Network-Centric Buffer Cache Organization

Gang Peng Srikant Sharma Tzi-cker Chiueh
Computer Science Department, Stony Brook University
{gpeng, srikant, chiueh}@cs.sunysb.edu

Abstract

A pass-through server such as an NFS server backed by an iSCSI[1] storage server only passes data between the storage server and NFS clients. Ideally it should require at most one data copying operation on sending or receiving, as in normal IP routers. In practice, pass-through servers actually incur multiple data copying operations because they are implemented using a layered architecture where each layer has its own internal data representation. This paper describes the design, implementation and evaluation of a novel network-centric buffer caching scheme called NCache *that minimizes data copying overhead in pass-through servers without requiring significant changes to their existing implementation. By organizing the data being passed or cached in a network friendly format,* NCache *is able to eliminate all unnecessary data copying. The key innovation in* NCache *is that it exploits the fact that pass-through servers do not interpret data, by replacing physical copying with logical copying in a way transparent to existing software. This transparency enables* NCache *to be easily portable to many operating systems. We have successfully built a Linux-based* NCache *prototype that can be applied to in-kernel NFS and static Web servers. Empirical performance measurements collected from this prototype show that by reducing the CPU load associated with data copying,* NCache *is able to provide up to 92% improvement in throughput for NFS server and up to 47% for Web server.*

1 Introduction

Network storage architecture separates bit movement from control processing. In this architecture, traditional network file servers now mainly support the functions of name translation, access control, and relaying the bits between network storage servers and clients. Because most of the bits exchanged between clients and network storage servers pass through the network file servers without additional interpretation, in theory the file servers should be able to relay them without incurring additional data copying overheads, just like normal IP routers. In practice, however, this is rarely the case. Following on the legacy implementations, most modern network file servers are implemented in a layered fashion, with each layer having its own special internal data format. For example, NFS daemons are typically built on top of local file systems, which in turn may rest on top of an iSCSI module, which in turn sits on top of the TCP/IP stack, etc. Copying and transforming data across layers is the simplest way to maintain the modularity of this layered architecture. Following on these issues, the goal of this research is to develop techniques that minimize data copying overhead for "pass-through" servers such as NFS servers backed by network storage, while preserving their modular software architecture.

Unlike IP routers, a network file server backed by network storage not only relays bits between network storage servers and clients but also satisfies network file access requests using its local file cache, which is a very common case in practice. The reliance of modern network file servers on local file system cache makes it difficult to reduce the number of data copying operations in the pass-through servers because the format of file blocks in the file system cache is very different from their counterparts in the network stack. For example, in a Linux-based NFS server backed by iSCSI storage server, data is stored in the format of 1500-Byte `sk_buff` in the network stack and as contiguous 4-KByte or 8-KByte buffer chunks in the page/buffer cache. This requires them to be explicitly copied during the movement between the file system cache and the network protocol stack.

Recognizing that *all* data cached on a pass-through network file server will eventually be sent out on the network, we propose that the buffer/page cache in a pass-through NFS server be organized in a network-ready format and passing data between file system cache and network stack be through pointer manipulation [18]. Each data item in the network-centric buffer cache (*NCache*) is called a *network-centric buffer*, which consists of a payload part that stores the file system data, and a metadata part that stores headers of various network protocol layers such as NFS, RPC, TCP/UDP, IP, Ethernet, etc. When a network packet containing normal file system data (i.e., non-metadata), for example an iSCSI read response or an NFS write request, arrives at the server, it is read into the network stack and cached in the network-centric cache without any modification. From this point

on, both the in-kernel NFS server and the network stack access these cached data through pointers. When a cached data item is to be sent out over the network, it is moved directly from the network-centric buffer cache to the network interface card. Network packets that contain file system metadata are sent through the protocol stack in the usual way, because the pass-through network file server needs to interpret and maybe modify them. Since these packets are typically small, the overhead of physically copying them is not significant.

Two important performance benefits accrue from the proposed *NCache* architecture. First and foremost, it eliminates unnecessary data copying within the pass-through servers. Second, as cached data is stored in a network ready format, the amount of work required to send out a cached data item is reduced substantially. For example, the protocol headers do not need to be repeatedly allocated and deallocated, as they are pre-allocated and stored in the cache. Also, the checksum of a cached block can be either pre-computed or inherited from the payload's originator, and does not need to be calculated repeatedly every time the block is sent out.

Although the main goal of *NCache*, i.e., avoiding unnecessary data copying, is the same as many other research projects, there is one important difference: *NCache* is designed keeping in mind that the amount of modification to the kernel and the network file server daemon should be minimal. Our Linux prototyping efforts show that except the standalone *NCache* module, the amount of *modification* to the Linux kernel and NFS daemon is fewer than 150 lines of C code. This advantage not only makes the *NCache* approach more likely to be accepted in existing information technology infrastructure, but also makes it more portable to other platforms and applications.

2 Related Work

NCache shares some similarities with Payload caching [19] in the sense that both target at intermediaries in network whose main task is to forward or pass-through data. Payload caching caches payload on network interface cards (NICs) to save traffic through I/O bus; *NCache* stores the pass-through data in a network friendly format and eliminates unnecessary data copying within host system. As a result, Payload caching is suitable for forwarding services which require minimal state information, such as firewall, protocol translators, etc., while *NCache* is more beneficial to applications that usually cache data, e.g., cache proxies, and NFS server with network storage.

