CARDs: Cluster-Aware Remote Disks

Vlad Olaru Walter F. Tichy

Computer Science Department
Karlsruhe University, Germany
E-mail: {olaru,tichy}@ipd.uka.de

Abstract

This paper presents Cluster-Aware Remote Disks (CARDs), a Single System I/O architecture for cluster computing. CARDs virtualize accesses to remote cluster disks over a System Area Network. Their operation is driven by cooperative caching policies that implement a joint management of the cluster caches. All the CARDs of a given disk employ a common policy, independently of other CARD sets. CARD drivers have been implemented as Linux kernel modules which can flexibly accommodate various cooperative caching algorithms. We designed and implemented a decentralized policy called Home-based Serverless Cooperative Caching (HSCC). HSCC showed cache hit ratios over 50% for workloads that go beyond the limit of the global cache. The best speedup of a CARD over a remote disk interface was 1.54.

1 Introduction

High-speed System Area Networks (SAN) have latencies and bandwidths comparable to those of a memory subsystem. This makes a case for integrating cluster resources into Single System Image services. Integrating distributed resources into a single system was the subject of substantial prior research. User-space communication subsystems [23, 20] improved the SAN performance by removing the kernel from the critical path. However, message passing is not a handy programming model and high-level software abstractions (memory pages, disk blocks, etc.) were developed. A notable research effort in this direction was that of software Distributed Shared Memory systems [1, 24].

Cooperative caching network filesystems [2, 8] changed the distributed filesystem memory hierarchy (client cache, server cache, server disk) by letting client cache misses to be checked against other client caches before the server cache. Thus, the working set grew beyond the local memory limit while read latency improved because remote caches were accessed faster than the disk (even if it was local).

Flexible/extensible kernels [5, 9] have shown a better response to the challenges raised by the new class of highly intensive I/O-bound applications (mostly related to multimedia and Web/Internet) than conventional general-purpose kernels. These systems use a joint management of the resources. Applications manage alone their own resources while the system software continues to provide general mechanisms such as protection domains, resource allocation, scheduling, etc.

Our work is inspired by all these trends. CARD drivers hide the distributed nature of the cluster disk caches by offering the local hosts an interface to a global unified buffer cache (from hereon called *cooperative cache*). Similar to DSMs, CARDs use a high-level abstraction (disk blocks) to deal with remote resources and cooperative caching algorithms [8] to jointly manage the cluster caches. They rely on the low communication latencies of powerful interconnects to minimize block access times. Applications may download their own caching policies into the CARD driver. Thus, the kernel provides the block access mechanism (the CARD driver) while applications can specify their block management policy at will. Different caching policies can be in use at the same time in the cluster, but the set of CARDs of a given disk must employ a common policy.

This paper evaluates the performance of CARDs as a distributed storage system. We present and evaluate a simple and efficient decentralized cooperative caching policy, Home-based Serverless Cooperative Caching. We emphasize the flexibility of policy choice in our system by implementing and evaluating another cooperative caching algorithm, Hash Distributed Caching [8].

2 Cluster-Aware Remote Disks

CARDs are block devices that virtualize remote disk accesses over a SAN. They can be mounted on the local system as regular block devices. Without cooperative caching, a CARD behaves like a remote disk interface (RD). Every miss in the local buffer cache is checked by the RD also against the remote buffer cache at the physical disk node. A set of CARDs using a common cooperative caching pol-

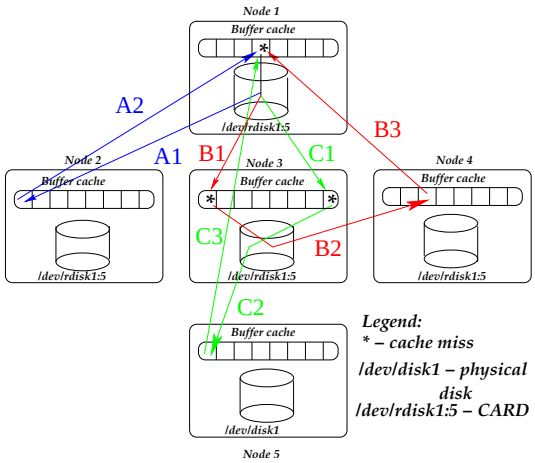

Figure 1. Cooperative caching with CARDs **Case A**: client-to-client cooperation; **Case B**: three-client cooperation; **Case C**: client-to-client cooperation fails. The block must be retrieved from disk

icy implements a joint management of their corresponding caches. CARDs offer a flexible and easy to use scheme of building the cooperative cache. Just by mounting CARDs of a given disk on a set of cluster nodes, one gets shares of each buffer cache in the cluster for cooperative use. The cooperative cache size is thus the sum of these shares.

