DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality

Song Jiang
Performance & Architecture Laboratory
Computer & Computational Sciences Div.
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
sjiang@lanl.gov

Xiaoning Ding, Feng Chen,
Enhua Tan and Xiaodong Zhang
Department of Computer Science and Engineering
Ohio State University
Columbus, OH 43210, USA
{dingxn, fchen, etan, zhang}@cse.ohio-state.edu

Abstract

Sequentiality of requested blocks on disks, or their spatial locality, is critical to the performance of disks, where the throughput of accesses to sequentially placed disk blocks can be an order of magnitude higher than that of accesses to randomly placed blocks. Unfortunately, spatial locality of cached blocks is largely ignored and only temporal locality is considered in system buffer cache management. 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 locality in 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 disk more sequential, significantly increasing the effectiveness of I/O scheduling and prefetching for disk performance improvements.

DULO has been extensively evaluated by both trace-driven simulations and a prototype implementation in Linux 2.6.11. In the simulations and system measurements, various application workloads have been tested, including Web Server, TPC benchmarks, and scientific programs. Our experiments show that DULO can significantly increase system throughput and reduce program execution times.

1 Introduction

A hard disk 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 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 limited 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 a stream of sequential blocks on the same track achieves a much higher disk throughput than that accessing several random blocks does.

In 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 disk. To minimize disk activities in the number of requested blocks, all the current replacement algorithms are designed by adopting 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 management. In the context of this paper, spatial locality specifically refers to the sequentiality of continuously requested blocks' disk placements.

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 re-

quest queue into a dispatching order that results in minimal seeks and thereafter maximal global disk throughput. Example schedulers include Shortest-Seek-Time-First (SSTF), CSCAN, as well as the Deadline and Anticipatory I/O schedulers [15] adopted in the current Linux kernel.

The third effort is prefetching. The data prefetching manager predicts the 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 continuously 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 schemes 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 buffer cache management. Hence, in the worst case, a stream filled with random accesses prevents I/O scheduling and prefetching from helping, because no spatial locality is left for them to exploit.

Concerned with the missing 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 locality, which we call the *DU*al *LO*cality scheme *DULO*. DULO introduces dual locality into the caching component in the OS by tracking and utilizing the disk placements of in-memory pages in buffer cache management[1]. Our objective is to maximize the sequentiality of I/O requests that are served 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 from applications so that more sequential block requests and less 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 Illustrating 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 mixed with sequential and random blocks. In 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 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 6.5 ms average seek time and 3.0 ms average rotation time are taken from the specification of the Hitachi Ultrastar 18ZX 10K RPM drive.

Table 1 shows the reference stream and the on-going 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*). In the 5th access, prefetching is activated and all the four sequential blocks are fetched because the prefetcher knows the reference (to block X1) starts at the beginning of the file. The difference in the cache states between the two schemes here is that *traditional* lists the blocks in the strict LRU order, while *dual* re-arranges 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 four random block disk request stream, and the total cost is 95.0 ms. For *dual*, it is {X1, X2, X3, X4} at the 13th access, a four sequential blocks, and the total cost is only 66.5 ms.

If we do not enable prefetching, the two schemes have the same number of misses, i.e., 16. With prefetching enabled, *traditional* has 10 misses, while *dual* has only 7 misses. This is because *dual* generates higher quality I/O requests (containing more sequential accesses) to provide more prefetching opportunities.

2.2 Challenges with Dual Locality

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

In current cache management, replacement algorithms only consider temporal locality (a position in the queue in the case of LRU) to make a replacement decision. While introducing spatial locality necessarily has to compromise

	Block Being Accessed	*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

Table 1: An example showing that a dual locality conscious scheme can be more effective than its traditional counterpart in improving disk performance. Fetched blocks are boldfaced. The MRU end of the queue is on the left.

the weight of temporal locality in the replacement decision, the role of temporal locality must be appropriately retained in the decision. In the example shown in Table 1, 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 have few re-accesses showing sufficient 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 only has a small number of blocks to amortize the almost fixed 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 how to make the tradeoff between temporal locality (recency) and spatial locality (sequence size) with the goal of maximizing disk performance.

3 The DULO Scheme

We now present our DULO scheme to exploit 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. A *sequence* is defined as a number of blocks whose disk locations are adjacent[3] and have been accessed during a limited time period. Because a sequence is formed from the blocks in a stack segment of limited size, and all blocks enter into the stack due to their references, the second condition of the definition is automatically satisfied. Specifically, a random block is a sequence of size 1. (2) Sorting the sequences in the LRU stack according to their recency (temporal locality) and size (spatial locality), with the objective that sequences of large recency and size are placed close to the LRU stack bottom. Because the recency of a sequence is changing while new sequences are being added, the order of the sorted sequence should be adjusted dynamically to reflect the change.

3.1 Structuring LRU stack

To facilitate the operations presented above, we partition the LRU stack into two sections (shown in Figure 1 as a vertically placed queue). The top part is called *staging section* used for admitting newly fetched blocks, and the bottom part is called *evicting section* used for storing sorted sequences to be evicted in their orders. We again divide the staging section into two segments. The first segment is called *correlation buffer*, and the second segment is called *sequencing bank*. The correlation buffer in DULO is similar to the *correlation reference period* used in the LRU-K replacement algorithm [26]. 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 size of the

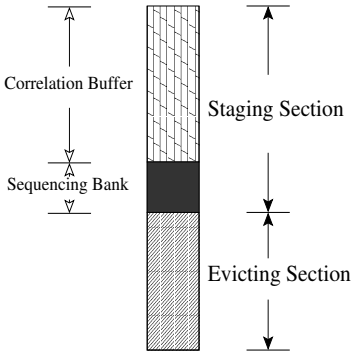

Figure 1: LRU stack is structured for the DULO replacement algorithm.

buffer is fixed. 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 evicting section holds the rest of the stack. When a block leaves the evicting section at its bottom 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 evicting section, the size of sequencing bank is *BANK_MAX*. We refill the evicting section by taking the blocks in the bank, forming sequences out of them, and inserting them into the evicting section in a desired order. There are three reasons for us to maintain two interacting sections and use the bank to conduct sequence forming: (1) The newly admitted blocks have a buffering area to be accumulated for forming potential sequences. (2) The sequences formed at the same time must share a common recency, because their constituent blocks are from the same block pool — 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 evicting section for a replacement order reflecting both their sequentiality and their recency.