Network-Attached Secure Disk (NASD) [14] is for high performance data delivery in distributed file system. There is no dedicated file server involved in the data path of NASD architecture; data is transmitted directly between end clients and NASD drives. In contrast, *NCache* is designed to work with the traditional NFS environment where NFS server is involved in data transmission.

H.K. Jerry Chu [10] describes a transparent copy avoidance approach to remove data copying and touching in TCP protocol stack by using virtual memory remapping and copy-on-write (COW) techniques. Genie [8] extends the transparent copy avoidance to support not only network I/O, but also file I/O, with a technique called *emulated copy*. This technique addresses the virtual memory page alignment problem and allows transparent copy-free network access under certain conditions. As in [10], it also leverages virtual memory remapping. A more generic scheme supporting transparent copy avoidance is Fbuf [12]. Fbuf provides a copy-free facility to move data among multiple protection domains via IPC.

However, utilizing these transparent copy avoidance schemes is usually difficult. They often incurs problems with virtual memory remapping or have specific interfaces that are quite different from those legacy applications may use. *NCache* does not reply on virtual memory remapping and keeps the legacy read/write semantics completely unchanged.

IO-Lite [17] gives up the transparency provided by the previous schemes and proposes new copy-free I/O interfaces used by various subsystems of operating systems based on a data structure called "*buffer aggregate*". Data is passed among different subsystems by moving references to buffer aggregates, rather than copying data. It also integrates this data structure into file system cache. As a result, the data in file system cache is stored as lists of buffer aggregates, instead of conventionally used pages. This integration obviously helps to reduce redundant stored data, but is intrusive to file system cache and is complicated to implement. The authors acknowledge these drawbacks in [17]. Unlike IO-Lite, *NCache* keeps the file system and file system cache abstractions intact so that it can be readily ported to popular general purpose operating systems, such as Linux, and FreeBSD.

A more aggressive approach to facilitate data transfer within operating system subsystems is to revamp the operating system design. Scout [16] introduces the use of explicit paths as an important abstraction in operating system design to improve performance. Extensible kernels [13] address various problems associated with existing operating systems, including performance issues, such as I/O bottleneck. Compared with these approaches, *NCache* is directly applicable and portable on existing operating systems and beneficial to legacy applications, as it does not propose revamping of operating systems.

In addition to the software approaches described so far, users have also resorted to special hardware support, especially for storage interconnects. TCP Offload Engine (TOE) [9] or Remote Direct Memory Access (RDMA) [4] are the typical techniques that are being explored. These techniques may have the potential of boosting data I/O performance of applications [15], but they often need the applications to use new file access

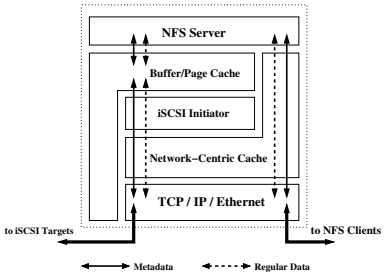


Figure 1. The software modules and data path within an NFS server that is backed by iSCSI storage and equipped with network-centric cache. The solid lines represent physical copying of metadata, whereas the dashed lines denote logical copying of regular data.

semantics, such as Direct Access File System (DAFS) semantics [11], to maximize the possible performance gain. Furthermore, all the hosts involved must be furnished with advanced NICs to eliminate every single data copying operation within the application servers, along the data path between storage servers and end users. This is not always practical, especially for end users, such as NFS and Web clients, which usually do not install the advanced NICs for economic reasons. Considering this, we design *NCache* to be capable of running over common commodity NICs, rather than relying on hardware with special support.

3 Network-Centric Buffer Design

Conceptually, a pass-through NFS server should behave just like an IP router, in that it should simply forward packets between iSCSI targets and NFS clients. So it should be relatively straightforward to emulate the single data copying behavior of modern routers. However, there are certain complications. First, although an iSCSI-based NFS server does not need to "touch" most packets' contents, it does need to interpret the packets that correspond to NFS metadata, examples of which include inodes, directory files, and the superblocks of the file system. Fortunately the majority of the bytes passing through an iSCSI-based NFS server do not require any additional processing beyond forwarding. These bytes only need to be copied into and out of the memory once. Second, an iSCSI-based NFS server, unlike other pass-through devices, caches data. Moreover, the organization of this cache is typically dictated by the file system, which may require additional data copying. The proposed network-centric buffer cache organization (*NCache*) successfully addresses the above issues while minimizing modifications required to the existing software base.

3.1 Basic NCache Structure

Under the *NCache* architecture, when a data packet arrives at an NFS server, it is read into the network stack and cached in the network-centric cache without any modification. When a cached data item is to be sent out, it is retrieved from the network-centric cache and transmitted on the network. No additional data copying is needed as the network-centric cache and the network stack represent data in the same way.

