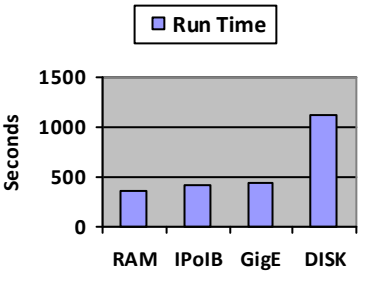

Fig. 5. Job 2 Performance Results

Figure 4 and Figure 5 gives the pictorial view of the results. In case of Job 1 we have got 1.85 times speedup with NRBD-IPoIB and 1.18 times speedup with NRBD-GigE as compared to disk. For Job 2 we have got 2.7 times speedup with NRBD-IPoIB and 2.5 times speedup with NRBD-GigE as compared to disk.

Figure 6 gives the pictorial view of HPLinpack results. For HPLinpack we have got very encouraging 5.4 times speedup with NRBD-IPoIB and 4.5 times speedup with NRBD-GigE as compared to disk.

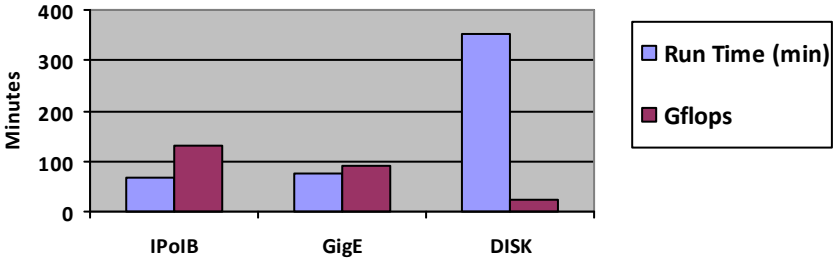

Fig. 6. HPLinpack Performance

5 Conclusions and Future Work

We have designed, developed and deployed remote memory based network swap over both InfiniBand and Gigabit Ethernet networks. Through experimentation we have demonstrated that significant benefits can be derived in improving turnaround time of the jobs. With our implementation using additional memory for the purpose of executing large program would be as simple as including a few machines with memory on the network. Furthermore our current work can be used for sizing the memory requirements of a given application and their number of instances to get an acceptable turnaround time by factoring in network delays.

Further, we wish to pursue research work on finding the swap block replacement policies and incorporate redundancy issues to address failure of a participating node. With the increase in speeds and reduced latencies of Network interconnects, utilizing remote memory for swapping will be increasingly deployed in clusters to gain performance benefits. Swapping on to remote memory itself might become the norm with a storage device (Like a SAN box) may be used occasionally as a backing store for storing the snapshots of the network based swap utility to address failure of participating node/nodes.

Acknowledgements

We would like to express our sincere gratitude to R. S. Mundada, Kislay Bhatt, D. D. Sonvane, Vaibhav Kumar, Vibhuti Duggal, Urvashi Karnani of Parallel Processing Group, Computer Division, BARC, for their insightful and constructive suggestions throughout the research work. We are thankful to A.G. Apte, Head, Computer Division, BARC, for providing us with the opportunity to undertake this project.

References

1. Acharya, A., Setia, S.: Availability and Utility of Idle Memory on Workstation Clusters. In: ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems, pp. 35–46 (May1999)

2. Arpaci, R.H., Dusseau, A.C., Vahdat, A.M., Liu, L.T., Anderson, T.E., Patterson, D.A.: The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In: ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 267–278 (1995)
3. Feeley, M.J., Morgan, W.E., Pighin, F.H., Karlin, A.R., Levy, H.M., Thekkath, C.A.: Implementing Global Memory Management in a Workstation Cluster. In: 15th ACM Symposium on Operating Systems Principles (December 1995)
4. Markatos, E.P., Dramitinos, G.: Implementation of a Reliable Remote Memory Pager. In: USENIX 1996 Annual Technical Conference (1996)
5. Bernard, G., Hamma, S.: Remote Memory Paging in Networks of Workstations. In: SUUG 1994 Conference (April 1994)
6. Anderson, E., Neefe, J.: An Exploration of Network RAM. Technical Report CSD-98-1000, UC Berkley (1998)
7. Liang, S., Noronha, R., Panda, D.K.: Swapping to remote memory over InfiniBand: an approach using a high performance network block device. In: IEEE Cluster Computing (2005)
8. Newhall, T., Finney, S., Ganchev, K., Spiegel, M.: Nswap: a network swapping module for linux clusters. In: Proc. Euro-Par 2003 International Conference on Parallel and Distributed Computing (2003)
9. Newhall, T., Amato, D., Pshenichkin, A.: Reliable Adaptable Network RAM. In: IEEE International Conference on Cluster Computing (2008)