3.2 *Block Table:* A Data Structure for Dual Locality

To complement the missing spatial locality in traditional caching systems, we introduce a data structure in the OS kernel called *block table*. The block table is analogous in structure to the multi-level page table used for process address translation. However there are clear differences between them because they serve different purposes: (1) The page table covers virtual address space of a process in the unit of page and page address is an index into the table, while the block table covers disk space in the unit of block, and block disk location is an index into the table. (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 page table lookup efficiency is much more demanding and performance-critical than that on the block table lookup efficiency because the former supports instruction execution while the latter facilitates I/O operations. That is the reason why a hardware TLB has to be used to expedite page table lookup, but there is no such a need for block table. (4) Each process owns a page table, while each disk drive owns a block table in memory.

In the system we set a global variable called *bank clock*, which ticks each time the bank in the staging section is used for forming sequences. Each block in the bank takes the current clock time as a timestamp representing its most recent access time. We then record the timestamp in an entry at the leaf level of the block table corresponding to the block disk location, which we called BTE (*Block Table Entry*). BTE is analogous in structure to PTE (*Page Table Entry*) of page table. Each BTE allows at most two most recent access times recorded in it. Whenever a new time is added, the oldest time is replaced if the BTE is full. In addition, to manage efficiently the memory space held by block table(s), a timestamp is set in each table entry at directory levels (equivalent to PGD (*Page Global Directory*) and PMD (*Page Middle Directory*) in the Linux page table). Each time the block table is looked up in a hierarchical way to record a new access time, the time is also recorded as a timestamp 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 entires when the table is viewed as a tree. The entries of the table are allocated in an on-demand fashion.

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

3.3 Forming Sequences

When it is the time to form sequences from a full bank, the bank clock is incremented by one. Each block in the bank then records the clock time as a new timestamp in the block table. After that, we traverse the table to collect all the sequences consisting of the blocks with the current clock time. This is a low cost operation because each directory at any level in a block table contains the most recent timestamp among all the BTEs under it. The traversal goes into only those directories containing the blocks in the bank. To ensure the stability of a sequence exhibited in history, the algorithm determines that the last block of a developing sequence should not be coalesced with the next block in the table if the

next block belongs to one of the following cases:

1. Its BTE shows that it was accessed in the current clock time. This includes the case where it has never been accessed (i.e., it has an empty timestamp field). It belongs to this case if the next block is a spare or defective block on the disk.

2. One and only one of the two blocks, the current block and the next block, was not accessed before the current clock time (i.e., it has only one timestamp).

3. Both of the two blocks have been accessed before the current clock time, but their last access times have a difference exceeding one.

4. The current sequence size reaches 128, which is the maximal allowed sequence size and we deem to be sufficient to amortize a disk operation cost.

If any one of the conditions is met, a complete sequence has been formed and a new sequence starts to be formed. Otherwise, the next block becomes part of the sequence, the following blocks will be tested continuously.

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 overhead associated with the DULO scheme. In the 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 evicting section is associated with a bank sequencing cost and a cost for sequence ordering in the evicting section. These additional costs that can incur in a system with few memory misses are unacceptable. In fact, the strict LRU algorithm is seldom used in real systems because of its overhead associated with every memory reference [18]. 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 *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 closely 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 evicting section refilling operations. While being compared with the miss penalty, these costs are very small.

The second issue is how sequences in the evicting section are ordered for replacement according to their temporal and spatial locality. We adopt an algorithm similar to

```
Initialize L = 0;
losses_of_evicting_section = 0;

/* Procedure to be invoked upon a reference
   to block b */

if b is in cache
   mark b as a young block;
else {
     while (block e at the stack bottom is young) {
       revoke the young block state;
       move it to the stack top;
       losses_of_evicting_section++;
       if (losses_of_evicting_section == BANK_MAX)
           refill_evicting_section();
     }
     replace block e at the stack bottom;
     s = e.sequence;
     L = H(s);
     losses_of_evicting_section++;
     if (losses_of_evicting_section == BANK_MAX)
         refill_evicting_section();
     place block b at the stack top as a young block
}

/* procedure to refill the evicting section */
refill_evicting_section()
{
  /* group sequences */
  for each block in sequencing bank
      place it in hierarchical block table;

  traverse the table to obtain all sequences;
  for each above sequence s
     H(s) = L + 1/size(s);
  sort the above sequences by H(s) into list L1;

  /* L2 is the list of sequences in evicting
     section */
  evicting_section = merge_sort(L1, L2);
  losses_of_evicting_section = 0;
}
```

Figure 2: The DULO Replacement Algorithm

GreedyDual-Size used in Web file caching [8]. GreedyDual-Size was originally derived from GreedyDual [37]. It makes its replacement decision by considering the recency, size, and fetching cost of cached files. It has been proven 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 file fetching cost is equivalent to the I/O operation cost for a sequence access. For sequences whose sizes are distributed in a reasonable range, which is limited by bank size, we currently assume their fetching cost is the same. Our algorithm can be modified to accommodate cost variance if necessary in the future.

The DULO algorithm associates each sequence with a value H, where a relatively small value indicates the sequence should be evicted first. The algorithm has a global inflation value L, which records the H value of the most

recent evicted sequence. When a new sequence s is admitted into the evicting 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. The sequences in the evicting 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 first. However, if a sequence of small size is not accessed for a relatively long time, it will be replaced. This is because a newly admitted long sequence could have a larger H value due to the L value, which is continuously being inflated by the 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 evicting 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 evicting section, we complete the refilling of the evicting section, and the staging section ends up with only the correlation buffer. The algorithm is described using pseudo code in Figure 2.

