Demotion-Based Exclusive Caching through Demote Buffering: Design and Evaluations over Different Networks

Jiesheng Wu Pete Wyckoff[†] Dhabaleswar K. Panda

Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
email: {wuj, panda}@cis.ohio-state.edu

[†] Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
email: {pw}@osc.edu

***Abstract*—Multi-level buffer cache architecture has been widely deployed in today's multiple-tier computing environments. However, caches in different levels are *inclusive*. To make better use of these caches and to achieve the expected performance commensurate to the aggregate cache size, *exclusive caching* has been proposed. Demotion-based exclusive caching [1] introduces a DEMOTE operation to transfer blocks discarded by a upper level cache to a lower level cache. In this paper, we propose a *DEMOTE buffering* mechanism over storage networks to reduce the visible costs of DEMOTE operations and provide more flexibility for optimizations. We evaluate the performance of DEMOTE buffering using simulations across both synthetic and real-life workloads on three different networks and protocol layers (TCP/IP on Fast Ethernet, IBNice on InfiniBand, and VAPI on InfiniBand). Our results show that DEMOTE buffering can effectively hide demotion costs. A maximum speedup of 1.4x over the original DEMOTE approach is achieved for some workloads. Speedups in the range of 1.08-1.15x are achieved for two real-life workloads. The vast performance gains results from overlapping demotions and other activities, reduced communication operations and high utilization of the network bandwidth.**

I. Introduction

Caching is designed to shorten access paths for frequently referenced items, and so improve the performance of the overall file and storage systems. With the increasing gap between processors and disks, and decreasing memory price, modern file and storage servers typically have large caches up to several or even tens of gigabytes to speed up I/O accesses [1]. In addition, the clients of these servers also devote a large amount of memory for caching [2]–[5]. Multiple clients may share file and storage resources through various storage networks. A typical scenario is a two-level hierarchy: the lower level cache can be a high-end disk array cache or a cluster file server cache, and the upper level can be a database server cache or a file client cache. We call a lower level cache a *server cache*. In contrast, we call an upper level cache as a *client cache*.

Most cache placement and replacement policies used in multi-level cache systems maintain the *inclusion property*: any block in an upper level buffer cache is also in a lower level cache. The drawbacks of inclusive caching have been observed in a rich set of literature [1], [6]–[8]. To aggregate the cache size of the multi-level cache hierarchy and to achieve *exclusive caching*, Wong and Wilkes [1] recently proposed a simple operation called *DEMOTE* as an additional interaction means between a client cache and a disk array cache to achieve *exclusive re-read cache*. The DEMOTE operation is used to transfer evicted data blocks from the client buffer cache to the disk array cache. Then the server cache uses different cache replacement policies for the demoted blocks and blocks recently read from disks to yield exclusive caching.

In our study, we study the overheads of DEMOTE operations and propose a *DEMOTE buffering* mechanism to reduce the impact of DEMOTE operations on the system performance. In the DEMOTE buffering mechanism, a small portion of buffer space is used to delay DEMOTE operations. When an evicted block needs to be sent to the server cache, the block is first placed in the DEMOTE buffer. Unlike eager demotions in the DEMOTE mechanism [1], DEMOTE buffering delays demotions and schedules them at an appropriate time in an efficient way. We discuss design issues in DEMOTE buffering and perform performance evaluations over different networks, including TCP/IP of Fast Ethernet, IBNice of InfiniBand [9], and VAPI of InfiniBand [9].

DEMOTE buffering has the potential to mask the DEMOTE overheads, to smooth the variance (burstiness) in the DEMOTE traffic, and to provide more flexibility for optimizations. In DEMOTE buffering, the design space for optimizations is broadened, including non-blocking network operations, remote direct memory access (RDMA), RDMA Gather/Scatter operations in networks such as InfiniBand [10], [11], and speculating demotions. Our performance evaluation through simulations across both synthetic and real-life workloads on different networks validates these potentials.

The rest of the paper is organized as follows. Background is presented in Section II. Section III describes the system architecture and various design issues in details. Section IV presents and analyzes the simulation results. We examine some related work in Section V and draw our conclusions and discuss possible future work in Section VI.

II. Background

We give a brief overview of demotion-based exclusive caching and analyze its performance overheads in this section. Some background information of InfiniBand is also presented.

