BUCS - A Bottom-Up Cache Structure for Networked Storage Servers

Ming Zhang and Qing Yang
Department of Electrical and Computer Engineering
University of Rhode Island
Kingston, RI 02881 USA
{mingz, qyang}@ele.uri.edu
Tel:1(401)874-2293
Fax:1(401)782-6422

Abstract

This paper introduces a new caching structure to improve server performance by minimizing data traffic over the system bus. The idea is to form a bottom-up caching hierarchy in a networked storage server. The bottom level cache is located on an embedded controller that is a combination of a network interface card (NIC) and a storage host bus adapter (HBA). Storage data coming from or going to a network are cached at this bottom level cache and meta-data related to these data are passed to the host for processing. When cached data exceed the capacity of the bottom level cache, some data are moved to the host RAM that is usually larger than the bottom level cache. This new cache hierarchy is referred to as ***bottom-up cache structure*** *(BUCS) in contrast to a traditional CPU-centric top-down cache where the top-level cache is the smallest and fastest, and the lower in the hierarchy the larger and slower the cache. Such data caching at the controller level dramatically reduces bus traffic and leads to great performance improvement for networked storages. We have implemented a proof-of-concept prototype using Intel's IQ80310 reference board and Linux network block device. Through performance measurements on the prototype implementation, we observed up to 3 times performance improvement of BUCS over traditional systems in terms of response time and system throughput.*

Key words*: cache structure, bus contention, networked storage, intelligent controller*

1. Introduction

Rapid technology advances have resulted in dramatic increase in CPU performance and network speed over the past decade. Similarly, throughput of data storages have also improved greatly due to technologies such as RAID and extensive caching. In contrast, the performance increase of system interconnects such as PCI bus have not kept pace with these improvements. As a result, it has become the major performance bottleneck for high performance servers. Extensive research has been done in addressing this bottleneck problem [2][4]. Most notable research efforts in this area aim at increasing the bandwidth of system interconnects by replacing PCI with PCI-X, PCI Express, or InfiniBand [1]. The InfiniBand technology uses a switch fabric as opposed to a shared bus thereby increasing bandwidth greatly [11].

It is interesting to recall the great amount of research efforts in designing various types of interconnection networks for multiprocessors for high communication bandwidth in the 80's and 90's, while at the same time there was also a great deal of research in minimizing communication by means of data caching. Both tracks of efforts contributed greatly to the architecture advance of parallel/distributed computing. We believe that it is both feasible and beneficial to build a cache hierarchy with an intelligent controller to minimize communication cost across the system interconnects.

Feasibility comes from the fact that embedded processor chips are becoming more powerful and less costly. This fact makes it cost-effective to offload many I/O and network functions at controller level [7][6] and to cache I/O and network data close to such embedded processors [12]. Table 1 lists performance parameters of three generation *I/O processors* (IOP) from Intel [10]. Compared to i960 processors that are still widely used in many RAID controllers, an IOP315 chip set has 6 times higher frequency and supports up to 12 GB on-board memory. Most Gigabit network adapters support checksum offloading by computing and checking packet checksums at the NIC level.

Potential benefit of such data caching is also fairly clear because data localities exist in many applications. Kim, Pai and Rixner have shown in their recent research [12] that

I/O Processor	i960	IOP310	IOP315
Bus Speed	33 MHz	66 MHz	133 MHz PCI-X
Bus Width	64/32-bit	64-bit	64-bit
CPU Speed	100 MHz	733 MHz	733 MHz
Memory Type	32/64-bit	64-bit	32/64-bit
Max Memory	128 MB	512 MB	12 GB

Table 1. Performance parameters of three generations of I/O Processor.

data locality exists in web applications and significant performance gain can be obtained with network interface data caching. The research work by Yocum and Chase [19] also showed the benefit of "payload caching" for network intermediary servers. By temporarily storing the payload of a packet in a NIC, they were able to improve system performance substantially. These existing research works indicate a great potential for reducing unnecessary data traffic across a system bus by means of data caching.

