Toward Effective NIC Caching: A Hierarchical Data Cache Architecture for iSCSI Storage Servers *

Xiaoyu Yao and Jun Wang

Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, NE 68588-0115
{xyao,wang}@cse.unl.edu

Abstract

In this paper, we present a hierarchical Data Cache Architecture called DCA to effectively slash local interconnect traffic and thus boost the storage server performance. DCA is composed of a read cache in NIC card called NIC cache and a read/write unify cache in host memory called Helper cache. NIC cache services most portion of read requests without fetching data via PCI bus, while Helper cache 1) supplies some portions of read requests given partial NIC cache hits; 2) directs cache placement for NIC cache and 3) absorbs most transient writes locally. We developed a novel State Locality Aware cache Placement algorithm called SLAP to improve NIC cache hit ratio for mixed read and write workloads. To demonstrate the effectiveness of DCA, we developed a DCA prototype system and evaluated it with open source iSCSI implementation under representative storage server workloads. Experimental results showed that DCA can boost iSCSI storage server throughput by up to 121% and slash the PCI traffic by up to 74% compared with an iSCSI target without DCA.

1. Introduction

The iSCSI [8] technique has been gradually pushing current networked storage systems evolving from Storage Area Network (SAN) [10] and Network Attached System (NAS) [6] to economic SAN built over the mature TCP/IP network infrastructure. As playing a core role in the economic SAN, the iSCSI-based IP storage servers determine the overall performance of storage system.

In an iSCSI storage server, both volume storage traffic and network traffic travel across the local interconnect bus and easily overwhelm the bus. Today, many storage servers typically employ a 32 bits, 33 MHz PCI bus, which can

only support 133 MB/sec of maximum raw bandwidth. This speed is only half of the bandwidth of a full-duplex Gigabit Ethernet. In recent years, new local interconnects such as PCI-X and PCI-Express have been proposed to mitigate the PCI bus bottleneck, but none of them can solve all the problems. First, the aggregate I/O traffic resulting from link aggregation with the usage of multiple high-speed I/O devices can easily saturate the local interconnect bandwidth. Second, the improvement pace of networking speed and storage bandwidth have exceeded that of local interconnect bus bandwidth. Recently, Feng *et al.* found that the peak bandwidth of a 133 MHz, 64-bit PCI-X bus in a PC is 8.5 Gb/s, which is less than half the 20.6 Gb/s bidirectional data rate that the Intel 10 GbE adapter can support [5]. As a result, the traffic burden on local interconnect will be even worse. Based on the above observations, solving the local interconnect bottleneck becomes a pressing problem today.

Orthogonal to the aggressive PCI bus innovations, few work have been done on reducing local interconnect traffic through an effective caching approach. Kim *et al.* in reference [9] first proposed adopting a small read cache in programmable Ethernet NIC to reduce PCI traffic in a web server. The results showed that a single NIC cache can work efficiently in a read dominant web server with a small working set and good locality. However, the mixed read and write traffic in a storage server is much more intensive. More importantly, the locality of the block-level I/O accesses in a storage server is typically poor as most locality has already been filtered by multiple higher level buffer caches. Therefore, a simple NIC cache for web server cannot work well in storage environment.

Zhang and Yang developed a novel bottom-up cache structure called BUCS in a networked HBA card [12]. They used synthetic read-exclusive, write-exclusive traces and a TPC-C (99% read ratio) trace in the experiments. Their experimental results showed that the response time and system throughput can be improved by up to a factor of three. However, BUCS requires a hardware IOP board to process both storage and network I/Os. The internal PCI bus and

*This research is sponsored in part by NSF under grant #CCF-0429995 and University of Nebraska-Lincoln Layman Fund.

In this paper, we present a hierarchical *Data Cache Architecture* called *DCA* to boost the iSCSI Storage server performance through effective NIC caching. It is notable that a large body of previous research in storage servers focus on hierarchical cache collaboration [11, 4, 7] between clients and servers to reduce much unnecessary network traffic. Effective NIC caching by DCA essentially extends the benefit of network traffic reduction between clients and servers to the local interconnect bus within a storage server.