A. Demotion-Based Exclusive Caching

Wong and Wilkes [1] proposed a simple operation called *DEMOTE* as an additional interaction means between a client cache and a disk array cache to achieve *exclusive re-read cache*. The DEMOTE operation is used to transfer evicted data blocks from the client cache to the disk array cache. To achieve exclusive caching, the server cache should choose appropriate cache placement and replacement policies to manage blocks demoted from the client cache and blocks that have been read from the disk. One of exclusive cache schemes discussed is shown in Figure 1. The client cache uses the LRU policy to read blocks from the server cache. The server cache puts blocks it has sent to a client at the head (earliest to be discarded) of its LRU queue, while puts demoted blocks from the client cache at the tail. This cache management policy most closely achieves a single unified LRU cache [1]. Ideally, exclusive caching has the potential to double the effective cache size with client and server caches of equal size.

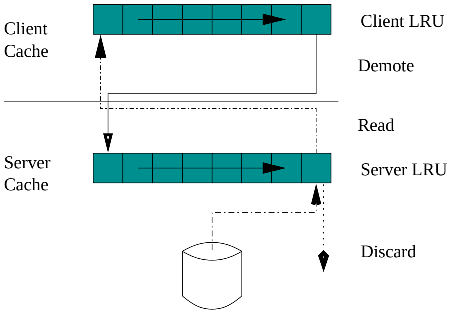

Fig. 1. Cache management in the DEMOTE exclusive caching.

When a client is to discard a clean block from its cache to make space for other blocks, it first sends the block metadata in advance of the block data [12]. This control message can be used to avoid transferring the block data in some cases. For example, from the metadata, the server can determine whether or not it has cached the block, and if it has it can signal to the client to abort the transfer.

Demotion-based exclusive caching relies on transferring demoted blocks through network between the clients and the server. Further, the DEMOTE approach performs eager DEMOTE operations with assumption that the network is fast. This method offers design simplicity. However, it introduces the following performance overheads, which may offset the benefits of exclusive caching for some workloads and networks.

- Eager DEMOTE operations increase request access time. A request may be delayed because it needs to wait for the completion of a DEMOTE operation to make space for it. That is, the average client cache miss penalty will be increased due to the cost of DEMOTE operations.
- DEMOTE operations increase the network traffic. In the worst case, each read incurs a DEMOTE operation, the network traffic is more than doubled (some control traffic is also counted).
- Eager DEMOTE operations provide little design space for optimizations. A DEMOTE operation must be finished before a demand request can have space in the client cache. Therefore, features such as non-blocking network operations can not be applied. Besides, each block needs a control message. In addition, the size of cache blocks is usually small (e.g. 4 kB or 8kB), one cache block per DEMOTE operation may not utilize the network bandwidth efficiently.

We propose DEMOTE buffering to hide and/or reduce these overheads. DEMOTE buffering provides more flexibility for optimizations. These optimizations can make better use of network features, such as non-blocking network operations and gather/scatter operations. A control message in DEMOTE buffering can contain multiple blocks' metadata. This can reduce control traffic and amortize control cost over multiple blocks.

B. Overview of InfiniBand

The InfiniBand Architecture [10] defines a System Area Network for interconnecting both processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. The InfiniBand specification defines three basic components: a Host Channel Adapter (HCA), a Target Channel Adapters (TCA), and a fabric switch. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Host Channel Adapters (HCAs) and Target Channel Adapters

(TCAs), respectively. These components are intelligent hardware devices. These hardware devices provide highly reliable, fault-tolerant communication to enable improved bandwidth, latency, and reliability of the system.

The HCA resides in the processor node and provides the path from the system memory to the InfiniBand network. It has a programmable direct-memory-access (DMA) engine with special protection and address-translation features that allow DMA operations to be initiated locally or remotely by another HCA or a TCA. An abstraction interface for HCAs is specified in the form of InfiniBand Verbs.

The TCA resides in the I/O unit and provides the connection between an I/O device (such as a disk drive) or I/O network (such as Ethernet or Fibre Channel) to the InfiniBand network. It implements the physical, link, and transport layers of the InfiniBand protocol. The interface of the TCA is vendor implementation specific.

VAPI is a user-level software interface for InfiniHost HCAs from Mellanox [13]. The interface is based on InfiniBand verbs layer. It supports both send/receive operations and remote direct memory access (RDMA) operations. Gather/Scatter are also supported in RDMA operations. RDMA write operation can gather multiple data segments together and write all data into a contiguous buffer on the peer side in one single operation. RDMA read operation can read data from a contiguous buffer on the peer side into several local buffers.

IBNice [13] is a kernel-level TCP/IP implementation over InfiniHost HCAs. It provides full compatibility of legacy TCP/IP protocol to applications.

