Collaborative Memory Pool in Cluster System

Nan Wang[1, 2], Xuhui Liu[1,2], Jin He[2], Jizhong Han[2], Lisheng Zhang[2], Zhiyong Xu[3]

1. Graduate University of the Chinese Academy of Sciences
2. Institute of Computing Technology, Chinese Academy of Sciences
3. Suffolk University, Boston MA, USA

{wangnan06, mafish, hejin, hjz, zhang}@ict.ac.cn,zxu@mcs.suffolk.edu

Abstract

With the developments of network technologies, many mechanisms have been introduced to improve system performance in cluster systems by exploiting remote idle memory. However, none of them can satisfy the requirements from different applications. Most methods can only improve the performance of a particular type of applications but not for others. One important reason is they failed to provide unified interfaces. In this paper, we propose Collaborative Memory Pool (CMP) to solve this problems. CMP brings scalability and high performance. It has five features: 1) Providing malloc-like interfaces, block device interfaces and kernel API for different applications, which benefit both user-level and kernel-level applications; 2) Retaining traditional VM mechanism, programmers and uses have the freedom to select CMP or not; 3) Improving kernel applications performance by eliminating remote swapping; 4) Avoiding loan while in debt *problem with dynamic workload; 5) Providing optional memory servers to further improve performance. In our testbed with CMP-based swap devices, Qsort gets 83.28% improvement comparing with the case using disk-based swap devices.*

1. INTRODUCTION

Cluster systems have been widely used in commercial data centers. Unlike those used in high performance computing environments, which take performance as their first priority, these clusters have to balance between cost and performance. In general, those clusters consist of ordinary computers don't have large memory and storage space. As a consequence, in some cases, memory resource on one node can't satisfy application requirements. Therefore, some nodes have to frequently swap in and out data. At the same time, in those systems, memory space on some other nodes are idle. This phenomenon results in uneven utilization of system memory and degrade system performance[1].

Many researches have been conducted on improving the memory utilization on cluster systems. These efforts can be divided into two categories: the first one is trying to evenly distribute the requests to achieve load balancing, which is beyond our paper. The second one is trying to build distributed sharing memory to exploit remote idle memory. In this case, remote idle memory can be plugged into traditional memory hierarchy as shown in Figure 1. It is admitted that the usage of remote memory can fill the widening performance gap between main memory and local hard disk through today's high-speed, low-latency network. Previous strategies and implementation [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] provided many feasible mechanisms to increase memory utilization, but complexities inherited in traditional network and consistency protocols counteract their advantages. Also, the lack of flexible interfaces limits their usage for end-users and developers. For instance, most of these mechanisms only support either computing-intensive applications or data-intensive applications. No single system can fulfill the requirements of both applications. Moreover, today's kernel applications have higher and higher memory requirements, but those mechanisms failed to provide enough support. Clearly, a new memory collaboration strategy is needed.

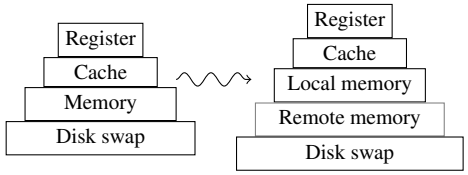

Figure 1. Remote memory in the memory hierarchy

With widely deployment of today's new generation networks, such as 10GbE and InfiniBand, it's feasible to create an efficient protocol to bundle memory spaces located on different nodes together as a resources pool. In case one node uses all of its local memory, it can utilize memory in other machines at low cost. Many popular memory hungry applications, such as multimedia, geographical information processing, etc. will benefit from this.

In this paper, we propose Collaborative Memory Pool (CMP) for this propose. Figure 2 introduces the structure of CMP system.

CMP has the features listed as following:

Flexible interface CMP has rich interfaces for applications either computing-intensive or data-intensive, either user-level or kernel-level. Malloc-like calls are used to `malloc()` memory from and `free()` them to CMP. Meanwhile, virtual disks can be built over CMP memory. Those disks can host file systems, or be used for swapping. Kernel APIs are provided so that kernel-level applications can use remote memory to improve performance as well;