The above observations motivate us to introduce a new caching structure for the purpose of minimizing data traffic over a system bus. The idea is to form a bottom-up cache hierarchy in a server. The bottom level cache is located on an embedded controller that is a combination of a *network interface card* (NIC) and a storage *host bus adapter* (HBA). Storage data coming from or going to a network are cached at this bottom level cache and meta-data related to these storage data are passed to the host for processing. When cached data exceeds the capacity of the bottom level cache, some data are moved to the host RAM that is usually larger than the RAM on the controller. We call the cache on the controller *level-1* (L-1) cache and the host RAM *level-2* (L-2) cache. This new system is referred to as **bottom-up cache structure** (BUCS) in contrast to a traditional CPU-centric top-down cache where the top-level cache is the smallest and fastest, and the lower in the hierarchy the larger and slower the cache. BUCS tries to keep frequently-used data at a lower-level cache as much as possible to minimize data traffic over the system bus as opposed to placing frequently used data at a higher-level cache as much as possible in a traditional top-down cache hierarchy. For storage read requests from a network, most data are directly passed to the network through a L-1 cache. Similarly for storage write requests from the network, most data are directly written to the storage device through a write-back L-1 cache without copying them to the host RAM as being done in existing systems. Such data caching at a controller level dramatically reduces traffic on the system bus such as PCI bus and leads to a great performance improvement for networked storages. We have implemented a proof-of-concept prototype using Intel's IQ80310 reference board and Linux NBD (network block device). Measured results show that BUCS improves response time and system throughput over traditional systems by as much as a factor of 3.

The research contributions of this paper are four fold. Firstly, we proposed a new concept of a bottom-up cache structure. This caching structure clearly minimizes data traffic over a PCI bus and increases throughput for networked storages. Secondly, we proposed a marriage between a storage HBA and a NIC with unified cache memory. Although there exist controller cards containing both a storage HBA and a NIC [13], these cards mainly concentrate on the direct data bypass rather than the functional marriage of the two that share RAM and other resources as we proposed here. Such a marriage is both feasible with high performance embedded processors and beneficial because of short circuit of data transfer between storage device and network interface. Third, we have implemented a proof-of-concept prototype BUCS using Intel's IQ80310 with Intel XScale^{TM} IOP310 processor and embedded Linux. And finally, we have carried out performance measurements on the prototype to show that the BUCS system provides up to 3 times performance improvement in terms of response time and system throughput over a traditional server.

The paper is organized as follows. The next section gives a detailed description of BUCS architecture and designs. Section 3 presents the prototype implementation of BUCS and reports the performance results. Related work is discussed in Section 4. Section 5 concludes the paper.

2. Bottom-Up Cache Structure and Design

Data flow inside a normal storage server as results of read/write requests is shown in Figure 1. The server system consists of a system RAM, an HBA card controlling the storage device, and a NIC interconnected by a system interconnect typically a PCI bus. Upon receiving a read request over the NIC, the server host OS checks if the requested data are already in the host RAM. If not, the host OS invokes I/O operations to the storage HBA and loads the data from the storage device via the PCI bus. After the data are loaded to the host RAM, the host generates headers and assembles response packets to be transferred to NIC via the PCI bus. The NIC then sends the packets out to the requesting client. As a result, data are moved across the PCI bus at least once or even twice. Upon receiving a write request over the NIC, the host OS first loads the data from NIC to host RAM via the PCI bus and then stores the data into the attached storage device later, via the PCI bus again . Therefore, for write operations, one piece of data travels through the PCI bus twice. An important thing here is that the host never examines the data contents for either read operations or write operations except for moving data from one peripheral to another.

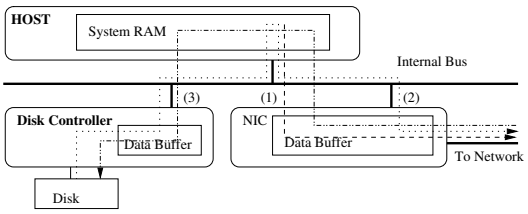


Figure 1. Data flows in a traditional system for three cases: (1) A network read request finds data in the system RAM. Data go through bus once. (2) The requested data is not in the system RAM. Data goes through bus twice: one from storage device to the RAM and the other from the RAM to network. (3) A write request from the network goes through the bus twice: one from network to the system RAM and the other from the system RAM to the storage.

The idea of BUCS is very simple. Instead of moving data back and forth every time between a peripheral device and host RAM across a PCI bus, we try to keep frequently used data in a cache at a controller level, the lowest level. Only meta-data that describe storage data and commands are transferred to the host system every time for necessary processing. Most of storage data do not travel through the PCI bus between host RAM and controllers because of effective caching at the low level. Since the lowest level cache (L-1 cache) is usually limited in size because of power and cost constraints, we use the host RAM as a L-2 cache to cache data replaced from the L-1 cache. The two level caches work together and are managed by a BUCS cache manager residing in the kernel of host OS.

