A Buffer Cache Management Scheme Exploiting Both Temporal and Spatial Localities

XIAONING DING
The Ohio State University
SONG JIANG
Wayne State University
and
FENG CHEN
The Ohio State University

On-disk sequentiality of requested blocks, or their spatial locality, is critical to real disk performance where the throughput of access to sequentially-placed disk blocks can be an order of magnitude higher than that of access to randomly-placed blocks. Unfortunately, spatial locality of cached blocks is largely ignored, and only temporal locality is considered in current system buffer cache managements. Thus, disk performance for workloads without dominant sequential accesses can be seriously degraded. To address this problem, we propose a scheme called *DULO* (*DU*al *LO*cality) which exploits both temporal and spatial localities in the buffer cache management. Leveraging the filtering effect of the buffer cache, DULO can influence the I/O request stream by making the requests passed to the disk more sequential, thus significantly increasing the effectiveness of I/O scheduling and prefetching for disk performance improvements.

We have implemented a prototype of DULO in Linux 2.6.11. The implementation shows that DULO can significantly increases disk I/O throughput for real-world applications such as a Web server, TPC benchmark, file system benchmark, and scientific programs. It reduces their execution times by as much as 53%.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Main memory*; C.4 [**Performance of Systems**]: *Design studies*

General Terms: Design, Performance

A preliminary version of the article was published in Proceedings of the 4th UNENIX Conference on File and Storage Technologies. The research was partially supported by the National Science Foundation under Grants CNS-0405909 and CCF-0602152.

Authors' addresses: S. Jiang, Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI 48202; email: sjiang@ece.eng.wayne.edu; X. Ding and F. Chen, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210; email: {dingxn,fchen}@cse.ohio-state.edu.

Additional Key Words and Phrases: Caching, temporal locality, spatial locality, file systems, hard disk

ACM Reference Format:

Ding, X., Jiang, S., and Chen, F. 2007. A buffer cache management scheme exploiting both temporal and spatial localities. ACM Trans. Storage 3, 2, Article 5 (June 2007), 27 pages. DOI = 10.1145/1242520.1242522 http://doi.acm.org/ 10.1145/1242520.1242522

1. INTRODUCTION

The hard drive is the most commonly used secondary storage device supporting file accesses and virtual memory paging. While its capacity growth pleasantly matches the rapidly increasing data storage demand, its electromechanical nature causes its performance improvements to lag painfully far behind processor speed progress. It is apparent that the disk bottleneck effect is worsening in modern computer systems, while the role of the hard disk as the dominant storage device will not change in the foreseeable future, and the amount of disk data requested by applications continues to increase.

The performance of a disk is constrained by its mechanical operations, including disk platter rotation (*spinning*) and disk arm movement (*seeking*). A disk head has to be on the right track through seeking and on the right sector through spinning for reading/writing its desired data. Between the two moving components of a disk drive affecting its performance, the disk arm is its Achilles' heel. This is because an actuator has to move the arm accurately to the desired track through a series of actions including acceleration, coast, deceleration, and settle. Thus, accessing of a stream of sequential blocks on the same track achieves a much higher disk throughput than that accessing of several random blocks does.

In the current practice, there are several major efforts in parallel to break the disk bottleneck. One effort is to reduce disk accesses through memory caching. By using replacement algorithms to exploit the temporal locality of data accesses where data are likely to be re-accessed in the near future after they are accessed, disk access requests can be satisfied without actually being passed to a disk. To minimize disk activities in the number of requested blocks, all current replacement algorithms are designed by choosing block miss reduction as the sole objective. However, this can be a misleading metric that may not accurately reflect real system performance. For example, requesting ten sequential disk blocks can be completed much faster than requesting three random disk blocks where disk seeking is involved. To improve real system performance, spatial locality, a factor that can make a difference as large as an order of magnitude in disk performance, must be considered. However, spatial locality is unfortunately ignored in current buffer cache managements. In the context of this article, spatial locality specifically refers to the sequentiality of the disk placements of the continuously requested blocks.

Another effort to break the disk bottleneck is reducing disk arm seeks through I/O request scheduling. I/O scheduler reorders pending requests in a block device's request queue into a dispatching order that results in minimal

seeks and thereafter maximal global disk throughput. Example schedulers include Shortest-Seek-Time-First (SSTF), C-SCAN, as well as the Deadline and Anticipatory I/O schedulers [Iyer and Druschel 2001] adopted in the current Linux kernels.

The third effort is prefetching. A prefetching manager predicts future request patterns associated with a file opened by a process. If a sequential access pattern is detected, then the prefetching manager issues requests for the blocks following the current on-demand block on behalf of the process. Because a file is usually contiguously allocated on disk, these prefetching requests can be fulfilled quickly with few disk seeks.

While I/O scheduling and prefetching can effectively exploit spatial locality and dramatically improve disk throughput for workloads with dominant sequential accesses, their ability to deal with workloads mixed with sequential and random data accesses, such as those in Web services, databases, and scientific computing applications, is very limited. This is because these two strategies are positioned at a level lower than the buffer cache. While the buffer cache receives I/O requests directly from applications and has the power to shape the requests into a desirable I/O request stream, I/O scheduling and prefetching only work on the request stream passed on by the buffer cache and have very limited ability to recatch the opportunities lost in the buffer cache management. Hence, in the worst case, a stream filled with random accesses makes I/O scheduling and prefetching largely ineffective because no spatial locality is left for them to exploit.

Concerned with the lack of ability to exploit spatial locality in buffer cache management, our solution to the deteriorating disk bottleneck is a new buffer cache management scheme that exploits both temporal and spatial localities which we call the *DU*al *LO*cality scheme (DULO). DULO introduces dual locality into the caching component in an operating system by tracking and utilizing disk placements of in-memory pages in its buffer cache management.[1] Our objective is to maximize the sequentiality of I/O requests that are serviced by disks. For this purpose, we give preference to random blocks for staying in the cache, while sequential blocks that have their temporal locality comparable to those random blocks are replaced first. With the filtering effect of the cache on I/O requests, we influence the I/O requests made by applications so that more sequential block requests and fewer random block requests are passed to the disk thereafter. The disk is then able to process the requests with stronger spatial locality more efficiently.

2. DUAL LOCALITY CACHING

2.1 An Illustrative Example

To illustrate the differences that a traditional caching scheme could make when equipped with dual locality ability, let us consider an example reference stream

[1]We use *page* to denote a memory access unit, and *block* to denote a disk access unit. They can be of different sizes. For example, a typical Linux configuration has a 4KB page and a 1KB block. A page consists of one or multiple blocks if it has a disk-mapping.

Table I. An Example Showing that a Dual-Locality-Conscious Scheme Can Be More Effective Than its Traditional Counterpart in Improving Disk Performance (Blocks being fetched are boldfaced. The MRU end of the queue is on the left.)

	Block	*Traditional*	Time(ms)	*Dual*	Time(ms)
1	A	[**A** - - - - - - -]	9.5	[**A** - - - - - - -]	9.5
2	B	[**B** A - - - - - -]	9.5	[**B** A - - - - - -]	9.5
3	C	[**C** B A - - - - -]	9.5	[**C** B A - - - - -]	9.5
4	D	[**D** C B A - - - -]	9.5	[**D** C B A - - - -]	9.5
5	X1	[**X4 X3 X2 X1** D C B A]	9.5	[D C B A **X4 X3 X2 X1**]	9.5
6	X2	[X2 X4 X3 X1 D C B A]	0	[D C B A X2 X4 X3 X1]	0
7	X3	[X3 X2 X4 X1 D C B A]	0	[D C B A X3 X2 X4 X1]	0
8	X4	[X4 X3 X2 X1 D C B A]	0	[D C B A X4 X3 X2 X1]	0
9	Y1	[**Y4 Y3 Y2 Y1** X4 X3 X2 X1]	9.5	[D C B A **Y4 Y3 Y2 Y1**]	9.5
10	Y2	[Y2 Y4 Y3 Y1 X4 X3 X2 X1]	0	[D C B A Y2 Y4 Y3 Y1]	0
11	Y3	[Y3 Y2 Y4 Y1 X4 X3 X2 X1]	0	[D C B A Y3 Y2 Y4 Y1]	0
12	Y4	[Y4 Y3 Y2 Y1 X4 X3 X2 X1]	0	[D C B A Y4 Y3 Y2 Y1]	0
13	X1	[X1 Y4 Y3 Y2 Y1 X4 X3 X2]	0	[D C B A **X4 X3 X2 X1**]	9.5
14	X2	[X2 X1 Y4 Y3 Y2 Y1 X4 X3]	0	[D C B A X2 X4 X3 X1]	0
15	X3	[X3 X2 X1 Y4 Y3 Y2 Y1 X4]	0	[D C B A X3 X2 X4 X1]	0
16	X4	[X4 X3 X2 X1 Y4 Y3 Y2 Y1]	0	[D C B A X4 X3 X2 X1]	0
17	A	[**A** X4 X3 X2 X1 Y4 Y3 Y2]	9.5	[A D C B X4 X3 X2 X1]	0
18	B	[**B** A X4 X3 X2 X1 Y4 Y3]	9.5	[B A D C X4 X3 X2 X1]	0
19	C	[**C** B A X4 X3 X2 X1 Y4]	9.5	[C B A D X4 X3 X2 X1]	0
20	D	[**D** C B A X4 X3 X2 X1]	9.5	[D C B A X4 X3 X2 X1]	0
		total time	95.0	total time	66.5

