Cluster-Aware Cache for Network Attached Storage*

Bin Cai, Changsheng Xie, Qiang Cao

National Storage System Laboratory, Department of Computer Science,
Huazhong University of Science and Technology, Postfach 430074,
Wuhan, P.R.China
hust_caibin@sohu.com

Abstract. Decentralized, cooperative and large-scale distributed storage systems that consist of a cluster of storage nodes attached with local disks can deliver high resource utilization, high availability and easy scalability. This paper describes the design and prototype implementation of a novel Cluster-Aware Cache (CAC) algorithm that shares memories between nodes in cluster to construct an efficient and cooperative cache-to-disk accesses policy. The difference between our scheme and previous studies is that processes on different node can access the same page concurrently. Furthermore, CAC algorithm is also well suited to heterogeneous clusters where one or more nodes may have larger amounts of memory than the others. The performance measurements with a Web server on our system show dramatic performance improvements with increasing number of nodes.

1 Introduction

Large-scale distributed storage systems that consist of a cluster of storage nodes with local disks have become a cost-effective solution for wide range of applications, ranging form enterprise-class storage backend, HPC (High-Performance Computing) to data mining and Internet services. Such systems can be realized at little or no extra cost, can offer an inherently scalable aggregate I/O bandwidth, and can take advantage of existing cluster installations through double-use or upgrade of older hardware. Although the parallelism offered by the numerous disks in a cluster can alleviate the I/O bandwidth problem, it does not really address the latency issue which is largely limited by seek and rotational costs. Caching data blocks in memory is a well known way of reducing I/O latencies, provided we can achieve good hit ratio.

In this paper, we describe the design of a storage cluster using inexpensive PCs equipped with local disk. In our system, large files are stored in a scalable fashion by striping the data across multiple nodes to obtain high aggregate bandwidth. In order to solve the disks latency issue, we present the design and prototype implementation of a novel Cluster-Aware Cache (CAC) scheme, which changes the cache hierarchy

* This research is supported by National 973 Great Research Project of P.R.China under the grant No. 2004CB318200 and National Natural Science Foundation under grant No. 60273037 and No. 60303031.

of traditional distributed system (client cache, server cache, server disk) by letting one node cache misses to be checked against other node caches before the local storage devices. Thus, the working set can grow beyond the local memory limit while applications read latency can be alleviated tremendously because remote caches were accessed faster over high-speed network than the disk even if it is local.

The remainder of the article is organized as follows: In section2, we describe the related work about cooperative caching scheme. In section3, we introduce the architecture of our storage cluster system, and detail the CAC scheme in section 4. The experimental results are evaluated at section 5. The conclusion comes at section 6.

2 Related Work

Using regular nodes as storage nodes has previously been suggested in the Slice [1,2] and OPIOM [3] projects, but where they primarily focus on using dedicated storage nodes, we examine the possibilities for distributing the load across all nodes in a cluster. The Network Block Device (NBD) [4] and GNBD/VIA [5] also provide network access to a remote block device, but the architectures are neither modular nor extensible. The xFS [6,7] introduced the notion of cooperative caching. Other I/O buffer cache management schemes exist on global memory management and cooperative caching [8] by extending the use of a shared distributed buffering mechanism to the I/O devices themselves. PACA [9] is another cooperative file system cache. It attempts to avoid replication and the associated consistency mechanisms by allowing only one cached block copy in the entire cluster-wide cache. That is possible since PACA uses a *memory copy* mechanism (a sort of Remote DMA) to send the data from the cache to the user memory. However, every data access has to go through this *memory copy* mechanism which is clearly much slower than accessing a local block copy. Other low level approaches to remote I/O include Swarm [10] and Network-Attached Secure Disks (NASDs) [11]. Swarm offers the storage abstraction of a *striped log* while NASDs provide an object-oriented interface.

3 System Architecture

In this section, we describe the main components of NAS storage cluster system and how they work together. We first provide an overview of the architecture, and then we cover the CAC algorithm in more detail.

The physical layout of such storage cluster is shown in figure 1. To provide an interface of a single virtual cluster server, each cluster is assigned with a multicast IP address. All participating cluster members joint in this multicast group, whose IP address is known to each other. The main advantage of NAS approach is that internally the design can seamlessly integrate major storage components to work closely together. All members in this system work collaboratively to construct a storage system with a unified storage space.