Using IBNice can reduce development efforts, however, applications can not fully realize the hardware capability. In contrast, applications programmed with VAPI can take full advantage of InfiniBand user-level networking and RDMA features and potentially achieve the highest performance.

III. DEMOTE Buffering

In this section, we first describe the structure of DEMOTE buffering and its potential benefits. Then we discuss its design issues.

A. The Architecture of DEMOTE Buffering

In DEMOTE buffering, a small memory space, the *DEMOTE buffer*, is used on the client side for buffering demoted blocks, as shown in Figure 2. DEMOTE buffering works as follows: when a client encounters a cache miss and is about to demote a clean block from its cache (e.g. to make space for a READ), it first moves the demoted block into the DEMOTE buffer. This operation is a local operation. Then it initiates a request to the server cache to read the demanded block. Unless the number of demoted blocks in the DEMOTE buffer is up to a certain threshold, requests will not encounter any overhead of DEMOTE operations.

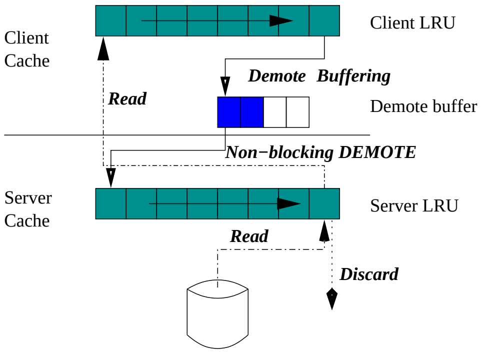

Fig. 2. Architecture of DEMOTE Buffering.

Double-buffering technique is used in our DEMOTE buffering mechanism. The threshold could be half of the DEMOTE buffer size in blocks. Thus, when the DEMOTE buffer is half full, non-blocking DEMOTE operations are initiated.

When client cache misses are too bursty, it is possible that there is no space left in the DEMOTE buffer when a client needs to demote a block. The client cache then checks the completion of previously initiated demotions and reclaims resources for future demoted blocks.

B. Benefits of DEMOTE Buffering

DEMOTE buffering has the following potential benefits.

1) **Reduced visible DEMOTE cost.** DEMOTE buffering tries to hide the cost of DEMOTE operations by increasing the overlap between demotions and other activities. This is achieved through buffering demoted blocks in the DEMOTE buffer and using non-blocking I/O to perform DEMOTE operations.
2) **Exploiting idle network bandwidth.** DEMOTE buffering provides opportunities to use idle network bandwidth to transfer demoted blocks in the DEMOTE buffer. The network link between a client and a server is free when the client cache hits occur or the client is not performing I/O operations. If DEMOTE operations occur during this period, the impact of the increased traffic on the system performance is minimized. This benefit can be realized easily with one-sided communication such as RDMA operations in the InfiniBand network. For example, the server can monitor the network usage and then initiate RDMA Read operations to retrieve demoted blocks from a client's DEMOTE buffer when the link is free.
3) **Better network utilization.** In the DEMOTE buffering mechanism, multiple control messages can be aggregated into one single message for all buffered blocks. The control messages can be piggybacked with request

and reply messages. Effective scheduling or batching on the transmission of multiple blocks in the DEMOTE buffer can be performed. For example, RDMA Gather/Scatter operations in InfiniBand can be used to retrieve multiple blocks in one operation, even though they are not contiguous. This can achieve better network utilization.

4) **More flexibility for optimizations.**
Demote buffering enables more flexibility for optimizations. One example is speculating demotion. When the client cache misses are too bursty for the DEMOTE buffering to hide the DEMOTE costs, the client can only send metadata information of the demoted blocks to the server. The server then speculates about cold blocks [8] which are accessed infrequently and thus are unnecessary to demote from the clients. After the client receives the server's speculation information, it can replace those unnecessarily demoted blocks with the newly demoted blocks directly. Speculating demotion can be applied without DEMOTE buffering, however, the control message cost is significant. In Demote buffering, the control message cost can be amortized effectively across multiple blocks.

In summary, Demote buffering has the potential to hide the demotion cost through overlapping communication and other activities, to reduce the number of communication operations, to achieve better utilization of the network bandwidth, and to allow more flexible optimizations to reduce the impact of demotion cost on the system performance.

C. Design Issues in DEMOTE Buffering

The DEMOTE buffering mechanism shows very attractive potential benefits over the eager DEMOTE mechanism, however, several issues need to be addressed for this mechanism to be used in real systems to achieve high performance.