Figure 1 shows the structure of an NFS server using *NCache*. Data passing through the server is classified into two categories: *metadata* carrying information such as directory files and superblocks of a file system, and *regular data* corresponding to actual file blocks. Metadata is moved around within the NFS server just as in normal NFS servers, as shown by the solid lines in Figure 1. *NCache* stores regular data packets in a network-ready format and moves them around different modules through a *logical copying* mechanism, which copies the *keys* rather than the payload of regular data packets. These keys can later be used to retrieve the corresponding data packets. The dashed lines in Figure 1 denote the logical copying operations. Because keys are much smaller, logical copying significantly decreases the memory copying overhead.

3.2 Data Movement

Figure 2 shows the usefulness of network-centric cache by exhibiting the data movement within an NFS server using *NCache*. The target file block of an NFS read request is not in the server's cache, and an access request needs to be sent to the backend iSCSI storage server to retrieve the target block. The figure depicts the data flow of the response packets returned by the iSCSI server. When the response packets reach the NFS server, they are in the form of network buffer list. The *NCache* module intercepts the packets and determines if the packets carry regular data or metadata. If the packets carry metadata, they are physically copied across different modules as in standard NFS servers. However, if the packets carry regular data, the *NCache* module puts them into the network-centric buffer by hashing the logical block numbers (LBNs) contained in the response packets (step 2). Next, instead of physically copying the packets' payload, the *NCache* module only copies the associated keys (step 3, 4 and 5) through the logical copying interfaces. Since an LBN is much smaller than a file block (normally 4 KBytes), logical copying incurs significantly less overhead than physical copying.

To service an NFS read request, the NFS server checks the file system buffer cache, and logically copies the target block if it is in the cache (step 4). Although the retrieved block contains only a key and some "junk" data, nonetheless the NFS server can still compose a valid NFS read reply from the block, because it does not interpret the block's data. Following this the NFS server

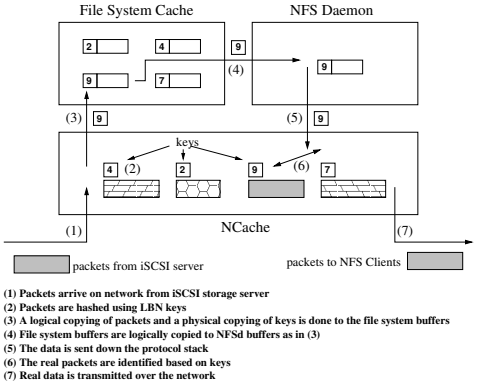


Figure 2. Data movement during the service of an NFS read request, assuming the target file block is not in cache. Packets returned by the iSCSI storage server come with logical block numbers, which can serve as keys. Only keys are copied around rather than physical data.

sends out the NFS read reply (step 5). When this reply is about to be sent out, the *NCache* module uses the key in the reply to retrieve the corresponding data block from the network-centric cache, substitutes the block for the reply (step 6), and sends the resulting reply back to the requesting NFS client (step 7).

3.3 Distinguishing Metadata from Data

A key design issue in network-centric cache is how to distinguish regular data packets from metadata packets, as the NFS server handles them differently: regular data packets are logically copied among modules whereas metadata packets are physically copied among modules. To differentiate regular data packets from metadata packets, the network-centric cache module has to rely on the higher-level protocol headers, iSCSI or NFS in our case. The Remote Procedure Call (RPC) field in NFS messages specifies the operation type. Among incoming NFS packets, only the payloads of NFS write request packets are cached in the network-centric cache, and among outgoing NFS packets only the payloads of NFS read replies are replaced with entries in the network-centric cache before they are sent out. The network-centric cache lets all other NFS packets through without any change.

For the iSCSI protocol, things are more complicated, as from the iSCSI protocol header alone one cannot decipher if the payload of a packet is carrying metadata or regular data. The only hint lies in the type of file system object on which an iSCSI command operates. If an iSCSI read or write command accesses a directory or a block device, it is accessing metadata; otherwise, the iSCSI command is accessing regular files. It is not possible to identify metadata by examining only the headers of iSCSI read responses. However, the page data structure associated with iSCSI requests contains the inode type information, which actually specifies whether the requests are for regular file data or metadata.

3.4 Cache Management

The network-centric cache in an NFS server is decomposed into two parts: an LBN cache and an FHO cache, because there are two sources of data and they provide different ways of uniquely identifying data items. The LBN cache stores data packets returned from the iSCSI storage server as responses to iSCSI read requests. These packets are indexed based on the logical block numbers (LBNs) in the corresponding requests. The FHO cache stores data packets that are NFS write requests from NFS clients. These packets are indexed based on a unique identifier for the associated file block: a file handle and a file offset (FHO). When the file system buffer cache is full, first clean buffers are reclaimed and then dirty buffers are flushed and reclaimed. When a dirty file system cache buffer is flushed, the *NCache* module intercepts the corresponding iSCSI request, uses the FHO key in the payload portion of the iSCSI request to identify the corresponding entry in the FHO cache, moves this FHO cache entry to the LBN cache, and changes the entry's key from the original FHO to an LBN specified in the iSCSI request header. If the LBN cache already has an entry with the same LBN, the FHO cache entry is overwritten on it because data in the FHO cache is always more up-to-date. This procedure of converting FHO cache entries to LBN cache entries is called *remapping*. Figure 3 shows detailed life time snapshots of a data block inside an NFS server using *NCache*, one step of which is this remapping.