Each of the storage device members in the cluster runs a program, called daemon. Daemon communicates with each other and provides some functions, including transferring file data, transferring control message, and performing statistical information. When a node needs to get file at other node, the daemon finds the file firstly, and then gets the file from remote node. Daemon checks usage of the CAC at a fixed interval in order to provide reasonable cache replacement policy. When a node's residual cache capacity is less than the threshold value, the daemon will move some blocks from its local cache to the remote cache in order to balance the load.

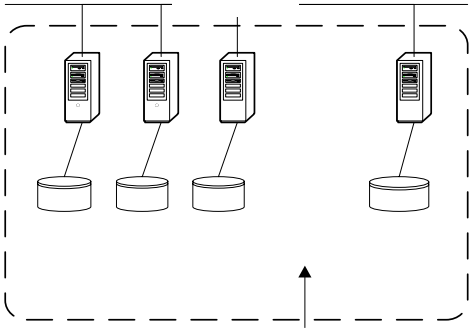

Fig. 1. Architecture of NAS Storage Cluster

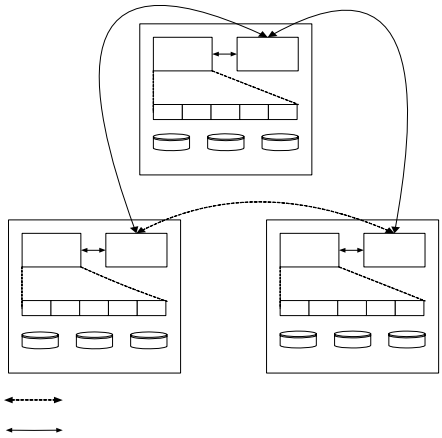

Fig. 2. Daemon and CAC Kernel Module

Each of the storage device members in the cluster also has a kernel module. It divides the node's total memory into two parts: one is the node's local cache; the remainder memory at each node therefore makes of the CAC cache spaces. The typical setup and possible scenarios are shown in figure 2.

CAC hides the distributed nature of the cluster node's caches by offering the local hosts an interface to a global unified buffer cache. Similar to GMS [3], CAC uses a high-level abstraction (disk blocks) to deal with remote resources and cooperative cache algorithms to jointly manage the cluster caches. It rely on the low communication latencies of powerful interconnects to minimize block access times.

4 Implementation of CAC Scheme

Each node has a local cache to cater to the individual process requests at that node; and upon a miss goes to a shared cooperative cache running on one or more nodes of the cluster which can possible satisfy requests that come from different nodes.

4.1 Local Cache

We opted to implement the local cache within the Linux kernel that can be shared across all the processes running on that node. Only when the request misses in this cache (either all or some of the request cannot be satisfied locally), is an external request initiated out of that node to the cooperative cache. This cache is implemented using open hashing with second chance LRU replacement. There is a dirty list, a free list, and a buffer hash to chain used blocks for faster retrieval and access. The hashing function takes as parameters the inode number of the file and the block number to index the buffer hash table. There are two kernel threads called *flusher* and *harvester* in the implementation. Writes are normally non-blocking (except the *sync write* explained later), and the flusher periodically propagates dirty blocks to the cooperative cache. The harvester is invoked whenever the number of blocks in the free list falls below a low water mark, upon which it frees up blocks till the free list exceeds a high water mark. A block size of 4K bytes is used in our implementation. Note that such a kernel implementation automatically allows multiple applications/processes to share this local cache, thus making more effective use of physical memory.

4.2 Global Unified Cache

The cooperative cache, as explained earlier, adds one more level to the storage hierarchy before the disk at one node to be accessed, and we go over it in the following discussion, explaining the base algorithm in our implementation.

Currently, we use a separate cooperative cache for each file. If there is little file sharing across applications, or even across parallel processes of the same application, then the requests would automatically distribute the load more evenly with this approach. Since we would also like to be able to perform inter-application optimizations based on sharing patterns, we have opted to share the cooperative cache across applications. This can help one application benefit from the data brought in earlier by another from the cache. This feature is one key difference between our system and GMS [8] where the global cache is intended for optimizations within the processes of a single application. Similar to the local cache implementation, we implement the cooperative cache within the Linux kernel.