2007 International Conference on Parallel Processing (ICPP 2007)
0-7695-2933-X/07 $25.00 © 2007 IEEE

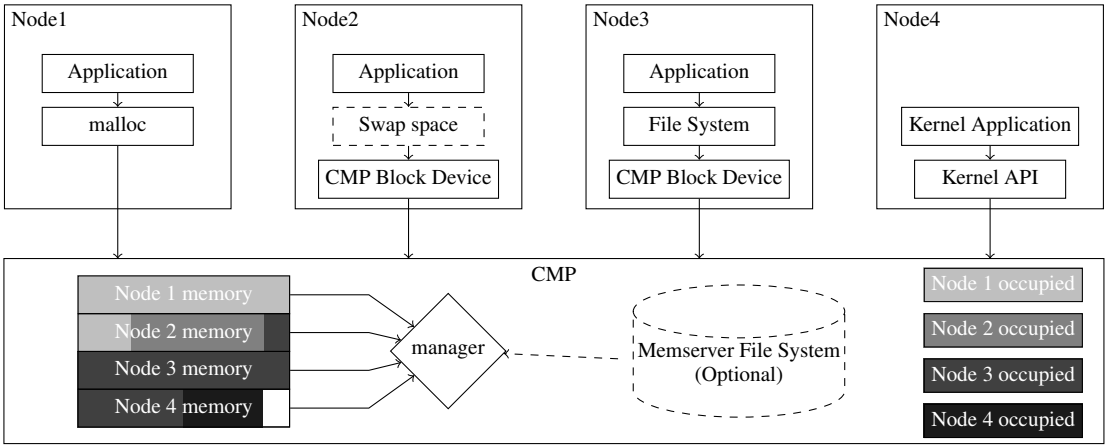

Figure 2. CMP Model

Complementing VM mechanism CMP is not a substitute of traditional memory allocation mechanism. Programmers and users can choose to use CMP memory or not. Though CMP provides huge memory space, critical applications may get worse performance by using CMP instead of using local memory only. To avoid this, they can bypass CMP to guarantee their behavior and performance. Meanwhile, new applications can choose which data can be saved to remote memory and which data should stay local;

Eliminating swapping for kernel applications CMP allows user-level applications access remote memory transparently by *Remote paging*. However, the usage of swapping in remote paging makes it a high-cost mechanism. For small, random accesses, reading or writing 1 byte may cause 8Kbytes (2 pages) or more data transfers. CMP provides memcpy-like kernel APIs to access those data segments directly, and eliminates that cost. When kernel applications require small data segments from CMP memory, they can select those APIs. Only those small segments are transferred;

Avoiding *loan while in debt* problem In case of workload thrashing, it is possible that some nodes loan their local memory to others while themselves are using remote memory, vice versa (nodes may borrow remote memory while themselves have idle memory). Such a nasty topology wastes network bandwidth and reduces performance. We call it *loan while in debt* problem. CMP's migration algorithm guarantees that each node tries to use its local memory to satisfy local applications at large.

Optional memory servers The structure of CMP can be serveless, but adding memory servers is allowed. Memory servers provide large memory space by using their file systems and disks, but have lower performance. CMP tries to satisfy processes requirements with their local memory first, then remote idle memory. The activation of memory servers usually means there is no idle CMP memory left within cluster.

The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 describes the implementation of CMP in detail. Section 4 presents several experiments to evaluate the performance of CMP. Finally, section 5 gives out our conclusions and future work.

2. RELATED WORKS

There are many researches on pooling idle memory within cluster systems for different purposes. They provide many feasible mechanisms and implementations. These researches can be classified into two categories. One is to provide large, sharing virtual memory space for computing-intensive applications, the other is to build a storage device, that can host file-systems or be used as swap space, over remote memory.

GMM[2] is an early attempt of exploiting remote memory in client-server database system. In GMM, clients interact with each other via a central server. When a client read a page, it tries to get the page from server's memory first. If that page is not there, the server checks weather another client caches it. If so, the server asks that client to forward its copy to the reader workstation. GMM brought up a four-level memory hierarchy: *local client memory, server memory, remote client memory, server disk*. In other words, it uses remote memory as caches of server disk. The remote memory is shared by all clients. Due to the special data consistency requirement in database applications, delicate care must be taken. GMM has to sacrifice performance by not coordinating the contents of client caches; [4] solves this problem with algorithms requiring global knowledge of client cache contents, however, it increases complexity.