Managing the cooperative cache is a task orthogonal to that of physically moving blocks to and fro and is expressed by means of a cooperative caching policy. Essentially, such a policy defines how to find blocks in the cooperative cache, what should be done with the evicted blocks, whether multiple copies of the block may exist in the cooperative cache and, if so, how consistency is handled. Blocks not found in the local caches can be looked for in the cooperative cache. Only when there is no available copy in the entire cooperative cache, the block request is serviced from the disk. A typical setup and possible scenarios are shown in **Figure 1**.

Cooperative caching policies Cooperative caching has been originally developed for network file systems ([8, 2]). Our approach tackles the topic at storage system level. Instead of reasoning about a filesystem hierarchy, we prefer to view cooperative caching as a way to enlarge local disk caches by using remote caches. Thus, the upper layers are offered the view of a unified buffer cache across the cluster. This approach seems more general to us, as one can build on top of our storage system not only filesystems but also databases or Web storage systems.

A cooperative caching policy is a set of four operations executed by the CARD drivers in response to various events during the lifetime of a disk block in the cooperative cache. These operations exercise in fact a distributed management of the cooperative cache. Writing a cooperative caching policy means to provide code for these operations. By default, they are void and CARDs behave like remote disk interfaces. The C definition of a cooperative caching policy looks like this:

```
struct coop_caching_ops {
    int    (*lookup)(struct request*);
    int    (*handle_eviction)(struct
                        buffer_head*);
    int    (*keep_consistency)(kdev_t,
                        struct sk_buff*);
    kdev_t (*handle_request)(kdev_t,
                        struct sk_buff*);
};
```

Block requests missing in the local cache are sent by the CARD driver to a node caching the block or to the disk. Searching the cooperative cache for a block copy is the job of *lookup*. It returns a node id.

Cooperative caching saves evicted blocks in idle remote caches in order to reduce the disk I/O activity and to improve read latencies and cache hit ratios. Single cached copies of a block (*singletons*) should be handled specially because discarding them means that the next request for that block anywhere in the cluster will have to go to the disk. *handle_eviction* looks for remote hosts for locally evicted blocks. It does not interfere with the native kernel eviction algorithm because CARDs are ordinary block devices and they obey this policy as any other local disk would do. *handle_eviction* extends this algorithm by taking advantage of the cooperative cache. On success, it returns the node id of the new block host. Otherwise, the block is dropped.

If cooperative caching uses block replication, a special operation to support consistent writes is needed. *keep_consistency* should define the actions to be taken in response to a block write event.

Handling incoming block requests is the task of *handle_request*. It decides whether or not a block request can be satisfied from the local cache. On a cache hit, the method serves the block and returns the corresponding local CARD or physical disk device identifier. Otherwise, the request is forwarded to another host. *handle_request* also defines how to deal with evicted blocks forwarded from other nodes.

3 HSCC: Home-based Serverless Cooperative Caching

According to the above guidelines, we designed Home-based Serverless Cooperative Caching (HSCC), a decentralized globally coordinated cooperative caching policy which attempts to offload the physical disk node by distributing the block delivery task. It is *home-based* because each block gets a *home* node which handles requests for that block. Homes are assigned to blocks by using a hashing *modulo n* scheme. Since most clusters don't change dynamically in

size, the choice of a fixed *n* is reasonable. The home may cache the block in its own buffer cache or may simply keep a hint telling which is the node caching the most recent block copy. Blocks get loaded at their homes lazily (on-demand) and only if there is enough room. Thus, HSCC partitions statically and evenly only the meta-data, not the actual data. The policy is called *serverless* because the disk blocks are not served by a traditional centralized server.