mixed with sequential and random blocks. Among the accessed blocks, we assume blocks A, B, C, and D are random blocks dispersed across different tracks. Blocks X1, X2, X3, and X4 as well as blocks Y1, Y2, Y3, and Y4 are sequential blocks located on their respective tracks. Furthermore, two different files consist of blocks X1, X2, X3, and X4, and blocks Y1, Y2, Y3 and Y4, respectively. Assume that the buffer cache has room for eight blocks. We also assume that the LRU replacement algorithm and a Linux-like prefetching policy are applied. In this simple illustration, we use the average seek time to represent the cost of any seek operation, and we use average rotation time to represent the cost of any rotation operation.[2] We ignore other negligible costs such as disk read time and bus transfer time. The 6.5ms average seek time and 3.0ms average rotation time are taken from the specification of the Hitachi Ultrastar 18ZX 10K RPM drive.

Table I shows the reference stream and the ongoing changes of cache states as well as the time spent on each access for the traditional caching and prefetching scheme (denoted as *traditional*) and its dual-locality-conscious alternative (denoted as *dual*). At the 5th access, prefetching is activated and all of the four sequential blocks are fetched because the prefetcher knows the reference (to block X1) starts at the beginning of a file. The difference in the cache states between the two schemes here is that traditional places the blocks in strict LRU

[2]With a seek reduction disk scheduler, the actual seek time between consecutive accesses should be less than the average time. However, this should not affect the legitimacy of the discussions in the section as well as its conclusions.

order, while dual rearranges the blocks and places the random blocks at the MRU end of the queue. Therefore, the four random blocks A, B, C, and D are replaced in traditional, while sequential blocks X1, X2, X3, and X4 are replaced in dual when the 9th access incurs a four-block prefetching. The consequences of these two choices are two different miss streams that turn into real disk requests. For traditional, it is {A, B, C, D} from the 17th access, a disk request stream consists of four random blocks, and the total cost is 95.0ms. For dual, it is {X1, X2, X3, X4} at the 13th access, four sequential blocks, and the total cost is only 66.5ms. Using the dual-locality-conscious scheme, we can significantly reduce I/O costs by reducing random accesses.

2.2 Challenges with Dual Locality

Introducing dual locality in cache management raises challenges that do not exist in a traditional system, which is evident even in the aforementioned simple illustrative example.

In the current cache managements, replacement algorithms only consider temporal locality (a position in a queue in the case of LRU) to make a replacement decision. While introducing spatial locality necessarily has to compromise the weight of temporal locality in a replacement decision, the role of temporal locality must be appropriately retained in the decision. In the example shown in Table I, we give random blocks A, B, C, and D more privilege of staying in cache by placing them at the MRU end of the queue due to their weak spatial locality (weak sequentiality), even though they have weak temporal locality (large recency). However, we certainly cannot keep them in cache forever if they do not have sufficient reaccesses that indicate temporal locality. Otherwise, they would pollute the cache with inactive data and reduce the effective cache size. The same consideration also applies to the block sequences of different sizes. We prefer to keep a short sequence because it has only a small number of blocks to amortize the cost of an I/O operation. However, how do we make a replacement decision when we encounter a not-recently-accessed short sequence and a recently-accessed long sequence? The challenge is essentially how to make the trade-off between temporal locality (recency) and spatial locality (sequence size) with the goal of maximizing disk performance.

3. THE DULO SCHEME

We now present the DULO scheme that exploits both temporal locality and spatial locality simultaneously and seamlessly. Because LRU or its variants are the most widely used replacement algorithms, we build the DULO scheme by using the LRU algorithm and its data structure—the LRU stack—as a reference point.

In LRU, newly fetched blocks enter into its stack top, and replaced blocks leave from its stack bottom. There are two key operations in the DULO scheme. (1) Forming sequences is one of the key operations. A *sequence* is defined as a number of blocks whose disk locations are close to each other and have been accessed continuously without an interruption during a limited time period. Additionally, a sequence is required to be stable so that blocks in it would be

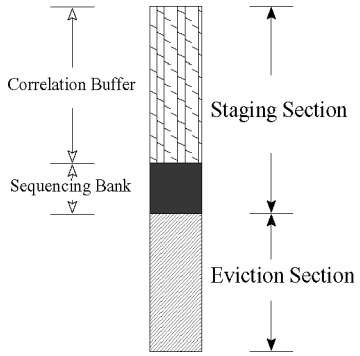

Fig. 1. LRU stack is structured for the DULO replacement algorithm.

fetched together next time when they are read from disk. Specifically, a random block is a sequence of size 1. (2) Sorting sequences in an LRU stack according to their recency (temporal locality) and size (spatial locality) with the objective that sequences of large recency and size are close to the LRU stack bottom. Because recency of a sequence changes when new sequences are added, the order of the sorted sequences should be adjusted dynamically to reflect the change.

3.1 Structuring the LRU Stack

To facilitate the operations, we partition the LRU stack into two sections (shown in Figure 1 as a vertically placed queue). The top part is called the *staging section* and is used for admitting newly fetched blocks, and the bottom part is called the *eviction section* and is used for storing sorted sequences to be evicted in their order. We further divide the staging section into two segments. The first segment is called the *correlation buffer*, and the second segment is called the *sequencing bank*. The correlation buffer in DULO is similar to the *correlation reference period* used in the LRU-K replacement algorithm [O'Neil et al. 1993]. Its role is to filter high-frequency references and to keep them from entering the sequencing bank so as to reduce the consequential operational cost. The sequencing bank is used to prepare a collection of blocks to be sequenced, and its size ranges from 0 to a maximum value, *BANK-MAX*.

Suppose we start with an LRU stack whose staging section consists of only the correlation buffer (the size of the sequencing bank is 0), and the eviction section holds the rest of the stack. When a block leaves the eviction section and a block enters the correlation buffer at its top, the bottom block of the correlation buffer enters the sequencing bank. When there are *BANK-MAX* blocks leaving the eviction section, the size of the sequencing bank is *BANK-MAX*. We then refill the eviction section by taking the blocks in the bank to form sequences out of them and insert them into the eviction section in the desired order. There are three reasons for us to maintain two interacting sections and use the bank to conduct sequence formation. (1) The newly admitted blocks have a buffering area where they are accumulated for forming potential sequences. (2) The sequences formed at the same time must share a common recency because their

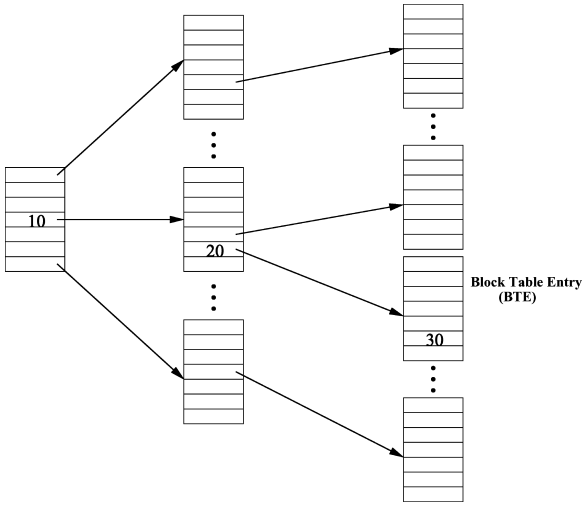

Fig. 2. Block table. There are three levels in the example block table: two directory levels and one leaf level. The table entries at different levels are fit into different memory pages. An entry at the leaf level is called Block Table Entry (BTE). Suppose one page can hold 512 entries. The access time information about LBN 2,631,710 (i.e. $10 \times 512^2 + 20 \times 512 + 30$) is recorded at the BTE entry marked as 30, which can be efficiently reached with a given LBN via directory level entries marked as 10 and 20.

constituent blocks are from the same block pool, namely, the sequencing bank in the staging section. By restricting the bank size, we make sure that the block recency will not be excessively compromised for the sake of spatial locality. (3) The blocks that are leaving the stack are sorted in the eviction section for a replacement order reflecting both their sequentiality and their recency.