Dirty blocks in the LBN cache can also be flushed to the iSCSI storage server when the *NCache* module runs short of memory. However, because the file system cache is configured to be much smaller than the network-centric cache, dirty file system cache buffers are more likely to be flushed than dirty blocks in the network-centric cache. As a result, remapping of a dirty FHO cache block to the LBN cache always takes place before the corresponding LBN cache block is flushed back to the iSCSI storage server.

Because the network-centric cache consists of an LBN cache and an FHO cache, the *NCache* module needs to consult with the appropriate cache when substituting NFS read replies or iSCSI write requests. Moreover, some NFS read replies may contain both an FHO key and an LBN key, because the corresponding data block is accessed through an NFS read request followed by an NFS write request. For these NFS read replies, the *NCache* module needs to consult the FHO cache using the FHO key first, and then the LBN cache using the LBN key. This way, it guarantees that NFS clients always receive the most up-to-date data.

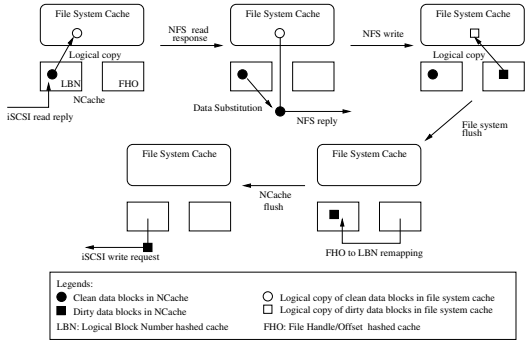


Figure 3. Typical life time of a data block inside an NFS server. The incoming data from storage server is put in LBN cache where data is indexed using disk LBNs. A logical copy of this data is maintained in file system cache. NFS replies are serviced using data from *NCache*. NFS writes result in dirty data blocks which are cached using file handle and offset indexing. A logical copy of these dirty data blocks is maintained in file system cache.

Physically the network-centric cache consists of fixed-sized data chunks, each of which consists of a list of network buffers. In addition, all the data chunks in the network-centric cache are chained into a linked list. Whenever a data chunk is accessed, it is moved to the end of the list. When all data chunks are used up, *NCache* picks the chunks from the head of the list as candidates for reclamation. If the candidate chunk is clean, it is simply freed; if it is dirty, e.g., data carried by an NFS write request, *NCache* writes back the dirty data to the remote storage server, and then reclaims the candidate chunk. The above design is an instance of the classical LRU cache replacement algorithm.

As shown in Figure 1, the file system buffer cache and the network-centric cache both cache data but are separate. This may lead to a piece of data being cached twice, which we refer to as double buffering, resulting in memory wastage. *NCache* resolves this problem by limiting the file system buffer cache size, which is possible in most modern operating systems, such as Linux, FreeBSD and Windows NT. Even though a small file system buffer cache may lead to excessive cache misses and thus extra disk accesses, most of these disk accesses are caught and serviced by a much larger network-centric cache, which thus acts as a second-level cache with respect to the file system buffer cache.

3.5 Generalizing NCache

The idea of NCache is applicable to all pass-through servers whose major task is to channel data between external parties with little or no interpretation of the data being relayed. An NFS server backed by iSCSI storage is only one instance of pass-through server. Other examples of pass-through server include Video-On-Demand server or static Web server using networked storage, content router, and web caching proxies.

Two issues need to be addressed when applying NCache to a given pass-through server. The first issue is how to distinguish metadata from regular data that goes through the server. This requires application-specific and protocol-specific customization, more specifically, checking fields of higher-level protocol headers. For example, for NFS NCache checks the RPC field in NFS messages, and for HTTP some specific string patterns in HTTP response header, like "\r\n\r\n", to determine if a packet corresponds to regular data or metadata. The second issue is how to handle mismatch between data block sizes used by different protocols. NCache first determines the data block size of various protocols to detect possible mismatch, for example, from size field in NFS replies or from response header for HTTP. Then NCache needs to transform an input data block to one or multiple output data blocks by splitting or merging so that they are properly aligned.

4 NCache Implementation

We discuss the implementation of *NCache* on Linux and how it is applied to NFS server as well as an in-kernel static Web server. Porting *NCache* to FreeBSD is also described.

4.1 Linux Implementation

To support logical copying, three kernel modifications are required. First, the read/write and sendfile interfaces that an NFS server uses to access the file system buffer cache is changed so that both reads and writes move only keys in file blocks rather than payloads of file blocks. Second, to eliminate the data copying operation required when invoking the network stack, the read/write interface of the network stack is also modified such that only the keys are copied rather than the payloads. The NFS server and the iSCSI initiator can benefit from this new interface. In addition, iSCSI initiator is modified to leverage this new interface. Third, the *NCache* module is inserted into the layer between the network stack and the Ethernet device driver to perform on-the-fly packet caching and replacement. As these kernel modifications are minimal and the network-centric caching functionality is implemented as a loadable module, *NCache* is largely transparent to the Linux kernel and thus can be easily ported to other operating systems. Table 1 summarizes the required kernel modifications in Linux. Not including the standalone *NCache* module, the total number of lines of C code modified in the kernel is fewer than 150. Overall, the *NCache* implementation under Linux is fairly self-contained, and as a result does not need to touch any complicated system

Module	Locations Modified
NFS/Web server daemon	None
buffer cache	None
iSCSI initiator	two functions invoking socket interface changed
network stack	TCP/IP socket interfaces extended

Table 1. Modifications to components of the Linux kernel. The network stack and iSCSI initiator are slightly modified; the NFS/Web server daemon and the buffer cache remain intact. In total, fewer than 150 lines of C code are added.

modules, such as the buffer cache and the application server.