HSCC employs a per-node cache index to keep track of the nodes caching particular blocks of the disk. The node uses this index to forward block requests that miss in the local cache. In general, this may prove to be too space-expensive and may endanger scalability. For instance, using a bitmap to mark the nodes caching the block requires 16 bytes per entry to keep track of 128 nodes. 1024 nodes require 128 bytes per entry and so on, the index size grows steadily with the size of the cluster. For 1024 nodes with 256 MB each, 128 bytes for every 4KB block require roughly 8 GB (3.1% out of 256 GB).

To avoid this inconvenience, we chose to keep the index entry size constant, regardless of the cluster size. We store only two node IDs per index entry. When storing a block, a node records in the corresponding index entry the ID of the node delivering the copy. We call it the *previous* ID. The second ID is that of the next node requesting the block from the local node. We call it the *next* ID. Now all the block copies are in a distributed double-linked list in which *next* points to more recent block copies while *previous* refers to older copies. This solution doesn't reduce the overall size of the index, but distributes its information and thus reduces the local memory usage.

The overall index size can be reduced significantly if the block size (or equivalently, the cooperative caching unit) is increased to values larger than 4KB (a typical value for SCSI disks). But larger caching units aggravate the *false sharing* problem pointed out by the DSM experience.

HSCC_lookup If available, a valid *next* ID for the block is returned. In turn, the CARD driver directs the request to that node. Otherwise, the request is sent to the home.

HSCC_handle_eviction An evicted block is considered singleton if the *next* ID is void and the *previous* ID is the physical disk node ID. For a void *next* ID and a *previous* ID different than the physical disk node ID, the block is considered the most recent copy (it may even be a singleton if all the other copies have been evicted meanwhile). Only singletons and most recent copies are considered for saving. All the other copies are dropped.

HSCC_handle_eviction sends singletons to the least loaded node in the cluster, as perceived from the local perspective. Choosing such a node is based on a priority queue storing the numbers of blocks cached on behalf of each home participating in the cooperative cache. The home on behalf of which the local node caches the least number of blocks is considered to be the least loaded node. If the chosen node can host the block, it will send an index update with the new *next* ID to the block home. If not, the target node further forwards the block to its home. If there is no room at home either, the block is discarded.

Most recent copies are sent to their *previous* node. If this node has a copy, the block is discarded. Else, if there is enough space, it stores the block and sends an update to the home with its own ID as the new *next* ID for the block. If there is not enough room, it forwards the block further to its own *previous* node. In practice, this process can be quite long, so a *depth* count set in the evicted block message helps to restrict the forwarding to *depth* stages. If no copy is found in *depth* steps, the block is stored locally instead of a "non-CARD" block in order to avoid an eviction ripple effect. By default, *depth* is 2.

When the index update reaches the home, if there are no newer block requests, an invalidate index message is sent to the evicting node. This node discards its cache index entry and forwards the invalidation to the *previous* node. Each *previous* node does the same until the invalidation reaches the node caching the new most recent copy.

HSCC_keep_consistency The consistency algorithm is a flavor of write-through with invalidation. A written block is sent to the physical disk node and both the home and the *previous* node are invalidated. The home forwards the invalidation to the node caching the most recent copy. Each node receiving an invalidation message invalidates its own copy. If *previous* is not the physical disk node, the invalidation is forwarded to it. Otherwise, the message is dropped since the physical disk node already got the written block so the invalidation would be wrong. As noticed, there are no guarantees for concurrent writes. This would need a locking scheme in the upper kernel layers (at filesystem level).

HSCC_handle_request If a block request arrives at home, the request handler looks for a block copy in the local buffer cache. If found, the block is returned to the requester. Otherwise, the request is forwarded to another node that can satisfy it. This node is either that identified by the *next* ID from the corresponding index entry, if any, or the physical disk node, otherwise. Either way, the requester is registered in the home index as the *next* ID.

A forwarded block request is handled the following way. If the node holds a block copy, this one is delivered directly to its original requester in order to save network bandwidth and to avoid an extra and unnecessary hop. Otherwise, the request is forwarded directly to the physical disk node which in turn will deliver the block.

4 The software architecture of CARDs

Device identification and block addressing Both the physical disk and the CARD are uniquely identified in the cluster by a tuple *(major number, minor number, physical*

ID). The *major* and *minor* numbers form the conventional pair used to identify a block device in standalone kernels. This pair may differ from CARD to CARD, but all the CARDs of a disk share the same *physical ID*. This is the ID of the physical disk node and helps to distinguish among different CARDs with the same *(major,minor)* pair. Remotely cached blocks are first checked against their *physical ID* to identify the actual CARD on that machine. Then, the native addressing scheme of the local kernel is used.