4 Performance Evaluation

We use both trace-driven simulations and a prototype implementation to evaluate the DULO scheme and to demonstrate the impact of introducing spatial locality into replacement decisions on different access patterns in applications.

4.1 The DULO Simulation

4.1.1 Experiment Settings

We built a simulator that implements the DULO scheme, Linux prefetching policy [28], and Linux Deadline I/O scheduler [30]. We also interfaced the Disksim 3.0, an accurate disk simulator [4], to simulate the disk behaviors. The disk drive we modeled is the Seagate ST39102LW with 10K RPM and 9.1GB capacity. Its maximum read/write seek time is 12.2/13.2ms, and its average rotation time is 2.99ms. We selected five traces of representative I/O request patterns to drive the simulator (see Table 2). The traces have also been used in [5], where readers are referred for their details. Here we briefly describe these traces.

Trace *viewperf* consists of almost all-sequential-accesses. The trace was collected by running SPEC 2000 benchmark viewperf. In this trace, over 99% of its references are to consecutive blocks within a few large files. By contrast, trace *tpc-h* consists of almost all-random-accesses. The trace was collected when the TPC-H decision support benchmark runs on the MySQL database system. TPC-H performs random references to several large database files, resulting in only 3% references to concecutive blocks in the trace.

The other three traces have mixed I/O request patterns. Trace *glimpse* was collected by using the indexing and query tool "glimpse" to search for text strings in the text files under the */usr* directory. Trace *multi1* was collected by running programs *cscope*, *gcc*, and *viewperf* concurrently. *Cscope* is a source code examination tool, and *gcc* is a GNU compiler. Both take Linux kernel 2.4.20 source code as their inputs. *Cscope* and *glimpse* have a similar access pattern. They contain 76% and 74% sequential accesses, respectively. Trace *multi2* was collected by running programs *glimpse* and *tpc-h* concurrently. *Multi2* has a lower sequential access rate than *Multi1* (16% vs. 75%).

In the simulations, we set the sequencing bank size as 8MB, and evicting section size as 64MB in most cases. Only in the cases where the demanded memory size is less than 80MB (such as for *viewperf*), we set the sequencing bank size as 4MB, and evicting section size as 16MB. These choices are based on the results of our parameter sensitivity studies to be presented in Section 4.1.3. In the evaluation, we compare the DULO performance with that of the CLOCK algorithm. For generality, we still refer it as LRU.

4.1.2 Evaluation Results

Figures 3 and 4 show the execution times, hit ratios, and disk access sequence size distributions of the LRU caching and DULO caching schemes for the five workloads when we vary memory size. Because the major effort of DULO to improve 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 differences for workloads running on the LRU caching scheme and on the DULO caching scheme. For this purpose, we use CDF curves to show how many percentages (shown on Y-axis) of requested blocks appear in the sequences whose sizes are less than a certain threshold (shown on X-axis). For each trace, we select two memory sizes to draw the corresponding CDF curves for LRU and DULO, respectively. We select the memory sizes according to the execution time gaps between LRU and DULO shown in execution time figures — 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.

First, examine Figure 3. The CDF curves show that for the almost-all-sequential workload *viewperf*, more than 80% of requested blocks are in the sequences whose sizes are larger than 120. Though DULO can increase the sizes of short sequences a little bit, and hence reduce execution time by 4.4% (up to 8.0%), its influence is limited. For the almost-all-random workload *tpc-h*, apparently DULO cannot create sequential disk requests from the application requests consisting of pure random blocks. So we see almost no improve-

Application	Num of block accesses (M)	Aggregate file size(MB)	Num of files	sequential refs
viewperf	0.3	495	289	99%
tpc-h	13.5	1187	49	3%
glimpse	3.1	669	43649	74%
multi1 *(cscope+gcc+viewperf)*	1.6	792	12514	75%
multi2 *(glimpse+tpc-h)*	16.6	1855	43696	16%