Distributed Shared Memory (DSM)[3, 14, 15] is another way of using remote memory. It provides a piece of virtual address space shared among processes on loosely coupled

processors. DSM maps remote memory into process address space, it seems that programmers can build their memory intensive programs on it. But like multiprocessor caches, DSM has to use consistency protocols as well. It also has same problems such as complexity and low performance.

Network Block Device (NBD) [16] and its derivatives have a totally different view on remote resource utilization. NBD is a Linux device driver (and has been marked deprecated by kernel developers) that allow machines to use remote block devices through TCP network. If remote devices are ram-disks, NBD can be viewed as a mechanism to use remote memory. NBD was proposed to substitute NFS, but it failed because it couldn't provide a data-sharing model like its competitor. However, NBD can be used for swapping purpose that original NFS cannot (there are also some researches on swapping over NFS, like [8]), it implements Figure 1's memory hierarchy in a direct way. Network swapping (Nswap[12], SDNMS[13], HPBD[10] and recent dRamDisk[17]) is an attractive idea and gets reasonable performance. It groups idle memory together, then splits it into pieces. Each node get one (or more) piece(s) and install them as swap devices. Some of these approaches have special memory-server while others depend on nodes donating memory. [7, 18] still focus on building file-systems over NBD-like remote disks under specific impetus.

However, virtual disks always have fixed capacity and hard to stretch or shrink. They often have static provider-and-consumer topology while serving and always need manual configuration when deploying. That means with a careless configuration, or unpredictable workload, idle nodes may deny offering their idle memory, even worse it may occupy remote memory (even busy nodes' memory) while busy nodes painfully swapping from local disks. Moreover, in a large, complex system, with serverless structure, sometimes nodes loan out their memory while themselves in debt.

[19] introduces so called *memory server* nodes to support *remote paging*. Memory servers are nodes whose memory is used for fast backing storage. Computation nodes can page from memory servers. When page fault occurs, operating system retrieves pages from memory servers, and in exchange, some local pages have to be stored on servers. With the kernel support, it can provide malloc-like interface for remote memory. They can also use `free()` to release remote memory when applications exit. The size of remote memory occupied by one node is dynamical, it varies while workload changes. However, in client/server model, the idle memory on clients is still wasted. In fact, adding memory servers is equal to adding low-performance memory. Though performance is improved, the resources utilization rate is unchanged if not worse.

GMS[6, 11] provides a serverless method for remote paging, and it solves the *loan while in debt* problem with its migrating algorithm. However, GMS implementation depends on specific OS — OSF/1. GMS substitutes OS's VM mechanism, forces applications to use remote memory even it may reduce the performance of critical applications. The global LRU algorithm in GMS needs the global knowledge of pages age within cluster. The requirement of the essential disk is nasty, results in low performance.

With the fast development of network storage technologies like iSCSI, kernel applications and devices drivers are becoming more and more memory hungry. [20, 21] show the significance of utilizing memory on iSCSI target; [22] presents a remote iCache model. Such a cache must reside in kernel mode because of performance consideration. However, none of above technologies can make kernel applications benefit from the utilization of remote memory.

We compare CMP with some of the existing related systems in Table1.

3. DESIGN AND IMPLEMENTATION

The CMP architecture is shown in Figure 3.

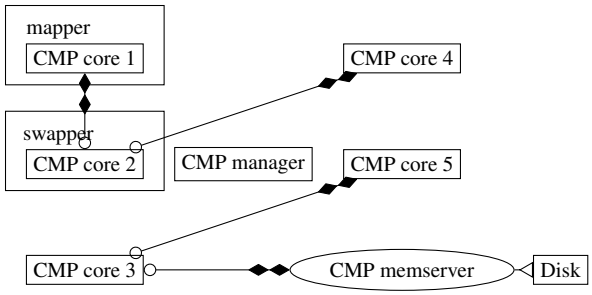

Figure 3. CMP Structure

CMP consists of three parts: CMP core, CMP manager, CMP memserver. They are connected with high speed network. CMP proposes a new concept of connector which makes it can be applied under various network conditions conveniently.

CMP provides a set of kernel APIs. Other kernel modules could use them directly. Two applications are constructed based on Kernel API: mapper and swapper. Mapper offers a *malloc-like* interface for user-mode programs. Swapper provides a virtual block device. Data-intensive applications could benefit from swapper, and computing-intensive applications could benefit form mapper.

CMP Core

Functions

CMP core is the core component of CMP. It manages local idle memory as a part of memory pool. It contains four main functions:

- Providing an interface to access memory pool for local applications.