Design and implementation issues If a block request misses in the buffer cache, the kernel allocates a new buffer for the block, locks the buffer and queues a request in the disk driver. The process making the request goes to sleep while the driver is responsible for serving the request. When the driver finishes the service, it dequeues the request, unlocks the buffer and wakes up any process that might wait for that block to become available. The main difference between a real disk driver and a CARD driver is that the latter serves the requests from remote caches. That requires certain design and implementation decisions that are described in what follows.

Handling incoming block requests may follow either a blocking model (using a kernel thread server) or an interrupt driven one, akin to systems like Active Messages [22], which serves the blocks using a software interrupt handler run in interruptible context. This handler is executed at the end of the hardware interrupt handler triggered by the SAN card interrupt. The thread solution offers a well-defined protection domain and the thread gets charged for the server computation. That avoids unfairness as pointed out the by the research experience with network subsystems [3, 4]. The disadvantage consists in a lower degree of responsiveness. An incoming request triggers a network card interrupt, the server thread is woken up and placed in the run queue. However, this doesn't ensure immediate response as only the scheduler decides who gets the processor next. The interrupt driven solution shows better responsiveness. Cached blocks are delivered at interrupt time as quick as possible. If the block is not cached, the software interrupt handler sets up a callback function associated with the buffer. This will deliver the block (also in interruptible context) when the disk interrupt handler will signal that the disk driver finished loading the block. The disadvantage is that the interrupted process is going to be unfairly charged for unasked computation. In our solution, cached blocks are serviced at interrupt time in order to improve responsiveness. The uncached blocks service is deferred to a kernel thread because the disk latency is dominant in this case and saving a few microseconds wouldn't help much.

Linux itself raises problems as well. It uses two cache systems to speed up disk access. Raw and special data (inodes, for instance) are accessed through the buffer cache. However, regular files read/write their data through the *page cache*. Unlike the buffer cache, which is indexed by *disk ID* and *block number*, the page cache is indexed by *inode* and *offset* within the file. It seems to have appeared as a response to an efficient *mmap* implementation because in its pages it stores disk blocks contiguous in the logical file layout (a byte stream for Unix files) but potentially uncontiguous on disk. A CARD asking another CARD to deliver an already cached block sends the disk ID tuple and the block number but the CARD storing the block cannot use them to search the page cache. To solve the problem, we unified the buffer and the page cache so that blocks in the page cache can be also indexed through the buffer cache hash list.

The separation of the various buffering systems in the kernel affects the CARD performance. The network interface uses specialized socket buffers to send/receive messages while the file system buffer cache uses its own buffering system (the buffer head cache). Therefore, fetching blocks from remote caches implies two extra copies. To avoid them, one needs a Remote DMA (RDMA) engine that allows direct read/write access to remote memories over the SAN. Alternatively, one could use a unified network and cache buffering system such as IO-Lite [18]. To date, our prototype does not incorporate any of these facilities.

CARDs are independent of any disk format and may access the physical disk through a raw interface, but using a non-distributed filesystem (Linux EXT2) on top of them may lead to severe inconsistencies. Any file system has specialized directory and inode (file meta-data) caches. Because CARDs look like ordinary disks to the system, such a file system will fail to take the appropriate measures for keeping these caches consistent across the cluster. We chose not to involve the CARDs in filesystem meta-data consistency handling for two reasons. First, CARDs are meant to be storage devices independent of the actual data format on the disk. Second, our plans are oriented towards integrating CARDs with Clusterfile [15], a parallel file system for clusters. Clusterfile will take care of meta-data consistency.

