A New Buffer Cache Design Exploiting both Temporal and Content Localities

Jin Ren and Qing Yang
Dept of Electrical, Computer, and Biomedical Engineering
University of Rhode Island, Kingston, RI
Email: {rjin, qyang}@ele.uri.edu

Abstract: *This paper presents a **L**east **P**opularly **U**sed buffer cache algorithm to exploit both temporal locality and content locality of I/O requests. Popular data blocks are selected as reference blocks that are not only accessed frequently but also identical or similar in content to other blocks that are being accessed. Fast delta compression and decompression are used to satisfy as many I/O requests as possible using the popular reference blocks together with small deltas inside the buffer cache. The popularity of a reference block is calculated based on the statistical analysis of data contents and access frequency. A prototype LPU has been implemented as a new cache layer for Kernel Virtual Machine (KVM) on Linux system. Experimental results show LPU is effective for a variety of workloads with the maximum speed up of over 300% compared with LRU.*

1 Introduction

While the capacity of disk drives grows rapidly, their electromechanical parts have held down the improvement of their performance. Buffer cache plays a critical role in modern operating systems bridging the gap between disk drive and main memory. Most existing storage cache algorithms, such as Least Recently Used (LRU), are based on the analysis of the sequence of disk addresses of I/O requests. Content locality has not been one of the major considerations to traditional cache designs.

Content locality refers to the fact that many data blocks in disk storage share similar or even same content. Such content locality has been well known by the storage community and has been exploited in the design of backup storage and data replications. Content Addressable Storage (CAS) [1,2] and data de-duplication [3,4] are two examples of such data storage system designs. CAS is the technique that stores and retrieves information based on its content, not its location. Data de-duplication techniques try to find identical blocks during data backups or replications so that only one copy is stored or transferred to save storage space or network bandwidth [5]. Research in both academia [6,7,8,9,10] and industry [4] has shown the effectiveness of such designs indicating the existence of strong content locality in data storage.

The recent advancement of machine virtualization [11] has made content locality even stronger. In such a virtual machine environment, each guest virtual machine is allocated a virtual disk image for guest OS, application codes, and data. As a result, many data blocks in such data storage share similar or same content. Liguori and Hensbergen have found high redundancy (79%-96%) of virtual disk images among different installations of Fedora 9 with Office, SDK, and Web [12]. For better resource consolidation, lower power consumption, easy management, and strong process isolation, data center servers in future cloud computing will have tens and even hundreds of virtual machines to provide service to thousands of computers and mobile devices [13] resulting in much more data redundancy than that running single OS. This increasing data redundancy in virtual machines has been successfully exploited recently by Difference Engine [8] and Satori [10] to reduce memory consumption by means of page coalescing.

The objective of this paper is to exploit the ever increasing content locality in buffer cache design to minimize disk I/O operations that are still the main bottleneck in computer systems. The idea is to dynamically identify the most popular data blocks that not only have the most access frequency and recency but also contain the information contents that are shared or resembled by other blocks being accessed. Such popular blocks are called *reference blocks* because their content may be exactly the same as or similar to other active blocks with very small differences. By keeping such popular reference blocks and small deltas from other active blocks, most disk I/O operations are served by combining a reference block with the corresponding delta inside the buffer cache as opposed to going to the slow disk. A new cache replacement algorithm, Least Popularly Used (LPU), is developed based on the statistical analysis of frequency spectrum of both I/O addresses and I/O contents. LPU increases hit ratio of buffer cache greatly for the same cache size because of the strong content locality.

A prototype LPU has been implemented on Linux platform as a new application level cache for Kernel Virtual Machine (KVM) [14]. A special metadata structure has been developed to effectively keep track of popular reference blocks. The popularity of a reference block is determined based on two orthogonal parameters, reference frequency and content signatures. Using the prototype, we have carried out extensive experiments to measure the performance of LPU as compared to LRU cache and data de-duplication. Experimental results have shown superb advantages of LPU over existing buffer cache algorithms.

We define block popularity and present the replacement algorithm in the next section. Section 3 presents the implementation of LPU on KVM. Experimental settings are discussed in Section 4, followed by related work in Section 5. Section 6 concludes the paper.

2 Overview of LPU Design

The key to LPU is how to find blocks that are both accessed frequently and resembled by as many other blocks as possible. This section presents the algorithm to calculate popularity and the design of LPU.

1063-6927/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDCS.2010.26

2.1 Popularity

In order to allow the buffer cache to be managed based on popularity, we need to determine and keep track of both access frequency and content signature of a cached block. For this purpose, each cache block is divided into *S* sub-blocks. A sub-signature is calculated for each of the *S* sub-blocks. A special two dimensional array, called Heatmap, is maintained in our LPU buffer cache design. The Heatmap has *S* rows and *Vs* columns, where *Vs* is the total number of possible signature values for a sub-block. For example, if the sub-signature is 8 bits, Vs = 256. Each entry in the Heatmap keeps a popularity value that is defined as the number of accesses of the sub-block matching the corresponding signature value. As an example, consider Figure 1 that shows the 8×256 Heatmap. In this example, each data block is divided into 8 sub-blocks and has 8 corresponding signature values. When a block is accessed with sub-block signatures being 55, 00, and so on as shown in Figure 1, the popularity value corresponding to column number 55 of the 1st row is incremented. Similarly, column number 0 of second row is also incremented. In this way, Heatmap keeps popularity values of all sub-signatures of sub-blocks.

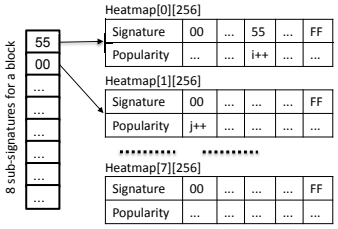

Figure 1.Sub-signatures and the Heatmap.