Table 2: Characteristics of the traces used in the simulations

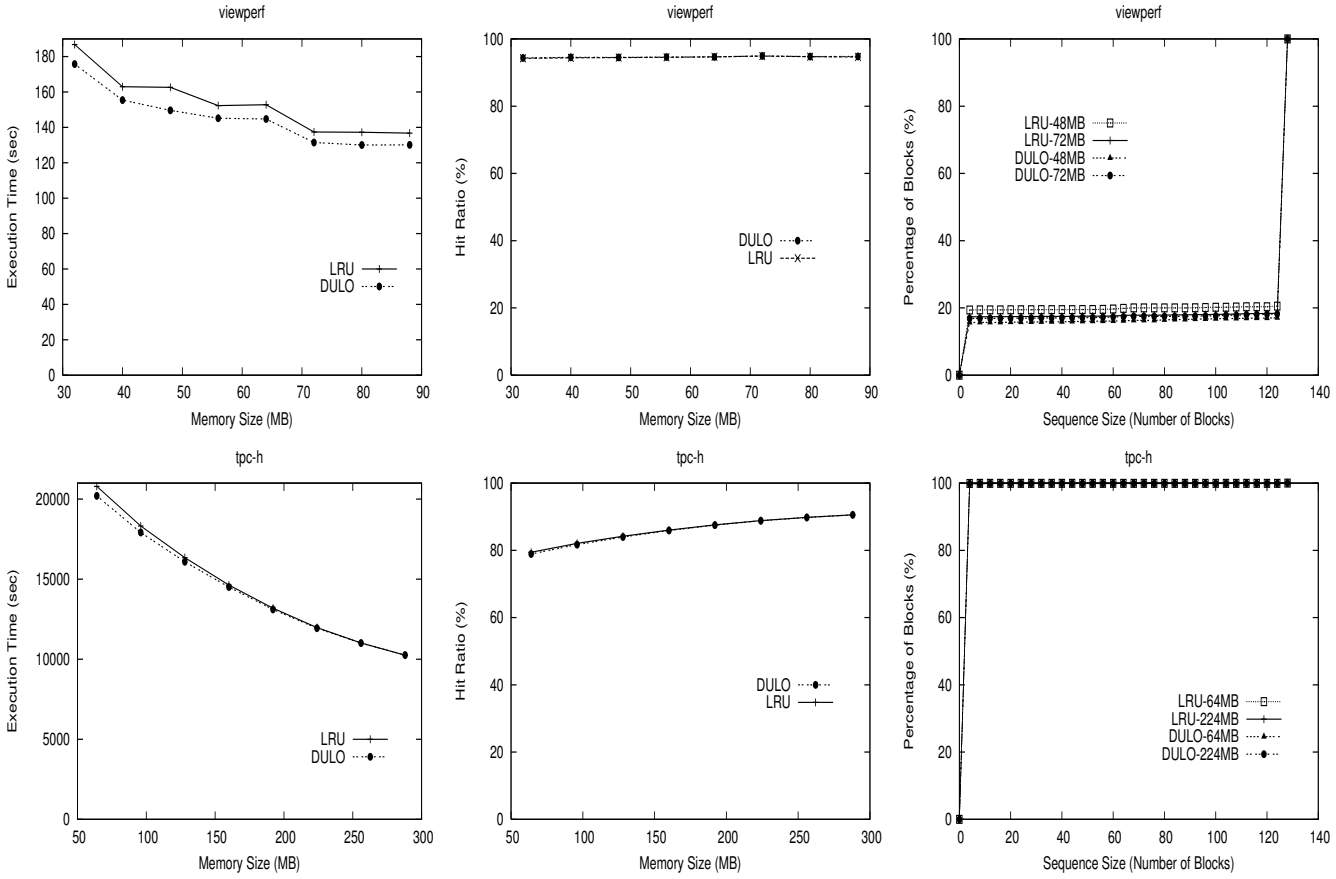

Figure 3: Execution times, hit ratios, and disk access sequence size distributions (CDF curves) of LRU caching and DULO caching schemes for *viewperf* with sequential request pattern and *tpc-h* with random request pattern.

ments from DULO.

DULO achieves substantial performance improvements for the workloads with mixed request patterns (see Figure 4). There are several observations from the figures. First, the sequence size increases are directly correlated to the execution time and hit ratio improvements. Let us take *multi1* as an example, with the memory size of 80MB, DULO makes 16.1% requested blocks appear in the sequences whose sizes are larger than 40 compared with 13.7% for LRU. Accordingly, there is an 8.5% execution time reduction and a 3.8% hit ratio increase. By contrast, with the memory size of 160MB, DULO makes 24.9% requested blocks appear in the sequences whose sizes are larger than 40 compared with 14.0% for LRU. Accordingly, there is a 55.3% execution time reduction and a 29.5% hit ratio increase. The correlation clearly indicates that requested sequence size is a critical factor affecting disk performance and DULO makes its contributions through increasing the sequence size. DULO can increase the hit ratio by making prefetching more effective with long sequences and generating more hits on the prefetched blocks. Second, the sequential accesses are important for DULO to leverage the buffer cache filtering effect. We see that DULO does a better job for *glimpse* and *multi1* than for *multi2*. We know that *glimpse* and *multi1* have 74% and 75% of sequential accesses, while *multi2* has only 16% sequential accesses. The small portion of sequential accesses in *multi2* make DULO less capable to keep random blocks from being replaced because there are not sufficient sequentially accessed blocks to be replaced first. Third, *multi1* has more pronounced performance improve-