5 Performance evaluation

We evaluated the performance of our CARD prototype using HSCC and *Hash Distributed Caching* [8] (from hereon designated as HDC). CARDs of a disk formatted with a native Linux filesystem format (EXT2) were mounted remotely. Because of the potential inconsistencies that we have mentioned in Section 4, the mount was read-only. The filesystem was not aged and therefore mostly contiguous. All the tests consist of running the Unix *find* command on a CARD to scan a directory for a given string. The typical layout of the command was:

```
find <dir> -exec grep <str> {} \;
```

Experimental Setup We ran our experiments on a 3-node Linux cluster interconnected through a Myrinet switch

and LANai 7 cards. The Myrinet cards have a 133 MHz processor on board. They achieve 2 Gb/sec. in each direction. The host interface is a 64 bit/66 MHz PCI that can sustain a throughput of 500 MB/sec. The Myrinet cards were controlled by the GM 1.4 driver of Myricom [20]. The PCs were 350 MHz Pentium II machines with 256 MB of RAM. All the systems ran Linux 2.2.14. The disk used for tests was an IBM DCAS-34330W Fast/Ultra-SE SCSI disk. Only a 1.7 GB partition of it was remotely mounted for the experiments that we further describe.

We approximated the extent to which the buffer cache of Linux can grow by scanning a directory whose size was larger than the local memory. On the physical disk node, the buffer cache grew up to 240 MB. On a CARD node, the buffer cache grew up to 225 MB. So we can consider a value of roughly 690 MB of RAM for our cooperative cache. However, this figure is just an upper bound as the Linux memory management algorithm dynamically trades off application memory for kernel memory. This behavior makes it hard to precisely determine the buffer cache size which may vary significantly depending on the machine load.

	CARD	Local Disk
Throughput (MB/s)	2.14	2.93
Memory copies	80 μs	-
Service handling	15 μs	-
Network overhead	205 μs	-
Average cached read time	300 μs	30 μs
Average disk read time	12300 μs	12000 μs

Table 1. CARD vs. Local Disk Comparison
Throughput and 4k block read access times

5.1 CARD vs. Local Disk comparison

We evaluated the performance of a CARD acting as a simple remote disk interface. The outcome was compared to that of the corresponding physical disk. We mounted remotely the SCSI partition and measured the CARD throughput and the read access times (both cached and uncached). The throughput was computed by timing a Unix *find* command scanning all 1.7 GB of data stored on that partition. We repeated the experiment locally on the machine hosting the disk. The figures are presented in **Table 1**. Notice that accessing a remotely cached block is 40 times faster than getting it from the local disk. Also, the CARD achieves 73.03% of the local disk throughput.

5.2 CARD operation analysis

The cache-cooperative operation of the CARD driver was evaluated using six workloads whose sizes were roughly 310 MB, 420 MB, 490 MB, 571 MB, 681 MB and 741 MB respectively. All the workloads were combinations of subdirectories of a typical */usr* Unix directory, namely */usr/bin*, */usr/lib/*, */usr/share*, */usr/src*. The first four size choices aim at evaluating the behavior of the cooperative cache for workloads bigger than any cluster node memory (240 MB) but smaller than the size of the global cache (690 MB). The last two size choices intend to show the system performance at the limit of the global cache and beyond.

We warmed up the cooperative cache by running the *find* command at a CARD node and then we took measurements by running it at another CARD node. We compared the cache hit ratios of the two policies and evaluated the general benefits of saving the evicted blocks.

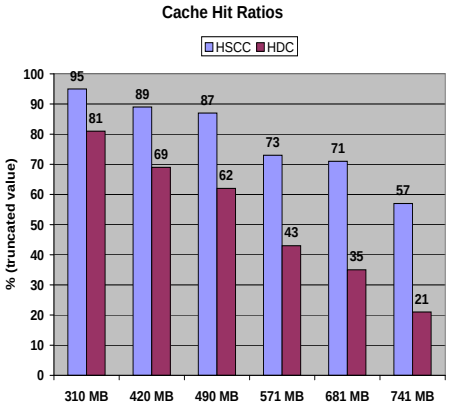

Figure 2. Cache hit ratio comparison

Cache hit ratios comparison A comparison between HSCC and HDC in terms of cache hit ratios is presented in **Figure 2**. Notice that HSCC performs better overall and that HDC's performance degrades faster with the increasing size of the workloads than that of HSCC. Indeed, for the first workload the cache hit ratio of HDC is roughly 85% of the HSCC figure, while close to the global cache limit HSCC achieves at least twice as many cache hits as HDC. HDC degradation becomes even more severe beyond this limit, as for the last workload HSCC's ratio is roughly 2.7 times that of HDC. **Figure 3** offers more insight on the CARD operation by showing cache access breakdowns.