We developed a DCA prototype system based on an Intel [2] iSCSI target under Linux kernel 2.4.20. We used two real-world block-level storage server traces (cello99 and TPC-D) to conduct experiments. The results show that DCA can significantly boost storage server performance.

In a storage server cache, the locality distance [4] is usually longer than that of a higher level cache. This is because multi-level storage caches deployed on client-side and application server have already filtered out most of application locality. As a consequence, the PCI bus traffic in a storage server is more intensive than that of application servers such as Web server. We have simulated single-level LRU based NIC cache proposed in reference [9] with a set of configurations ranging from 2 MB to 128 MB under representative storage server traces, resulting in only negligible NIC hit rate and local interconnect traffic reduction. This motivates us to explore new cache architecture that can best exploit the block-level locality and NIC cache space for an iSCSI based storage server.

For the consideration of compatibility, we design a unified read and write cache called *Helper Cache* at iSCSI application level instead of being merged with kernel level socket buffer or disk I/O buffer cache that have already been optimally implemented for different purposes. The size of a helper cache is larger than that of NIC cache. It is used to assist NIC cache in making placement decision and absorb transient writes by employing a write-back policy with periodical flushes.

In DCA, the PCI traffic reduction is largely determined by three factors: the NIC cache hit ratio, helper cache hit ratio and cache coherency overhead between NIC cache and helper cache. As an on-board cache, NIC cache has a limited memory space. Our initial experimental results indicated that current separate NIC cache and system I/O buffer cache solutions cannot deal with mixed read and write traf-

fic well. To efficiently and effectively utilize the limited memory space, we developed a new effective NIC caching solution-DCA. As a two level storage cache architecture, more issues need to be addressed than a single level NIC cache.

3.1 NIC Cache Organization

To efficiently organize the limited space on NIC cache, we consider the following four issues in our design.

3.1.1 Basic Cache Line Unit

Single disk block is chosen as an atomic cache line unit in both NIC cache and helper cache for the following considerations. A block-level cache implementation using single block as basic cache line unit can take advantage of the partial hit to best capture the locality in storage servers. Given part of the requested blocks are found in NIC cache, DCA only sends the remaining blocks from helper cache to NIC. After that, NIC combines both parts together into a complete package and send it out. By this way, every block cached at NIC is effectively utilized. More importantly, there are no alignment problem and extra overhead involved here at all.

Although client-informed application-level hints can also be utilized to effectively capture the locality in lower-level storage cache [11, 7], it is expensive and sometime prohibitive to modify the client-side software to provide such hints. As a result, it is imperative to employ single block as a basic cache line unit to best capture locality and make DCA be compatible with current applications.

3.1.2 Managing Cache Space

Even given a NIC cache in the several hundreds of megabytes, without a good organization, the hit rate could still be very low because the interleaved volume iSCSI data traffic with multiple targets and poor locality. A flat NIC cache space allocation scheme may work well in a Web server environment with a small working set. For an iSCSI based storage server, a simple flat cache space shared by multiple targets can be easily polluted by capacity misses. To solve the problem, we organize the space of NIC cache into a hierarchical structure. As shown in Figure 1, at the first level, we use a *target directory* to maintain per iSCSI target related information such as basic block size, maximum LBA address and cache space currently allocated for the specific target. Each entry in target directory is indexed by a unique pair of IP address and TCP port. At the second level, we save the cache data and related meta-data (i.e., cache directory) separately for each target. We maintain the first level target directory and second level per target cache directory in host system to reduce the PCI traffic due

to cache lookup, leaving real data blocks in the NIC cache. In this way, the NIC cache space is exclusively utilized for data caching.

3.1.3 Inclusive vs. Exclusive

Previous research in multi-level cache architectures between storage server and application server tells us that the exclusive cache property is preferred because their size and price are usually commensurable [11, 4]. In this environment, an inclusive cache architecture would waste half of the cache space in storing redundant data. Given an exclusive cache architecture, no data would be saved in different level caches at the same time. However, exclusive cache does not fit well for DCA in that the DEMOTE and Reload operations [11, 4] overhead itself may overwhelm the local interconnect. The reload overhead here over PCI bus can not be avoided by applying techniques proposed in reference work [4]. In addition, the size of the NIC cache is quite precious and relatively small compared with the cheaper and larger host memory. We make helper cache and NIC cache to be inclusive so that NIC cache can take the advantage of volume helper cache space as a backup reservoir upon a miss. The write-back policy in helper cache also makes the blocks stay at host memory for a longer time so that more block information can be used to make better NIC cache placement decision.