Because the system has a network-centric cache as well as a file system buffer cache, it is essential to reduce the size of the file system buffer cache to minimize resource wastage due to double buffering. Since Linux does not provide a configuration option for controlling the buffer cache size directly, we use an indirect approach to reduce the buffer cache size. All the buffers cached in the network-centric cache are in fact network buffers allocated by the network device driver for packet reception. As these buffers are allocated in the device driver context, they are automatically pinned down in the physical memory. As a side effect, the Linux kernel and therefore the file system buffer cache can only use the part of physical memory that has not been allocated to *NCache*. Therefore, by carefully controlling the amount of memory allocated to *NCache*, one can control the amount of memory left to the file system buffer cache.

4.2 Porting to FreeBSD

Porting the Linux-based *NCache* implementation to FreeBSD is relatively straightforward, as both operating systems share a very similar structure. The network buffer structure used in FreeBSD is *mbuf*, while in Linux kernel it is *sk_buff*. However, both structures and their relevant routines in the operating systems support variable-size buffer operations very well, which is desirable for handling communication protocol packets. Hence, using `mbuf`, rather than `sk_buff`, does not lead to any structural change to *NCache*. In FreeBSD, to exploit logical copying, the read/write interface of the network stack exposed to the upper layer needs to be modified slightly, just as in Linux. FreeBSD simplifies the problem of limiting buffer cache size by allowing users to specify the buffer cache size limit directly. The mechanism for distinguishing metadata from regular data used in Linux can be readily used in FreeBSD as well. Finally, just like the Linux kernel, entries in the network-centric cache can be allocated in the device driver context. This means that these packets are automatically pinned down unless *NCache* explicitly frees them.

4.3 Applying NCache to Web Server

Besides the NFS server, we have also applied *NCache* to another pass-through server, kHTTPd [3], an in-kernel static Web server on Linux. kHTTPd handles only *static* web-pages, and passes all requests for non-static information to a regular user-space web server, e.g., Apache. To optimize performance, kHTTPd uses the *sendfile* interface to copy data directly from the file system buffer cache to the network stack when serving HTTP requests.

To support kHTTPd, we extended the *NCache* module to track HTTP streams sent out from kHTTPd. For packets carrying HTTP reply headers, *NCache* lets them go through without any action; for packets associated with web page contents, *NCache* retrieves the real content from its own cache and substitutes them for the intercepted packets, similar to the handling of NFS read replies. No further changes are needed for other kernel components and kHTTPd.

5 Performance Evaluation

5.1 Baseline Pass-through Servers

NCache aims to eliminate redundant data copying operations that occur within pass-through servers, but it may introduce its own overhead, e.g., overhead of managing network-centric buffer cache. To determine the maximal performance gain that is possible when all unnecessary data copying operations are eliminated, we removed all the data copying operations in the original Linux NFS server and kHTTPd, and used their performance as the base case for comparison.

We simply changed the data movement interfaces among three modules, buffer/page cache, NFS server/kHTTPd, and network stack, so that regular data copying operations are eliminated while metadata are copied through as usual. We distinguished between metadata and regular data by adding several pointer checks for each disk request, which do not incur much overhead. After this modification, the server daemon, NFS server or kHTTPd, neither retrieves regular data from buffer cache nor puts them through network stack. So the packets that are actually sent back to clients contain only random bits as payload. Use of random packets does not affect the performance measurement result reported here as the NFS clients in our experiments do not interpret the payloads.

From this point onwards, we refer to the original NFS server as *NFS-original*, the one with *NCache* kernel as *NFS-NCache* and the "baseline" case described above as

	Read Path		Write Path	
	Hit	Miss	Overwritten	Flushed
NFS server	2	3	1	2
kHTTPd	1	2	N/A	N/A

Table 2. Number of data copying operations per request in NFS server or kHTTPd for reading data or writing data. kHTTPd copies data directly from file system buffer cache to network stack when serving requests and thus for each request incurs one less copying operation than NFS server does in the read path.

NFS-baseline. Similar naming is applied to kHTTPd as well.

5.2 Testbed Setup

The testbed used to evaluate the performance impact of *NCache* consisted of a storage server, an application server running NFS or kHTTPd, and two clients. The operating system on all nodes was RedHat Linux with kernel version 2.4.19. The storage server was a Pentium III 1-GHz node with 512 MB RAM, 64-bit 66-MHz PCI bus and two Promise 66-MHz dual-channel IDE controllers. The storage was provided by 4 IDE disks (IBM DTLA-307075) configured as RAID-0. The application server was a Pentium III 1-GHz node with 896 MB RAM. The clients had a similar configuration with 384 MB RAM. All nodes were equipped with Intel Pro/1000 MT Server Gigabit Ethernet cards. The checksum offloading support on the Intel card was enabled by default, and the default Ethernet MTU size of 1500-Byte was used. The iSCSI reference implementation used is described in [2]. All the machines were connected through a NetGear Gigabit switch.