3.2 Block Table: A Data Structure for Dual Locality

To implement the missing spatial locality in traditional caching systems, we introduce a data structure in the OS kernel called *block table*, which is shown in Figure 2. The block table is analogous in structure to the multilevel page table used to process address translation, however, there are clear differences between them due to the different purposes they serve. (1) The page table covers virtual address space of a process in the unit of page where a page address is the index into the table, while the block table covers disk space in the unit of block where a logical block number (LBN) of a block is the index into the table. A LBN is a unique number assigned to each addressable block contained in a disk drive. Disk manufacturers usually make every effort to ensure that accessing blocks with consecutive LBNs has a minimal disk head positioning cost [Schlosser et al. 2005]. (2) The page table is used to translate a virtual address into its physical address, while the block table is used to provide the times of recent accesses for a given disk block. (3) The requirement on the efficiency of looking up the page table is much more demanding and performance-critical than that on the efficiency of looking up the block table because the former supports instruction execution, while the latter facilitates I/O operations. This is the reason why a

hardware TLB has to be used to expedite page table look-up, while there is no such need for the block table. (4) Each process owns a page table, while each disk drive owns a block table.

In the system, we set a global variable called a *disk access clock*, which ticks each time a block is fetched into memory and stamps the block being fetched with the current clock time. We then record the timestamp in an entry at the leaf-level of the block table, which is determined by the LBN of the block. We call the entry a block table entry (BTE). When the block is reclaimed, we reset the information recorded for that block to prevent new block allocated to the same disk location from inheriting the stale information. A BTE is analogous in structure the to the page table entry (PTE) of a page table. Each BTE allows at most two most recent access timestamps to be recorded in it. Whenever a new timestamp is added, the oldest timestamp is replaced if the BTE is full. In addition, to efficiently manage the memory space held by the block table, the timestamp is also recorded in each table entry at directory levels (equivalent to page global directory (PGD) and page middle directory (PMD) in the Linux page table). Each time the block table is looked up in a hierarchical way to record a new access timestamp, the timestamp is also recorded in each directory entry that has been passed. In this way, each directory entry keeps the most recent timestamp among those of all its direct/indirect children entries when the table is viewed as a tree. The entries of the table are allocated in the same on-demand fashion as Linux uses with the page table.

The memory consumption of the block table can be flexibly controlled. When the system memory pressure is so high that the system needs to reclaim memory held by the table, it traverses the table with a specified timestamp threshold for reclamation. Because the most recent access timestamps are recorded in the directories, the system will remove a directory once it finds that its timestamp is smaller than the threshold, and all the subdirectories and BTEs under it will be removed accordingly.

3.3 Forming Sequences

When the sequencing bank is full, it is time to examine blocks in the bank to aggregate them into sequences. We first sort the blocks, according to their LBNs in ascending order, into a list, then form sequences starting from the end of the list holding blocks of small LBNs. To ensure the sequentiality and stability of a sequence, we follow a rule by using the timestamp information recorded in the block table about the blocks. According to the rule, the last block, denoted as A, of a developing sequence should not be coalesced with its succeeding block in the list, denoted as B, if the two blocks belong to one of the following cases.

(1) Block B is not close enough to block A. DULO takes 4 blocks as the distance threshold to determine if two blocks are close to each other. If the distance between block B and block A is within the threshold, the read-ahead mechanism built in most hard drives, which is enabled by default in most systems, can fetch block B into disk caches automatically after it fetches block A. So the reading of block B is inexpensive once block A is read.

(2) Block B and block A are not continuously fetched from disk. If the most recent timestamp of block B is not greater than the most recent timestamp of block A by 1, the accesses of block A and block B are interleaved with the accesses of other blocks, and so high cost disk head seeks could be involved.
(3) Block B and block A were not continuously fetched from disk. This includes two cases. (a) One of the two blocks has only one timestamp (representing its current access) while the other block has two timestamps (representing both its current and previous accesses). (b) The nonrecent timestamp of block B is not greater than the nonrecent timestamp of block A by 1.
(4) The current sequence size reaches 128 blocks which we deem to be a sufficient maximal sequence size to amortize a disk operation cost.

If any one of the conditions is met, a complete sequence has been formed, and the formation of a new sequence starts. Otherwise, block B becomes part of the current sequence, and the remaining blocks continue to be tested.

3.4 The DULO Replacement Algorithm

There are two challenging issues to be addressed in the design of the DULO replacement algorithm. The first issue is the potentially prohibitive time overhead associated with the DULO scheme. In a strict LRU algorithm, both missed blocks and hit blocks are required to move to the stack top. This means that a hit on a block in the eviction section is associated with a bank sequencing cost and a cost for ordering sequence in the eviction section. These additional costs that could even incur in a system with few memory misses are unacceptable. In fact, a strict LRU algorithm is rarely used in real systems because of its overhead associated with every memory reference [Jiang et al. 2005]. Instead, its variant, the CLOCK replacement algorithm, has been widely used in practice. In CLOCK, when a block is hit, it is only flagged as a young block without being moved to the stack top. When a block has to be replaced, the block at the stack bottom is examined. If it is a young block, it is moved to the stack top and its young-block status is revoked. Otherwise, the block is replaced. It is known that CLOCK simulates LRU behaviors very well, and its hit ratios are very close to those of LRU. For this reason, we build the DULO replacement algorithm based on the CLOCK algorithm, that is, it delays the movement of a hit block until it reaches the stack bottom. In this way only block misses could trigger sequencing and the eviction section refilling operations. Compared with miss penalty where disk operations are involved, their costs are very small.

The second issue is how sequences in the eviction section are ordered for replacement according to their temporal and spatial localities. We adopt an algorithm similar to GreedyDual-Size used in Web file caching [Cao and Irani 1997]. GreedyDual-Size was originally derived from GreedyDual [Young 1998]. It makes its replacement decision by considering the recency, size, and fetching cost of cached files. It has been proved that GreedyDual-Size is online-optimal, which is k-competitive, where k is the ratio of the size of the cache to the size of the smallest file. In our case, file size is equivalent to sequence size, and the file fetching cost is equivalent to the I/O operation cost for a sequence access.

For sequences whose sizes are distributed in a limited range which is bounded by the bank size, we currently do not discriminate their fetching costs for two reasons. First, moving a disk head onto the first block of a sequence constitutes the major part of the time to fetch the sequence, while the costs of reading the remaining blocks are relatively insignificant. Usually fetching a long sequence may take only slightly longer time than fetching a short one. Second, most long sequences are generated by prefetching, which can be overlapped (at least partially) by computation, while short sequences are usually generated by on-demand requests, which can hardly be overlapped. Our algorithm can be easily modified to accommodate the cost variance if necessary in the future.

The DULO algorithm associates each sequence with an attribute H, where a relatively small H value indicates its associated sequence should be evicted earlier. The algorithm has a global inflation value L, which is initiated as 0. When a new sequence s is admitted into the eviction section, its H value is set as $H(s) = L + 1/size(s)$, where $size(s)$ is the number of the blocks contained in s. When a sequence is evicted, we assign the H value of the sequence to L. So L records the H value of the most recently evicted sequence. The sequences in the eviction section are sorted by their H values with sequences of small H values at the LRU stack bottom. In the algorithm, a sequence of large size tends to stay at the stack bottom and to be evicted earlier. However, if a sequence of small size is not accessed for a relatively long time, it would be replaced. This is because a newly admitted long sequence could have a larger H value due to the L value, which keeps being inflated by evicted blocks. When all sequences are random blocks (i.e., their sizes are 1), the algorithm degenerates into the LRU replacement algorithm.

As we have mentioned before, once a bank size of blocks are replaced from the eviction section, we take the blocks in the sequencing bank to form sequences and order the sequences by their H values. Note that all these sequences share the same current L value in their H value calculations. With a merge-sorting of the newly ordered sequence list and the ordered sequence list in the eviction section, we complete the refilling of the eviction section, and the staging section ends up with only the correlation buffer. The algorithm is described using pseudocode in Figure 3.

4. PERFORMANCE EVALUATION

To demonstrate the performance improvements of DULO on a modern operating system, we implement it in the recent Linux kernel 2.6.11. We then evaluate the DULO scheme on a wide range of benchmark programs and real-world applications. We show that DULO achieves significant performance improvements for I/O-intensive workloads by reducing random accesses to the hard disk.