3.1.4 Cache Coherency

For multi-level inclusive caches, maintaining the coherence property of multiple data copies is an important problem. *Aggressive update* scheme can easily overwhelm the PCI bus due to the heavy update traffic between helper cache and NIC cache. *Conservative invalidate* scheme usually keeps the blocks of helper cache and NIC cache in an inconsistent state. To strike a good tradeoff, we choose a hybrid policy combining the advantages of both aggressive update and conservative invalidate schemes.

3.2 NIC Cache Placement

Since a simple access-based placement algorithm for NIC cache cannot work well with mixed read and write traffic, we need to develop a new algorithm which can identify the blocks which access frequency exceeds a pre-defined threshold but still have high possibility of future reads. We call these blocks as *hot blocks*. The objective of our cache placement algorithm is to cache most hot blocks in NIC cache, save *warm blocks* as reservoir in helper cache and kick *cold blocks* out of cache.

A large body of research work [11, 4, 7] have been done in client-oriented hierarchical cache architecture in storage environment. Their experimental results showed that

cache hit rate can be significantly improved by collaborative placement. Inspired by their work, DCA hands over the NIC cache placement decision to the helper cache instead of simply caching every recently accessed block.

Although DCA shares some similarities with hierarchical cache architecture, there exists significant differences between DCA and previous work. First, DCA cache placement decision is made within a storage server and transparent to the clients while current approaches collaborate with storage client. Second, DCA maintains inclusive property between helper cache and NIC cache. Third, current hierarchical cache architecture solutions aim to improve hit rate at a higher level and reduce demotion overhead over the wide area network. But DCA aims to maximize the lower-level NIC cache hit rate, thus slashing local interconnect traffic. To the best of our knowledge, this work is among the first to develop the hierarchical cache placement algorithm to effectively reduce PCI traffic within storage servers.

3.2.1 Off-line Analysis

We define a new metric called *State Locality Distance* to measure the locality in storage traffic, namely, RR (Read after Read) distance, WR (Read after Write) distance, WW (Write after Write) distance and RW (Write after Read) distance. Taking both access type and distance into consideration, the new metric indicates how many correlated read and write patterns exist in a block reference sequence.

Figure 2. Finite State Machine for Cache Blocks

Figure 2 illustrates a seven-state finite state machine that models the state transition of a cacheline in DCA, namely S (Start), SR (Single Read), SW (Single Write), RR (Read after Read), RW (Write after Read), WR (Read after Write) and WW (Write after Write). In the diagram, any block begins with the start state S. Given current request type as input, the state of a block is changed according to the transition arrows shown in Figure 2.

To detect the typical access patterns in storage traffic, we conduct an off-line analysis on representative storage

server traces (cello96, cello99, and TPC-D) at block level, mining clues on the strength of locality with the help of state locality distance and finite state machine model.

Figure 3 draws the histogram of RR distance, WR distance, WW distance, and RW distance distribution in an one-hour cello99 trace segment (one of the busiest hour between 03:00AM and 04:00AM on April, 29, 1999). The X-axis shows the length of state locality distance. The Y-axis denotes the number of occurrences that corresponds to a specific length of state locality distance. To derive the state locality distance from the traces, each request is decomposed into a unique block access event and assigned a new sequence number. The state locality distance is calculated based on the access type (r/w) and difference between sequence number of last two accesses on the same block.

Figure 3. Categorized Cello99 Trace Reference Distance Histogram

As seen in Figure 3 (c), the read after write distance is clustered at about 220,000, 360,000 and 380,000, which implies the weakest locality among the four types of state locality. The number of read after write cases is 4306, which only accounts for 2% of overall re-visited blocks. Therefore, this type of state locality may be considered as low priority to be cached in NIC cache.