5.3 Workload Description

We compared the three versions of NFS server using micro-benchmarks and macro-benchmarks. There were two types of micro-benchmark workloads. One is to sequentially read a big file (2 GB) from the NFS server, which resembles *all-miss* workload. The other is to repetitively access a small file (5 MB) from the NFS server, which represents *all-hit* workload. The number of data copying operations each NFS read/write request may incur under these two micro-benchmarks is shown in Table 2. The workload was generated by means of synthetic traces and an *Active Trace Player* [20].

The macro-benchmark used in this study is SPECsfs V3 benchmarks [6]. The total NFS file system size was configured to be 2 GB, and the size of file set to be accessed was chosen to be 10% of the total file system size. We also used the default size distribution for regular data requests, in which small sized requests (< 16 KB) dominate. The ratio of NFS read and write requests was maintained at the default value, 5:1 . We measured the NFS server's throughput in terms of operations per second while varying the ratio between NFS reads and NFS writes.

We evaluated the three versions of kHTTPd using micro-benchmarks and macro-benchmarks. The micro-benchmark workload is to access a small working set (5 MB) repetitively from web server, which we refer to as an *all-hit* workload.

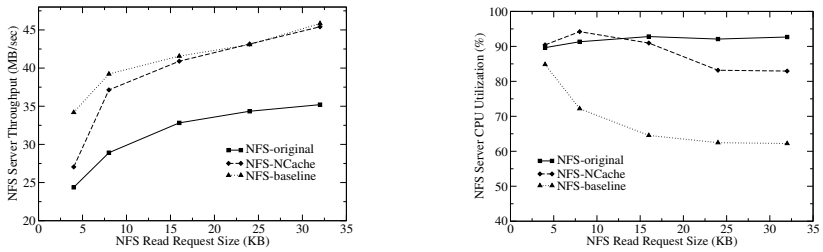
The macro-benchmark used was the SPECweb99 benchmark [5]. Only static web page requests were used as kHTTPd does not support non-static page requests. The distribution of web page access frequency was in compliance with Zipf's law [7]. The average web page size accessed was about 75 KB. Table 2 shows the number of data copying operations each web page access may incur in case of cache hit or miss.

5.4 Throughput of NFS Server

We ran the two micro-benchmarks against all three NFS server configurations. The workload comprised of read requests of size from 4 KB to 32 KB. The file system read ahead window was tuned appropriately so that the average disk request size matches with the NFS request size. The number of NFS server daemons was also adjusted to reach the best performance.

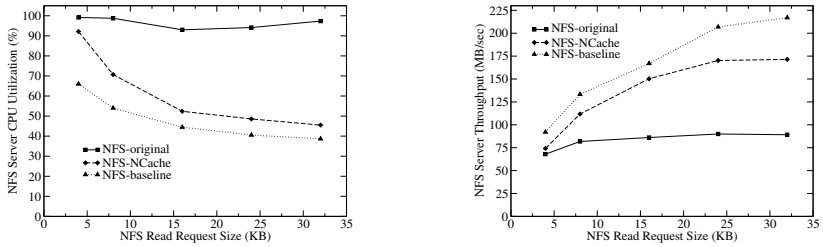
Figure 4 shows the throughput and CPU utilization for all three NFS server configurations when all NFS requests miss in the server's buffer cache. When the request size is 16 KB or larger, the throughput improvement of NFS-NCache over NFS-original ranges between 29% to 36%, similar with NFS-baseline's improvement. This improvement mainly comes from the elimination of unnecessary data copying operations, as shown by the difference in NFS server CPU utilization ratio. Between NFS-original and NFS-baseline, this difference is around 30% and attributed to reduction in the number of data copying operations. Between NFS-NCache and NFS-baseline, the difference is around 20% and due to the management overhead of network-centric buffer cache. Because of additional data copying, the server CPU for NFS-original is always saturated. However, for NFS-NCache, the server CPU utilization decreases as the request size increases, as in NFS-baseline. Despite this, the throughput improvement of NFS-NCache and NFS-baseline over NFS-original remains constant after the request size reaches 16 KB, because the storage server's CPU remains saturated from this point onwards.

When the request size is smaller than 16 KB, the per-packet overhead dominates the per-byte overhead (which includes data copying overhead). Therefore, the performance gain from eliminating unnecessary data copying is not as apparent. As the request size increases, the per-byte overhead plays an increasingly more important role, and the throughput difference between NFS-



(a) (b)

Figure 4. Throughput (a) and NFS server CPU utilization (b) for all three NFS server configurations under the all-miss workload.



(a) (b)

Figure 5. Throughput and NFS server CPU utilization for the three NFS server configurations under the all-hit workload. (a) shows the NFS server's CPU utilization when only one NIC is installed. In this case, the network link is the bottleneck for all configurations. (b) shows the throughput of the three NFS server configurations when two NICs are installed. In this case, the CPU is more likely to be the bottleneck.