Table 1. Comparison CMP with related system

System	Sharing accessing	Memory-like Interface	Disk-like Interface	Kernel Interface	Serverless	Dynamical while Serving	Potential loan while in debt
CMP	No	Yes	Yes	Yes	Optional servers	Yes	No
Share memory:							
GMM[2]	Yes	No	No	No	No	No	–
DSM[3]	Yes	Yes	No	No	Yes	Yes	Yes
Memto[15]	No (low level) Yes (Full system)	Yes	No	No	Yes	Yes	Yes
Remote swapping:							
Anemone[8]	No	Yes	Yes	No	No	No	–
Nswap[12]	No	Yes	No	No	No	Yes	–
SDNMD[13]	No	Yes	Yes	No	No	No	–
HPBD[10]	No	Yes	Yes	No	No	No	–
Remote paging:							
Memserver[19]	No	Yes	No	No	No	Yes	–
Reliable Remote Memory Pager[11]	No	Yes	No	No	Yes	Yes	Yes
GMS[6]	Yes	Yes	No	No	Yes	Yes	No

- Reporting memory usage to CMP manager.
- Offering local idle memory to other CMP cores.
- Requesting remote idle memory on other nodes for local applications.

Working mechanism

In CMP, the memory in one node is divided into two parts, one consists of the used memory and part of idle memory, ans this part is managed by OS. The rest memory is managed by CMP core. Those memory on different nodes is bundled to establish a big memory pool (called Collaborative Memory Pool). When an application runs out of memory, it asks for CMP's help. CMP core in this node will find free blocks for the application. The proceduce is transparent to user. The application has to make an exchange, it sends a local page to remote node and then get the free block. This strategy is to ensure that the number of block managed by CMP core will not decrease.

Key data structure and algorithm

CMP uses `cmp_block` structure to describe a block. All these blocks are linked in five lists which are free_list, LLL, LRL, RLL, RRL. Free_list is a list for free blocks. The detailed functions of other four lists are described in table 2.

Two principles should be indicated in the design of CMP core:

- Unless all local blocks are used by local applications, local applications will not take blocks from other remote nodes.
- When releasing blocks in LLL, if LRL is not empty, CMP core will get blocks in LRL and move back them to LLL.

Example of allocating, accessing and freeing blocks

Table 2. LL, LR, RL, RR lists

List Name	User	Provider
LLL (Local-Local list)	local node	local node
LRL (Local-Remote list)	local node	Remote node
RLL (Remote-Local list)	Remote node	Local node
RRL (Remote-remote list)	Remote node	Remote node

Taken a three-node system as example, there are node A,B and C. Node A and B offer two blocks individually and node C provides one block. As shown in figure 4.

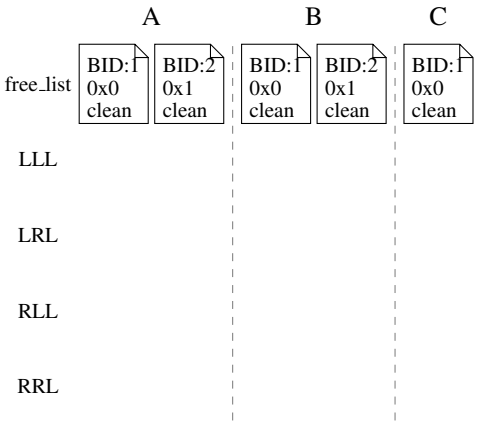

Figure 4. The initial state

As shown in figure 5, node A requests for one block and node B requests for two. In node A, CMP core gets a

free block from free_list and links it at the tail of LLL. The procedure is similar in node B.

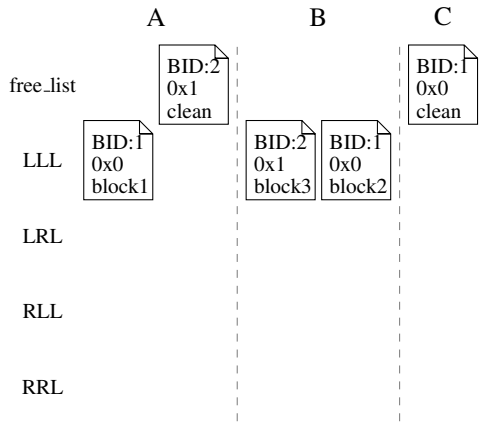

Figure 5. Node A requests for one block and node B requests for two

In Figure 6, node B requests for one more block . CMP core asks CMP manager to locate a free block. The manager can return with a block on node A or C. We assume node A is selected. Node B's CMP core contacts with node A, transfers the content of block 2 (the last block at LLL) to it and node A put that block into RLL. At the same time, node B relink block 2 to it's LRL, the memory block it used before belongs to block 4 now.