To illustrate how Heatmap is organized and maintained as I/O requests are issued, consider a simple example where each cache block is divided into 2 sub-blocks and each sub-signature has only four possible values, i.e. Vs = 4. The Heatmap of this example is shown in Table 1 for a sequence of I/O requests accessing data blocks at addresses *LBA1, LBA2, LBA3*, and *LBA4*, respectively. Assume that all possible contents of sub-blocks are *A, B, C*, and *D* and their corresponding signatures are *a, b, c*, and *d,* respectively. The Heatmap in this case contains 2 rows corresponding to 2 sub-blocks of each data block and 4 columns corresponding to 4 possible signature values. As shown in this table, all entries of the Heatmap are initialized to {(0, 0, 0, 0), (0, 0, 0, 0)}. Whenever a block is accessed, the popularities of corresponding sub-signatures in the Heatmap are incremented. For instance, the first block has logical block address (*LBA*) of *LBA1* with content (*A, B*) and signatures (*a, b*). As a result of the I/O request, two popularity values in the Heatmap are incremented corresponding to the two sub-signatures, and the Heatmap becomes {(1, 0, 0, 0), (0, 1, 0, 0)} as shown in Table 1. After 4 requests, the Heatmap becomes {(2, 1, 1, 0), (0, 1, 0, 3)}.

In our current design, the size of a cache block is fixed at 4 KB. Each 4KB block is divided into 8 512-bytes sub-blocks resulting in 8 sub-signatures to represent the content of a block. Unlike many existing content addressable storage systems, each sub-signature is 1 byte representing the sum of 4 bytes in a sub-block at offsets 0, 16, 32, and 64, respectively, instead of the hash value of the sub-block. In this way, the computation overhead is substantially reduced. What is more important is that our objective is to find the similarity rather than identical blocks. Hashing is efficient to detect identical blocks, but it also lowers the chance of finding similarity because a single byte change results in a totally different hash value. Therefore, additional computation of hashing does not help in finding more similarities. On the other hand**,** LPU calculates the *sha1* value of the whole block to determine identical blocks.

I/O sequence	Content	Signature	Heatmap[0] a b c d	Heatmap[1] a b c d
		Initialized	0 0 0 0	0 0 0 0
LBA1	A B	a b	1 0 0 0	0 1 0 0
LBA2	C D	c d	1 0 1 0	0 1 0 1
LBA3	A D	a d	2 0 1 0	0 1 0 2
LBA4	B D	b d	2 1 1 0	0 1 0 3

Table 1. The buildup of heatmap. Each block has 2 sub-blocks represented by 2 sub-signatures each having 4 possible values Vs=4.

With 4KB blocks, 512B sub-blocks, and 8 bits sub-signature for each sub-block, we have Heatmap with 8 rows corresponding to 8 sub-blocks and 256 columns to hold all possible signatures that a sub-block can have. Each time a block is read or written, its 8 1-byte sub-signatures are retrieved and the 8 popularity values of corresponding entries in the Heatmap are increased by one. This frequency spectrum of content is the key difference between LPU and other cache algorithms. It is able to capture both the temporal locality and the content locality. If a block of the same address is accessed twice, the increase of corresponding popularity value in the Heatmap reflects the temporal locality. On the other hand, if two similar blocks with different addresses are accessed once each, the Heatmap can catch the content locality since the popularity values are incremented at entries that have matched signatures.

In order to capture the content locality dynamically at runtime, LPU scans cached blocks after certain number of I/O requests. This number of I/O requests defines a scanning window. Within the scanning window, LPU examines the popularity values in the Heatmap and chooses most popular blocks as *reference blocks*. A reference block should contain the most frequently accessed sub-blocks so that many frequently accessed blocks share same contents with it. Our objective is to select a reference block in such a way to maximize the number of remaining blocks that have small differences from the reference block. In this way, more I/O requests can be served by combining the reference block with small deltas.

LBAs	Block	Popularity	LRU	Reference			
				A B	C D	A D	B D
LBA1	A B	2+1 = 3	A B	A B	A B	_ B	A B
LBA2	C D	1+3 = 4	C D	C D	C D	C _	C _
LBA3	A D	2+3 = 5	A D	_ D	A _	A D	A _
LBA4	B D	1+3 = 4	B D	B D	B _	B _	B D
		Cache space	4	3.5	3	2.5	3

Table 2. Selection of a reference block. The popularities of all blocks are calculated according to the Heatmap of Table 1.

Table 2 shows the calculation of popularity values and the cache space consumption using different choices of reference block for the example of Table 1. The

popularity value of a data block is the sum of all its sub-block popularity values in the Heatmap. As shown in the table, the most popular block here is the data block at address *LBA3* with content (*A, D*) and its popularity value is 5. Therefore, block (*A, D*) should be chosen as the reference block. Once the reference block is selected, LPU uses delta-coding to eliminate data redundancy. The result shows that using the most popular block (*A, D*) as the reference, cache space usage is minimum, about 2.5 cache blocks assuming perfect delta encoding. Without considering content locality, a simple LRU would need 4 cache blocks to keep the same hit ratio. The saved space can be used by LPU to cache more data. Figure 2 shows the data layout after selecting block (*A, D*) as the reference block.

2.2 Cache Management

LPU divides cache into 3 parts as shown in Figure 2: virtual block list, data blocks, and delta blocks. The virtual block list, referred to as *LPU queue*, stores all the information of cached disk blocks with each entry containing the address, the signature, the pointer to the reference block, and the pointer to delta blocks for the corresponding cached block. However, data is not stored in the LPU queue allowing a large number of virtual blocks to be managed as an ***LRU*** queue. The data pointer of a virtual block is NULL if the disk block represented by this virtual block has been evicted. The delta blocks are managed in 64-bytes small blocks. A virtual block can have one or more delta blocks due to (i) this virtual block refers to a reference block; (ii) this virtual block is a reference block and has been written since it was selected as a reference. As long as there are sufficient delta blocks, a virtual block can always keep its delta block even its data block is evicted. Our current implementation assumes a fixed portion of LPU cache for delta blocks and making the size of deltas adaptive to workloads is our future research work.

A virtual block can be one of three different types: *reference block, associate block, or independent block*. An associate block is a virtual block that is associated with a reference block together with a delta that is the difference between the content of the associate block and the reference block. An independent block is a virtual block that has no associated reference block in the cache.