NCache/NFS-baseline and NFS-original becomes more significant. When the request size is 8 KB, the NFS-NCache case shows a small peak in the NFS server's CPU utilization ratio, because the optimal number of NFS server daemons that achieves the best throughput varies for different request size.

We also ran the all-hit workload against the three NFS server configurations. The results are shown in Figure 5. Under this workload, no access to the storage server is required. So the NFS server's throughput is limited by either the CPU or the network link. By installing one NIC in the NFS server, we can compare the three NFS server configurations when the network link is the bottleneck. By installing an additional NIC in the NFS server, we can compare the three NFS server configurations when the CPU is the bottleneck.

Figure 5(a) shows the NFS server CPU utilization for the three NFS server configurations when the bottleneck is the network link. As in the all-miss case, the NFS server's CPU utilization saturates throughout for NFS-original, but decreases with request size for NFS-NCache and NFS-baseline. When the request size is smaller than 32 KB, the saving in CPU utilization of NFS-NCache over NFS-original is up to 42%, comparable with 49% of NFS-baseline over NFS-original. However, because the network link is already saturated, reduction in NFS server CPU utilization does not lead to additional throughput gain.

Figure 5(b) shows the throughput of the three NFS server configurations when the bottleneck is the NFS server CPU. The throughput of NFS-original grows when the request size increases from 4 KB to 8 KB, but completely saturates after that. In contrast, the throughput of NFS-NCache grows continuously with the request size – at the request size of 32 KB, the throughput of NFS-NCache is better than that of NFS-original by 92%. There are two reasons for this significant performance gain. First, under the all-hit workload, data copying is likely to be the dominant performance cost since no disk access is required, and therefore *NCache*'s ability to eliminate redundant data copying can have huge impact. Second, because the network link is no longer the bottleneck, saving in the CPU utilization of NFS-NCache over NFS-original is directly translated into throughput gain. For example, when the request size is 32 KB, the observed difference in CPU utilization between NFS-NCache and NFS-original is around 52%, as shown in Figure 5(a), which is converted to the throughput difference between NFS-NCache and NFS-original, 82 MB/sec, which is 92% gain over NFS-original's throughput, as shown in Figure 5(b). NFS-baseline gives throughput improvement up to 143% over NFS-original.

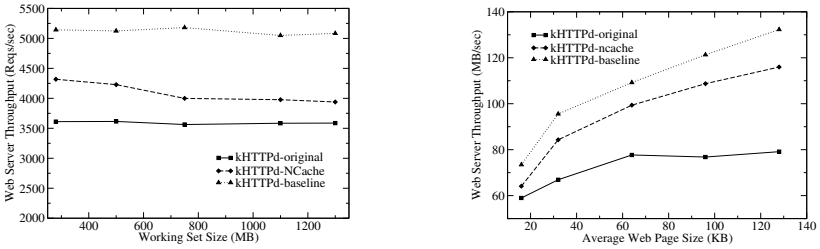


Figure 6. Throughput of kHTTPd. (a) Performance evaluation of kHTTPd with SPECweb99 benchmarks. (b) Performance evaluation of kHTTPd with varying request size.

The performance difference between NFS-baseline and NFS-NCache is due to certain additional operations in *NCache*, such as packet substitution and *NCache* buffer management.

We also measured the throughput of the three NFS server configurations using the SPECsfs benchmarks and varying the percentage of NFS requests that access regular data (as opposed to metadata). Figure 7 shows the resulting throughput measurements in terms of operations per second. As expected, NFS-NCache consistently performs better than NFS-original. When the percentage of regular data requests is 30%, NFS-NCache can sustain a throughput that is 16.3% higher than that of NFS-original. When the percentage of regular data requests grows to 75%, NFS-NCache can sustain a throughput that is 18.6% higher than that of NFS-original. The absolute throughput gain of NFS-NCache over NFS-original is not as significant because *NCache* does not offer any performance improvements for metadata operations or small regular data operations, which are dominant in the SPECsfs workload. Accordingly, the throughput gain of NFS-NCache over NFS-original increases as the percentage of regular data requests increases. Similar with the result shown in Figure 5, the management overhead and packet substitution overhead incurred by *NCache* contribute to the performance disparity between NFS-baseline and NFS-NCache.

5.5 Throughput of kHTTPd

We measured the throughput of the three web sever configurations under the SPECweb99 benchmark and the all-hit workload. The results using the SPECweb99 benchmark with various working set sizes are shown in Figure 6(a). kHTTPd-NCache consistently shows between 10% to 20% throughput improvement over kHTTPd-original. Due to *NCache*'s overhead, this improvement is not as large as 40% improvement of kHTTPd-baseline over kHTTPd-original. As the working set size increases, the throughput drops for all three configurations, because the buffer cache hit ratio decreases. But the throughput degradation of kHTTPd-NCache is particularly noticeable, especially when the working set size increases from 500 MB to 750 MB. This is because *NCache* requires additional memory to store the metadata of network-centric cache, and reduces the effective amount of memory available for data caching leading to higher cache miss ratio.