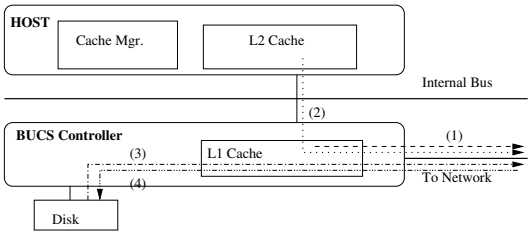


Figure 2. Data flows in a BUCS system for four cases: (1) A read request finds data in L-1 cache. Data do not go through system bus. (2) The requested data is not in L-1 cache but in L-2 cache. Data go through bus once. (3) The requested data is missed from both L-1 cache and L-2 cache. Data is loaded from the storage, cached at L-1 cache, and sent out to the network. No bus transfer is necessary. (4) Written data goes to L-1 cache and the storage device.

Figure 2 shows the data flow inside the server system as result of networked storage requests. For a read request, the BUCS cache manager checks if data are already in the L-1 or L-2 cache. If data is in the L-1 cache, the host prepares headers and invokes the BUCS controller to send data packets to the client over the network. If the data is in the L-2 cache, the cache manager moves the data from the L-2 cache to the L-1 cache. If the data is still in the storage device, the cache manager reads them out and puts them directly into the L-1 cache. In both cases, the host generates packet headers and transfers them to the BUCS controller. The controller then sends assembled packets including headers and data out to the client.

For a write request, the controller only generates a unique identifier for the data contained in a data packet and notifies the host of this id. The host then attaches this id with the meta-data in the corresponding previous command packet. The cache manager check if the old data are still in the L-1 cache or L-2 cache. If the data is in the L-1 cache, then new data will overwrite the old data directly. If the old data are in the L-2 cache, the old copy will be discarded and the actual data are stored in the L-1 cache directly. If old data are not found, the data are stored in the L-1 cache directly and later persisted into correct location in the attached storage device. The server simply responds the client with an acknowledge packet. Compared to the traditional system described above, BUCS does not transfer large bulk of data to host RAM but only related commands and meta-data via the PCI bus. As a result, the PCI bus is removed from the critical data path for most of operations.

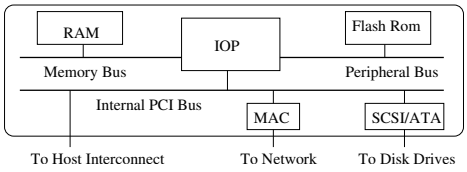


Figure 3. Functional block diagram of a BUCS controller. It integrates a storage HBA and NIC with a unified RAM.

BUCS controller is a marriage between a HBA and a NIC by integrating the functionalities of storage HBA and NIC, as shown in Figure 3. The firmware in the Flash ROM on-board contains the embedded OS code, the microcode of a storage controller, and some network protocol functions. Besides a high performance embedded processor, the BUCS controller has a storage controller chip that manipulates attached disks and a network *media access control* (MAC) chip that transmits and receives packets. An important component on the BUCS board is the on-board RAM, most of which is used as a L-1 cache except for some reserved space for on-board code. The BUCS controller is connected to a server host via a PCI or PCI-X interface.

Data placement and Identifying a data item in the cache: The basic unit for caching is a file block for file system level storage protocols or a disk block for block-level storage protocols allowing the system to maintain cache contents independently from network request packets. A special cache manager manages this two level cache hierarchy. All cached data are organized and managed by a single hashing table that uses the on-disk offset of a data block as its hash key. The size of each hash entry is around 20 bytes. If the average size of data represented by each entry is 4096 bytes, the hash entry cost is less than 5%, which is reasonable and feasible. When one data block is added to the L-1 or the L-2 cache, an existed hash entry is modified or a new hash entry is created by the cache manager, filled with meta-data about this data block, and inserted into the appropriate place in the hash table. Because the cache is managed by software, data mapping becomes trivial with addresses and pointers. We let a cache manager reside on the host and maintain meta-data in a host memory for both L-1 cache and L-2 cache. The cache manager sends different messages via APIs to the BUCS controller that acts as a slave to finish cache management tasks. The reason we choose this method is that network storage protocols are still processed at the host side, thus only the host can easily extract and acquire the meta-data about all cached data.