4.1 The DULO Implementation

Linux uses an LRU variant that is similar to the 2Q replacement [Johnson and Shasha 1994] in its memory management. This brings up some implementation issues. Let us start with a brief description of the Linux replacement policy before we introduce the implementation of DULO.

```
/* Initialize */
L = 0;
losses_of_eviction_section = 0;
disk_access_clock = 0;

/* Procedure to be invoked upon a reference to block b */
if b is in cache
   mark b as a young block;
else {
     increase disk_access_clock by 1;
     update the BTE of block b in the hierarchical block table;
     while (block e at the stack bottom is young) {
       revoke the young block state;
       move it to the stack top;
       losses_of_eviction_section++;
       if (losses_of_eviction_section == BANK_MAX)
           refill_eviction_section();
     }
     replace block e at the stack bottom;
     s = e.sequence; /* s is the sequence that e belongs to */
     L = H(s);
     losses_of_eviction_section++;
     if (losses_of_eviction_section == BANK_MAX)
         refill_eviction_section();
     place block b at the stack top as a young block
}

/* procedure to refill the eviction section */
    refill_eviction_section()
    {
      sort the blocks in sequencing bank according to their LBNs;
      /* group sequences */
      examine the sorted blocks to obtain all sequences;
      for each of the above sequences, s
          H(s) = L + 1/size(s);
      sort the above sequences by H(s) into list L1;

      /* L2 is the list of sequences in eviction section */
      eviction_section = merge_sort(L1, L2);
      losses_of_eviction_section = 0;
    }
```

Fig. 3. The DULO replacement algorithm.

4.1.1 *Linux Caching.* In the Linux replacement policy, all the process pages and file pages are grouped into two LRU lists called the *active list* and the *inactive list*. As their names indicate, the active list is used to store recently accessed pages, and the inactive list is used to store those pages that have not been accessed for some time. A faulted-in page is placed at the head of the inactive list. The replacement page is always selected at the tail of the inactive list. An inactive page is promoted into the active list when it is accessed as a file page (by *mark_page_accessed()*), or it is accessed as a process page and its

reference is detected at the tail of the inactive list. An active page is demoted to the inactive list if it is determined to have not been recently accessed (by *refill_inactive_zone()*).

4.1.2 *Implementation Issues.* In our prototype implementation of DULO, we do not replace the original Linux page-frame reclaiming code with a faithful DULO scheme implementation. Instead, we opt to keep the existing data structure and policies mostly unchanged, and seamlessly adapt DULO into them. We make this choice to serve the purpose of demonstrating what improvements a dual-locality design could bring to an existing spatial-locality-unaware system without radically changing the basic infrastructure of its replacement policy.

In Linux we partition the inactive list into a staging section and an eviction section because the list is the place where new blocks are added and old blocks are replaced. However, the inactive list is different with the LRU stack in the DULO scheme described in Section 3. The LRU stack manages all the memory pages in the system and has a fixed size, while the inactive list in Linux manages only inactive pages and has a variable size without a lower bound. The difference brings up two issues:

(1) The inactive list in Linux cannot guarantee a reasonably large eviction section size demanded by DULO to effectively hold random blocks. We know that in DULO random blocks are conditionally protected from eviction by the sequential blocks inserted as sequences at a position in the LRU stack that is closer to the stack bottom than that where random blocks stay. These sequential blocks would be evicted earlier than random blocks when some pages need to be reclaimed. With an excessively short eviction section, random blocks may be evicted prematurely before some sequential blocks are available to be formed into sequences and inserted in the eviction section to serve as a protection for the random blocks. Our parameter sensitivity studies in Section 4.6 show that an excessively short eviction section could increase the execution times by over 20% for some workloads.
(2) Prefetched blocks may be evicted too early even if they are to be accessed in the near future. To prevent active pages from being flushed by a large number of one-time accesses, Linux places newly fetched pages into the inactive list. These pages are evicted if they have not been accessed before they reach the tail of the inactive list. Prefetched blocks, which are sequential blocks, are usually evicted more quickly than other blocks in DULO. The difference is not evident if there is a long staging section. However, when the size of the inactive list is close to the size of the eviction section set by DULO, the staging section has to be so small that it loses the function of buffering newly fetched blocks. In such case, the prefetched pages may be quickly moved to the bottom of the eviction section and evicted prematurely, which could degrade the performance.

Because both of the issues are raised by an excessively short inactive list, we modified the function *refill_inactive_zone()*, which is used to refill inactive list, to keep the inactive list at least two times as long as the eviction section in DULO. As the eviction section usually holds less than 15% of memory size

in a system with more than 512MB memory, we believe that the impact of the modification is insignificant.

In the DULO scheme on-disk distance of two blocks is determined by their LBNs. However, in Linux anonymous pages do not have LBNs because they have no mappings on disk yet. In our implementation, the anonymous pages are treated as random blocks until they are swapped out and are associated with certain disk-mappings.

4.2 Experiment Setting

We conduct the experiments on a Dell desktop with a single 3.0GHz Intel Pentium 4 processor, 512MB memory, and a Western Digital 7200RPM IDE disk with a capacity of 160GB. The read-ahead mechanism built in the hard drive is enabled. The operating system is Redhat WS4 with its kernel updated to Linux 2.6.11. The file systems is Ext2. In the experiments we set the sequencing bank size as 4MB and the eviction section size as 64MB. These choices are based on the results of our parameter sensitivity studies presented in Section 4.6.

To reveal the impact of introducing spatial locality into replacement decisions on different workloads, we run two types of I/O-intensive applications on the original Linux kernel and on the kernel with the enhancement of DULO. The first type of applications access hard disk mainly for file reads/writes, while the second type of applications access hard disk mainly for virtual memory paging.

4.3 Experiments on File Accesses

4.3.1 *Benchmarks.* We select the following five benchmarks to evaluate DULO's performance for file accesses. These benchmarks have different access patterns, among which *TPC-H* consists of almost all sequential accesses, *diff* consists of almost all random accesses, and three other benchmarks, *BLAST*, *PostMark*, and *LXR*, have mixed I/O access patterns.

(1) *TPC-H* is a decision support benchmark that runs business-oriented queries against a database system [TPC-H 2006]. In our experiment, we use PostgreSQL version 7.3.3 as the database server. The scale factor of the database is 0.3. We choose query 6, which carries out a sequential scan over table *lineitem* and run the query five times to emphasize its caching effect.

(2) *diff* is a tool that compares two files in a character-by-character fashion. We run it on two Linux 2.6.11 source code trees in the experiment. *diff* accesses the files in strict alphabetical order, while the files are usually contiguously placed on the disk in the order in which they are created. Though *diff* scans each file sequentially, the mismatch of these two orders, combined with the fact that the Linux source code trees consist of only small files, makes *diff* have a random access pattern on disk.

(3) *BLAST* (basic local alignment search tool) is software from the National Center for Biotechnology Information [BLAST]. BLAST compares nucleotide or protein sequences to those stored in the given sequence databases to search for matched segments. In the experiments we run the

program to compare 5 nucleotide sequences against the *patnt* sequence database. The total size of the database including index files, data files, and header files is 1.3GB. While *BLAST* scans the data files sequentially, the index and header files are accessed randomly.

(4) *PostMark* is a benchmark designed by Network Appliance to test performance of systems, such as e-mail servers or news group servers whose workloads are dominated by operations on small files [Katcher 1997]. It first generates a file pool with file sizes that are randomly selected between the upper and lower bounds specified in a user configuration. Then a large number of operations, including *create or delete files*, *read or append files*, are carried out. The total number of the operations and the percentages of each type of operations are configurable. At the beginning of our experiment, PostMark creates 1,500 files whose sizes range from 512B to 512KB. Then 40,000 operations are performed on these files; 80% of these operations are reads and the remaining 20% are writes.

(5) *LXR* (Linux cross-reference) is a widely used source code indexer and cross-referencer [LXR]. It serves user queries for searching, browsing, or comparing source code trees through an HTTP server. In our experiment, the file set for querying consists of three versions of Linux kernels 2.4.20, 2.6.11, and 2.6.15. To simulate user requests, we use WebStone 2.5 [Trent and Sake 1995] to generate 25 clients which concurrently submit freetext search queries. To make a query, each client randomly selects a keyword from a pool of 50 keywords and sends it to the server. It sends its next query right after it receives the results of the previous query. We randomly select 25 Linux symbols from the file */boot/System.map* and another 25 popular OS terms such as lru, scheduling, and page as the pool of candidate query keywords. One run of the experiment lasts for 30 minutes. In each run, a client always uses the same sequence of keyword queries. The metric we use to measure system performance is the query throughput represented by MBits/sec, which is the number of megabits of query results returned by the server per-second. Due to intensive I/O operations in the experiment, this metric is suitable to measure the effectiveness of the memory and disk systems. This metric is also used for reporting WebStone benchmark results.