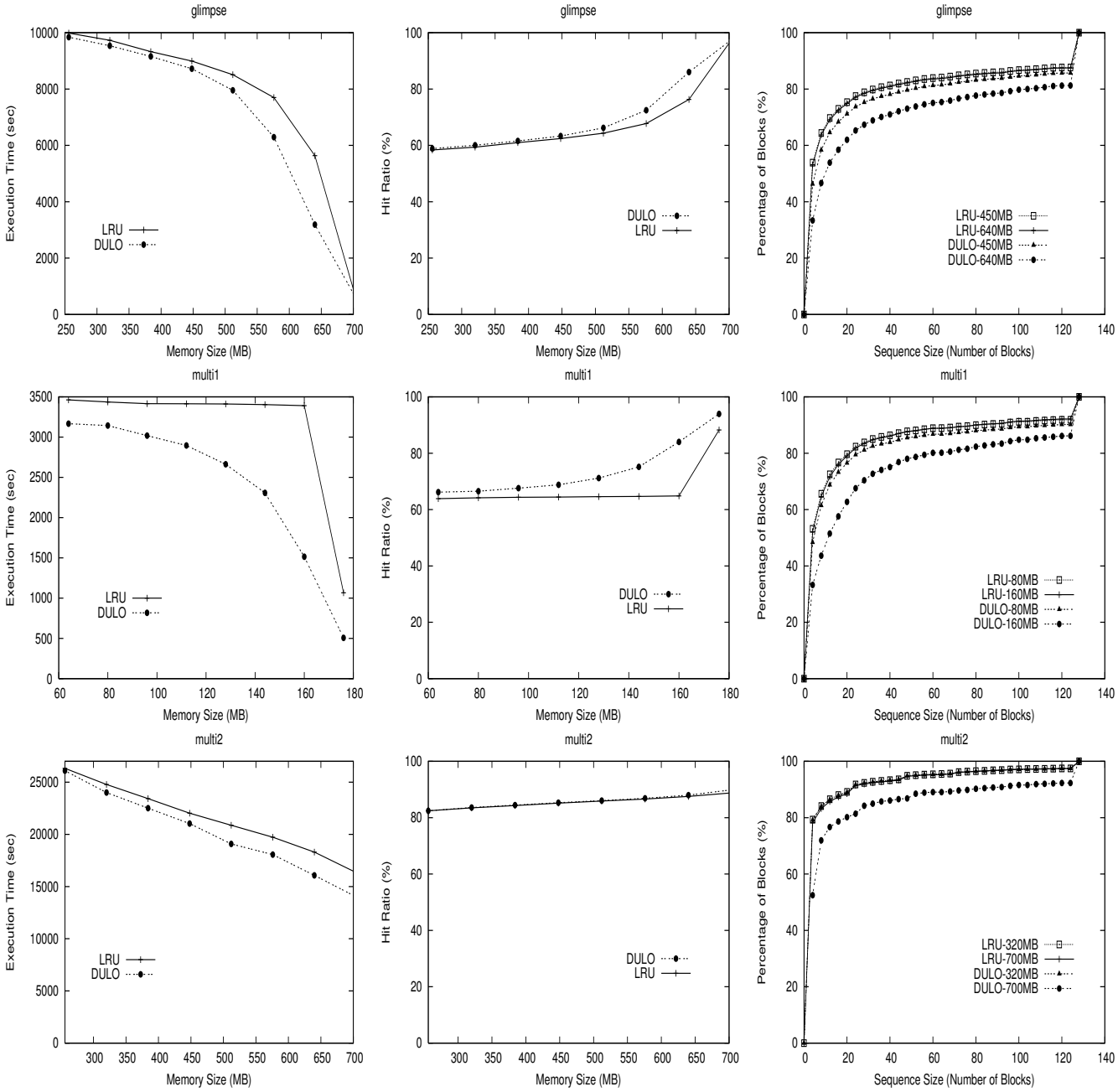

Figure 4: Execution times, hit ratios, and disk access sequence size distributions (CDF curves) of LRU caching and DULO caching schemes for *glimpse*, *multi1*, and *multi2* with mixed request patterns.

ments from DULO than *glimpse* does. This difference is mainly because DULO incidentally improves the LRU hit ratios by better exploiting temporal locality with the looping access pattern, for which LRU has well well-known inability (see e.g. [17]). By contrast, in the case of *multi2*, DULO can hardly improve its hit ratios, but is able to considerably reduce its execution times, which clearly demonstrates its effectiveness at exploiting spatial locality.

4.1.3 Parameter Sensitivity and Overhead Study

There are two parameters in the DULO scheme, the (maximum) sequencing bank size and the (minimal) evicting size. Both of these sizes should be related to the workload access patterns rather than memory size, because they are used to manage the sequentiality of accessed blocks. We use four workloads for the study, excluding *viewperf* because its memory demand is very small.

Table 3 shows the execution times change with varying

Bank Size (MB)	*tpc-h* (128M)	*glimpse* (640M)	*multi1* (160M)	*multi2* (640M)
1	16325	3652	1732	18197
2	16115	3611	1703	17896
4	15522	3392	1465	16744
8	15698	3392	1483	16737
16	15853	3412	1502	17001
32	15954	3452	1542	17226

Table 3: The execution times (seconds) with varying bank sizes (MB). Evicting section size is 64MB. Memory sizes are shown with their respective workload names.

Evicting Section Size (MB)	*tpc-h* (128M)	*glimpse* (640M)	*multi1* (160M)	*multi2* (640M)
32	15750	3852	1613	18716
64	15698	3392	1483	16737
128	-	3382	1406	16685
192	-	3361	-	16665
256	-	3312	-	16540
320	-	3342	-	16538

Table 4: The execution times (seconds) with varying evicting section sizes (MB). Sequencing bank size is 8MB. Memory sizes are shown with their respective workload names.

sequencing bank sizes. We observe that across the workloads with various access patterns, there is an optimal bank size roughly in the range from 4MB to 16MB. This is because a bank with too small size has little chance to form long sequences. Meanwhile, a bank size must be less than the evicting section size. When the bank size approaches the section size, the performance will degrade. This is because a large bank size causes the evicting section to be refilled too late and causes the random blocks in it to have to be evicted. So in our experiments we choose 8MB as the bank size.

Table 4 shows the execution times change with varying evicting section sizes. Obviously the larger the section size, the more control DULO will have on the eviction order of the blocks in it, which usually means better performance. The figure does show the trend. Meanwhile, the figure also shows that the benefits from the increased evicting section size saturate once the size exceeds the range from 64MB to 128MB. In our experiments, we choose 64MB as the section size because the memory demands of our workloads are relatively small.

The space overhead of DULO is its block table, whose size growth 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 it is associated with the misses. Our simulations show that a miss is associated with one block sequencing operation including placing the block into the block table and comparing with its adjacent blocks, and 1.7 merge sort comparison operation in average.

4.2 The DULO Implementation

To demonstrate the performance improvements of DULO for applications running on a modern operating system, we implement DULO in the recent Linux kernel 2.6.11. One of the unique benefits from real system evaluation over trace simulation is that it can take all the memory usages into account, including process memory and file-mapped memory pages. For example, due to time and space cost constraints, it is almost impossible to faithfully record all the memory page accesses as a trace. Thus, the traces we used in the simulation experiments only reflect the file access activities through system calls. To present a comprehensive evaluation of DULO, our kernel implementation and system measurements effectively complement our trace simulation results.

Let us start with a brief description of the implementation of the Linux replacement policy, an LRU variant.

4.2.1 Linux Caching

Linux adopts an LRU variant similar to the 2Q replacement [16]. The Linux 2.6 kernel groups all the process pages and file pages into two LRU lists called the *active list* and the *inactive list*. As their names indicate, the active list is used to store recently actively 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 inactive list tail. An active page is demoted to the inactive list if it is determined to have not been recently accessed (by *refill_inactive_zone()*).