Figure 3 (d) shows that there is a strong locality of transient write as evident by the fact that most of WW distance is less than 10,000. In other words, a block is often written and then overwritten instead of being visited by another read in between. Combined with the fact that WW pattern accounts for 15% of all re-visited blocks, it is wise to cache transient write in helper cache rather than NIC cache without discrimination.

For read access, strong locality strength does exist for Read after Read as evidenced by the fact that 45.1% of RR distance is less than 450 as shown in Figure 3 (b). Com-

Table 1. Statistics of State Locality Distance in Cello99

Category	RR	RW	WR	WW
TotalNumberOfCases	34935	120171	4306	28130
AverageLocalityDistance	85827.5	71736.6	318230.4	128695.5

paring the number of cases that fall into the four categories of state locality shown in Table 1, we can conclude that the block reference pattern that matches RR and RW sequence has more cases and shorter average locality distance than those of WR. WW is not considered here since only read-related access pattern is exploited by a read-only NIC cache. As a result, we need to distinguish the state locality of RR from that of RW in order to enforce effective PCI traffic reduction. However, it may be too expensive to maintain the state locality distance for all cache blocks at runtime. We need to seek an efficient way to track the four types of locality distance.

It is noticed that in a LRU based cache, the longer the locality distance, the higher the possibility one block will be kicked out. Therefore, for those blocks with strong RR state locality, the number of reads on a specific block (called read count) always exceeds that of writes (called write count). The reason is that given either RW or WR as subsequence, the read count and write count on a given block do not change. Only RR and WW break the tie of read count and write count on the block. However, WW is more likely to have longer state locality distance and fewer occurrences than RR as shown in both Figure 3 (d) and Table 1. We analyzed cello96 and TPC-D traces and got similar observations. Due to space limitation, we do not present these results here. We use these observations as a basis to develop a new heuristic *State Locality-Aware cache Placement* algorithm called *SLAP*, which will be discussed in the next section.

3.2.2 Online Algorithm

Based on the off-line analysis, we develop a heuristic placement algorithm that can 1) reduce the cold misses as well as capacity misses with an modest NIC cache size and 2) lower the overhead of cache coherency. The main idea is to use the difference between the read count and write count on a block as hints of state locality strength. The blocks that currently have more reads than writes are identified as hot blocks and are placed in NIC cache. By this way, not only the block-level locality is captured, but also the runtime overhead is minimized. Before describing the SLAP algorithm, we introduce four interfaces between NIC cache and Helper cache.

- `search_nic_cache()` Determine whether a block is cached in the NIC cache or not by searching the target directory.

- `mov_nic_cache()` Notify the NIC cache to move blocks from NIC send buffer to a specific target's portion of NIC cache. It may be noted that this operation does not introduce additional traffic over host PCI bus as any requested data have to pass the NIC anyway.
- `update_nic_cache()` Duplicate newly written blocks from helper cache to NIC cache. DCA is conservative on this operation because it introduces extra data block transfers over PCI bus.
- `invalidate_nic_cache()` Invalidate the corresponding cache block in NIC cache. Compared with `update_nic_cache()`, this operation does not involve block data transfer over PCI bus.

```

1 /*The algorithm runs on each block b in a request req*/
2   Update R/W count and state of block b;
3   if (req.type = DCA_READ)
4     if (NIL != search_nic_cache(b))
5       return NIC_READ_HIT; //b is in NIC Cache
6   else if (b not in Helper Cache)
7     Add b to disk read pending list;
8     return HELP_READ_MISS;
9   else{
10    if (ReadCount(x) > WriteCount(x))
11      mov_nic_cache(req);
12    return HELP_READ_HIT;}
13   else if (req.type = DCA_WRITE)
14     if (b in Helper Cache){
15       //hcache_entry is the directory entry of b in helper cache
16       if (hcache_entry.last_type = DCA_WRITE)
17         invalidate_nic_cache(req);
18       if (ReadCount(b) > WriteCount(b)) &&
19         (ReadCount(b) > MOV_THRESHOLD)
20         update_nic_cache(req);
21       return HELP_WRITE_HIT;}
22   else{
23     allocate new cache_entry in helper cache;
24     set dirty bit;
25     return HELP_WRITE_MISS;}

```

Figure 4. SLAP Algorithm in Pseudo-code

The SLAP algorithm is detailed in Figure 4. To predict future accesses more accurately, a placement policy based on a larger history window of state locality distance may be used. However, the maintenance overhead associated with each cacheline involves an $O(n)$ time operation upon each block access. To strike a good trade-off, SLAP algorithm

selectively chooses hot blocks based on state and access frequency, which requires only $O(1)$ time to make the NIC cache placement decision.