4.3.2 *Experimental Results.* Figures 4 and 5 show the execution times (throughputs for LXR), hit ratios, and distribution of disk access sequence sizes for the original Linux system and the system with DULO enhancement for the five workloads when we vary memory size. Because the major effort of DULO in improving system performance is to influence the quality of the requests presented to the disk, the number of sequential block accesses (or sequence-size), we show the sequence-size distributions for the workloads running on Linux as well as on DULO. For this purpose, we use CDF (cumulative distribution function) curves to show how many percentages (shown on Y-axis) of the sequences whose sizes are less than a certain threshold (shown on X-axis). For each workload, we select two memory sizes to draw the corresponding CDF curves for Linux and DULO. These two memory sizes are selected according to

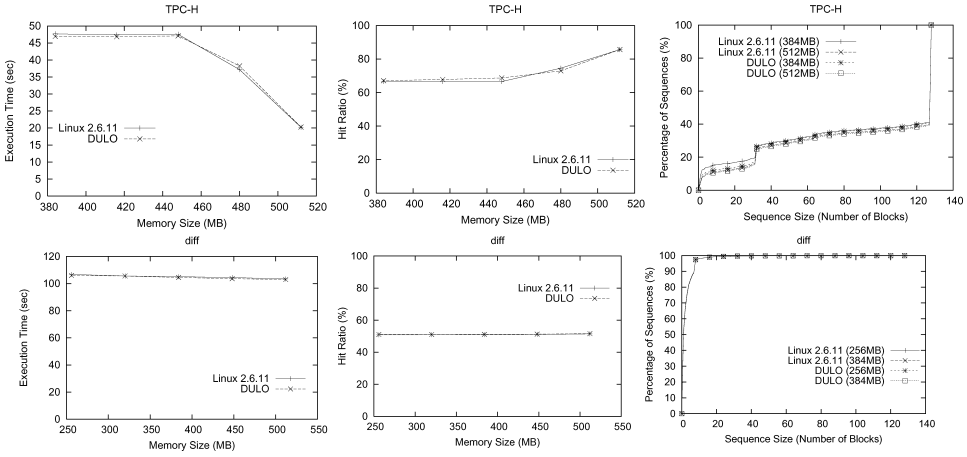

Fig. 4. Execution times, hit ratios, and disk access sequence-size distributions (CDF curves) of the Linux 2.6.11 caching and DULO caching schemes for TPC-H with the sequential request pattern and *diff* with the random request pattern.

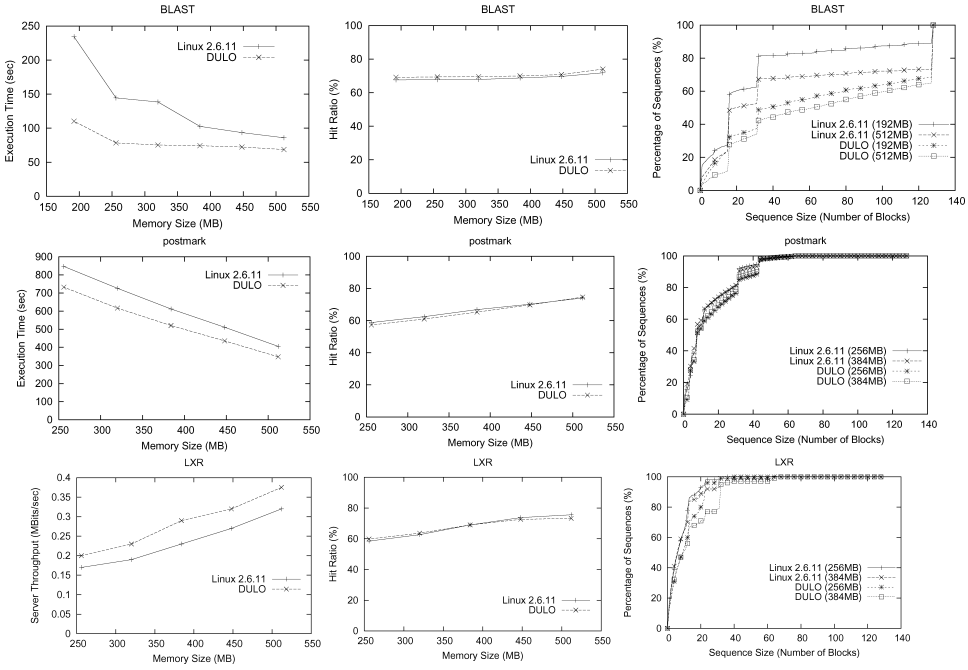

Fig. 5. Execution times (throughtputs for LXR), hit ratios, and disk access sequence-size distributions (CDF curves) of the Linux 2.6.11 caching and DULO caching schemes for blast, PostMark, and LXR with the mixed request pattern.

the execution time gaps between Linux and DULO shown in the figures about execution time, that is, one memory size is selected due to its small gap and another is selected due to its large gap. The memory sizes are shown in the legends of the CDF figures.

In Figure 4, the CDF curves show that in workload TPC-H more than 85% of the sequences are longer than 16 blocks. For this almost-all-sequential workload, DULO has limited influence on the performance. It can slightly increase the sizes of short sequences, and accordingly reduce execution time by 2.1% with a memory size of 384MB. However, for the almost-all-random workload *diff*, more than 80% of the sequences are shorter than 4 blocks. Unsurprisingly, DULO cannot create sequential disk requests from application requests consisting of purely random blocks. As expected, we see almost no improvements of execution times by DULO.

The other three benchmarks have a considerable amount of both short sequences and long sequences. For example, PostMark has more than 25% sequences shorter than 4 blocks and over 30% sequences longer than 16 blocks. DULO achieves substantial performance improvements for these workloads with mixed request patterns (see Figure 5). There are several observations from the figures. First, the increases of sequence sizes are directly correlated to the improvement of the execution times or throughputs. Let us take BLAST as an example. With a memory size of 512MB, Linux has 8.2% accesses whose sequence sizes equal 1, while DULO reduces this percentage to 3.5%. At the same time, in DULO, there are 57.7% sequences whose sizes are larger than 32, compared with 33.8% in Linux. Accordingly, there is a 20.1% execution time reduction by DULO. In contrast, with the memory size of 192MB, DULO reduces random accesses from 15.2% to 4.2% and increases sequences longer than 32 from 19.8% to 51.3%. Accordingly, there is a 53.0% execution time reduction. The correlation clearly indicates that the size of a requested sequence is a critical factor affecting disk performance and DULO makes its contributions through increasing sequence sizes. Second, DULO increases the sequence size without excessively compromising temporal locality. This is demonstrated by the small difference of hit ratios between Linux and DULO for different memory sizes shown in Figures 4 and 5 . For example, DULO reduces the hit ratios of PostMark by 0.53% $\sim$ 1.6%, while it slightly increases the hit ratio of BLAST by 1.1% $\sim$ 2.2%. In addition, this observation also indicates that reduced execution times and increased server throughputs are results of the improved disk I/O efficiency rather than the reduced I/O operations in terms of the number of accessed blocks, which is actually the objective of traditional caching algorithms. Third, sequential accesses are important in leveraging the buffer cache filtering effect by DULO. We see that DULO achieves more performance improvement for BLAST than it does for PostMark and LXR. From the CDF curves, we observe that BLAST has over 40% sequences whose sizes are larger than 16 blocks, while PostMark and LXR have only 30% and 15% such sequences, respectively. The small portion of sequential accesses in PostMark and LXR make DULO less capable of keeping random blocks from being replaced because there are not sufficient sequentially accessed blocks to be replaced first.

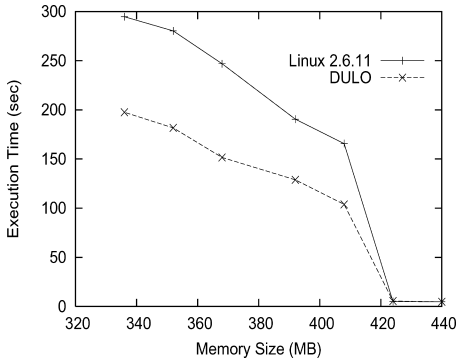

Fig. 6. SMM execution times on the original Linux kernel and DULO instrumented kernel with varying memory sizes.

4.4 Experiments on VM Paging

In order to study the influence of the DULO scheme on VM paging performance, we use a representative scientific computing benchmark, namely, sparse matrix multiplication (SMM) from a NIST benchmark suite SciMark2 [Pozo and Miller 2000]. The SMM benchmark multiplies a sparse matrix with a vector. The matrix is of size $N \times N$, and has M nonzero data regularly dispersed in its data geometry, while the vector has a size of N ($N = 3 \times 2^{20}$ and $M = 3 \times 2^{23}$). Each element in the matrix or the vector is a double precision value of 8 bytes long. In the multiplication algorithm the matrix is stored in a compressed-row format so that all the nonzero elements are contiguously placed in a one-dimensional array with two index arrays recording their original locations in the matrix. The total working set, including the result vector and the index arrays, is around 348MB. To cause the system paging and stress the swap space accesses, we have to adopt small memory sizes from 336MB to 440MB, including the memory used by the kernel and applications.