When a disk block is accessed the first time and brought in the cache, a virtual block and a data block are allocated to cache it. Before this virtual block is selected as a reference block or associate block, it is an independent block so that data is read from or written to its data block. Its signature is updated upon every write request. Once it is selected as a reference block or associate block, one or more delta blocks are allocated for this virtual block. A write request to a virtual block that is an associate block needs to read its reference block first, calculate the difference using delta-coding, and write the difference to the delta block. Read request to an associate block combines its delta and the reference block to obtain its data. As a result, a reference block is always ahead of its associate blocks in the LPU queue because accesses to its associate blocks also need to access the reference block. Similarly, write requests to a reference block need update its delta blocks. But the signature of the block does not change since its data is being referred. Read requests to the changed reference block needs combine with its delta block.

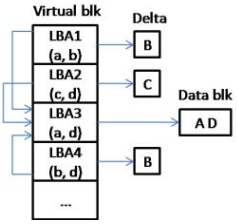

Figure 2. The data layout of LPU buffer cache.

To manage cached data blocks described above, we need to consider 3 kinds of replacements. The first is virtual block replacement when there is no available virtual block. LPU searches from the end of the LPU queue and replaces the first non-reference block. This kind of replacement rarely happens because the number of virtual blocks is large. The second is data block replacement. LPU searches from the end of LPU queue and replaces the first data block. The data block of a reference block also can be evicted indicating that the reference block and its associate blocks have not been accessed for a long time. The third is delta replacement which leads to virtual block replacement. LPU searches from the end of the LPU queue, replaces the first block that has delta and is not a reference. Both the data block and the delta block are released. To save delta block space, a block disconnects the link to its reference if this block has been changed a lot. The criterion in current implementation is whether the delta is larger than 768 bytes which will be discussed later. Once the link is disconnected, all read and write requests go to its data block directly.

3 Implementation

We have implemented LPU on Linux platform with about 5,000 lines of code. LRU and data de-duplication are also implemented for the purpose of performance comparison with LPU.

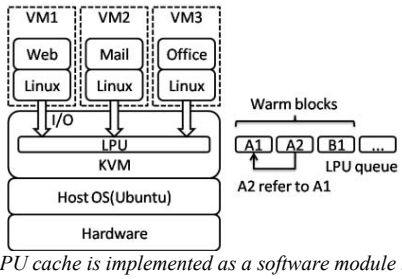

Figure 3. LPU cache is implemented as a software module inside KVM.

3.1 System architecture

Our LPU is designed as a cache layer within KVM. As shown in Figure 3, it works at application level providing a block level cache to guest operating systems making use of the virtual disk driver of KVM. The I/O function of KVM depends on QEMU [15] that is able to emulate many virtual devices including virtual disk drive. In the guest operating system, the kernel driver for the virtual disk receives I/O requests and sends them to the

QEMU host kernel driver in the host operating system. The QEMU host kernel driver passes these I/Os to QEMU application module to perform actual disk I/O. Upon completion of an I/O, the QEMU application module informs QEMU host kernel module, and then the driver in guest operating system, and finally data is returned to the guest operating system. LPU provides disk operation APIs such as open/close, read/write to be called by the QEMU application module. Caching is done within the LPU module and is transparent to QEMU. Working in the application module of KVM also enables LPU to collect more information of I/O requests such as filename and virtual machine id which are not available at kernel block level.

The I/O requests captured by LPU are identified as virtual blocks and managed by the LPU queue. The LPU queue is scanned after every 50,000 I/O requests to select reference blocks based on their popularities. Popularity value alone is not sufficient for determining reference blocks because some highly popular blocks are very similar to each other. For example, "00" is observed to be the most frequently used signature. If the signature of the first block is "00 00 00 00 00 00 00 00", the signature of the second one is likely to be "80 00 00 00 00 00 00 00". For good performance, the second block should refer to the first block instead of being selected as a reference block. Therefore, LPU picks up 20,000 warm blocks in the LPU queue and groups them to ensure reference blocks are sufficiently different from each other so that we can have as many associate blocks as possible. For this purpose, the signature of a candidate reference block is compared with the reference blocks that have been selected. The candidate reference block is discarded if 7 or more sub-block signatures match those of any one of other reference blocks. Our experimental evaluation shows this method works very effectively in selecting reference blocks. The associate-to-reference delta compression ratio with this selection method ranges from 20:1 to 30:1 on average.

3.2 Caching in KVM

To emulate a virtual IDE drive, QEMU monitors the IDE command port, for example 0x170-0x177, to get IDE commands sent by the guest virtual machine. The read and write IDE commands are handled by registered callback functions and are finally translated into read and write requests to the virtual disk image file in the host machine. The entire processing of an I/O request is at application level which makes it possible to develop a cache layer in QEMU. However QEMU does not provide storage cache because data sharing in application level is not as straight forward as in kernel level. Several issues arise to share data and exchange message between virtual machines and the LPU cache.

The first issue is how to share data between virtual machines. LPU uses mapping files to build a shared buffer for all virtual machines. Each virtual machine opens the same temporary file, e.g. "/tmp/vmshare", and call mmap() function to establish a mapping between its memory address and the temporary file. The shared data stores both cache block and the mutex to protect the consistency of shared data. Since each process in Linux has its own memory space so data cannot be transferred by passing the pointer from one virtual machine to another. The buffer Id replaces the use of pointer in most cases especially for organizing queue data structure.

The second issue is asynchronous I/O. QEMU uses both synchronous I/O and asynchronous I/O to read and write data. We need to implement all I/O interfaces to replace the corresponding system calls. Table 3 lists the APIs provided by LPU. For synchronous I/Os, QEMU calls open_share() to get the handle of target file. Read and write requests are sent to read_share() and write_share(), respectively, and the results are returned at the end of the calls. Asynchronous I/O aio_read_share() and aio_write_share() return immediately without finishing the request. The request is queued to a separate aio thread to be processed later. The caller process can either query the status using aio_error_share() or wait for the completion signal. We need to emulate the signal of what kernel driver does after finishing a request. The aio thread setups the signal mask to be the same as QEMU and sends out SIGUSR2 on which QEMU is waiting.