Data replacement policy: To make a room for new data to be placed in a cache upon cache full, we implement a LRU replacement policy in our cache manager while other replacement policies can be used as well. Most frequently used data are kept at L-1 cache. After the L-1 cache is full, least recently used data is replaced from the L-1 cache to the L-2 cache. The dirty data will be written to the storage device before be replaced to the L-2 cache. The cache manager updates the corresponding entry in the hash table to reflect such replacement. When a piece of data in the L-2 cache is accessed again and needs to be placed in the L-1 cache, it is prompted back to the L-1 cache. A hash entry is unlinked from the hash table and discarded by the cache manager when the data is discarded.

Write operations: Because a single cache manager manages the cache hierarchy, it is fairly easy to make sure that data in L-1 cache and L-2 cache are exclusive rather than inclusive. Data exclusivity not only makes efficient use of RAM space but also simplifies write operations because of no consistency issue between the two caches. Therefore, write operation is performed only at the cache where the data is located and there is no write-back or write-through issue between the two caches. Between cache and storage device, a write-back policy is used.

Interfaces between BUCS and Host OS: With a new integrated BUCS controller, interactions between host OS and controllers are changed and thus the interface between OS and BUCS has to be carefully designed. Our current design is to make the host system treat BUCS controller as a normal NIC with some additional functionalities. This greatly simplifies our implementation and keep changes to OS kernel minimum as opposed to introducing a whole new class of devices. In this way, interface design is similar to that of [12] with some modifications to satisfy our specific requirements. We add codes in the host OS to export several APIs that can be utilized by other parts of the OS and also add corresponding microcodes in BUCS controller. The APIs we have implemented include *bucs_cache_init()*, *bucs_append_data()*, *bucs_read_data()*, *bucs_write_data()*, *bucs_destage_l1()*, *bucs_prompt_l2()*, and etc. The detailed description of these APIs can be found in [20]. For each API, the host OS writes a specific command code and parameters to the registers of BUCS controller, and the command dispatcher invokes the corresponding microcode onboard to finish desired tasks.

3. Experimental Study

3.1. Prototype Implementation

We have implemented a proof-of-concept prototype of our BUCS using an Intel IQ80310 reference board driven by an IOP310 CPU. The board is plugged into a PCI slot of a host and is driven by an embedded Linux. We run the BUCS microcode as a kernel module running under this embedded Linux. The code uses the DMA engine and Message Unit provided by the hardware to communicate with the cache manager on the host. We modified the Linux kernel on the host to add a preliminary cache manager and to implement the APIs. Because of the time limitation, we did not finish the integration with the host OS and can not provide transparent support to all user space programs. Instead we reimplemented one of the data access protocols, NBD, to utilize our BUCS. We believe that fully integration with OS only need more programming work and our current prototype is sufficient to demonstrate the potential benefits of BUCS. The reason we choose NBD is that it is a simple protocol that can be easily modified and customized. A NBD client in Linux OS is a kernel module that exports a block device that can be partitioned, read, and written just like a normal disk device. It connects to a NBD server and redirects I/O requests to the server. We rewrite the user space NBD server to be a kernel module so that it can accept requests from clients in kernel and/or interact with the BUCS directly.

3.2. Experimental Environment and Workloads

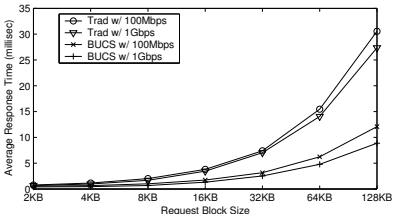
Using the prototype implementation discussed above, we carried out performance measurements of BUCS compared to traditional servers. Five PCs are used in our experiments

with one PC acting as a server and remaining 4 PCs acting as clients. All the PCs have single Pentium III 866 MHz CPU, 512 MB PC133 SDRAM, 64bit-33MHz PCI bus, and Maxtor 10 GB IDE Disk. They are interconnected by an Intel NetStructure 470T Gigabit Switch via Intel Pro1000 Gigabit NICs. An Intel IQ80310 board acts as a BUCS controller and is plugged into one of the PCI slots in the server PC. It has an Intel Pro1000T Gigabit NIC, an Adaptec 39160 Ultra 160 SCSI controller, and a Seagate Cheetah 15,000 rpm SCSI disk (ST318453LW) as storage device. We reserved 128 MB memory on IQ80310 as the L-1 cache and 256 MB memory in the host as the L-2 cache.