4 Implementation

Considering different iSCSI target implementations [2, 3], we have several ways to implement DCA. The helper cache can be implemented in non-pageable host memory area with interfaces to both iSCSI target and NIC cache. The NIC cache resides in NIC on-board memory and accepts placement direction from Helper cache via driver interfaces. The target directory and per target cache directory information of NIC cache are maintained by NIC driver in host system. We initially attempted to use common programmable NIC card with firmware source code available. However, most of them provide up to 2 MB on-board memory which may only be applicable for higher-level caching such as static web pages. For storage servers, some latest NIC cards provide on-board memory extension such as Adaptec 7211c card [1] that supports 512 MB memory and is priced less than 500\$. However, its firmware source code is not publicly available for modification as some programmable NIC cards. In addition, its memory space is either used as an I/O buffer or simple cache without interaction with higher level caches.

Therefore, we choose to adopt a non-programmable NIC to emulate DCA prototype system based on open source software of iSCSI target provided from Intel [2]. We implement DCA software components by modifying the request process flow of the iSCSI target module and integrate it with helper cache and NIC cache management modules. To truly replay the data traffic over the PCI bus, given any NIC cache hit (including partial hit), we forward those blocks that miss in NIC cache along with corresponding iSCSI response headers to the NIC card. The NIC card only sends out the data from the host system, leaving the blocks that hit in the NIC cache unsent. By this scheme, the data traffic over PCI is unchanged. Since we use a non-programmable NIC in experiments, the traffic over Ethernet is changed because the blocks that hit in NIC cache are not composed and sent to the network. We believe that this would lead to trivial impact on our results for the following reasons: 1) all the results are collected at iSCSI target side which are not affected by the modified network traffic any way; 2) current full-duplex Gigabit Ethernet is not the system bottleneck compared with PCI bus that services both network and storage I/Os; 3) the CPUs in a storage server are powerful enough to deal with both I/O requests and DCA cache management.

5 Performance Evaluation

5.1 Experimental Methodology

To evaluate the performance of DCA, we setup an iSCSI based storage system testbed. Two commodity PCs act as an iSCSI client (or initiator) and an iSCSI storage server (or target) respectively. They are connected to a NetGear GS105 Gigabit Ethernet switch by two Intel Pro 1000/MT Gigabit Ethernet cards. Both machines have 1 GB DDR PC2700 RAM and run Linux kernel 2.4.20. The machine running as iSCSI storage server is installed with an Adaptec 39160 SCSI adaptor and a 73 GB Maxtor Atlas10K4 Ultra 320 SCSI disk. And the iSCSI client machine uses a Western Digital IDE disk with 200 GB capacity.

The current DCA prototype system has been developed based on open source software iSCSI implementation from Intel [2], with about 700 lines of the iSCSI target code and 500 lines of iSCSI initiator code modified. On iSCSI target side, two-level DCA cache management and the SLAP algorithm module are integrated into the user-space iSCSI target module. To run real-world workloads for performance evaluation, we developed a trace generator by modifying the iSCSI initiator module code. The modified iSCSI initiator fetches requests from real-life block-level storage server traces and sends them to the iSCSI target via Ethernet. The performance results in terms of cache hit/miss ratio and PCI traffic along with iSCSI storage server throughput were all collected at iSCSI target side.

Two real-world storage server traces cello99 and TPC-D provided by HP were chosen to drive the experiments as they represent modern file server and decision-making database server applications. Both contain mixed read and write requests with different read-to-write ratio from multiple users, which fits well with iSCSI applications. Cello99 trace segment has a 1,203 MB data set with 57.3% reads (collected between 02:00AM and 03:00AM on May, 02, 1999) while the TPC-D trace segment we use is collected during Q10 benchmark with a 105, 075 MB data set in total with an read ratio of 82.5%.