API name	Description
open_share	Open file
close_share	Close file handle
read_share	Synchronous read data
write_share	Synchronous write data
aio_read_share	Asynchronous read data
aio_write_share	Asynchronous write data
aio_error_share	Asynchronous query status
aio_return_share	Asynchronous complete request
aio_cancel_share	Asynchronous cancel request

Table 3. I/O APIs provided by LPU

The third issue is context switching. This issue occurs when flushing evicted block and accessing reference blocks. For example, virtual machine ***A*** needs to replace *block_b* in the cache, and *block_b* belongs to *file_b* which is opened by virtual machine ***B***. Though the handle of the *file_b* is stored in the cache table, virtual machine ***A*** cannot use that handle to write *block_b* back because file handle is not allowed to be shared between processes. So virtual machine ***A*** has to open *file_b* itself to write *file_b* back. To reduce the unnecessary file open/close operations, LPU maintains a small temporary buffer for blocks that are loaded and evicted by different processes. When the temporary buffer is full, the current process opens related files to flush all temporary blocks.

The last but not the least issue is the impact of Linux buffer cache. File system cache and buffer cache make it difficult to evaluate LPU accurately compared with other cache algorithms. To deal with this issue, we first limit the size of allocated RAM for each virtual machine to 192MB to meet the minimum requirement for all benchmarks so that little free RAM is left for Linux buffer cache that makes use of all available free RAM. Secondly, LPU bypasses the host system cache by specifying O_DIRECT flag while opening files and aligns the buffer to 512 bytes for read and write request. Therefore, we can evaluate LPU as a second level storage cache without any underlying cache.

3.3 Scanning and Coalescing

Each time when the LPU queue is scanned to select reference block, 20,000 warm blocks are picked up and divided into two groups: the reference blocks and the remaining warm blocks as the candidate for associate blocks. For each candidate associate block, LPU tries to find the most similar reference block. To make the search fast, an array, reference_index[8][256], is used to store pointers to reference blocks with matching sub-signature. For example, the pointers to the reference blocks which have the first sub-signature of "10" are stored in reference_index[0][10]. Each candidate associate block finds the maximum similarity between its signature and any of reference blocks using reference_index[8][256].

Two threshold values are used to connect an associate block with a reference block: similarity threshold and delta threshold. When the maximal similarity between a candidate associate block and a reference block exceeds the similarity threshold, zdelta compression [16] is performed for the candidate associate block. Then we examine the length of the delta result. If it is less than or equal to a delta threshold, the candidate associate block is selected to be associated with the reference block. Otherwise, it is false positive and the candidate associate block remains as an independent bock. The similarity threshold determines whether a candidate associate block needs to do delta-coding and the delta threshold determines how many eligible associate blocks can be found. The smaller the similarity threshold is, the more delta encodings are needed increasing the computation overhead. The smaller the delta threshold is, on the other hand, the less number of associate blocks can be found giving rise to more false positive. We use 7 for similarity threshold and 768 bytes as delta threshold in our current implementation based on our empirical observations. The average number of false positive is controlled to below 50 for each scan.

LPU does not allow cascading references, which increases the coding complexity and degrades performance. If an associate block is selected as a reference, it will be replaced using its reference block instead.

4 Experiments and Evaluations

4.1 Experiment Settings

Two server machines are used to characterize LPU's performance. The primary server, which hosts virtual machines, is a Dell PowerEdge T410 with 1.8GHz Xeon CPU, 2GB RAM, and 160G Dell SATA drive. A Dell Precision 690 with 1.6GHz Xeon CPU, 2GB RAM, and 400G Seagate SATA drive is the secondary server to generate client requests for benchmarks. The two servers are connected using a gigabit Ethernet switch. The host OS and virtual guest OS are both Ubuntu 8.10 server edition while the host OS has GUI needed to run KVM.

To be able to evaluate the performance of LPU, we need real world I/O workloads that have meaningful contents as well as access patterns similar to real applications. We have selected 7 standard benchmarks in our performance evaluation experiments as shown in Table 4. Each of the first 6 benchmarks runs with the same configuration on 5 virtual machines, and the Mixed workload runs 5 different benchmarks concurrently.

Abbrev.	Name	Description
RU	RUBiS	e-Commerce web server workload
TP	TPCC-UVA	Database server workload
SM	SPECmail2009	Mail server workload
SB	SPECwebBank	Online banking
SE	SPECwebEcommerce	Online store selling computers
SS	SPECwebSupport	Vendor support website
MX	Mixed	Heterogeneous workload

Table 4. Benchmarks used to evaluate LPU.

RUBiS is an auction site prototype which is similar to eBay [17]. It evaluates the performance of an e-commerce server simulating the auction operations such as selling, browsing and bidding. Each virtual machine has Apache, Mysql, PHP, and RUBiS client, and the database is initialized using the sample database provided by RUBiS. Each virtual machine is tested by 20 clients with 60 minutes running time. TPC-C is a benchmark to model the operations on real-time transactions. We us TPCC-UVA [18] on Postgres database with 2 warehouses, 10 clients, and 60 minutes measure time.

SPECmail2009 [19] measures the ability of a system to act as an enterprise mail server using the Internet standard protocols SMTP and IMAP4. We install Postfix as the SMTP service and Dovecot as the IMAP service. SPECmail2009 is configured to use 20 clients and 30 minutes measure time. SPECweb2009 [20] provides the capabilities of measuring both SSL and non-SSL request/response performance of a web server. One workload generator emulates the arrivals and activities of 10 clients to each virtual web server under test. Each virtual server is installed with Apache and PHP support. The secondary server works as a backend application and database server to communicate with each virtual server. The SPECwebEcommerce is designed to simulate a web server that sells computer systems which allows end user to search, browse, customize, and purchase product. The SPECwebSupport simulate the workload of a vendor's support web site. 10 clients are setup to test each virtual server for both SPECwebEcommerce and SPECweb-Support. The duration of each test is 30 minutes.

The Mixed benchmarks setup 5 virtual machines running RUBiS, TPC-C, SPECmail2009, SPECwebBank, and SPECwebEcommerce concurrently. It is used to evaluate LPU's effectiveness under heterogeneous workloads. We stop all 5 virtual machines after running for about 40 minutes.

For the purpose of performance comparison, we have implemented two baseline buffer cache algorithms. The first baseline system is the traditional LRU cache that manages the buffer cache using the least recently used algorithm. The second baseline system adds data de-duplication mechanism in the LRU buffer cache (De-Dup). In this De-Dup buffer cache system, only one of identical data blocks is cached in the buffer cache. Duplications are eliminated leaving only pointers to the cached identical block. Upon a write to one of the identical blocks, a copy is made in the buffer cache.