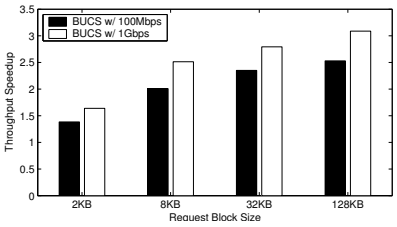
Two kinds of workloads are used in our measurements. The first one is a micro-benchmark that generates continuous storage requests with pre-specified data sizes. The purpose of the benchmark is to observe performance behaviors of the BUCS system under various request data sizes and request types. The second workload is a real block level trace downloaded from the Trace Distribution Center at Brigham Young University. They had run TPC-C benchmark with 20 data warehouses using Postgres database on Redhat Linux 7.1 and collected the trace using their kernel level disk trace tool, DTB. We ignored the time stamp values in the trace and sent one request immediately after the previous request is completed since we are interested in the response time value under two different systems. The trace file contains more than 3 million requests with size ranging from 4KB to 124KB. It has an average request size (mean value) about 92.4 KB and is read-dominated with 99% of its operations being read operations. The average request rate (mean value) is about 282 requests/second.

3.3. Numerical Results and Discussions

Our first experiment is to measure performances of the BUCS and the traditional server for read requests generated from one NBD client over the network using the micro-benchmark. Figure 4 shows our measured results for both BUCS and the traditional server. In this figure, response times and throughput speedup are plotted as functions of data sizes of read requests. Throughput speedup is defined as the ratio between the average throughput of the BUCS and the throughput of the traditional system with same speed network. For each system, two sets of data were collected corresponding to 100 Mbps network and 1Gbps network, respectively. As shown in Figure 4, performances of the BUCS are significantly better than that of the traditional system. As the data size increases, performance improvements of the BUCS increase. This result agrees well with our initial expectation. The larger data size is, the more data will travel through the PCI bus between host RAM and the NIC/HBA in the traditional system. As a result, BUCS shows greater advantages because of effective data caching and minimization of PCI traffic. The performance improvement goes as high as a factor of 3. For example, the average response time of BUCS is 8.88 ms compared to 27.36 ms of the traditional system for data size of 128KB on the 1Gbps network. The throughput speedup goes as high as 3.09 with 1 Gbps network.



(a) Average Read Response Time

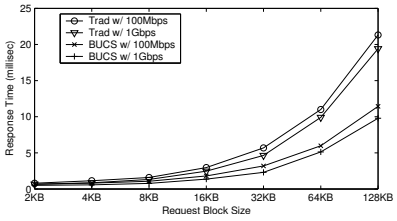


(b) Read Throughput Speedup

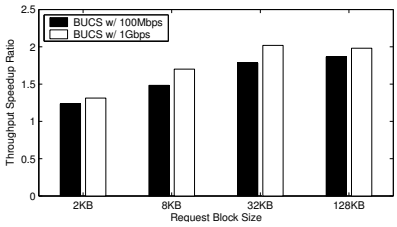
Figure 4. Measured read performance of the traditional system and the BUCS system with single NBD client.

The throughput speedup increases when we increase the network speed from 100 Mbps to 1 Gbps as shown in Figure 4. For example, with data block size being 128 KB, the speedup of BUCS is 2.53 with 100 Mbps network while it becomes 3.09 with 1 Gbps network. It is clearly shown that our BUCS benefits more from network improvement than the traditional system. While throughput speedup increases greatly as network bandwidth increases, the response time change is not as significant. This is because network latency does not reduce linearly with the increase of network bandwidth. For example, we measured average network latencies (mean value) over 100 Mbps and 1 Gigabit Ethernet switches to be 128.99 and 106.78 microseconds, respectively [9]. These results indicate that network latencies resulting from packet propagation delays do not change dramatically as Ethernet switches increase their bandwidth from 100 Mbps to 1 Gbps.