The internal data structures and activities of the cooperative cache are more or less similar with those for the local cache that were described earlier. One could designate such global caches on different nodes, particularly on those nodes with larger physical memory (DRAM). Consequently, this architecture is also well suited to heterogeneous clusters where one or more nodes may have larger amounts of memory than the others. The base algorithm of CAC is described in following pseudo-code:

```
Application issues file request;
if (file is at local cache){
  give the file to application;
  return;
}else{
  if (file is at remote cache){
repeat-remote:
```

```
    use Daemons Communication to fetch the file;
    if ( the file is hot){
repeat-local:
      if (local cache has space){
        add the file to local cache;
        give the file to application;
        return;
      }else{
        give the file to application;
        return;
      }
    }else{
      give the file to application;
      return;
    }
  }else{
    if (file is at local storage devices){
      goto repeat-local;
    }else{
      if (file is at remote storage devices){
        goto repeat-remote;
      }else{
        can not find the file;
        return error;
      }
    }
  }
}
```

4.3 Daemon Communication

Each node runs a user-level daemon program for the purpose of transferring file data, transferring control message, and performing statistical information. TCP/IP sockets are being explicitly used for sending messages to it from the individual local caches regardless of which application process is making a call. The convenience and flexibility of a user-level implementation has led us to implement the daemon running on each node of our cluster serving requests to a specific file running on a cluster node, to which explicit requests are sent by the local caches, and is shared by different applications.

When a node needs to get file at other node, the daemon finds the file firstly, and then gets the file from remote node. Daemon checks usage of the CAC at a fixed interval in order to provide reasonable cache replacement policy. When a node's residual cache capacity is less than the threshold value, the daemon will move some blocks from its local cache to the remote cache in order to balance the load. The process of communication between daemon and CAC is presented as following pseudo-code:

```
wakeup (Daemons Communication);
if (available CAC cache size < threshold_ CAC _size){
  discard some blocks;   // replacement policy
```

```
  sleep;
}else{
  if (available local cache size < thresh-
old_local_size){
    find the node with more available cache space;
    migrate the blocks to that node;
    sleep;
  }else{
    sleep;
  }
}
```

5 Experimental Results

In this section, we present an evaluation of the performance of the CAC prototype. The performance goal of CAC is to have a performance close to that of local cache and to have a small overhead on the nodes hosting the disks.

For the purpose of our experiments, we constructed a small cluster with five equivalent Pentium4/2GHz PCs with 256MB DRAM running Linux operating system with the version of the 2.4 kernel. Each PC was equipped with a single Gigabit network card as well as a single 160G IDE hard disk to provide storage space. No special kernel optimizations were done to optimize I/O or inter-process communications. One node was dedicated to servicing HTTP requests and the other four nodes were available to service data storage. Each node was running a daemon and a CAC kernel module, and shared 128MB RAM to consist of cooperative cache, therefore, the cooperative cache cache capacity is 640MB. Each client was a Windows PC running the WebBench [12] suite of e-commerce tests. WebBench is a benchmark program that measures the performance of Web servers using PC clients. In our test, the file set was 1.5GB and was placed on the disk of Web server initially, and we used 10 clients running WebBench to generate 1000 WebBench-client requests to Web server simultaneously. The experiments were done according to the request and cache hit ratio, irrespectively.

A comparison between remote disk, local disk, remote cache, and local cache in terms of the number of request operations is presented in figure 3. Notice that Web server's performance upgrades fast with the increasing number of the node. Large parts of requests hitting in Web server's local cache improves the I/O performance of Web server dramatically. Similar comparison in terms of request bytes is presented in figure 5.

A comparison between cache hit ratios in terms of the number of request operations is presented in figure 4. Notice that I/O in Web server's local disk degrades fast with the increasing number of the node, large parts of requests hitting in Web server's local cache or remote cache improves the Web server's I/O performance. Similar comparison in terms of request bytes is presented in figure 6.