4.2.2 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, which has to tolerate some compromises of the original DULO design, to serve the purpose of demonstrating what improvements a dual locality consideration could bring to an existing spatial-locality-unaware system without changing its basic underlying replacement policy.

In Linux, we partition the inactive list into a staging section and an evicting section because the list is the place where new blocks are added and old blocks are replaced. Two LRU lists used in Linux instead of one assumed in the DULO scheme challenge the legitimacy of forming a sequence by using one bank in the staging section. We know

that the sequencing bank in DULO is intended to collect continuously accessed pages and form sequences from them, so that the pages in a sequence can be expected to be requested together and be fetched sequentially. With two lists, both newly accessed pages and not recently accessed active pages demoted from the active list might be added into the inactive list and probably be sequenced in the same bank[4]. Hence, two spatially sequential but temporally remote pages can possibly be grouped into one sequence, which is apparently in conflict with the sequence definition described at the beginning of Section 4. We address this issue by marking the demoted active pages and sequencing each type of page separately. Obviously, the Linux two-list structure provides fewer opportunities for DULO to identify sequences than those in one stack case where any hit pages are available for a possible sequencing with faulted-in pages.

The anonymous pages that do not yet have mappings on disk are treated as random blocks until they are swapped out and are associated with some disk locations. To map the LBN (Logical Block Number) of a block into a one-dimensional physical disk address, we use the technique described in [33] to extract track boundaries. To characterize accurately block location sequentiality, all the defective and spare blocks on disk are counted. We also artificially place a dummy block between the blocks on a track boundary in the mapping to show the two blocks are non-sequential.

There are two types of I/O operations, namely file access and VM paging. In the experiments, we set the sequencing bank size as 8MB, and the evicting section size as 64MB, the same as those adopted in the simulations.

4.2.3 Case Study I: File Accesses

In the first case we study the influence of the DULO scheme on file access performance. For this purpose, we installed a Web server running a general hypertext cross-referencing tool — Linux Cross-Reference (LXR) [24]. This tool is widely used by Linux developers for searching Linux source code. The machine we use is a Gateway desktop with Intel P4 1.7GHz processor, a 512MB memory, and Western Digital WD400BB hard disk of 7200 RPM. The OS is SuSE Linux 9.2 with the Linux 2.6.11 kernel. The file system is Ext2. We use LXR 0.3 search engine on the Apache 2.0.50 HTTP Server, and use Glimpse 4.17.3 as the freetext search engine. The file set for the searching is Linux 2.6.11.9 source code, whose size is 236MB. Glimpse divides the files into 256 partitions, indexes the file set based on partitions, and generates a 12MB index file showing the keyword locations in terms of partitions. To serve a search query, glimpse searches the index file first, then accesses the files included in the partitions matched in the index files. On the client side, we used WebStone 2.5 [36] to generate 25 clients concurrently submitting freetext search queries. Each client randomly picks up a keyword from a pool of 50 keywords and sends it to the server. It sends its next query request once it receives the results of its previous query. We randomly select 25 Linux symbols from file */boot/System.map* and another 25 popular OS terms such as *"lru", "scheduling", "page"* as the pool of candidate query keywords. Each experiment lasts for 15 minutes. One client always uses the same sequence of keyword queries in each experiment. The metric we use is the query system throughput represented by MBits/sec, which means the number of Mega bits of query results returned by the server per second. This metric is also used for reporting WebStone benchmark results. Because in the experiments the query processing is I/O-intensive, the metric is suitable to measure the effectiveness of the memory and disk systems.

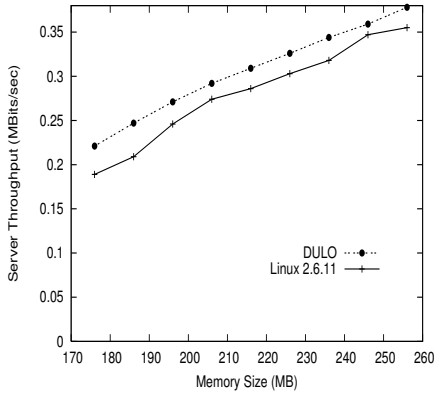

Figure 5: Throughputs of LXR search on the original Linux kernel and DULO instrumented kernel with varying memory sizes.

Figure 5 shows the server throughputs for the original Linux 2.6.11 kernel and its DULO instrumented counterpart with various memory sizes. The reported memory sizes are those available for the kernel and user applications. We adopt relatively small memory sizes because of the limited size of the file set for search. The figure shows that DULO helps improve the benchmark throughput by 5.1% to 16.9%, and the trend also holds for the hit ratio curves (not shown in this paper). To understand the performance improvements, we examine the disk layout of the glimpse partitions (i.e., the sets of files the glimpse searches for a specific keyword). There are a small percentage of files in a partition that are located non-continuously with the rest of its files. The fact that a partition is the glimpse access unit makes accesses to those files be random accesses interleaved in the sequential accesses. DULO identifies these isolated files and keeps them in memory with priority. Then the partition can be scanned without abruptly moving the disk head, even if the partition contains isolated small files. To prepare the aforementioned experiments, we *untar* the compressed kernel 2.6.11.9 on the disk with 10% of its capacity occupied. To verify our performance explanation, we manually copy the source code files to an unoccupied disk and make all the files in a glimpse partition be closely allocated on the disk. Then we repeat the experiments. This time, we see little difference between the

DULO instrumented kernel and the original kernel, which clearly shows that (1) DULO can effectively and flexibly exploit spatial locality without carefully tuning system components, which is sometimes infeasible; (2) The additional running overhead introduced by DULO is very small.