Similar performance results were observed for write operations as shown in Figure 5. Again, performance improvement of BUCS increases with the increase of data size. However, such increase is not monotonic. At size 32KB, the speedup of BUCS with 1Gbps network reaches a peak point and comes down for larger sizes. While analyzing this result, we noticed that the on-chip data cache of IOP310 is 32KB [10]. Our current implementation utilizes the TCP/IP



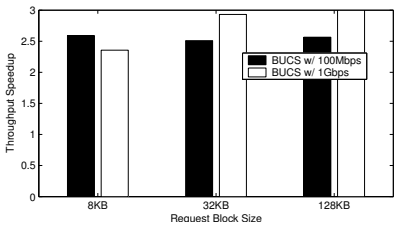
(a) Average Write Response Time



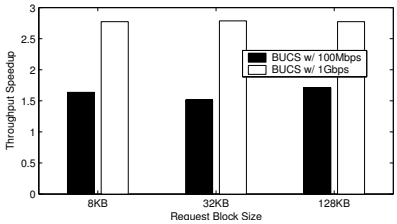
(b) Write Throughput Speedup

Figure 5. Measured write performance of the traditional system and the BUCS system with single NBD client.

stack of the embedded Linux that carries out more memory data copies when receiving data than sending out data. As a result, when write operations are performed the cache effects make a more difference in throughput. Such effects are not clearly shown for read operations and for lower speed network. Therefore, it further implies that there is room for additional performance improvement through code optimizations on the BUCS board.



(a) Read Throughput Speedup



(b) Write Throughput Speedup

Figure 6. Speedup measured using four clients.

Our next experiment is to increase the number of clients that generate networked storage requests in order to observe how BUCS performs with higher workloads. Figure 6 shows the server throughput speedup with four clients in the network. Three sets of bars were drawn for three different data sizes, 8KB, 32KB, and 128KB, respectively. It is observed that the throughput speedup of BUCS is similar to the one-client case implying that BUCS can handle higher load and scale fairly well with more clients. Similar to the single client case, up to a factor of 3 performance improvement was observed for the 4-client cases. Therefore, we can claim with experimental backing that BUCS effectively reduces bottleneck of the system bus.

Figure 7 shows the response time plots measured for TPC-C trace. Each dot in this figure represents the exact response time of one storage request. We plotted 10,000 requests randomly selected from the 3 million requests. It is clearly seen from this figure that BUCS dramatically reduces response time as compared to traditional systems. For the 100 Mbps network, most requests finish within about 12 ms with BUCS system as opposed to 30 ms with traditional systems. For 1Gbps network, similar performance differences are observed between BUCS and traditional systems with slightly lower response times. To further observe the distribution of response times, we summarized number of requests finished in different time intervals in Table 2. As shown in Table 2, BUCS can finish over 50% of storage requests within 10 ms and close to 100% remaining requests within 20 ms. With traditional system, only about 20% of requests can finish within 10 ms and over 58% of requests complete in longer than 20 ms. For 1Gbps network, 99.89% of requests complete within 10 ms with BUCS while only 24.29% of requests finish within 10 ms and majority take more than 20 ms with traditional systems. The average response time of BUCS is about three times faster than that of traditional systems. Performance results for TPC-C trace are fairly consistent with the results of the micro-benchmarks describe above. Although other researchers [21] had observed the poor temporal locality of TPC-C I/O accesses after being filtered by a large database buffer cache, our BUCS still provides a data "shortcut" to avoid extra PCI bus traffic.

4. Related Work

The bus contention problem is not new and was pointed out a few years ago by Arpaci-Dusseau et *al* [2] for streaming applications. Barve et *al* [4] found that the overall performance of the storage subsystem is bound by the performance of the I/O bus that connects multiple disks because all disk services are serialized due to bus contention.

An interesting work done by Krishnamurthy et *al* [13] offloads scheduling functions to network interface equipped with an i960 processor, improving greatly scalability in media services. Their approach allows direct data forwarding

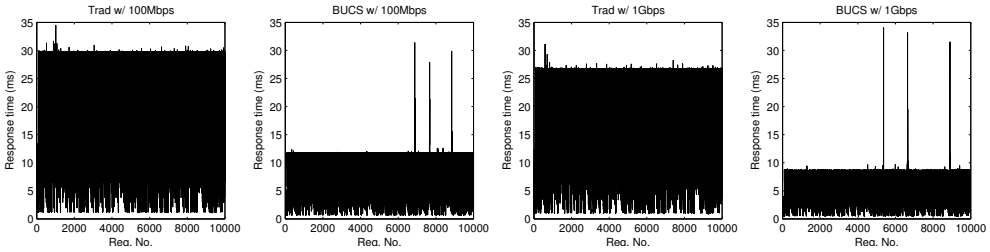


Figure 7. Measured response times (millisecond) of the BUCS and the traditional system using TPC-C trace.