To increase spatial locality of swapped-out pages in the disk swap space, Linux tries to allocate contiguous swap slots on the disk to sequentially reclaimed anonymous pages in the hope that they would be efficiently swapped-in in the same order. However, the data access pattern in SMM foils the system effort. SMM first initializes the arrays one-by-one. This thereafter causes each array to be swapped out continuously and allocated on the disk sequentially when the memory cannot hold the working set. However, in the computation stage, the elements that are accessed in the vector array are determined by the matrix locations of the elements in the matrix array. Thus, those elements are irregularly accessed, but they are contiguously located on the disk. The swap-in accesses of the vector arrays turn into random accesses, while the elements of matrix elements are still sequentially accessed. This explains the time differences for SMM between its execution on the original kernel and that on the DULO instrumented kernel (see Figure 6). DULO significantly reduces the execution times by up to 38.6%, which happens when the memory size is 368MB. This is because DULO detects the random pages in the vector array and caches them with a higher priority. Because the matrix is a sparse one, the vector array

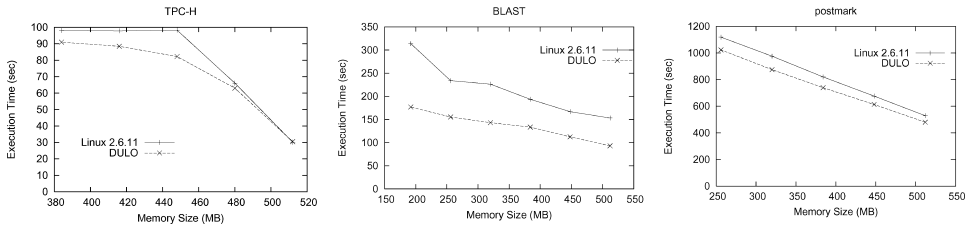

Fig. 7. Execution times of TPC-H, BLAST, and PostMark on the original Linux kernel and DULO instrumented kernel with varying memory sizes with an aged file system.

cannot obtain sufficiently frequent reuses to allow the original kernel to keep them from being paged out. In addition, the similar execution times between the two kernels when there is enough memory (exceeding 424MB) to hold the working set shown in the Figure 6 suggest that DULO's overhead is small.

4.5 Experiments with an Aged File System

The free space of aged file systems is usually fragmented, and sometimes it is difficult to find a large chunk of contiguous space for creating or extending files. This usually causes large files to consist of a number of fragments of various sizes and files in the same directory to be dispersed on the disk. This noncontiguous allocation of logically related blocks of data worsens the performance of I/O-intensive applications. However, it could provide DULO more opportunities to show its effectiveness by trying to keep small fragments in memory.

To show the performance implication of an aged file system on DULO, we selected three benchmarks, TPC-H, BLAST, and PostMark, as representatives, and run them on an aged file system. As we know, TPC-H is dominated by large sequential accesses, while BLAST and PostMark have a pattern mixed with sequential and random accesses. In our experiment, we choose an aging tool called agesystem [Loizides] to emulate an aged file system by repeatedly creating and deleting files on a fresh file system. Agesystem first creates 100 directories, each of which has 50 subdirectories. It then creates and deletes files randomly under these subdirectories. Agesystem creates files with different sizes, among which 89% have small sizes, 10% have medium sizes, and 1% have large sizes. The file sizes follow the normal distribution with the minimum file size equal to 0KB, 4KB, and 64KB for each type of the files, respectively, and average file size equal to 4KB, 64KB, and 1MB, respectively. The locations of new files and the files to be deleted are randomly chosen. It depends on the size of the free space whether to create or to delete a file. The probability of deleting a file increases when the size of the free space in the file system decreases. In our experiment, we stop the aging process when 50% of the disk space is allocated.

Figure 7 shows the execution times of the selected benchmarks with the aged file system. For benchmarks dominated with long sequential accesses such as TPC-H, an aged file system degrades its performance. For example, with a memory size of 448MB, the execution time of TPC-H on an aged file system is 107% more than on a fresh file system. This is because, on an aged file

system, large data files scanned by TPC-H are broken into pieces of various sizes. Accessing of small pieces of data on the disk significantly increases I/O times. Dealing with sequences of various sizes caused by an aged file system, DULO can reduce the execution time by a larger percentage than it does on a fresh file system. For TPC-H, with a fresh file system, DULO can hardly reduce the execution time as is shown in Figure 4. With an aged file system, DULO manages to identify sequences of small sizes and give them a high caching priority so that their high I/O costs can be avoided. This results in a 16.3% reduction of its execution time with the memory size of 448MB.

For benchmarks with patterns mixed of sequential accesses and random accesses, such as BLAST and PostMark, an aged file system has different effects on DULO's performance, depending on the sequentiality of the workloads and memory sizes.

For BLAST, which abounds in long sequences, DULO reduces its execution time by a larger percentage on an aged file system than it does on a fresh file system when memory size is large. For example, when the memory size is 512MB, DULO reduces the execution time by 20.1% with a fresh file system, while it reduces the execution time by 39.2% with an aged file system. There are two reasons for this. One reason is that the noncontiguous allocation of the large data files scanned by BLAST provides DULO with new opportunities to reduce execution times by holding small pieces of the data files in memory. The other reason is that the index and header files are also fragmented in an aged file system. This causes increased costs for the random accesses of these files. Accordingly, reducing these random accesses generates additional execution time reduction. However, DULO may become less effective when memory size is small. For example, when the memory size is 192MB, DULO reduces the execution time by 53.0% with a fresh file system, while it reduces the execution time by 43.6% with an aged file system. This is because the fragmentation increases the number of random blocks accessed by BLAST. A small memory cannot accommodate these random blocks as well as sufficient sequential blocks to protect random blocks, which limits DULO's ability to improve execution times.

For PostMark, which has a relatively small percentage of long sequences, the reduction of long sequences makes its access pattern close to that in almost all random applications, such as diff, where the lack of sufficient long sequences causes short sequences to be replaced quickly. Thus we expect that DULO may reduce less execution time with an aged file system than it does with a fresh file system. This is confirmed by our experimental results for PostMark. Its execution time is reduced by 8.6% $\sim$ 10.3% with various memory sizes on an aged file system, instead of 13.6% $\sim$ 15.1% on a fresh file system.

In summary, this experiment clearly shows that the fragmentation of file system increases DULO's performance advantages for applications that are dominated with sequential accesses. Meanwhile, the performance advantages can be slightly compromised for applications that contain a relatively small percentage of sequential accesses, especially when memory size is small. While programs and file systems are designed to preserve sequential accesses for efficient disk accesses, DULO is important in keeping system performance from

Table II. The Execution times (Seconds) for BLAST, PostMark, and SMM, and Server Throughputs (MBit) for LXR with Varying Bank Sizes (MB). (Eviction section size is 64MB. Memory sizes are shown with their respective benchmark names.)

Bank Size (MB)	*BLAST* (320MB)	*PostMark* (448MB)	*LXR* (384MB)	*SMM* (368MB)
1	77.27	440.62	0.24	170.90
2	76.78	437.74	0.26	158.25
4	75.33	435.65	0.29	151.64
8	79.25	436.02	0.30	148.07
16	85.61	436.28	0.27	184.59

degradation due to an aged file system and in retaining the expected performance advantage associated with sequential accesses.

4.6 Parameter Sensitivity and Overhead Study

There are two parameters in the DULO scheme, the (maximum) sequencing bank size and the (minimal) eviction section size. To test the performance sensitivity to these parameters, we select benchmarks BLAST, PostMark, LXR, and SMM for the study, while the other two benchmarks, TPC-H and diff, are insensitive to the spatial locality management in the DULO scheme. We vary the sequencing bank size from 1MB to 16MB and vary the eviction section size from 16MB to 128MB. For each size, we run the benchmarks with selected memory sizes. For each benchmark except BLAST, we select the memory size for which DULO achieves the largest performance improvement so that the performance variation caused by changing the sequencing bank size and the eviction section size can be clearly demonstrated. For BLAST, DULO achieves the largest performance improvement with 192MB memory, which is too small to provide a 128MB eviction section. Therefore, we select a memory size of 320MB for it.

Table II shows the different execution times or server throughputs with varying sequencing bank sizes. We observe that across the benchmarks with different access patterns, their performance is not sensitive to the variation of the parameter within a large range. Meanwhile, there exist bank sizes, roughly in a range from 4MB to 8MB, that are most beneficial to the performance, for two reasons. (1) A bank of too small size has little chance to form long sequences. (2) When bank size becomes large, approaching the eviction section size, the large bank size causes the eviction section to be refilled too late and forces the random blocks in the section to be evicted. This foils the DULO's effort to take the spatial locality into replacement decision. Therefore, we choose 4MB as the bank size in our experiments.