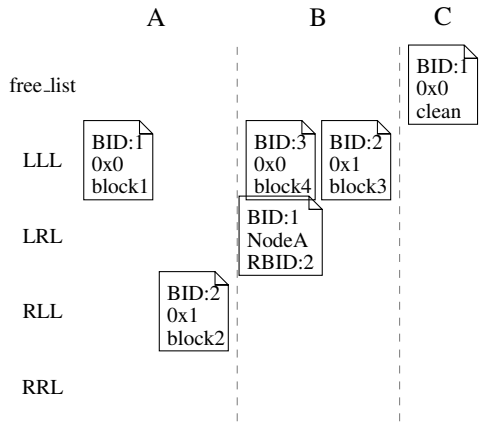

Figure 6. NodeB requests for one more block

In Figure 7, it's node A's turn to request for one more block. CMP core in node A finds that the free_list is empty and it has provided block to other node. So, it need recycle the shared block first. It needs a free block and the manager returns node C's free block. CMP core then turns to node C. After receiving acknowledgement, node A moves the content of node B's block 2 (node A doesn't care who is the host of block 2) to node C and moves its description to RRL. In node C, it inserts it into RLL. After that, node A gets a free block, it uses that block to fulfill its application.

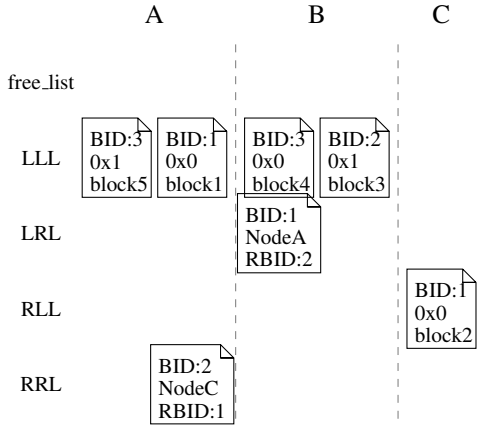

Figure 7. Node A requests for one more block

In figure 8, when node B accesses block 2, the LRL shows that the wanted block resides in node A and remote BID is 2. Then node A finds that the target block is located on node C now. It replies node B about that and deletes the related block description from it's RRL. Node B then turns to node C and finally finds the target block. It will take the following actions:

- Gets a block from the tail of LLL, which is block 3 in our case;
- Exchanges the content in block 3 and block 2 (BID:1), this action relinks block 3 to LRL, and makes block 2 linked to LLL. Physical memory used to hold block 3 now belongs to block 2;
- Finally, swaps the content with node C's block to retrieve block 2 and puts block 3's content to node C's block.

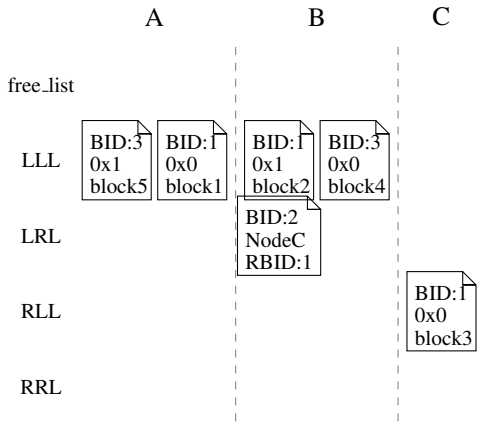

Figure 8. Node B accesses block 2

In figure 9 node B wants to free block 4. It checka if

LRL is empty. In our case, it's not, so it retrieves the content of the first block in LRL back from node C, then move the block to the tail of LLL.