	Number of requests with response time t				Avg. Response Time
	$t < 10$ ms	$10 \leq t < 20$ ms	$20 \leq t < 30$ ms	30 ms $\leq t$	(mean value)
Traditional w/ 100 Mbps	2218	1856	5897	29	20.553 ms
BUCS w/ 100 Mbps	5116	4879	4	1	8.207 ms
Traditional w/ 1 Gbps	2429	2151	5419	1	18.513 ms
BUCS w/ 1 Gbps	9989	8	0	3	6.103 ms

Table 2. The number of requests finished in different time intervals using TPC-C trace. We sampled 10000 requests randomly.

from a disk to network eliminating traffic from host CPU and host RAM by using the co-processor. Our work differs from this work in that we propose a general cache hierarchy for general storage servers as opposed to scalable scheduling for stream media applications.

There are several research projects that study data caching at network adapter level. The "payload caching" [19] by Yocum and Chase employs cache on a NIC as a short-term buffer for payloads of packets in a network intermediary to reduce bus traffic. It is shown in their research that substantial performance gains are obtained by caching part of incoming packet stream and directly forwarding them from the cache to the network. A research work by Kim et *al* [12] introduces a network interface data caching to reduce local interconnect traffic on networking servers. They have shown such caching reduces PCI traffic and increases web server throughput significantly because of high data locality that exists in web applications. Walton et *al* [17] combined two Myrinet cards on a same bus for efficient IP forwarding by transferring data directly from one card to another, called *peer DMA*, without involving the host processor. Although our BUCS shares some common objectives with the above work of network interface caching, the difference of our work from that of above is the integration of the storage controller cache and NIC cache eliminating data copy between storage subsystem and network subsystem via PCI bus. Furthermore, our two-level cache hierarchy and efficient management of the caches allow much larger cache size to be seen by clients than a single on-board cache that is usually limited because of cost and power constraints.

Recently, there has been a great deal of research in offloading computation tasks to programmable and intelligent devices to improve system performance. Most of the modern Gigabit NICs can compute the checksum in hardware, which was proposed in RFC1936 [16]. By utilizing this feature and other optimizations, Trapeze/IP [8] yielded a TCP bandwidth of 988 Mbps on a Gigabit network. In order to reduce the CPU utilization and achieve wire speed in a Gigabit network environment, various TOE [18] products and iSCSI accelerators are under production. Buonadonna and Culler [5] proposed a new system area network architecture, Queue Pair IP, that provides a QP abstraction on existing inter-network protocols and effectively reduces the CPU utilization by offloading part of the network protocol to a programmable NIC. EMP [15] also provides a zero-copy OS-bypass message passing interface by offloading the protocol to programmable NICs. TCP Server [14] and Split-OS [3] decouple various OS functionalities and offload them to different intelligent devices. TCP Server achieves performance gains of up to 30% by using dedicated network processors on a symmetric multiprocessor server or dedicated nodes on a cluster-based server. Our intention is to improve the server performance by reducing bus traffics. Instead of offloading computation tasks, our work "offloads" frequently used data to an intelligent adapter.

5. Conclusions and Future Work

This paper has introduced a new caching structure, BUCS, to address the bus contention problem, which limits the further performance improvement of a storage server.

In a BUCS system, a storage controller and a NIC are replaced by a BUCS controller that integrates the functionalities of both and has a unified cache memory. By placing frequently used data in the on-board cache memory, the L-1 cache, many read requests can be satisfied directly. A write request from a client can be satisfied by putting data in the L-1 cache directly, without invoking any bus traffic. The data in the L1 cache will be replaced to the host memory, the L2 cache, when needed. With effective caching policy, this two level cache can provide a high speed and large size cache for networked storage data accesses. Through experiments on our prototype implementation using micro-benchmark and real world traces, we have observed up to 3 times performance improvement of BUCS over traditional systems in terms of response time and system throughput. We are currently working on our prototype and hope to have a fully functional and highly optimized implementation in the near future.