5.2 Results and Analysis

5.2.1 PCI Traffic Reduction

To evaluate the DCA performance, we conducted a comprehensive set of experiments, varying the NIC cache size from 8 MB to 128 MB and helper cache size from 128 MB to 512 MB. The maximum size of helper cache in experiments is set to be 512 MB due to the physical memory limitation on the iSCSI Target server. The amount of PCI traffic is collected by summarizing all the data (i.e., block transfers, movements and updates from helper cache to NIC cache) and control messages (i.e., iSCSI headers) that pass the PCI

Figure 5. Normalized PCI Traffic of Cello99

bus. We present the results for cello99 and TPC-D in Figure 5 and Figure 6 respectively. All the numbers shown in Y-axis are PCI traffic normalized to that of a standard iSCSI storage server without DCA support, namely a baseline system. Although DCA has a large helper cache, we believe that the comparison here is fair because current Linux 2.4 kernel adopts an aggressive I/O buffer allocation policy. As a result, the size of I/O buffer cache in baseline system is always commensurate to the sum of the helper cache size and I/O buffer cache size in DCA prototype system during our experiments.

Given a DCA with a 128 MB helper cache, the percentage of the PCI traffic reduction remains at 27% for cello99 and 18% for TPC-D. However, given a DCA with a 512 MB helper cache, increasing the NIC cache size from 8 MB to 128 MB results in remarkable PCI traffic reduction, as seen from the fact that the normalized PCI traffic curve drops sharply from 64% to 51% for cello99. The same trend can be observed conspicuously in TPC-D workload, which changes from 53% to 26% when increasing the NIC cache size from 8 MB to 128 MB, given a 512 MB helper cache.

Based on the above results, we conclude that DCA matches our conjectures of effective PCI traffic reduction. DCA always beats the kernel-level I/O buffer cache scheme (i.e., baseline system) and the separate NIC cache solution in terms of PCI traffic reduction because the LRU algorithm cannot work well for a storage server workload with long locality distances. To have a further insight look at PCI traffic reduction, we analyze the cache Hit/Miss ratio next.

5.2.2 Anatomy of Cache Hit/Miss Ratio

We first decompose the cache hits in DCA into three categories, namely NRH (NIC Read Hit), HRH (Helper Read Hit) and HWH (Helper Write Hit). Since NIC cache only serves read requests, we interchangeably use NIC hit and NRH in the rest of this paper. In DCA, both read and write misses eventually resort to the helper cache, which leads to either HRM (Helper Read Miss) or HWM (Helper Write

Figure 6. Normalized PCI Traffic of TPC-D

Miss). Figure 7 and Figure 8 present the anatomy of cello99 and TPC-D experimental results with different DCA configurations. In both figures, X-axis denotes the NIC cache size and Y-axis denotes the stacked ratio of NRH, HRH, HWH, HRM and HWM in a bottom-up order. The left, middle and right sub-graphs of each figure show the results of a 128 MB, 256 MB and 512 MB helper cache respectively.

As seen in Figure 7, the larger the NIC cache, the higher the NIC cache hit ratio. However, Figure 7.b shows that the highest hit ratio of NIC cache with a 128 MB helper cache is always lower than 7.2%, even when the NIC cache size is increased to 128 MB. This indicates that a helper cache with less than 128 MB is not large enough to mine useful hints for NIC cache and thus resulting in a limited hit ratio gain. For simple NIC cache schemes (if there is no helper cache), the mixed read and write traffic can easily pollute the NIC cache space by indiscriminately filling it up with all blocks accessed recently. When the helper cache is enlarged to 512 MB, we observe that the increase of NIC cache size is much more productive than both 128 MB and 256 MB helper cache cases. The hit ratio of NIC cache increases from 4.4% to 19.7% when the cache size changes from 8 MB to 128 MB.