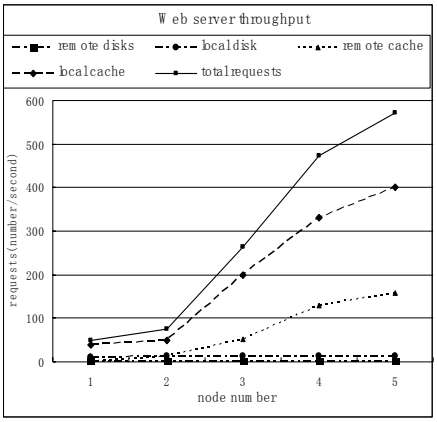

Fig. 3. The Comparison of Request Number

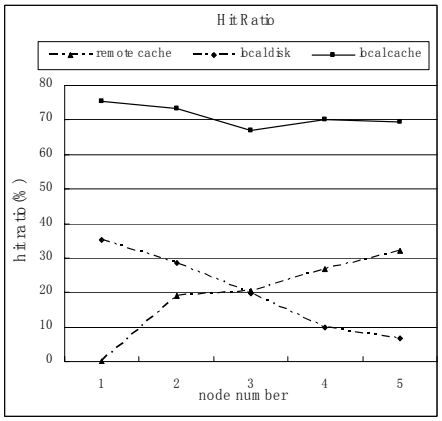

Fig. 4 The Comparison of Hit Ratio

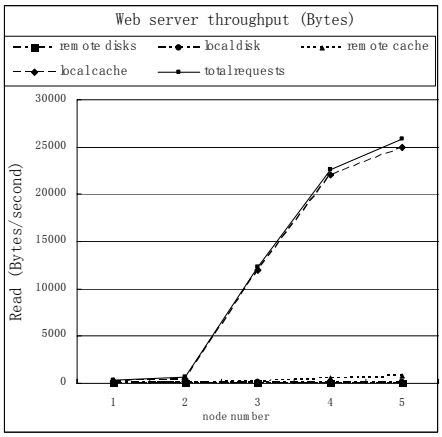

Fig. 5. The Comparison of Request Bytes

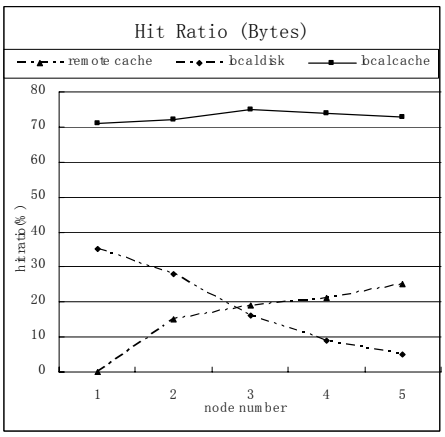

Fig. 6. The Comparison of Hit Ratio

6 Conclusion

By leveraging the high-speed communication afforded by the cluster interconnect such as Fast/Gigabit Ethernet, large files can be stored in a scalable fashion by striping the data across multiple nodes; by distributing the disks across a sufficient number of cluster nodes, high aggregate bandwidth can be easily obtained with current hardware. In order to solve the disks latency issue, we present the design and prototype implementation of a novel cluster caching scheme, which changes the cache hierarchy of traditional distributed system (client cache, server cache, server disk) by letting one node cache misses to be checked against other node caches before the local storage devices. Thus, the working set can grow beyond the local memory limit

while applications read latency can be alleviated tremendously because remote caches were accessed faster over high-speed network than the disk even if it is local. Performance measurements of such a system are encouraging, showing that the I/O performance of Web server improves fast with the increasing number of node.

References

1. D.C.Anderson, J.Chase, and A.Vadat, "Interposed request routing for scalable network storage", *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.
2. J.Chase, D.Anderson, A.Gallatin, A.Lebeck, and K.Yocum, "Network I/O with trapeze" *Proceedings of 1999 Hot Interconnects Symposium*, August 1999.
3. P.Geoffray, "OPIOM: Off-processor I/O with myrinet", *Proceedings of the first ACM/IEEE International Symposium on Cluster Computing and Grid*, May 2001.
4. P.T.Breuer, A.M.Lopez, and A.G.Ares, "The network block device", *Linux Journal*, (73), May 2000.
5. K.Kim, J.Kim. and S.Jung, "BNBD/VIA: A network block device over virtual interface architecture on Linux", *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, April 2002.
6. M.Dahlin, R.Yang, T.Anderson, and D.Patterson, "Cooperative Caching: Using remote client m emory to improve file system performance", *Proceedings of first Symposium on Operating Systems Design and Implementation*, November 1994.
7. T.Anderson, M.Dahlin, J.M.Neefe, D.Patterson, D.Rosseli, and R.Y.Wang, "Serverless network file systems", *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.
8. M.I.Feeley, W.E.Morgan, F.H.Pighin, A.R.Karlin, and H.M.Levy, "Implementing global memory management in a workstation cluster", *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 201-212, December 1995.
9. T.Cortes, S.Girona, and L.Labatra, "PACA: A distributed file system cache for parallel machines. Performance under Unix-like workload", *Technical Report UPC-DAC-RR-95/20 or UPC-CEPBA-RR-95/13*, Department d'Arquitectura de Computadors, Universittat Politecnica de Catalunya, 1995.
10. J.H.Hartman, I.Murdock, and T.Spalink, "The Swarm scalable storage system", *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS 99)*, June 1999.
11. G.A.Gibson, D.F.Nagle, K.Amiri, F.W.Chang, H.Gobioff, E.Riedel, D.Rochberg, and J.Zelenka, "File systems for network-attached secure disks", *Technical Report CMU-CS-97-118*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, July 1997.
12. http://www.veritest.com/benchmarks/webbench/