Eviction statistics The number of evicted blocks stored by a CARD node running on the warm cache is reported in **Figure 4**. From this figure and **Figure 3** one can infer that handling too many evicted blocks is a waste. The reason lies in the way the two policies handle the cooperative cache. HDC evicts blocks only at the server cache (physical disk node) and does it irrespective of load by sending the evicted blocks to their homes. As **Figure 3** shows, this improves the local cache hits. Overall, HDC has better local hit ratios than HSCC. On the contrary, HSCC has better global hit ratios overall because it handles evictions at homes as well and saves blocks trying to even out the load.

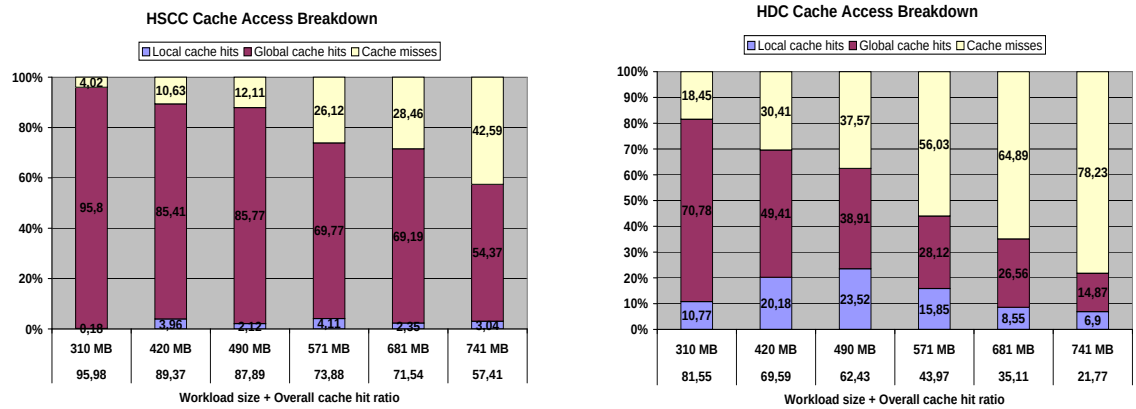

Figure 3. Cache Access Breakdowns Local cache hits, global cache hits and cache misses for HSCC and HDC

For heavy workloads (the last three for instance), local cache hits become less important when compared to global cache hits. In this case, HSCC outperforms HDC by far exactly because it maintains a higher global hit ratio. As it can be seen from **Figure 3**, saving too many evicted blocks (as in HDC's case) under memory pressure turns out to be ineffective as both the local and global cache hit ratios seem to diminish at the same pace. Thus, handling evictions must be made with care in order to balance the loads of the caches.

5.3 CARD Speedup/Slowdown

We ran the workloads on a CARD acting as a remote disk interface and measured the running time using the Unix *time* command. We also ran the workloads on CARDs with cooperative caching enabled, both on cold and warm caches. The results are presented in **Figure 5**.

For HSCC, the best speedup was that of the first workload. The CARD running on a warm cache achieved a speedup of 1.54 over the remote disk. Even the workload larger than the global cache experienced speedup, although smaller (see the last line of the x-axis of the **Figure 5**). The slowdown of a CARD running on a cold cooperative cache when compared to a remote disk is negligible (see the middle line of the x-axis of the **Figure 5**).

For HDC, practically only the first workload exhibited speedup since that of the second workload is negligible. All the other workloads experienced only slowdowns, both when running the CARDs on a cold cache and on a warm one. Moreover, the slowdowns of the CARD running on the cold cache are less severe than those of the CARD running on a warmed up cache. Notice however that for big loads (the last three), the HDC slowdowns of a CARD running on a cold cache are better than the corresponding ones of HSCC because HDC is a simpler policy than HSCC. Nevertheless, the warm cache figures show that aggressively trying to save evicted blocks is not only a waste but also induces running time penalties.

Some of the remote disk numbers look weird. The 490 MB load takes more time than the 571 MB one. Similar for the (681 MB, 741 MB) pair. This is also true when running the load on the physical disk. Therefore, the problem is not related to the CARD driver. Both the 490 MB and 681 MB loads include */usr/share* which is broader and deeper than the other workload directories (*/usr/bin*, */usr/lib*, */usr/src*). The running time breakdowns show indeed that */usr/share* needs more time per MB than the other directories.