4.2.4 Case Study II: VM Paging

In the second case we study the influence of the DULO scheme on VM paging performance. For this purpose, we use a typical scientific computing benchmark — sparse matrix multiplication (SMM) from an NIST benchmark suite SciMark2 [31]. The system settings are the same as those adopted in the previous case study. The SMM benchmark multiplies a sparse matrix with a vector. The matrix is of size $N \times N$, and has M non-zero data regularly dispersed in its data geometry, while the vector has a size of N ($N = 2^{20}$ and $M = 2^{23}$). The data type is 8Byte *double*. In the multiplication algorithm, the matrix is stored in a compressed-row format so that all the non-zero elements are continuously 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 116MB. To cause the system paging and stress the swap space accesses, we have to adopt small memory sizes, from 90MB to 170MB, including the memory used by the kernel and applications. The benchmark is designed to repeat the multiplication computation 15 times to collect data.

To increase spatial locality of swapped-out pages in disk swap space, Linux tries to allocate continuous swap slots on disk to sequentially reclaimed anonymous pages in the hope that they would be swapped-in in the same order efficiently. However, the data access pattern in SMM foils the system effort. In the program, SMM first initializes the arrays one by one. This thereafter causes each array to be swapped out continuously and be 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 continuously located on the disk. The swap-in accesses to the vector arrays turn into random accesses, while the elements of matrix elements are still sequentially accessed. This explains the SMM execution time differences between on the original kernel and on DULO instrumented kernel (see Figure 6). DULO significantly reduces the execution times by up to 43.7%, which happens when the memory size is 135MB. This is because DULO detects the random pages in the vector array and caches them with priority. Because the matrix is a sparse one, the vector array cannot obtain sufficiently frequent reuses to allow the original kernel to keep them from being paged out. Furthermore, the similar execution times between the two kernels when there is enough memory to hold the working set shown in the figure suggest that the DULO overhead is small.

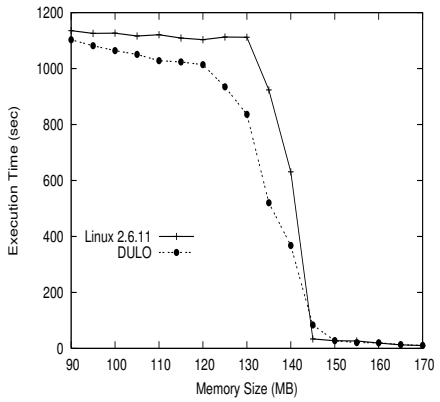

Figure 6: SMM execution times on the original Linux kernel and DULO instrumented kernel with varying memory sizes.

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 area 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 re-accessed 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 [16], MQ [38], ARC [25], LIRS [17] et al. 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 page fault rate. None of these algorithms considers spatial locality, i.e., how the stream of faulted pages is related to their disk locations. Because of the non-uniform 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 [5, 28]. System-wide file access history has been used in probability-based predicting algorithms, which track sequences of file access events and evaluate probability of file occurring in the sequences [11, 19, 20, 23]. 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 coincide with sequential block requests. 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 resulted 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 use of application-disclosed hints in the integration [2, 6, 12, 21, 7, 22, 27, 35]. Sharing the same weakness as those in current caching policies, this research only utilizes the temporal locality of the cached/prefetched blocks and uses hit ratio as metric in 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 hit ratio [5].

Most of the proposed integration schemes rely on application-level hints about I/O access patterns provided by users [6, 7, 22, 27, 35]. 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 [21, 12] does not require additional user support and thus is more related to our DULO design.

In paper [21], 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[5]. It takes a reasonable assumption — 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 multi-process environment, where multiple processes that access different files interleavingly feed their blocks into the common LRU stack. Prefetching requests involving disk seeks make little sense to improving I/O performance, and can hurt the performance due to possible wrong predictions. If we re-order 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 paper [12], 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 compare constantly 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, i.e., sequential blocks can be brought into the cache much faster than an equivalent 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.

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

5.4 Other Related Work

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 [1, 3] or reorganize selected disk blocks [14], 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 more accesses 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. [10] considers the difference in performance across heterogeneous storage devices in storage-aware file buffer replacement algorithms, which explicitly give those blocks from slow devices higher weight to stay in cache. To do so, the algorithms can adjust the stream of block requests to have more fast device requests by filtering slow device requests to improve caching efficiency. In papers [29, 39, 40], the authors propose to adapt 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 separated by long idle time to increase disk power-down opportunities in the single disk case, or 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 specific performance characteristics of disks for their respective objectives, which is the philosophy shared by our DULO. The uniqueness of DULO is that it influences disk access streams to make them more sequential to reduce disk head seeks.

6 Limitations of our Work

While the DULO scheme exhibits impressive performance improvements in average disk latency and bandwidth, as well as the application runtimes, there are some limitations worth pointing out. First, though DULO attempts to provide random blocks with a caching privilege to avoid the expensive I/O operations on them, there is little that DULO can do to help I/O requests incurred by compulsory misses or misses to random blocks that have not been accessed for a long time. In addition, the temporal-locality-only caching policy is able to cache frequently accessed random blocks, and there is no need for DULO's help. This discussion also applies to those short sequences whose sizes cannot well amortize the disk seeking cost. Second, we present our DULO scheme based on the LRU stack. For implementation purposes, we adapt it to the 2Q-like Linux replacement policy. The studies of how to adapt DULO on other advanced caching algorithms and understanding the interaction between DULO and the characteristics of each algorithm are necessary and in our research plan. Third, as we have mentioned, it remains as our future work to study the integration between caching and prefetching polices in the DULO scheme.

7 Conclusions

In this paper, we identify a serious weakness of lacking 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 locality. Our experiment results show that DULO can effectively reorganize application 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 tradeoff random accesses with sequential accesses and evaluate it using traces representing representative access patterns. Besides extensive simulations, we have also implemented DULO in a recent Linux kernel. The results of performance evaluation on both buffer cache and virtual memory system show that DULO can significantly improve the performance up to 43.7%.