- **Reducing DEMOTE buffering overhead.** There are two ways to buffer a demoted block. One is to copy the demoted blocks into the DEMOTE buffer. The method can have demoted blocks in a contiguous memory space which can be used to optimize communication in some networks. There is no change to the client cache space. These advantages are achieved at the cost of memory copy. The second one is to exchange the positions of a free block in the DEMOTE buffer and a demoted block in the client cache. There is no memory copy. However, the client cache space and the DEMOTE buffer space change with time. Noncontiguous data transmission may occur [14]. Tradeoff must be made between the cost of memory copy and the performance of noncontiguous data transmission in the studied network.
- **Tuning the size of the DEMOTE buffer.** The size of the DEMOTE buffer affects the ability of the DEMOTE buffering mechanism to mask the demotion overhead. From the server cache point of view, a DEMOTE operation is similar to a WRITE operation, and the DEMOTE buffer is similar to a write-behind buffer. Ideally speaking, the DEMOTE buffer size must be large enough to cover the burstiness of the client cache misses. This is similar to a write-behind buffer to cover the variance in the write workload [15], [16]. On the other hand, the DEMOTE buffer should not consume too much memory since most memory should be devoted to the client cache.
- **Maintaining cache hits.** DEMOTE buffering introduces a new complication: the delayed demotions may result in cache misses in the client and server caches. To address this issue, first the DEMOTE buffer should be considered as a part of the client cache. When a client cache miss occurs, the client should look at the DEMOTE buffer to see if the requested block is stored. Thus, from maintaining the client cache hit point of view, the DEMOTE buffer works as a victim buffer in victim caching [17]. In a single-client system, this method solves the issue completely. However, it is a little bit more complicated in a multi-client system.
- **Handling remote client cache hits.** In a multi-client system, it is possible that the demoted blocks in one client's DEMOTE buffer may be expected by others. That is, the delayed demotions might decrease the server cache hits for some workloads. One solution to this problem is to let a server cache maintain a directory of which blocks are in which client's DEMOTE buffer. Since the number of blocks in a DEMOTE buffer is small, the total space for this directory is limited. Then when a request from a client encounters a server cache miss, the server can potentially retrieve the block from a different client which is holding it in its demote buffer. Data transfer between clients is also possible, as with Cooperative Caching [5], [18]–[20].

IV. Performance Evaluation

We developed a simulator to simulate the DEMOTE buffering between the clients and the server over various networks. For comparison, the original DEMOTE mechanism is also simulated. Our simulator is built over *fscachesim* [1] by adding communication details. The simulator takes synthetic workloads and traces as input.

A. Experimental setup

Our experimental testbed consists of a cluster system consisting of 8 nodes built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node has two Intel Xeon 2.4 GHz processors with a 512 kB L2 cache and a 400 MHz front side bus. The machines are connected with Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The Mellanox InfiniHost HCA SDK version is thca-x86-0.1.2-build-001. The adapter firmware version is fw-23108-rel-1_17_0000-rc12-build-001. Each node has a Fujitsu Ultra3 SCSI (Model MAM3184MC) disk, which is a 18.4GB and 15,000 rpm drive. We used the Linux RedHat 7.2 operating system.

We perform tests over three different networks: TCP/IP over Fast Ethernet (referred as *FE*), IBNice (TCP/IP over InfiniBand), and native InfiniBand (using the VAPI library). We intend to use these three networks to represent roughly 0.1 Gb/s, 1 Gb/s, and 10 Gb/s networks. Table I lists their minimum latency, maximum bandwidth, and effective bandwidth when the message size is 4 kB bytes.

TABLE I

Network Performance

	FE	IBNice	VAPI
Latency (μs)	68	36	5.4
eff. Bandwidth (MiB/s)	11	109	718
max. Bandwidth (MiB/s)	11.2	130	831

B. Single-client synthetic workloads

Three synthetic workloads: Random, Sequential, and Zipf [21], are generated using the *fscachesim* package [1]. The size of cache blocks is assumed 4 kB, the client and server caches each have 16384 blocks.