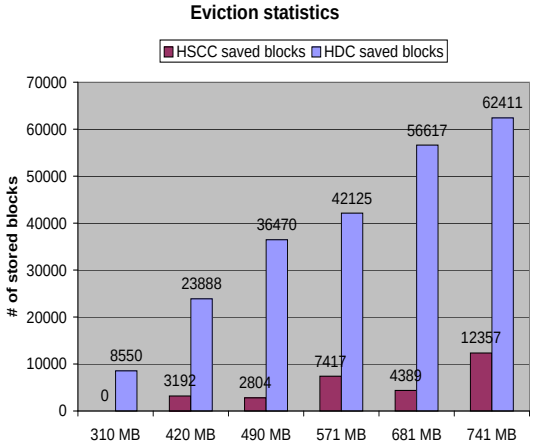

Figure 4. Eviction statistics The number of evicted blocks saved by the CARD for each policy

6 Related work

Remote I/O systems can be broadly classified in: user-level, filesystem or low-level systems. User-level systems are mostly a work-around: block devices [16] or filesystems (PVFS [14]) in user space, remote I/O libraries [10] over MPI-IO [6]. However, traditional kernels are unaware of the distributed nature of these systems and their inner mechanisms and policies fail to match the expectations of

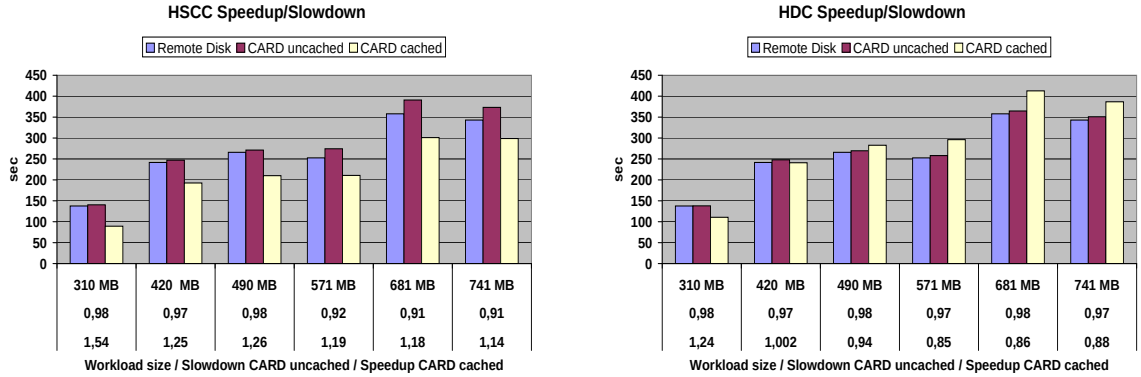

Figure 5. Speedup/Slowdown CARDs compared to remote disk interfaces

the user space driven computation. The end result is performance penalty. Also, by moving typical kernel code in user space, the application software complexity grows and development gets difficult.

Frangipani [21] is a distributed filesystem built on Petal [17] distributed virtual disks. Petal is a block-oriented storage system that manages a pool of distributed physical disks by providing an abstraction called *virtual disk*. It is globally accessible and offers a consistent view to all its clients. Petal delegates the meta-data consistency problem to Frangipani which implements a distributed lock service.

The xFS project [8, 2] introduced the notion of cooperative caching. HSCC resembles somewhat to *Hash Distributed Caching* as both are serverless and use a hash function on the block number. HSCC and *N-Chance Forwarding* eviction algorithms share the notion of *recirculation count*, called *depth* in HSCC, although used in different contexts: the *recirculation count* applies to singletons while the *depth* is used for the most recent block copy. *N-Chance Forwarding* and HSCC handle eviction differently. *N-Chance Forwarding* randomly chooses a node to forward an evicted singleton. Instead, HSCC takes an informed decision by sending the block to the node which appears to be the least loaded one from the local perspective.

PACA [7] is another cooperative file system cache using an algorithm akin to HSCC. 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.