We run 5 virtual machines to generate a variety of I/O workloads following the benchmark specs. In addition to I/O rates, space efficiency and computational

overheads of LPU are also measured as compared with the LRU baseline and the De-Dup baseline.

4.2 Numerical Results and Evaluations

4.2.1 Performance Evaluations and Comparisons

Our first experiment is to measure the cache hit ratios of the three buffer cache systems: LRU cache, De-Dup cache, and LPU cache. During our experiments, we observed substantial improvements in terms of cache hit ratios. For example, when all the virtual machines are running SPECweb-Ecommerce benchmark with buffer cache size of 256MB, we measured the cache hit ratios to be 37%, 49%, and 57% for LRU, De-Dup, and LPU caches, respectively. LPU shows 54% and 16% performance improvements over LRU cache and De-Dup cache, respectively. Similar performance improvements have been observed for other benchmarks. When we run mixed workloads on 5 virtual machines with the same cache size (256MB), the measured hit ratios are 49%, 49%, and 80% for LRU, De-Dup, and LPU caches, respectively, indicating 63% improvement of LPU over LRU and De-Dup caches.

The primary reason why LPU cache shows such great hit ratio improvement over the De-Dup cache in this case is that when mixed workloads are running on the virtual machines, De-Dup is not able to find many identical cached data blocks. On the other hand, LPU can find many similar cached data blocks and is able to exploit such content locality much better than De-Dup cache by using delta compression and decompression to keep small deltas and reference blocks in the buffer cache.

Comparing hit ratios of LPU cache with those of LRU and De-Dup caches is not a fair performance comparison because LPU needs online compression and decompression. In other words, even though LPU has higher hit ratios than the other two baseline caches, it may not show better I/O performance because of the extra computation overheads. In order to provide a fair performance comparison, we concentrated our experiments on the ultimate I/O performance in terms of I/O rates of the benchmarks.

Figure 4 shows the I/O rate of the three buffer cache systems while running RUBiS benchmark. The cache size ranges from 64MB to 1GB. The size of the delta cache of LPU is fixed at *min{16MB, total_cache_size/8}*. It can be seen from Figure 4 that LPU performs much better than LRU. For the cache size of 512MB, LPU cache triples the performance of LRU cache, i.e. LPU performs 3 times as good as the LRU cache. This substantial performance improvement clearly indicates the effectiveness of the new cache algorithm. Compared to the De-Dup cache, LPU shows about 20% performance improvement for the same cache size. To understand the performance improvement better, we analyzed the I/O activities and found that a reference block has 3 associated blocks on average. Therefore, the effective virtual cache size is doubled if 1/3 of the cache blocks are reference blocks. This space saving goes higher as more virtual machines are added because of the increase of data redundancy.

Performance results of TPCC-UVA benchmark are shown in Figure 5 in terms of I/O rates. Because of large working set of the TPC-C benchmark, we observed that the performance difference among the three buffer cache systems increases as the cache size increases. The performance improvements of LPU over LRU and De-Dup caches are about 100% and 15%, respectively, for cache size of 1GB. We noticed that TPCC-UVA shows less performance improvement than RUBiS because of its high write I/O rate. The read to write ratio of TPCC-UVA is about 1:1 while that of RUBiS is about 3:1. Write requests affect LPU's performance negatively because of the following reasons. A write to a reference block is redirected to the reference's delta block. In the worst case, a reference block can occupy twice as much as the original block implying less effective use of cache space. Such writes may also cause other blocks to be evicted to make room for the additional delta block for the reference block. Another reason is that when the write request is to an associate block, LPU needs to do delta-coding test to see if the change is too large. Such test involves reading old data and calculating new delta. The new data might become an independent block if the new delta is larger than 768B. This process is relatively expensive compared with LRU.

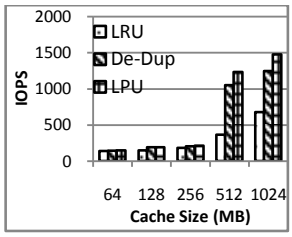

Figure 4. I/O rate of RUBiS.

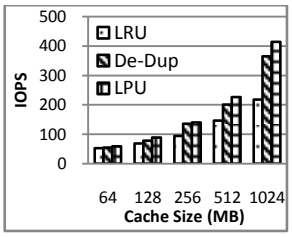

Figure 5. I/O rate of TPCC-UVA.

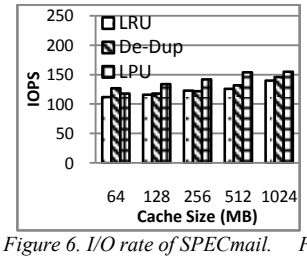

Figure 6. I/O rate of SPECmail.

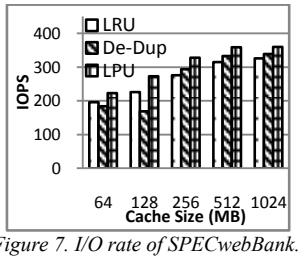

Figure 7. I/O rate of SPECwebBank.

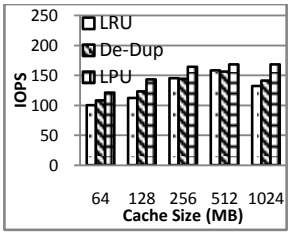

Figure 8. SPECwebEcommerce.

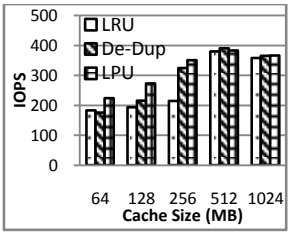

Figure 9. SPECwebSupport.

The SPECMail and SPECWeb performances are shown in Figures 6 to 9. While LPU cache performs constantly better than the other two, we noticed that the performance gains of these four benchmarks are not as high as the benchmarks discussed previously. To understand the reason of such performance differences, we recorded and analyzed the I/O sequence of each benchmark. The results reveals that LPU performs better for RUBiS and TPCC-UVA because 70% of the blocks

are accessed at least 2 times, while over 60% of the blocks of SPECmail, SPECwebBank, SPECwebEcommerce, and SPECwebSupport are accessed only once. The increased virtual cache size is not able to contribute to the hit ratio with such low repetitive access patterns.