Acknowledgments

This research is sponsored in part by National Science Foundation under grants CCR-0073377 and CCR-0312613. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We thank Dr. Xubin He at Tennessee Technological University for valuable suggestions and comments. We also thank Performance Evaluation laboratory at Brigham Young University for providing the TPC-C trace.

References

[1] Infiniband^{TM} specification. http://www.infinibandta.org, June 2001.

[2] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. The architectural costs of streaming I/O: A comparison of workstations, clusters, and SMPs. In *Proceedings of The Fourth International Symposium on High-Performance Computer Architecture*, pages 90–102, Las Vegas, Nevada, Feb. 1998.

[3] K. Banerjee, A. Bohra, S. Gopalakrishnan, M. Rangarajan, and L. Iftode. Split-OS: An operating system architecture for clusters of intelligent devices. In *Work-in-Progress Session at the 18th Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[4] R. D. Barve, P. B. Gibbons, B. Hillyer, Y. Matias, E. A. M. Shriver, and J. S. Vitter. Round-like behavior in multiple disks on a bus. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, pages 1–9, Atlanta, GA, May 1999.

[5] P. Buonadonna and D. E. Culler. Queue Pair IP: A hybrid architecture for System Area Networks. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 247–256, Anchorage, Alaska, May 2002.

[6] Y. Coady, J. S. Ong, and M. J. Feeley. Using embedded network processors to implement global memory management in a workstation cluster. In *The Eighth IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, CA, Aug. 1999.

[7] M. Fiuczynski, R. Martin, B. Bershad, and D. Culler. SPINE: An operating system for intelligent network adapters. Technical Report 98-08-01, University of Washington, Seattle, Aug. 1998.

[8] A. Gallatin, J. Chase, and K. Yocum. Trapeze/IP: TCP/IP at near-Gigabit speeds. In *Proceedings of the 1999 USENIX Annual Technical Conference (Freenix Track)*, Monterey, California, June 1999.

[9] X. He, Q. Yang, and M. Zhang. Introducing SCSI-To-IP Cache for Storage Area Networks. In *Proceedings of the 2002 International Conference on Parallel Processing*, pages 203–210, Vancouver, Canada, Aug. 2002.

[10] Intel. Intel I/O processor overview. http://www.intel.com/design/iio/index.htm.

[11] E. J. Kim, K. H. Yum, C. R. Das, M. Yousif, and J. Duato. Performance enhancement techniques for InfiniBand^{TM} architecture. In *Preceedings of the Ninth International Symposium on High-Performance Computer Architecture*, pages 253–262, Anaheim, CA, Feb. 2003.

[12] H. Kim, V. S. Pai, and S. Rixner. Increasing web server throughput with network interface data caching. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 239–250, San Jose, CA, Oct. 2002.

[13] R. Krishnamurthy, K. Schwan, R. West, and M.-C. Rosu. A network co-processor-based approach to scalable media streaming in servers. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 125–134, Toronto, Canada, Aug. 2000.

[14] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, R. Bianchini, L. Iftode, and W. Zwaenepoel. TCP servers: Offloading TCP/IP processing in internet servers. design, implementation, and performance. Technical Report DCS-TR-481, Rutgers University, Department of Computer Science, Mar. 2002.

[15] P. Shivam. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *Proceedings of SC 01*, Denver, Colorado, Nov. 2001.

[16] J. Touch and B. Parham. RFC 1936: Implementing the Internet checksum in hardware. http://www.ietf.org/rfc/rfc1936.txt.

[17] S. Walton, A. Hutton, and J. Touch. Efficient high speed data paths for IP forwarding using host based routers. In *Proceedings of the 10th IEEE Workshop on Local and Metropolitan Area Networks*, Sydney, Australia, Nov. 1999.

[18] E. Yeh, H. Chao, V. Mannem, J. Gervais, and B. Booth. Introduction to TCP/IP Offload Engine (TOE). http://www.10gea.org/SP0502IntroToTOE_F.pdf.

[19] K. Yocum and J. Chase. Payload caching: High-speed data forwarding for network intermediaries. In *Proceedings of 2001 USENIX Annual Technical Conference*, pages 305–317, Boston, MA, June 2001.

[20] M. Zhang and Q. Yang. BUCS - a bottom-up cache structure for networked storage servers. Technical report, Dept. ECE, Univ. of Rhode Island, May 2004.

[21] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX 2001 Annual Technical Conference*, pages 91–104, Boston, MA, June 2001.