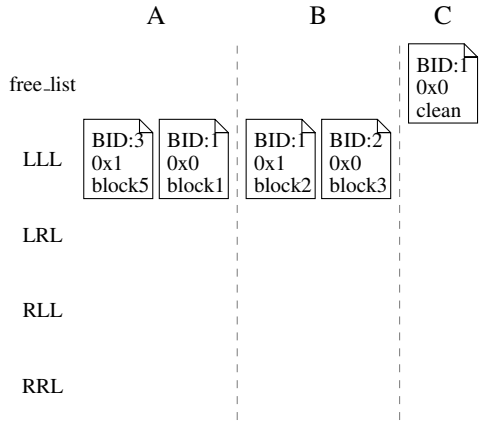

Figure 9. Node B frees block 4

CMP Manager

CMP manager is a user-mode program whose function is to locate free memory blocks. Every node connects to the manager and registers its quantity of free memory offered to CMP. Then the manager records all the information of connections and computing nodes and memory servers. When computing nodes request for memory, the manager looks for idle memory by polling and responds to nodes which have idle memory. According to the information returned from manager, CMP cores on the computing nodes communicate with those nodes. The manager does not need to perform complicated computation and avoids heavy data transfer. Therefore it can be deployed in either computing or memory servers nodes.

Memory Server

Memory servers could be considered as normal nodes with plenty of idle memory. In CMP architecture, memory servers are optional. However, memory servers can offer high capacity virtual memory based on disks. Meanwhile, in order to take full advantage of memory resources, memory servers offer storage space through filesystems. Linux filesystems optimize disks accesses with page buffers and write-back technology, keep disk's content into memory to improve performance of read and write system calls[23, 24]. On the contrary, accessing block device directly in [6, 9] can not benefit from these technologies.

Connector

Connector defines an abstract layer offering connect function in CMP. Different types of connectors are implement to provide a unified interface over different networks such as TCP/IP and InfiniBand without modifying core code in CMP. Connector's interfaces mirror UNIX socket's interfaces.

Kernel API

CMP provides two types of kernel APIs. The first one is based on remote paging (using CMP core), the other bases on *direct accessing*. When kernel application needs CMP memory, it can choose to retrieve the whole page by remote paging or to retrieve only the requested data by direct accessing. Direct accessing provides memcpy-like interface. It grabs only required data to specified kernel buffer, and keeps the whole page remotely. It is mush faster than remote paging which must pull the whole page back and push another page out.

Swapper and Mapper

Swapper and mapper are two kernel applications based on CMP. Swapper maps memory in CMP into virtual block devices. Applications can create filesystems on them or use block devices as swapping partitions. Mapper implements the *mmap()* system call to offer an *malloc-like* interface to applications. Applications can use CMP's memory like local physical memory.

4. EVALUATION

Experiments Setup

The experiments are conducted on three Dell PowerEdge SC430 nodes called dell-172, dell-175, dell-176. They are configured with Intel(R) Pentium(R) CPU 2.80GHz with 1024KB cache, 2GB memory and PCI-X 133MHz buses. They are connected by both Gigabit Ethernet and InfiniBand network with IB driver of IBG2. Two disks resides on the nodes, one is seagate 40GB, the other is 80GB SATA disk that is manufactured by Westdigit. The operating system is SUSE 10.0 with Linux kernel 2.6.13.

Software are configured as following: Manager resides on dell-172. CMP runs on all three nodes. Both dell-172 and dell-175 contribute 512M memory to pool, and the contribution of dell-176 is a variable that will be specified in experiments. All the experiments are conducted on dell-176 without memory server in our system.

Experiments Tools

Qsort

Qsort can be used to evaluate performance of memory-intensive applications. It uses the code in Linux kernel (2.6.11.12) XFS (the code can be found in kernel source tree: fs/xfs/support/qsort.c) in user mode. Qsort benchmark can perform large in-memory sorting with quick-sort algorithm. The data array is initially ordered, Qsort sorts it into reverse ordered. The time complexity of quick-sort algorithm is $O(N \log N)$, but with ordered initial data, quick-sort is actually bubble sort, and its time complexity becomes

$O(n^2)$. With ordered initial data, the result in different sorting is comparable and analysable. In each experiment, It does three sorts and calculates the average processing time. In the following experiments, we use Qsort to sort 1GB data (134217728 `unsigned long` integers in x86_64 machine).

IOzone

IOzone is a filesystem benchmark which generates and measures a variety of file operations. IOzone is very useful for performing a broad filesystem analysis on a vendor's platform. The benchmark tests I/O performance for the following operations: Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread ,mmap, aio_read, aio_write.