From the performance results shown in Figures 4 to 9, we noticed that our LPU cache improves the performance of De-Dup cache by 10% to 20%. Our next experiment is to run the Mixed workloads and measured results are shown in Figure 10. In this case, the buffer cache contains data blocks from different applications and different data sets. It is interesting to observe in Figure 10 that the new LPU cache almost doubled the performance of both LRU cache and De-Dup cache for cache sizes of 128MB and 256MB. For small cache size of 64MB, the effective virtual cache size of LPU is much less than the working set of the benchmark resulting in limited performance gain. On the other hand, large cache of 1GB is able to hold most of data blocks making all three cache algorithms perform well. It is important to note the advantage of LPU cache in effectively managing the buffer cache with limited cache size that may not be large enough with respect to the working set of running applications. With ever increasing demands for larger data set of many applications, effective buffer cache designs that better exploit data locality under hardware constraints are clearly desirable. As a future work, we are currently looking at minimizing CPU overhead of LPU so that it performs better than traditional caches over a wide range of cache sizes and workload mixes. One possible solution is to offload some of the computations to the embedded CPU on a HBA card.

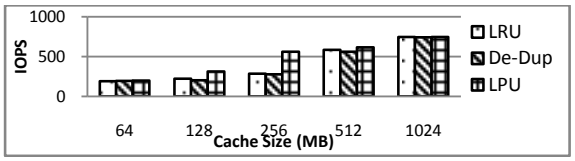

Figure 10. I/O rate of Mixed workloads.

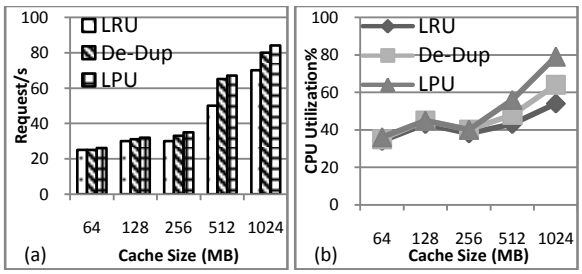

Figure 11. Application performance and CPU utilization of RUBiS.

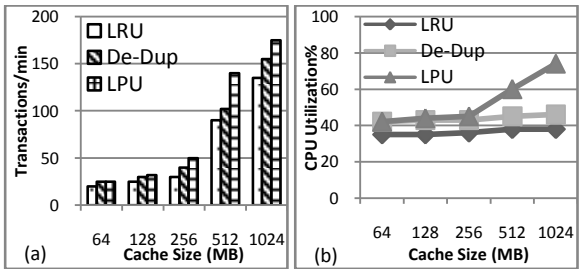

Figure 12. Application performance and CPU utilization of TPCC-UVA.

Figure 11(a) shows throughput comparisons from application view point. LPU shows 20% higher throughput than LRU, and 5% higher than De-Dup when cache size is 256MB or larger. Figure 12(a) shows the transaction rate measured using TPCC-UVA benchmark. When cache size is 256MB or larger, LPU is able to finish about 50% and 20% more transactions per minute than LRU and De-Dup, respectively. Figures 11(b) and 12(b) also show the corresponding CPU utilizations. The CPU utilization of LPU increases faster not only because of the computation overhead, but also because more instructions can be executed within the same time period because of reduced disk activities.

4.2.2 Coalescing Efficiency

The efficiency of a content based cache algorithm can be evaluated by looking at the number of blocks that are found to be similar. Figure 13 shows the ratio of reference blocks, associate blocks, and independent blocks for different benchmarks with cache size of 256MB. The figure shows that LPU can coalesce 30%-150% more blocks than De-Dup cache. For TPCC-UVA, SPECwebBank, SPECwebEcommerce, and SPECwebSupport, over 50% of the total blocks are coalesced by LPU. And the average delta size is less than 10MB.

More virtual machines have more data redundancy. Figure 14 shows the performance change as the number of virtual machines is increased running the RUBiS benchmark. It is clear from this figure that the advantages of LPU over LRU and De-Dup get larger as more virtual machines are spawned.

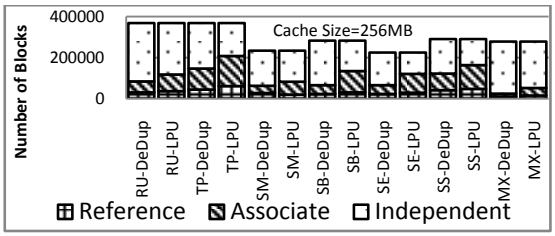

Figure 13. Comparison of the number of Coalesced blocks.

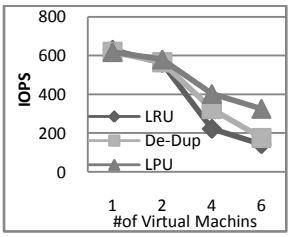

Figure14. I/O rate and #of machines.

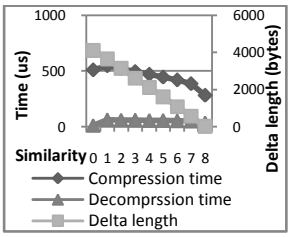

Figure15. Delta coding and similarity.

4.2.3 Overhead Evaluation

Virtual block list is the additional data structure required to implement LPU and hence all virtual blocks are the addition storage overhead. Each virtual block stores the addresses and signatures requiring total of 60 bytes. Therefore, 1GB cache needs 15MB virtual blocks if the cache block is 4KB giving rise to the space overhead of about 1.5%. All other metadata including the Heatmap is less than 100KB. Increasing the size of cache block can reduce the space overhead of metadata at the

cost of increasing computation cost for compressions and decompressions.

The additional computation overhead of LPU includes on-the-fly delta decompression and scanning overhead of each scanning window (after every 50,000 I/O requests). The on-the-fly delta decompression time was measured in our experiments to be less than 60 microseconds for each I/O access to an associate block. The computation cost of scanning after each scanning window is measured and shown in Figure 15. As shown in Figure 15, the time to do delta coding for two 4KB blocks is inversely proportional to the similarity between the source and the reference. It takes about 400 microseconds to compress if the similarity is 7. Among the 20,000 warm blocks, we can normally find 2,000 blocks as new reference blocks and about 2,000 other additional blocks can find reference blocks with similarity greater than or equal to 7. Therefore, the time to do delta coding in each scan is less than 1 second. One aspect of our future work is to use embedded system or GPU to offload the computation overhead from CPU.