We follow the following method as mentioned in [1] to compare DEMOTE buffering and DEMOTE mechanisms. We perform simulations over different networks with the above synthetic workloads. The size of the working set is 32768 blocks, 32767 blocks, and 49152 blocks for the Random, Sequential and Zipf workloads, respectively. In each test, the caches are "warmed up" with a working-set size set of READs. After the warm-up, another 10 timed READs are initiated. Time to randomly access a 4 kB disk block is set to 10 ms, the same value set in [1]. The DEMOTE buffer size is set to 10 cache blocks. The client cache hit ratios are 50%, 0%, and 86% for Random, Seqential, and Zipf workloads, respectively; with server cache hit ratios of 46%, 100%, and 9%. Note that these ratios are expressed as fractions of the total client READs. Since the cache hit results are important for us to understand the performance of DEMOTE buffering, we put these results in Table II for clearer reference.

The main metric for evaluating DEMOTE buffering is the mean latency of a READ at the client. DEMOTE buffering achieves same cache hits and server hits as DEMOTE does. The results in Table III show the mean latency of READs in each workload with DEMOTE (*DE* for short) and DEMOTE Buffering (*DB* for short) mechanisms.

TABLE II

Client and server cache hit rates for single-client synthetic workloads.

Workload	client	server
Seq	0%	100%
Random	50%	46%
Zipf	86%	9%

It can be observed that DEMOTE buffering achieves the highest speedup (1.44x) for the Sequential workload. This is because there is no cache hit on the client, and all accesses are cached in the server. Each access results in sending a demoted block to the server and receiving a block from the server.

In Random workload, the number of demote operations is 50% of the size of the working set due to 50% client cache hit ratio. There are 4% blocks needed to be read from disk. DEMOTE buffering achieves considerable improvement on Fast Ethernet and IBNice. However, the benefit diminishes on VAPI because the total demotion overheads are less significant.

DEMOTE buffering achieves the least improvement in the Zipf workload. This is actually expected, because of the highest client cache hit ratio and the lowest server cache hit ratio. The client cache hit ratio is 86%, indicating that there are only 14% blocks needed to be demoted. The server cache hit ratio is 9%, indicating that there are 5% blocks needed to be read from disk. Since the disk access time is much higher than the network access time, the total demotion overheads become less significant in the Zipf workload.

TABLE III

Mean READ latencies and speedups over DEMOTE for single-client synthetic workloads (DE: DEMOTE; DB: Demote Buffering)

		Seq (ms)	Random (ms)	Zipf (ms)
FE	DE	1.21	0.99	0.84
	DB	1.09 (1.11x)	0.92 (1.08x)	0.83 (1.01x)
IBNice	DE	0.26	0.52	0.73
	DB	0.18 (1.44x)	0.46 (1.13x)	0.71 (1.03x)
VAPI	DE	0.078	0.41	0.704
	DB	0.056 (1.4x)	0.40 (1.03x)	0.70 (1.01x)

We note that performance gain achieved by using DEMOTE buffering varies with the network performance, disk performance, as well as workload access patterns. It depends how significant the total demotion overheads are. It can be expected that workloads with high client cache miss rates can benefit more from DEMOTE buffering. In addition, the faster disks are, the more significance DEMOTE buffering is. Furthermore, the performance gain is closely related to the performance gap between network and disk. In Table III, we see DEMOTE buffering achieves less improvement on VAPI than on IBNice, this is because VAPI performs 6.5 times better than IBNice and tens of times better than disk, the total demotion overheads over VAPI are less significant than those over IBNice.

C. Effectiveness of DEMOTE buffering

To show the effectiveness of DEMOTE buffering directly, we profiled the total demotion overheads visible to the client in our tests. Results are shown in Table IV. With a small DEMOTE buffer (10 blocks), up to 34% demotion overheads are reduced by DEMOTE buffering.

TABLE IV

TOTAL DEMOTION OVERHEADS FOR SINGLE-CLIENT SYNTHETIC WORKLOADS

		Seq (s)	Random (s)	Zipf (s)
FE	DE	12.76	10.65	1.91
	DB	8.30	7.44	1.52
IBNice	DE	9.45	4.70	1.70
	DB	6.39	3.23	1.14
VAPI	DE	0.95	1.81	0.33
	DB	0.70	1.30	0.26

Note that in the above tests, one request is initiated immediately after the completion of the previous one. This incurs the highest burstiness of DEMOTE operations in all workloads studied. It is possible that client cache misses are too bursty to mask the DEMOTE overhead. That is, some of requests should wait for the completion of non-blocking demotions for spaces. Thus, the results in Table IV are actually the worst-case results for DEMOTE buffering. In the next subsection, we show that DEMOTE buffering can achieve better overlap if we reduce the access rate.