Sarkar et al. [19] describe a cooperative caching algorithm using hints. Like HSCC, it avoids centralized control. Reasonably accurate hints help locating a block master copy in the cache without involving the server. If the hints are not accurate enough, a fall-back mechanism gets a block copy. The master copy simplifies eviction: only such copies are saved. Unlike HSCC, the algorithm interferes with the native kernel eviction policy by using a "best-guess" replacement strategy when locally storing remotely evicted blocks. Erroneous decisions are offset by adding an extra cache at the server node, the *discard cache*. Cache consistency is file-based and not block-oriented as in HSCC.

CARDs are somewhat similar to the SIOS [13] Virtual Device Drivers (VDD). A VDD amasses the entire disk capacity of a node (both local and remote disks) much like a RAID (actually implemented as default technology). Cooperative caching is mentioned only once without any details.

Other low level approaches to remote I/O include Swarm [12] and Network-Attached Secure Disks [11]. Swarm offers the storage abstraction of a *striped log* while NASDs provide an object-oriented interface.

7 Conclusions

In this paper we presented a flexible solution for a cluster-wide cooperative caching system using Cluster-Aware Remote Disks. A collection of CARD drivers can employ a common cooperative caching policy in order to globally manage the contents of the buffer caches. We have implemented CARDs as Linux drivers and two policies to prove the flexibility of the solution. Our results show that cooperative caching reduces I/O activity and improves read latency. Even for heavy workloads our algorithm (HSCC) could achieve cache hit ratios above 50% without any slowdown. The best speedup observed was 1.54.

8 Acknowledgments

We thank Florin Isaila and Guido Malpohl for their suggestions on early drafts of this paper and the anonymous reviewers for their valuable comments. We are grateful

to Jürgen Reuter for thoroughly proofreading this paper as well as for his invaluable LaTeX assistance.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. In *IEEE Computer, Vol. 29, No. 2, pp. 18-28*, February 1996.

[2] T. Anderson, M. Dahlin, J. M. Neefe, D. Patterson, D. Rosseli, and R. Y. Wang. Serverless Network File Systems. In *The 15th Symp. on Operating System Principles*, December 1995.

[3] G. Banga, P. Druschel, and J. Mogul. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems . In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, October 1996.

[4] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems . In *Proc. of the 3rd Symp. on Operating System Design and Implementation* , February 1999.

[5] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symp. on Operating System Principles (SOSP-15)*, December 1995.

[6] P. Corbett, D. Feitelson, Y. Hsu, J.-P. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A parallel file I/O interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, Moffet Field, CA, January 1995.

[7] T. Cortes, S. Girona, and L. Labarta. 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, Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, 1995.

[8] M. Dahlin, R. Yang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *The 1st Symp. on Operating Systems Design and Implementation*, November 1994.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[10] I. Foster, Jr. D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast Access to Distant Storage. In *Proc. of the 5th Annual Workshop on I/O in Parallel and Distributed Systems*, November 1997.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems 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] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm Scalable Storage System. In *Proc. of the 19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS '99)*, June 1999.

[13] R. S. C. Ho, K. Hwang, and H. Jin. Single I/O space for Scalable Cluster Computing. In *Proc. of the 1st IEEE Int. Workshop on Cluster Computing*, December 1999.

[14] W. B. Ligon III and R. B. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the Extreme Linux Workshop*, June 1999.

[15] F. Isaila and W. Tichy. Clusterfile: A flexible physical layout parallel file system. In *3rd IEEE Int. Conf. on Cluster Computing*, October 2001.

[16] K. Kim, J.-S. Kim, and S. Jung. A Network Block Device Over Virtual Interface Architecture on LINUX. In *Proc. of the IEEE Int. Parallel and Distributed Symposium*, April 2002.

[17] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. of the 7th Int. Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System . In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation*, February 1999.

[19] P. Sarkar and J. H. Hartman. Efficient Cooperative Caching using Hints. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, October 1996.

[20] Myricom Inc. GM: the low-level message-passing system for Myrinet networks. *http://www.myri.com/scs/index.html*.

[21] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. of the 16th ACM Symp. on Operating System Principles*, 1997.

[22] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int. Symp. on Computer Architecture*, May 1992.

[23] T. M. Warschko, J. M. Blum, and W. F. Tichy. On the Design and Semantics of User-Space Communication Subsystems . In *Proc. of the Int. Conf. on Parallel and Distributed Processing, Techniques and Applications (PDPTA'99)*, June 1999.

[24] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, October 1996.