5 Related Work

5.1 Buffer Cache

Most of existing caching algorithms focus on exploring the temporal locality. FBR [21] factors out locality out of frequency by not increasing the reference count for the cache blocks within a section that starts from the head of LRU queue. The length of the section needs to be tuned for different workloads. O'Neil et.al proposed LRU-K [22] which is able to discriminate frequently referenced and infrequently referenced pages by tracking the times of the last K references to popular database pages. Specifically, the authors discussed LRU-2 for K = 2 which takes the last two references into account. LRU-2 is able to remove cold blocks quickly but less effective for workload without strong frequency. 2Q [23] introduces one FIFO queue $A1_{in}$, and two LRU lists: $A1_{out}$ and A_m. A missed block is placed in $A1_{in}$, and its address is stored in $A1_{out}$ if it is replaced. If this block is referenced again and its address is found in $A1_{out}$, it is promoted to A_m where frequently accessed blocks are stored. By tuning the size of $A1_{in}$ and $A1_{out}$, 2Q can achieve similar performance as LRU-2 with constant overhead. LRFU [24] calculates Combined Recency and Frequency (CRF) for each block which covers the spectrum between LRU and LFU. LRFU needs to adjust the value of λ for different applications.

MQ [25] was proposed to improve the performance of second level cache. Multiple LRU queues: Q_0, …, Q_{m-1}, and a history queue Q_{out} are used to keep warm blocks in high level LRU queue. MQ is effective to catch the frequently accessed blocks that have long distance between two accesses. ARC [26] dynamically balances recency and frequency by maintaining two LRU lists, L_1 and L_2. L_1 stores pages that have been seen only once to capture recency, and L_2 stores pages that have been seen at least twice to capture frequency. ARC is adaptive because the target size p of T_1 which is the top part of L_1 is adaptive to workload. LIRS [27] provides the ability to deal with both strong and weak locality. Cache blocks are replaced according to their recent Inter-Reference Recency (IRR). Cache is divided into High Inter-Reference Recency (HIR) blocks and Low Inter-reference Recency (LIR) blocks. LIRS replace HIR blocks and a block in HIR gets promoted to LIR if its IRR is smaller than the maximum recency of all LIR blocks.

GCLOCK [28] is a generalized version of CLOCK which associates a counter with each page to carry its weight. CAR [29] maintains two lists which are similar to ARC and make CLOCK adaptive to different workloads. CLOCK-Pro [30] uses the reuse distance instead of recency which is similar to LIRS while keeping the overhead low. CLOCK-Pro has shown its effectiveness and been adopted by Linux kernel. Jiang et al proposed DULO [31] to filter random requests and pass sequential requests to disk considering both temporal and spatial locality.

These cache algorithms are all based on tracking and analyzing the addresses of I/O requests. LPU's approach is orthogonal to existing buffer cache algorithms. It is a new method to analyze I/O requests that can be used on any of the algorithms described above.

5.2 Content Redundancy Reduction

LPU was inspired by the research studies and applications which aim to reduce the redundancy in storage systems. REBL [32] leverages delta-encoding to eliminate redundancy within large collections of files at block level. Venti [1] is a network storage systems which divides files into fixed-sized blocks and uses SHA1 value as identifier to coalesce duplicated blocks to reduce the consumption of disk storage space. Rsync [33] accelerates file synchronization by sending only difference between two files. Suel et al. enhanced the performance of Rsync by a two-phases protocol which includes map construction phase and delta compression phase [34]. TAPER [35] is another hierarchical protocol with each phase operates over decreasing data granularity for efficient data replication. Different from Rsync, LBFS [5] is able to find difference among multiple files using content-defined chunks. CBBC [3] maintains a single copy of duplicated blocks as well as eliminates silent writes. Spring and Wetherall proposed a technique to identify repetitive information of network traffic such as web proxy caching [36]. Yang et al leverage content similarity for write requests to reduce storage space for recovery [37][38][39]. Delta encoding has been used to extend the redundancy reduction research to similar data blocks, not only for identical blocks [40][41].

Liguori and Hensbergen [12] built a service-oriented file system on Venti to reduce the duplicated data between multiple virtual machine disk images. Foundation [6] works under multiple virtual machines to lower the size requirement for archiving snapshots of virtual disk images with inexpensive hardware. Waldspurger described that VMware ESX Server developed a content-based page sharing technique for identical memory pages [7]. Difference Engine [8] successfully enlarges memory space and improves memory performance through in-core memory compression by leveraging page patching. Memory Buddies [9] detects and collocates similar virtual

machines on the same physical host to increase the potential of sharing identical pages. Satori [10] monitors identical blocks at block driver level, and passes sharing hint to modified guest operating system. A sharing entitlement mechanism encourages guest virtual machines to share memory with others. Clements et al proposed DEDE [42] to avoid cross-host communication when eliminating duplication in VMFS cluster file system.

LPU differs from these existing works in the new solution to select reference blocks and a new buffer cache management algorithm to optimize both the space usage and the performance of buffer cache. Several interesting works on data de-duplication have been reported [43,44,45] after our paper submission to improve I/O performance. These results further showed potential benefits of exploiting content locality of I/O operations.

6 Conclusions

In this paper, we have presented a novel buffer cache design that exploits both temporal locality and content locality of disk operations. A new least popularly used (LPU) replacement algorithm has been developed that manages the buffer cache based on the popularity of a data block. A data block is considered to be popular if it is not only accessed frequently but also contains contents that are same to or resembled by other accessed blocks. A set of most popular blocks in the cache are selected to be reference blocks that can serve a large number of I/O requests by combining with small deltas. A prototype LPU buffer cache has been designed and implemented on KVM virtual machine environment of Linux OS. Extensive experiments have been carried out using the prototype LPU cache to measure its performance as compared to traditional LRU cache. Experimental results have shown that for some workloads LPU improves cache performance greatly while for other workloads it has limited performance gain.

We are currently working on efficient ways of calculating signatures to catch data redundancy among data blocks even after bit-wise shift happens, which, we believe, will allow LPU to improve cache performance even further. Our future research also includes offloading some of the necessary computations of LPU to HBAs or other types of embedded boards.