For TPC-D, we get similar results as in cello99 with a 128 MB helper cache. Although TPC-D represents a typical read-dominant decision making database workload, the amount of PCI traffic reduction is insensitive to the NIC cache size, given a 128 MB helper cache. This implies that simple NIC cache cannot work efficiently even for some read-dominant workloads. This is because LRU algorithm cannot work well for block-level caches, especially when the locality distance is larger than the LRU stack length. As seen from Figure 8.b, a 256 MB helper cache can effectively assist NIC cache in choosing real hot blocks for caching, peaking NIC hit ratio at 18.6% with a 128 MB NIC cache. Compared with Figure 8.a, the NIC hit ratio with the same NIC cache size is only 5.5%. We can obtain more than three times improvement on NIC cache hit ratio by applying SLAP algorithm to an appropriate helper cache

Figure 7. Hit/Miss Rate of cello99

size. As seen from Figure 8.b, the NRH ratio increases from 12% to 27% when the NIC cache size increases from 8 MB to 128 MB. Given a 512 MB helper cache, the maximum NIC cache hit ratio reaches 52% with a 128 MB NIC cache. Combined with the write hits at helper cache, the above anatomy results explain how DCA is able to slash the PCI traffic by up to 73%.

5.2.3 Server Performance Improvement

Due to the space limitation, we only present the results of a 512 MB helper cache and a 128 MB NIC cache. The result of cello99 shows that the server throughput has been improved by 76.9%, increasing from 4.95 MB/s to 8.76 MB/s. The result of TPC-D realizes a 121% server throughput improvement compared to an iSCSI storage server without DCA support. These numbers further prove our conjectures of DCA design. For TPC-D, the NIC cache hit ratio reaches 61.5% and thus delivers a much better overall performance than that of cello99.

6 Conclusion

In this paper, we have developed a novel Data Cache Architecture called DCA to effectively reduce local interconnect traffic. In DCA, a moderate-size NIC cache serves most read requests without fetching data from the host system via local interconnect while a large read/write unify helper cache employs a SLAP algorithm to direct cache placement for NIC cache. The proposed DCA architecture exhibits an extreme flexibility, as it can work either for a single server with multiple iSCSI targets, or a group of servers with multiple iSCSI targets (such as clustered storage servers). Our comprehensive experiments with representative real-life storage server workloads prove that DCA can effectively slash the PCI traffic by up to 74% and boost iSCSI storage server throughput by 121% compared with an iSCSI target without DCA support.

7 Acknowledgments

We thank Peng Gu for his participation and discussion in the early stage of this project. We also would like to thank

Figure 8. Hit/Miss Rate of TPC-D

Hewlett-Packard Labs for providing storage traces including cello99 and TPC-D.

References

- [1] Adaptec iscsi card7211c data sheet. http://graphics.adaptec.com/pdfs/iscsi_7211_datasheet.pdf.
- [2] Intel iSCSI Reference Implementation. <http://sourceforge.net/projects/intel-iscsi/>.
- [3] UNH-iSCSI project. <http://unh-iscsi.sourceforge.net/>.
- [4] Z. Chen, Y. Zhou, and K. Li. Eviction-based cache placement for storage caches. In *USENIX Annual Technical Conference, General Track 2003*, pages 269–281, 2003.
- [5] W. chun Feng, Justin, Hurwitz, H. B. Newman, S. Ravot, R. L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-gigabit Ethernet in networks of workstations, clusters, and grids: A case study. In *Proc. Of ACM/IEEE Supercomputing 2003: High-Performance Networking and Computing Conference*, 2003.
- [6] G. Gibson and R. Meter. Network attached storage architecture. *Communications of ACM*, 43(11):37–45, November 2000.
- [7] S. Jiang and X. Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 168–177, Tokyo, Japan, March 2004.
- [8] J. Satran, K. Meth, C. Sapuntzakis, and M. Chadalapaka. Internet Engineering Task Force, RFC 3720: Internet Small Computer System Interface (iSCSI). RFC 3720.
- [9] H. Kim, V. Pai, and S. Rixner. Increasing web server throughput with network interface data caching. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, San Jose, CA, October 2002.
- [10] B. Phillips. Have storage area networks come of age? *IEEE Computer*, 31(7):10–12, July 1998.
- [11] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference, General Track 2002*, pages 161–175, 2002.
- [12] M. Zhang and Q. Yang. BUCS: a bottom-up cache structure for networked storage servers. In *Proc. Of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 310–317, Montreal, Quebec, Canada, August 2004.