Performance Evaluation

Qsort result

We run Qsort in three scenarios: system memory with disk swap, CMP memory (512MB local and 512MB remote) with mapper and system memory with swapper as swap partition. Results in Figure 10 show that the average processing time in the three cases were 1008s, 168s, 687s. Mapper performs the best, it can achieve 83.28% improvement. While swapper's performance is a little poorer, it still runs 1.5 times faster than using disk swap.

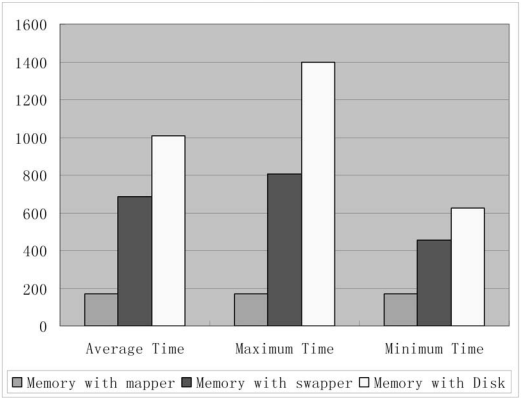

Figure 10. Qsort run time of memory with mapper, swapper and disk

IOzone result

In IOzone test, we use swapper as a block device, with ext3 filesystem on it, and then run IOzone on it.

The result of IOzone is not very promising. We list two figures here. Figure 11 is the result when the size of IOzone's test file is 512MB and accessing size is 16KB. In figure 12, the size of IOzone's test file is 1GB and accessing size is 16KB, too. From the results, we observe that the performance of swapper is close to disk.

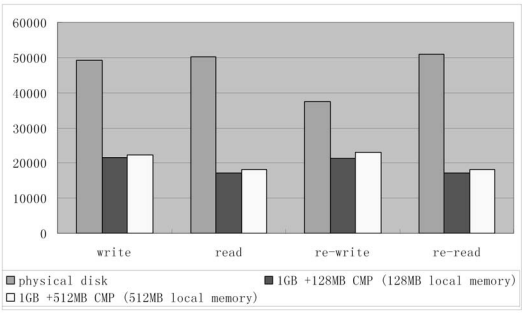

Figure 11. IOzone result when testing 1GB file

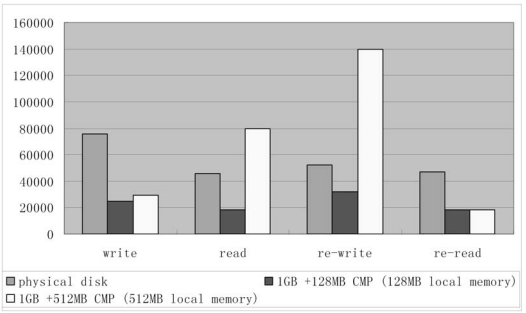

Figure 12. IOzone result when testing 512MB file

CMP's data access protocol is based on "swap", which means if we want to read or write a remote data block, CMP will send a block out and retrieve the full excepted block (in our implementation, block size is 4KB) even the wanted data is just 512 bytes. Furthermore, in QSort experiments, local memory acts as caches and cache hit rate is very high (nearly 70% in our experiments). But in IOZone, the local cache's hit rate is very poor. For above reasons, IOZone test result is not satisfied.

5. CONCLUSION AND FUTURE WORK

In this paper, we introduce a new mechanism to exploit idle memory in cluster system — CMP. Compared with old methods, CMP has many advantages. First, CMP has flexible interfaces that benefit both computing-intensive and data-intensive applications, both user-level applications and kernel-level applications. Second, CMP is Complementary to traditional VM mechanism, letting programmers and users to decide whether to use CMP memory or not, offers flexibility. Furthermore, CMP eliminates remote swapping in kernel applications, which reduces accessing cost. CMP also solves the *loan while in debt* problem with its migration algorithm. Finally, CMP has an *optional memory servers* structure that also increases flexibility.

We conducted experiments with QSort, the results show that CMP is practical. The performance improvement is great, it gains 83.28% improvement comparing with disk-

based swap. However, IOZone experiments show that in some scenarios, the performance of CMP is not as good as expected. We have explained the reasons of that. We believe with the "direct access" protocol, CMP can boost such applications as well as Qsort. In our future work, we will conduct those experiments to verify this protocol. We also intend to investigate designs that can eliminate copy and synchronization cost with zero-copy RDMA operations.

ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation Grant CNS-0509207 and National Basic Research Program of China (973 Program) 2004CB318202. We gratefully acknowledges the support of K. C. Wong Education Foundation, Hong Kong. We would also like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] A. Acharya and S. Setia, "Availability and utility of idle memory in workstation clusters," *ACM SIGMETRICS Performance Evaluation Review*, 1999.

[2] M. J. Frankling, M. J. Carey, and M. Livny, "Globla memory management in client-server dbms architectures," in *Proceeding of the 18th VLDB Conference*, Aug. 1992.

[3] Kai Li and Paul Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.

[4] A. Leff, J. L. Wolf, and P. S. Yu, "Replication algorithms in a remote caching architecture," *Parallel.and.Distributed.Systems,.IEEE Transactions.on.*, vol. 4, no. 11, pp. 1185–1204, 1993.

[5] E. A. Anderson and J. M. Neefe, "An exploration of network ram," Tech. Rep. CSD-98-1000, UC Berkley, Dec. 1994.

[6] M. Joseph Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing global memory management in a workstation cluster," *ACM SIGOPS Operating Systems Review*, pp. 201–212, 1995.

[7] M. D. Flouris and E. P. Markatos, "The network ramdisk: Using remote memory on heterogeneous nows," *Cluster Computing*, vol. 2, no. 4, pp. 281–293, 1999.

[8] M. R. Hines, M. Lewandowski, and K. Gopalan, "Anemone: Adaptive network memory engine," M.S. thesis, Florida State University, 2003.

[9] S. Koussih and S. Setia A. Acharyam, "Dodo:a user-level system for exploiting idle memory in workstation clusters," in *Proceeding of the Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.

[10] S. Liang, R. Notonha, and D. K. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," *IEEE Cluster Computing*, Sept. 2005.

[11] E. P. Markatos and G. Dramitions, "Implementation of a reliable remote memory pager," in *Proceeding of the 1996 Usenix Technical Conference*, 1996.

[12] T. Newhall, S. Finney, K. Ganchevm, and M. Spiegel, "Nswap:a network swapping module for linux clusters," in *Proceeding of Euro-Par'03 International Conference on Parallel and Distributed Computing*, Klagenfurt, Austria, Aug. 2003.

[13] H. Tang Sun, M. Chen, and J. Fan, "A scalable dynamic network memory service system," in *Proceeding of High-Performance Computing in Asia-Pacific Region*, 2005.

[14] B. Nitzberg and V. Lo, "Distributed shared memory: a survey of issues and algorithms," *Computer.*, vol. 24, no. 8, pp. 52–60, 1991.

[15] T. S. Trevisan, V. S. Costal, L. Whately, and C. L. Amorim, "Distributed shared memory in kernel mode," in *Proceeding of Computer Architecture and High Performance Computing*, 2002.

[16] P. Machek, "Network block device (tcp version)," http://nbd.sourceforge.net/.

[17] V. Roussev, G. Richard, III, and D. Tingstrom, "dramdisk: efficient ram sharing on a commodity cluster," in *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, 2006.

[18] Yun Mao, Youhui Zhang, Dongsheng Wang, and Weimin Zheng, "Lnd: a reliable multi-tier storage device in now," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 1, pp. 70–80, 2002.

[19] L. Iftode, K. Li, and K. Petersen, "Memory servers for multicomputers," in *Proceeding of the IEEE Spring COMPCON 93*, Feb. 1993, pp. 538–547.

[20] X. He, Q. Yang, and M. Zhang, "A caching strategy to improve iscsi performance," in *Proceeding of Local Computer Networks*, 2002.

[21] Jizhong Han, Dan Zhou, Xubin He, and Jinzhu Gao, "I/O profiling for distributed ip storage systems," in *Proceeding of The Second International Conference on Embedded Software and Systems*, Dec. 2005.

[22] Xuhui Liu, Nan Wang, Guozhong Sun, Jizhong Han, Lisheng Zhang, and Chengde Han, "Remote iscsi cache on infiniband: An approach to optimize iscsi system," *icppw*, vol. 0, pp. 527–534, 2006.

[23] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly, 3rd edition, 2005.

[24] Robert Love, *Linux Kernel Development*, Sams Publishing, 2nd edition, 2005.