D. Effects of Burstiness

The sequential workload results in the most bursty demote operations since each READ incurs a DEMOTE operation. To study how well DEMOTE buffering can mask the DEMOTE overheads under different client cache miss burstiness, we put some computation delay after each client read request. We expect that the larger the delay is, the better DEMOTE buffering can overlap demotions with the computation delay. Consequently, the visible DEMOTE overheads to the client decrease. In contrast, each read in the eager DEMOTE approach must pay the demotion cost no matter what the delay is.

The results in Figure 3 validate our expectations. The overhead visible to the client is expressed as the fraction of the total overhead visible to the client in the eager DEMOTE, shown in y-axis. The delays between two consecutive requests are shown in x-axis. First, DEMOTE buffering effectively reduces the demotion overheads visible to the client even with small delays. Second, the delay at which the best overlap is achieved is adversely proportional to the network bandwidth. For example, when the delay is 500 μs, DEMOTE buffering on Fast Ethernet can offer the best overlap, while similar benefits arises with 100 μs for IBNice, and 50 μs for VAPI. Third, another interesting observation is that still 36% and 32% overheads are visible to the client on Fast Ethernet and IBNice even with large delays, while only 12% overhead visible to the client on VAPI. This is due to different CPU overheads needed to transfer data in the three networks studied. Both fast Ethernet and IBNice use the kernel-based TCP/IP stack which incurs substantial overheads due to context switching and memory copies. In contrast, VAPI provides user-level networking which requires much less interference from the host CPUs and operating system.

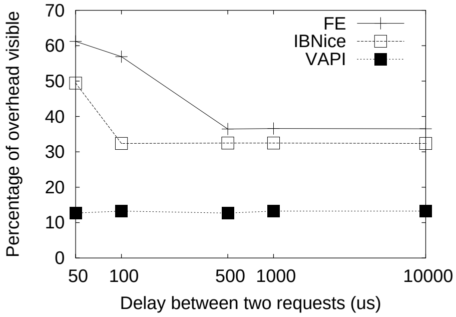

Fig. 3. Demotion overhead visible to the client with different request rates.

E. The Single-client DB2 workload

We used the DB2 workload [22] to evaluate the benefits of DEMOTE buffering for real-life workloads. The DB2 traces were generated by an eight-node IBM SP2 system. The size of data set is 5.2 GB. The eight client nodes access disjoint sections of the database. For single-client test, the eight access streams are combined into one. Unlike the above-mentioned tests, in this and the next tests, the disk accesses are not simulated. We use a Fujitsu Ultra3 SCSI (Model MAM3184MC) disk, which is a 18.4GB and 15,000 rpm drive. DB2 exhibits a behavior between the sequential and random workload styles [1]. The results of mean latencies achieved with different DEMOTE buffers are shown in Figure 4. Data points with zero block are results of DEMOTE. Overall, a 1.10 to 1.15x speedup over DEMOTE is achieved for Fast Ethernet and IBNice on InfiniBand, a 1.05x speedup for VAPI on InfiniBand. The mean latencies on IBNice and VAPI have little sensitivity to the size of DEMOTE buffer.

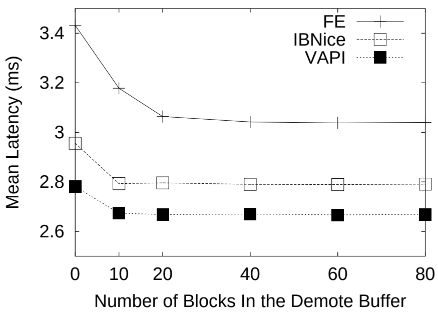

Fig. 4. Mean latencies of the DB2 workload

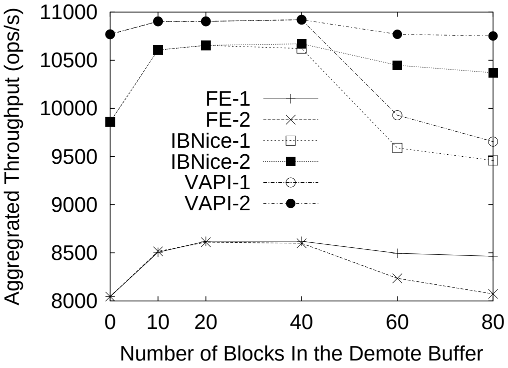

Fig. 5. HTTPD workload aggregate throughput

F. The Multi-client HTTPD workload

The HTTPD workload [22] was collected on a seven-node IBM SP2 parallel web server serving a 524 MB data set. There is a significant portion of blocks shared by clients. In this study, we evaluate the impact of DEMOTE buffering on the server cache hit ratio. For comparison, we implemented two methods to process server cache misses. One is to reload the missed blocks from disks, referred as *Reload*. The second one tries to retrieve the missed blocks first from the clients' DEMOTE buffers, referred as *Retrieve*. To keep the server updated of which blocks are in clients' DEMOTE buffers, the client eagerly piggybacks metadata information of the demoted block to the server cache in the request message.