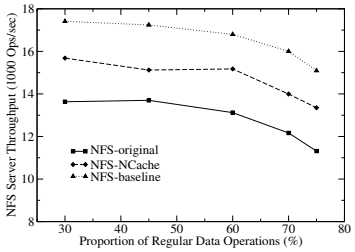


Figure 7. Performance evaluation of NFS server with SPECsfs benchmarks. The file system size is set at 2 GB and the total accessed file set is chosen to be 10% of the file system.

Figure 6(b) shows the performance comparison of kHTTPd for different request sizes under the all-hit workload, where the request sizes were varied from 16 KB to 128 KB. The overall performance improvement of kHTTPd increases with increase in request size. This is because of reduction in total number of requests reducing the aggregate per request overhead. For small request sizes the performance improvement of kHTTPd-NCache is around 8% and increases up to 47% for large request size of 128 KB. The overheads incurred by kHTTPd-NCache, in comparison to kHTTPd-baseline, are typically data substitution overhead and buffer management overhead.[1]

In general, the performance gain of kHTTPd-NCache over kHTTPd-original is smaller than that of NFS-NCache over NFS-original as in Figure 5(b) for two reasons. First, the data copying overhead per request in kHTTPd is inherently lower than that in NFS server, as

[1]Detailed measurement of overhead *NCache* may incur can be found in http://www.ecsl.cs.sunysb.edu/tr/TR177.ps.gz

shown in the read path column of Table 2, because of the sendfile interface. Second, the per-packet overhead of HTTP is higher than that of NFS because HTTP runs on TCP and NFS runs on UDP in our experiments.

6 Conclusion

Pass-through servers are servers that relay data packets between external entities without interpreting them. Because they don't need to interpret packets, they should be able to forward packets with minimal data coping, just like standard IP routers. An NFS server backed by an iSCSI storage server is an example of pass-through server. Unfortunately, standard pass-through servers still incur significant data copying overhead, mainly because they are implemented in a modular fashion and each module tends to have a different internal data representation.

This paper advocates a network-centric cache organization for pass-through servers that reduces the data copying overhead without drastic modifications. The result is a significant reduction in CPU utilization because of elimination of unnecessary data copying, which in turn leads to up to 92% improvement in throughput for NFS server and up to 47% for Web server. Comparison with modified servers that emulate the "ideal" zero-copy solution shows that the additional overhead of *NCache* is well within acceptable range. Although minimizing data copying is a well-known technique to improve system throughput, the proposed network-centric caching approach is novel in that it is able to substitute logical copying for physical copying whenever possible by exploiting the fact that pass-through servers do not interpret payloads, and closely approximates zero data copying without requiring wholesale changes to their existing implementations.

Network-centric file cache is rooted in the idea that modern file servers should be organized around networking rather than around computation because their main job is to move rather than process data. It is possible to take this idea one step further by organizing disk-resident data in a network-ready format, e.g., storing data on disk in an SSL-ready format, so that even non-pass-through file servers can also benefit from network-centric caching. We are currently exploring how to extend the *NCache* prototype to support this generalization.

Acknowledgment

We would like to thank the anonymous referees for their helpful comments and Ningning Zhu for her ATP tool. This research is supported by NSF awards ACI-0234281, CCF-0342556, SCI-0401777, CNS-0410694 and CNS-0435373 as well as fundings from Computer Associates Inc., New York State Center of Advanced Technology in Sensors, National Institute of Standards and Technologies, Siemens, and Rether Networks Inc.

References

[1] *Internet Small Computer Systems Interface (iSCSI)*. The Internet Engineering Task Force.

[2] *iSCSI reference implementation*. InterOperability Laboratory.

[3] *kHTTPd – Linux HTTP Accelerator*.

[4] *Remote Direct Memory Access*. RDMA Consortium.

[5] *SPECweb99 Benchmark*. Standard Performance Evaluation Corporation.

[6] *System File Server Benchmark SPEC SFS97_R1 V3.0*. Standard Performance Evaluation Corporation.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM (1)*, pages 126–134, 1999.

[8] J. C. Brustoloni and P. Steenkiste. Effects of Buffering Semantics of I/O Performance. In *Symposium on Operating Systems Principles*, October 1996.

[9] B.S.Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. Technical Report HPL-2001-8, HP Labs, January 2001.

[10] H. K. J. Chu. Zero-Copy TCP in Solaris. In *USENIX Annual Technical Conference*, pages 253–264, January 1996.

[11] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *USENIX Conference on File and Storage Technologies*, March 2003.

[12] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating Systems Principles*, 1995.

[14] G. Gibson, D. Nagle, K. Amiri, J. Butler, F. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[15] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.

[16] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Operating Systems Design and Implementation*, pages 153–167, 1996.

[17] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[18] G. Peng, S. Sharma, and T. Chiueh. A Case for Network-Centric Buffer Cache Organization. In *Symposium on High Performance Interconnects*, August 2003.

[19] K. Yocum and J. Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Annual USENIX Technical Conference,*, pages 305–318, June 2001.

[20] N. Zhu, J. Chen, T. Chiueh, and D. Ellard. *An NFS Trace Player for File System Evaluation*. ECSL, Stony Brook Tech. Report, 2003.