Table III shows the different execution times or server throughputs with varying eviction section sizes. Obviously the larger the section size is, the more control DULO would have on the eviction order of the blocks in the section, which usually means better performance. The data do show the trend. Meanwhile, the data also show that benefits from increasing the eviction section size saturate once the size exceeds the range from 64MB to 128MB. In our

Table III. The Execution Times (Seconds) for BLAST, PostMark, and SMM, and Server Throughputs (MBit) for LXR with Varying Eviction Section Sizes (MB). (Sequencing bank size is 4MB. Memory sizes are shown with their respective benchmark names.)

Eviction Section Size (MB)	*BLAST* (320MB)	*PostMark* (448MB)	*LXR* (384MB)	*SMM* (368MB)
16	78.32	485.74	0.24	246.60
32	76.68	459.32	0.27	235.33
64	75.33	435.65	0.29	151.64
128	74.92	414.03	0.29	150.12

experiments, we choose 64MB as the section size because the memory demands of our benchmarks are relatively small.

The space overhead of DULO is its block table. We monitor the sizes of the memory space occupied by the block table when we run the four benchmarks. The block tables for BLAST, PostMark, LXR, and SMM consume only 0.81%, 0.36%, 0.47%, and 0.03% of the total memory size, respectively. The size growth of block table corresponds to the number of compulsory misses. Only a burst of compulsory misses could cause the table size to be quickly increased. However, the table space can be reclaimed by the system in a grace manner as described in Section 3.2. The time overhead of DULO is trivial because the operations are associated with memory misses. On average, a miss incurs 1 operation of recording the timestamp into the block table, 10 comparison operations in sorting the sequencing bank, 1 operation of comparing its LBN and timestamps with those of its succeeding block, and 16 comparison operations to insert the block into the eviction section.

5. RELATED WORK

Because of the serious disk performance bottleneck that has existed over decades, many researchers have attempted to avoid, overlap, or coordinate disk operations. In addition, there are studies on the interactions of these techniques and on their integration in a cooperative fashion. Most of the techniques are based on the existence of locality in disk access patterns, either temporal locality or spatial locality.

5.1 Buffer Caching

One of the most actively researched areas on improving I/O performance is buffer caching, which relies on intelligent replacement algorithms to identify and keep active pages in memory so that they can be reaccessed without actually accessing the disk later. Over the years, numerous replacement algorithms have been proposed. The oldest and yet still widely adopted algorithm is the Least Recently Used (LRU) algorithm. The popularity of LRU comes from its simple and effective exploitation of temporal locality: a block that is accessed recently is likely to be accessed again in the near future. There are also a large number of other algorithms proposed such as 2Q [Johnson and Shasha 1994], MQ [Zhou et al. 2004], ARC [Megiddo and Modha 2003], LIRS [Jiang and Zhang 2002],

etc. All these algorithms focus only on how to better utilize temporal locality so that they are able to better predict the blocks to be accessed and try to minimize the page fault rate. None of these algorithms considers spatial locality, that is, how the stream of faulted pages is related to their disk locations. Because of the nonuniform access characteristic of disks, the distribution of the pages on disk can be more performance-critical than the number of the pages itself. In other words, the quality of the missed pages deserves at least the same amount of attention as their quantity. Our DULO scheme introduces spatial locality into the consideration of page replacement and thus makes replacement algorithms aware of page placements on the disk.

5.2 I/O Prefetching

Prefetching is another actively researched area on improving I/O performance. Modern operating systems usually employ sophisticated heuristics to detect sequential block accesses so as to activate prefetching, as well as adaptively adjust the number of blocks to be prefetched within the scope of one individual file [Pai et al. 2004]. System-wide file access history has been used in probability-based predicting algorithms, which track sequences of file access events and evaluate the probability of a file occurring in the sequences [Griffioen and Appleton 1994; Kroeger and Long 1996, 2001; Lei and Duchamp 1997]. This approach can perform prefetching across files and achieve a high prediction accuracy due to its use of historical information.

The performance advantages of prefetching coincides with sequential block requests. A recently proposed scheme called DiskSeen introduces disk layout information into the prefetch algorithm to make prefetching more effective in identifying and loading data contiguously located on the disk [Ding et al. 2007]. While prefetchers by themselves cannot change the I/O request stream in any way as the buffer cache does, they can take advantage of the more sequential I/O request streams that result from the DULO cache manager. In this sense, DULO is a complementary technique to prefetching. With the current intelligent prefetching policies, the efforts of DULO on sequential accesses can be easily translated into higher disk performance.

5.3 Integration between Caching and Prefetching

Many research papers on the integration of caching and prefetching consider the issues such as the allocations of memory to cached and prefetched blocks, the aggressiveness of prefetching, and the use of application-disclosed hints in the integration [Albers and Buttner 2003; Cao et al. 1995, 1996; Gill and Modha 2005; Kaplan et al. 2002; Kimbrel et al. 1996; Patterson et al. 1995; Tomkins et al. 1997]. Sharing the same weakness as those in current caching policies, this research only utilizes the temporal locality of the cached/prefetched blocks and uses the hit ratio as a metric for deciding memory allocations. Recent research has found that prefetching can have a significant impact on caching performance and points out that the number of aggregated disk I/Os is a much more accurate indicator of the performance seen by applications than the hit ratio [Butt et al. 2005].

Most of the proposed integration schemes rely on application-level hints about I/O access patterns provided by users [Cao et al. 1995, 1996; Kimbrel et al. 1996; Patterson et al. 1995; Tomkins et al. 1997]. This reliance certainly limits their application scope because users may not be aware of the patterns or source code may not be available. The work described in Kaplan et al. [2002], and Gill and Modha [2005] does not require additional user support, and thus is more related to our DULO design.

In Kaplan et al. [2002], a prefetching scheme called *recency-local* is proposed and evaluated using simulations. Recency-local prefetches the pages that are nearby the one being referenced in the LRU stack.[3] It takes the reasonable assumption that pages adjacent to the one being demanded in the LRU stack would likely be used soon because it is likely that the same access sequence would be repeated. The problem is that those nearby pages in the LRU stack may not be adjacent to the page being accessed on disk (i.e., sharing spatial locality). In fact, this is the scenario that is highly likely to happen in a multiprocess environment where multiple processes that access different files interleavingly feed their blocks into the common LRU stack. Prefetching that involves disk seeks makes little sense in improving I/O performance, and can hurt the performance due to possible wrong predictions. If we reorder the blocks in a segment of an LRU stack according to their disk locations so that adjacent blocks in the LRU stack are also close to each other on disk, then replacing and prefetching of the blocks can be conducted in a spatial locality conscious way. This is one of the motivations of DULO.

Another recent work is described in Gill and Modha [2005] in which cache space is dynamically partitioned among sequential blocks, which have been prefetched sequentially into the cache, and random blocks, which have been fetched individually on demand. Then a marginal utility is used to constantly compare the contributions to the hit rate between the allocation of memory to sequential blocks and that to random blocks. More memory is allocated to the type of blocks that can generate a higher hit rate so that the system-wide hit rate is improved. However, a key observation is unfortunately ignored here, that is, sequential blocks can be brought into the cache much faster than the same number of random blocks due to their spatial locality. Therefore, the misses of random blocks should count more in their contribution to final performance. In their marginal utility estimations, misses on the two types of blocks are equally treated without giving preference to random blocks even though the cost of fetching random blocks is much higher. Our DULO gives random blocks more weight for being kept in cache to compensate for their high fetching cost.

Because modern operating systems do not support caching and prefetching integration designs yet, we do not pursue this aspect in our DULO scheme in this article. We believe that introducing dual locality in these integration schemes will certainly improve their performance, and that it remains as future work to investigate the amount of its benefits.

[3]The LRU stack is the data structure used in the LRU replacement algorithm. The block ordering in it reflects the order of block accesses.

5.4 Other Related Work

The preliminary work for this article has been presented in Jiang et al. [2005] where DULO uses a coarse-grained timer to set timestamps of blocks. The timer ticks each time when the eviction section is refilled and makes the timestamps of the blocks unable to precisely reflect the order in which they are fetched from disk. For the blocks in the same sequence formed by the original algorithm in Jiang et al. [2005], it is possible that the accesses to these blocks are interrupted by other accesses and thus they should be grouped into separate sequences. In the article, we improve the algorithm by using a fine-grained timer, which ticks each time a block is fetched into memory to reflect the precise order of accesses. Moreover, the original algorithm forms sequences only for blocks contiguously located on the disk. We allow blocks sufficiently close to each other to be formed into same sequence. This enables more blocks to be formed into sequences and reflects the effect of read-ahead mechanism in disks. In this article, additional real-world applications are used to evaluate DULO's performance to represent a wider range of access patterns.