In our test, each client has an 8 MB cache. The server cache is 64 MB. This configuration results in 62% client cache hits on average, 12% server cache hits, and 16% server cache misses with the DEMOTE approach. In DEMOTE buffering, when the DEMOTE buffer is less than 40 blocks, the server cache misses remain same. When the DEMOTE buffer size is set to 80, the average client cache hit ratio is 63%, 9% server cache hit ratio, and 18% server cache miss ratio. Figure 5 shows the aggregated throughput (the total number of HTTPD requests finished per second) of seven clients with different DEMOTE buffer sizes. We use "1" to represent the Reload method, and "2" for the Retrieve method. In Fast Ethernet (FE), Reload performs better than Retrieve because of the poor network performance. In both IBNice and VAPI, the network access is much faster than the disk access; hence, retrieve performs better than Reload. Although there is some increase of the server cache misses when the DEMOTE buffer becomes large, the DEMOTE buffering with Retrieve scheme still provides better performance than the eager DEMOTE approach whose results are shown by data points with zero block in the figure. A speedup up to 1.08x can be achieved. For example, the eager DEMOTE can support 9859 ops/sec on IBNice, while DEMOTE buffering can support 10670 ops/sec, a factor of 1.08 improvement.

V. Related Work

In a multi-level cache hierarchy, there exist two types of interactions between cache entities: horizontal interaction and vertical interaction. Cooperative Caching [5], [18]–[20] can be considered to use horizontal interaction to aggregate caches in a same level. Our work focuses on the vertical interaction. Vertical interaction has been realized in different ways. Client-controlled caching policy [23], DEMOTE Exclusive caching [1], and eviction-based cache replacement [8] are typical examples to perform various vertical interactions to improve system performance. Perhaps the closest work to ours in spirit is eviction-based cache replacement. Both work are intended to avoid/reduce the costs of DEMOTE operations. However, different approaches have been taken. In eviction-based placement, the server cache tries to reload blocks from disks when they are evicted from the client cache. Our approach is still to demote blocks, however, we use DEMOTE buffering to mask the overheads of demotions and to provide flexibility for optimizations, including the tradeoff between reload and demotion.

Demote buffering also shares similarity with victim caching [17] proposed by Jouppi in the sense that a victim buffer is used to place data evicted from cache. However, we have a different goal in using a DEMOTE buffer. Victim caching was proposed as an approach to improve the miss rate of direct-mapped caches without affecting their access time. A victim buffer is essentially used as a cache between a processor cache and its refill path. Demote buffering was designed to achieve efficient demotion interaction between two caches in different levels and reduce/mask cost of demote operations. The benefits of DEMOTE buffering are mostly from the overlap of demotion communication and other activities, instead of the increase of cache hits as in victim caching.

VI. Conclusions and Future Work

DEMOTE buffering is proposed to hide the cost of DEMOTE operations by increasing the overlap between demotions and other activities in demotion-based exclusive caching. It also provides more flexibility for optimizations, such as non-blocking operations, aggregate of control messages, gather/scatter network operations, and speculating demotions. Results of experiments with synthetic workloads demonstrate that 1.11-1.44x speedups are achieved for the Sequential workload, up to 1.13x speedups for the Random workload. Simulation results with real-life workloads validate the benefits of DEMOTE buffering by 1.08-1.15x speedups over the DEMOTE approach.

We are planning to design and implement demotion-based exclusive caching with DEMOTE buffering in a cluster file system, named PVFS [14], [24], [25]. Another direction is to study how to choose appropriate cache management policies and make them work together in different cache levels.

ACKNOWLEDGMENT

The authors would like to thank Theodore M. Wong for his simulator and for his answers to our questions on the DEMOTE exclusive caching. We also want to thank anonymous reviewers for providing helpful comments on this paper. This work was supported in part by Sandia National Laboratory's contract #30505, Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052, #CCR-0204429, and #CCR-0311542. .