Acknowledgements

This research is supported in part by National Science Foundation under Grants CCF-0811333, CPS-0931820 and Natural Science Foundation of China under grant NSFC-60736013. 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. The authors are grateful to anonymous referees for their comments that improve the quality of the paper.

References

[1] S. Quinlan, and S. Dorward, "Venti: a new approach to archival storage," *In Proc. of FAST'02*, 2002.

[2] P. Nath, B. Urgaonkar, and A. Sivasubramaniam, "Evaluating the Usefulness of Content Addressable Storage for High-Performance Data Intensive Applications," In *Proc. of HPDC'08,* 2008.

[3] C. B. Morrey III and D. Grunwald, "Content-based block caching," *In Proc. of MSST'06*, May 2006.

[4] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," In *Proc. of FAST'08*, 2008.

[5] A Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," In *Proc. of SOSP'01*, 2001, pp 174–187.

[6] S. Rhea, R. Cox, and A. Pesterev, "Fast, inexpensive content-addressed storage in foundation," In *Proc. of the USENIX Annual Technical Conference*, 2008.

[7] C. A. Waldspurger, "Memory resource management in VMware ESX server," In *Proc. of the 5th OSDI, 2002.*

[8] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," In *Proc. of OSDI 2008*, Dec. 2008.

[9] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. Corner, "Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers," In *Proc. of VEE'09*, 2009.

[10] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," In *Proc. of USENIX Technical Conference*, 2009.

[11] M. Rosenblum, "The Reincarnation of Virtual Machines," *ACM Queue,* Vol 2, Issue 5, July 2004.

[12] A. Liguori and E. V. Hensbergen, "Experiences with content addressable storage and virtual disks," In *Proc. of WIOV '08*, Dec. 2008.

[13] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinsi, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," http://bishop.camp.clarkson.edu/Cloud_Computing_Papers/berkely-abovetheclouds.pdf, 2009.

[14] Kernel Virtual Machine, http://www.linux-kvm.org

[15]F. Bellard, "QEMU, a fast and portable dynamic translator," In *Proc. of the USENIX Ann Tech Conf*, 2005.

[16] D. Trendafilov, N. Memon, and T. Suel, "zdelta: a simple delta compression tool," Technical Report TR-CIS-2002-02, Polytechnic University, June 2002.

[17] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of EJB applications," In *Proceedings of the 17th ACMConference on Object-oriented programming, systems, languages, and applications*, 2002.

[18] J. Piernas, T. Cortes and J. M. García, "tpcc-uva: A free, open-source implementation of the TPC-C Benchmark," http://www.infor.uva.es/~diego/tpcc-uva.html, 2005.

[19] SPEC OSG Mailserver Subcommittee, "SPECmail-2009 Benchmark Architecture White Paper," http://www.spec.org/mail2009/docs/designdocument. pdf, December 2008.

[20] SPEC OSG Web Subcommittee, "SPECweb2009 Introduction and Setup Overview," http://www.spec.org/web2009/docs/SPECweb2009_Setup.pdf, October 2009.

[21] J. Robinson and M. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," In *SIGMETRICS-90*, 1990.

[22] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm For Database Disk Buffering," In *Proc. of SIGMOD-93*, 1993.

[23] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," In *VLDB-94*, 1994.

[24] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho and C. S. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," *Proc. of ACM SIGMETRICS,* May 1999, pp. 134-143.

[25] Y. Zhou, Z. Chen and K. Li. "Second-Level Buffer Cache Management," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 7, July, 2004.

[26] N. Megiddo, D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proc. of FAST '03*, March 2003, pp. 115-130.

[27] S. Jiang and X. Zhang, "LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. of SIGMETRICS '02*, June 2002.

[28] V. F. Nicola, A. Dan, and D.M. Dias, "Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing," *Proc. of 1992 ACMSIGMETRICS Conference*, June 1992, pp. 35-46.

[29] S. Bansal and D.Modha, "CAR: Clock with Adaptive Replacement," *Proc. of 3rd FAST,* March, 2004.

[30] S. Jiang, F. Chen and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," *Proc. of USENIX Annual Technical Conference*, 2005.

[31] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality," In *Proc. of FAST'05, 2005*.

[32] P. Kulkarni, F. Douglis, J. Lavoie, and J. M. Tracey, "*Redundancy* elimination within large collections of files," In *Proc. of the USENIX Annual Technical Conference*, 2004.

[33] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization*, PhD thesis, Australian National University, 1999.

[34] T. Suel , P. Noel, and D. Trendafilov, "Improved File Synchronization Techniques for Maintaining Large Replicated Collections over Slow Networks," In *Proc. of the 20th Int'l Conf. on Data Eng*, 2004.

[35] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replicas," In *Proc of FAST'05,* 2005.

[36] N. T. Spring and D. Wetherall, "A protocol independent technique for eliminating redundant network traffic," In *Proceedings of ACM SIGCOMM*, August 2000.

[37] Q. Yang, W. Xiao, and J. Ren, "TRAP-Array: A Disk Array Architecture Providing Timely Recovery to Any Point-in-time," In *Proc. of ISCA'06*, 2006.

[38] Qing Yang, Weijun Xiao, and Jin Ren "PRINS: Optimizing Performance of Reliable Internet Storages," In *Proc. of ICDCS'06*. 2006.

[39] Weijun Xiao and Qing Yang, "Can We Really Recover Data If *Storage* Subsystem Fails?" In *Proc. of ICDCS'08*, 2008.

[40] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection," In *Proc. of USENIX Technical Conference*, June 2003.

[41] Z. Ouyang, N. Memon, T. Suel, and D. Trendafilov, "Cluster-based delta compression of a collection of files," In *International Conference on WISE*, December 2002.

[42] A. T. Clements, I. Ahmad, M. Vilayannur, and J Li, "Decentralized Deduplication in SAN Cluster File Systems," In *Proc. of the USENIX Annual Technical Conference*, 2009.

[43] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Całkowski, C. Dubnicki, and A. Bohra "HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System," In *Proc. of FAST'10*, 2010.

[44] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal Content Defined Chunking for Backup Stream," In *Proc. of FAST'10*, 2010.

[45] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," In *Proc of FAST'10*, 2010.