Because disk head seek time far dominates I/O data transfer time, to effectively utilize the available disk bandwidth, there are techniques to control the data placement on disk [Arpaci-Dusseau et al. 2003; Black et al. 1991] or reorganize selected disk blocks [Hsu et al. 2003] so that related objects are clustered and the accesses to them become more sequential. In DULO, we take an alternative approach in which we try to avoid random small accesses by preferentially keeping these blocks in cache and thereby making accesses more sequential. In comparison, our approach is capable of adapting itself to changing I/O patterns and is a more widely applicable alternative to the disk layout control approach.

Finally, we point out some interesting work analogous to our approach in spirit. Forney et al. [2002] considers the difference in performance across heterogeneous storage devices in storage-aware file buffer replacement algorithms which explicitly gives those blocks from slow devices higher weight to stay in cache. To do so, the algorithms can adjust the stream of block requests so that it contains more fast-device requests by filtering slow-device requests to improve caching efficiency. In Papathanasiou and Scott [2004], and Zhu et al. [2004a, 2004b], the authors propose adapting replacement algorithms or prefetching strategies to influence the I/O request streams for disk energy saving. With the customized cache filtering effect, the I/O stream to disks becomes more bursty or requests are separated with long idle times to increase disk power-down opportunities in the single disk case, or the I/O streams becomes more unbalanced among the requests' destination disks to allow some disks to have longer idle times to power down. All this work leverages the cache's buffering and filtering effects to influence I/O access streams and to make them friendly to particular performance characteristics of disks for their respective objectives which is the philosophy shared by DULO. The uniqueness of DULO is that it influences disk access streams to make them more sequential to reduce disk head seeks.

6. CONCLUSIONS

In this article, we identify a serious weakness in spatial locality exploitation in I/O caching and propose a new and effective memory management scheme, DULO, which can significantly improve I/O performance by exploiting both temporal and spatial localities. Our experiment results show that DULO can effectively reorganize applications' I/O request streams mixed with random and sequential accesses in order to provide a more disk-friendly request stream with high sequentiality of block accesses. We present an effective DULO replacement algorithm to carefully trade off random accesses with sequential accesses. We implemented DULO in a recent Linux kernel and tested it extensively using applications from different areas. The results of performance evaluation on both buffer cache and virtual memory systems show that DULO can significantly improve a system's I/O performance.

ACKNOWLEDGMENTS

We would like to thank Professor Xiaodong Zhang for his advice, suggestions, and support to this work.

REFERENCES

ALBERS, S. AND BUTTNER, M. 2003. Integrated prefetching and caching in single and parallel disk systems. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'03)*. ACM Press, New York, NY, 109–117.

ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., BURNETT, N. C., DENEHY, T. E., ENGLE, T. J., GUNAWI, H. S., NUGENT, J. A., AND POPOVICI, F. I. 2003. Transforming policies into mechanisms with infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM Press, New York, NY, 90–105.

BLACK, D. L., CARTER, J., FEINBERG, G., MACDONALD, R., MANGALAT, S., SHEINBROOD, E., SCIVER, J. V., AND WANG, P. 1991. OSF/1 virtual memory improvements. In *Proceedings of USENIX MACH Symposium*. 87–104.

BLAST. NCBI BLAST. URL:http://www.ncbi.nlm.nih.gov/BLAST/.

BUTT, A. R., GNIADY, C., AND HU, Y. C. 2005. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*. ACM Press, New York, NY, 157–168.

CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. 1995. A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev. 23,* 1, 188–197.

CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. 1996. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst. 14,* 4, 311–343.

CAO, P. AND IRANI, S. 1997. Cost-aware WWW proxy caching algorithms. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems (USITS'97)*. Monterey, CA.

DING, X., JIANG, S., CHEN, F., DAVIS, K., AND ZHANG, X. 2007. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proceedings of USENIX Annual Technical Conference (USENIX'07)*. USENIX Association.

FORNEY, B. C., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2002. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*. USENIX Association.

GILL, B. S. AND MODHA, D. S. 2005. SARC: Sequential prefetching in adaptive replacement cache. In *Proceedings of the USENIX Annual Technical Symposium*. USENIX Association.

GRIFFIOEN, J. AND APPLETON, R. 1994. Reducing file system latency using a predictive approach. In *Proceedings of USENIX Summer*. 197–207.

HSU, W. W., SMITH, A. J., AND YOUNG, H. C. 2003. The automatic improvement of locality in storage systems. Tech. rep. UCB/CSD-03-1264, EECS Department, University of California, Berkeley, CA.

IYER, S. AND DRUSCHEL, P. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. ACM Press, New York, NY, 117–130.

JIANG, S., CHEN, F., AND ZHANG, X. 2005. CLOCK-Pro: An effective improvement of the clock replacement. In *Proceedings of the Annual USENIX Technical Conference*.

JIANG, S., DING, X., CHEN, F., TAN, E., AND ZHANG, X. 2005. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST'05)*. USENIX Association.

JIANG, S. AND ZHANG, X. 2002. LIRS: An efficient low interreference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*. ACM Press, New York, NY, 31–42.

JOHNSON, T. AND SHASHA, D. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 439–450.

KAPLAN, S. F., MCGEOCH, L. A., AND COLE, M. F. 2002. Adaptive caching for demand prepaging. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM'02)*. ACM Press, New York, NY, 114–126.

KATCHER, J. 1997. PostMark: A new file system benchmark. Tech. rep., TR 3022, Network Appliance Inc.

KIMBREL, T., TOMKINS, A., PATTERSON, R. H., BERSHAD, B., CAO, P., FELTEN, E. W., GIBSON, G. A., KARLIN, A. R., AND LI, K. 1996. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*. ACM Press, New York, NY, 19–34.

KROEGER, T. M. AND LONG, D. D. E. 1996. Predicting file-system actions from prior events. In *Proceedings of the Annual USENIX Technical Conference.* 319–328.

KROEGER, T. M. AND LONG, D. D. E. 2001. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 105–118.

LEI, H. AND DUCHAMP, D. 1997. An analytical approach to file prefetching. In *Proceedings of the USENIX Annual Technical Conference*.

LOIZIDES, C. Journaling-filesystem fragmentation project–tool: Agesystem. URL: http://www.informatik.uni-frankfurt.de/ loizides/ reiserfs/ agesystem.html.

LXR. Linux cross-reference. URL: http://lxr.linux.no/.

MEGIDDO, N. AND MODHA, D. S. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*. USENIX Association, 115–130.

O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*. ACM Press, New York, NY, 297–306.

PAI, R., PULAVARTY, B., AND CAO, M. 2004. Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Linux Symposium*.

PAPATHANASIOU, A. E. AND SCOTT, M. L. 2004. Energy efficient prefetching and caching. In *Proceedings of the USENIX Annual Technical Conference*.

PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM Press, New York, NY, 79–95.

POZO, R. AND MILLER, B. 2000. Scimark 2.0. URL: http://math.nist.gov/scimark2/.

SCHLOSSER, S. W., SCHINDLER, J., PAPADOMANOLAKIS, S., SHAO, M., AILAMAKI, A., FALOUTSOS, C., AND GANGER, G. R. 2005. On multidimensional data and modern disks. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST'05)*.

TOMKINS, A., PATTERSON, R. H., AND GIBSON, G. 1997. Informed multi-process prefetching and caching. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97)*. ACM Press, New York, NY, 100–114.

TPC-H. 2006. TPC benchmark H – standard specification. URL: http://www.tpc.org.

TRENT, G. AND SAKE, M. 1995. WebSTONE: The first generation in HTTP server benchmarking. URL: http://www.mindcraft.com/webstone/paper.html.

YOUNG, N. E. 1998. Online file caching. In *Proceedings of the 9th annual SIAM Symposium on Discrete Algorithms (SODA'98)*. SIAM, Philadelphia, PA, 82–86.

ZHOU, Y., CHEN, Z., AND LI, K. 2004. Second-level buffer cache management. *IEEE Trans. Paral. Distrib. Syst. 15,* 6, 505–519.

ZHU, Q., DAVID, F. M., DEVARAJ, C. F., LI, Z., ZHOU, Y., AND CAO, P. 2004a. Reducing energy consumption of disk storage using power-aware cache management. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE Computer Society.

ZHU, Q., SHANKAR, A., AND ZHOU, Y. 2004b. PB-LRU: A self-tuning power aware storage cache replacement algorithm for conserving disk energy. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)*. ACM Press, New York, NY, 79–88.

Received February 2007; accepted March 2007