REFERENCES

[1] T. M. Wong and J. Wilkes, "My cache or yours? making storage more exclusive," in *Proceedings of the 2002 USENIX Annual Technical Conference*. Monterey, CA: USENIX, June 2002, pp. 161–175. [Online]. Available: http://ssrc.cse.ucsc.edu/PaperArchive/wong-usenix02.pdf

[2] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li, "Experiences with VI communication for database storage," in *Proceedings of the 29th annual international symposium on Computer architecture*. IEEE Computer Society, 2002, pp. 257–268.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Y. Wang, "Serverless network file systems," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 41–79, Feb. 1996. [Online]. Available: http://ssrc.cse.ucsc.edu/PaperArchive/anderson-tocs96.pdf

[4] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Oct. 2001, pp. 174–187. [Online]. Available: http://ssrc.cse.ucsc.edu/PaperArchive/muthitacharoen-sosp01.pdf

[5] S. Narsimhan, S. Sohoni, and Y. Hu, "A Log-Based Write-Back Mechanism for Cooperative Caching," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[6] D. Muntz and P. Honeyman, "Multi-level caching in distributed file systems," University of Michigan Center for IT Integration (CITI), Technical Report 91-3, Aug. 1991. [Online]. Available: http://ssrc.cse.ucsc.edu/PaperArchive/muntz-multilevel91.pdf

[7] Y. Zhou, J. F. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the 2001 USENIX Annual Technical Conference*. Boston, Massachusetss: Usenix, June 2001, pp. 91–104.

[8] Z. Chen, Y. Zhou, and K. Li, "Eviction based cache placement for storage caches," in *USENIX Annual Technical Conference*. Usenix, June 2003.

[9] "Mellanox Technologies," http://www.mellanox.com.

[10] InfiniBand Trade Association, "InfiniBand Architecture Specification, Release 1.0," October 24, 2000.

[11] T. M. Pinkston, A. F. Benner, M. Krause, I. M. Robinson, and T. Sterling, "InfiniBand: The "De Facto" Future Standard for System and Local Area Networks or Just a Scalable Replacement for PCI Buses?" *Cluster Computing 6, pp. 95-103, 2003.*

[12] T. M. Wong, "Personal Communication," May 2003.

[13] Mellanox Technologies, "Mellanox InfiniBand InfiniHost Adapters," July 2002. [Online]. Available: http://www.mellanox.com

[14] J. Wu, P. Wyckoff, and D. K. Panda, "Supporting Efficient Noncontiguous Access in PVFS over InfiniBand," in *Proceedings of the IEEE International Conference on Cluster Computing*, Dec. 2003.

[15] K. Treiber and J. Menon, "Simulation study of cached RAID5 designs," in *Proceedings of the First Conference on High-Performance Computer Architecture*. IEEE Computer Society Press, January 1995, pp. 186–197.

[16] C. Ruemmler and J. Wilkes, "Unix disk access patterns," in *Proceedings of the Winter 1993 USENIX Technical Conference*, San Diego, CA, Jan. 1993, pp. 405–420. [Online]. Available: http://ssrc.cse.ucsc.edu/PaperArchive/ruemmler-usenix93w.pdf

[17] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 364–373.

[18] M. Dahlin, R. Wang, T. Anderson, and D. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 1994, pp. 267–280. [Online]. Available: http://ssrc.cse.ucsc.edu/PaperArchive/dahlin-osdi94.pdf

[19] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy, "Implementing cooperative prefetching and caching in a globally-managed memory system," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*. ACM Press, 1998, pp. 33–43. [Online]. Available: citeseer.nj.nec.com/voelker98implementing.html

[20] F. M. Cuenca-Acuna and T. D. Nguyen, "Cooperative caching middleware for cluster-based servers," in *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 2001.

[21] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM (1)*, 1999, pp. 126–134. [Online]. Available: citeseer.nj.nec.com/breslau98web.html

[22] A. A. Mustafa Uysal and J. Saltz, "Requirements of I/O Systems for Parallel Machines: An Application-driven Study. Technical Report, CS-TR-3802, University of Maryland, College Park," May 1997.

[23] P. Cao, E. W. Felten, and K. Li, "Implementation and performance of application-controlled file c aching," in *Proc. First USENIX Symposium on Operating Systems Design an d Implementation*, Nov. 1994, pp. 165–178.

[24] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*. Atlanta, GA: USENIX Association, 2000, pp. 317–327. [Online]. Available: citeseer.nj.nec.com/article/carns00pvfs.html

[25] J. Wu, P. Wyckoff, and D. K. Panda, "PVFS over InfiniBand: Design and Performance Evaluation," in *the 2003 International Conference on Parallel Processing (ICPP 03)*, Oct. 2003.