Acknowledgment

We are grateful to Ali R. Butt, Chris Gniady, and Y. Charlie Hu at Purdue University for providing us with their file I/O traces. We are also grateful to the anonymous reviewers who helped improve the quality of the paper. We thank our colleagues Bill Bynum, Kei Davis, and Fabrizio Petrini to read the paper and their constructive comments. The research was supported by Los Alamos National Laboratory under grant LDRD ER 20040480ER, and partially supported by the National Science Foundation under grants CNS-0098055, CCF-0129883, and CNS-0405909.

References

[1] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, F. I. Popovici, "Transforming Policies into Mechanisms with Infokernel", *Proc. of SOSP '03*, October 2003.

[2] S. Albers and M. Buttner, "Integrated prefetching and caching in single and parallel disk systems", *Proc. of SPAA '03*, June, 2003.

[3] D. Black, J. Carter, G. Feinberg, R. MacDonald, S. Mangalat, E. Sheinbrood, J. Sciver, and P. Wang, "OSF/1 Virtual Memory Improvements", *Proc. of the USENIX Mac Symposium*, November 1991.

[4] J. Bucy and G. Ganger, "The DiskSim Simulation Environment Version 3.0 Reference Manual", *Technical Report CMU-CS-03-102, Carnegie Mellon University*, January 2003.

[5] A. R. Butt, C. Gniady, and Y. C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms", *Proc. of SIGMETRICS '05*, June, 2005.

[6] P. Cao, E. W. Felten, A. Karlin and K. Li, "A Study of Integrated Prefetching and Caching Strategies", *Proc. of SIGMETRICS '95*, May 1995.

[7] P. Cao, E. W. Felten, A. Karlin and K. Li, "Implementation and Performance of Integrated Application-Controlled

Caching, Prefetching and Disk Scheduling", *ACM Transaction on Computer Systems*, November 1996.

[8] P. Cao, S. Irani, "Cost-Aware WWW Proxy Caching Algorithms", *Proc. of USENIX '97*, December, 1997.

[9] P. Cao, E. W. Felten and K. Li, "Application-Controlled File Caching Policies", *Proc. of USENIX Summer '94*, 1994.

[10] B. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Storage-Aware Caching: Revisiting Caching For Heterogeneous Storage Systems", *Proc. of FAST '02*, January 2002.

[11] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach", *Proc. of USENIX Summer '94*, June 1994.

[12] B. Gill and D. S. Modha, "SARC: Sequential Prefetching in Adaptive Replacement Cache," *Proc. of USENIX '05*, April, 2005.

[13] M. Gorman, "Understanding the Linux Virtual Memory Manager", *Prentice Hall*, April, 2004.

[14] W. W. Hsu, H. C. Young and A. J. Smith, "The Automatic Improvement of Locality in Storage Systems", *Technical Report CSD-03-1264, UC Berkeley*, July, 2003.

[15] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O", *Proc. of SOSP '01*, October 2001.

[16] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm", *Proc. of VLDB '94*, 1994, pp. 439-450.

[17] 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.

[18] S. Jiang, F. Chen and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement", *Proc. of USENIX '05*, April 2005.

[19] T. M. Kroeger and D.D.E. Long, "Predicting file-system actions from prior events", *Proc. of USENIX Winter '96*, January 1996.

[20] T. M. Kroeger and D.D.E. Long, "Design and implementation of a predictive file prefetching algorithm", *Proc. of USENIX '01*, January 2001.

[21] S. F. Kaplan, L. A. McGeoch and M. F. Cole, "Adaptive Caching for Demand Prepaging", *Proc. of the International Symposium on Memory Management*, June, 2002.

[22] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felton, G. Gibson, A. R. Karlin and K. Li, "A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching", *Proc. of OSDI '96*, 1996.

[23] H. Lei and D. Duchamp, "An Analytical Approach to File Prefetching", *Proc. USENIX '97*, January 1997.

[24] Linux Cross-Reference, *URL : http://lxr.linux.no/*.

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

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

[27] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, "Informed Prefetching and Caching", *Proc. of SOSP '95*, 1995.

[28] R. Pai, B. Pulavarty and M. Cao, "Linux 2.6 Performance Improvement through Readahead Optimization", *Proceedings of the Linux Symposium*, July 2004.

[29] A. E. Papathanasiou and M. L. Scott, "Energy Efficient Prefetching and Caching", *Proc. of USENIX '04*, June, 2004.

[30] R. Love, "Linux Kernel Development (2nd Edition)", *Novell Press*, January, 2005.

[31] SciMark 2.0 benchmark, *URL: http://math.nist.gov/scimark2/*

[32] A. J. Smith, "Sequentiality and Prefetching in Database Systems", *ACM Trans. on Database Systems*, Vol. 3, No. 3, 1978.

[33] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger, "Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics", *Proc. of FAST '02*, January, 2002.

[34] E. Shriver, C. Small, K. A. Smith, "Why Does File System Prefetching Work?", *Proc. of USENIX '99*, June, 1999.

[35] A. Tomkins, R. H. Patterson and G. Gibson, "Informed Multi-Process Prefetching and Caching", *Proc. of SIGMETRICS '97*, June, 1997.

[36] WebStone — The Benchmark for Web Servers, *URL : http://www.mindcraft.com/benchmarks/webstone/*

[37] N. Young, "Online file caching", *Proc. of SODA '98*, 1998.

[38] 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.

[39] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao, "Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management", *Proc. of HPCA '04*, February 2004.

[40] Q. Zhu, A. Shankar and Y. Zhou, "PB-LRU: A Self-Tuning Power Aware Storage Cache Replacement Algorithm for Conserving Disk Energy", *Proc. of ICS '04*, June, 2004

Notes

[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 is then composed of one or multiple blocks if it has a disk mapping.

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

[3] The definition of sequence can be easily extended to a cluster of blocks whose disk locations are close to each other and can be used to amortize the cost of one disk operation. We limit the concept to being strictly sequential in this paper because that is the dominant case in practice.

[4] This issue might not arise if the last timestamps of these two types of blocks cannot meet the sequencing criteria listed in section 3.3, but there is no guarantee of this.

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