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

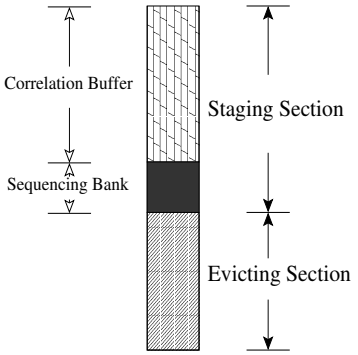


Figure 1: LRU stack is structured for the DULO replacement algorithm.

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

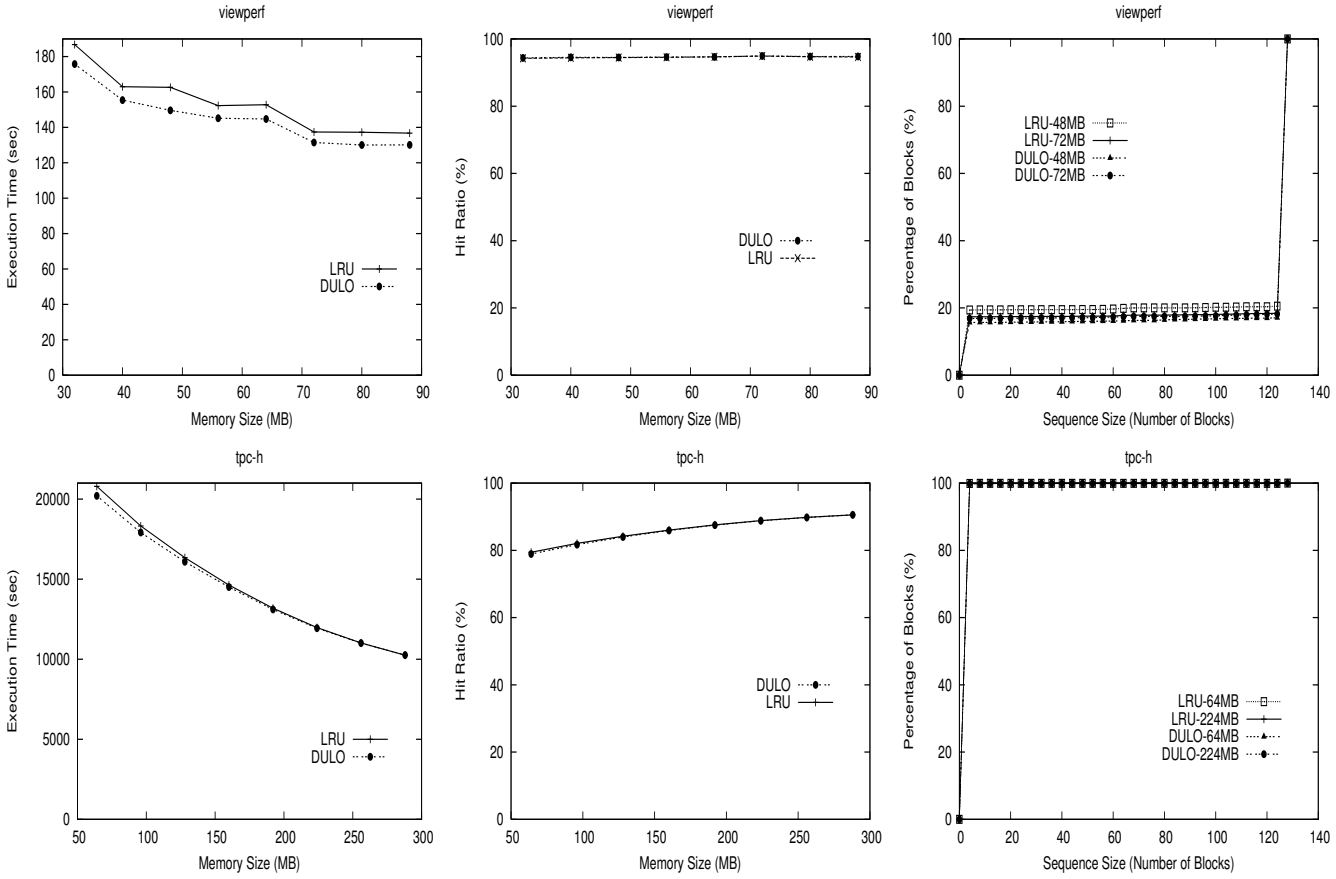


Figure 3: Execution times, hit ratios, and disk access sequence size distributions (CDF curves) of LRU caching and DULO caching schemes for *viewperf* with sequential request pattern and *tpc-h* with random request pattern.

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-

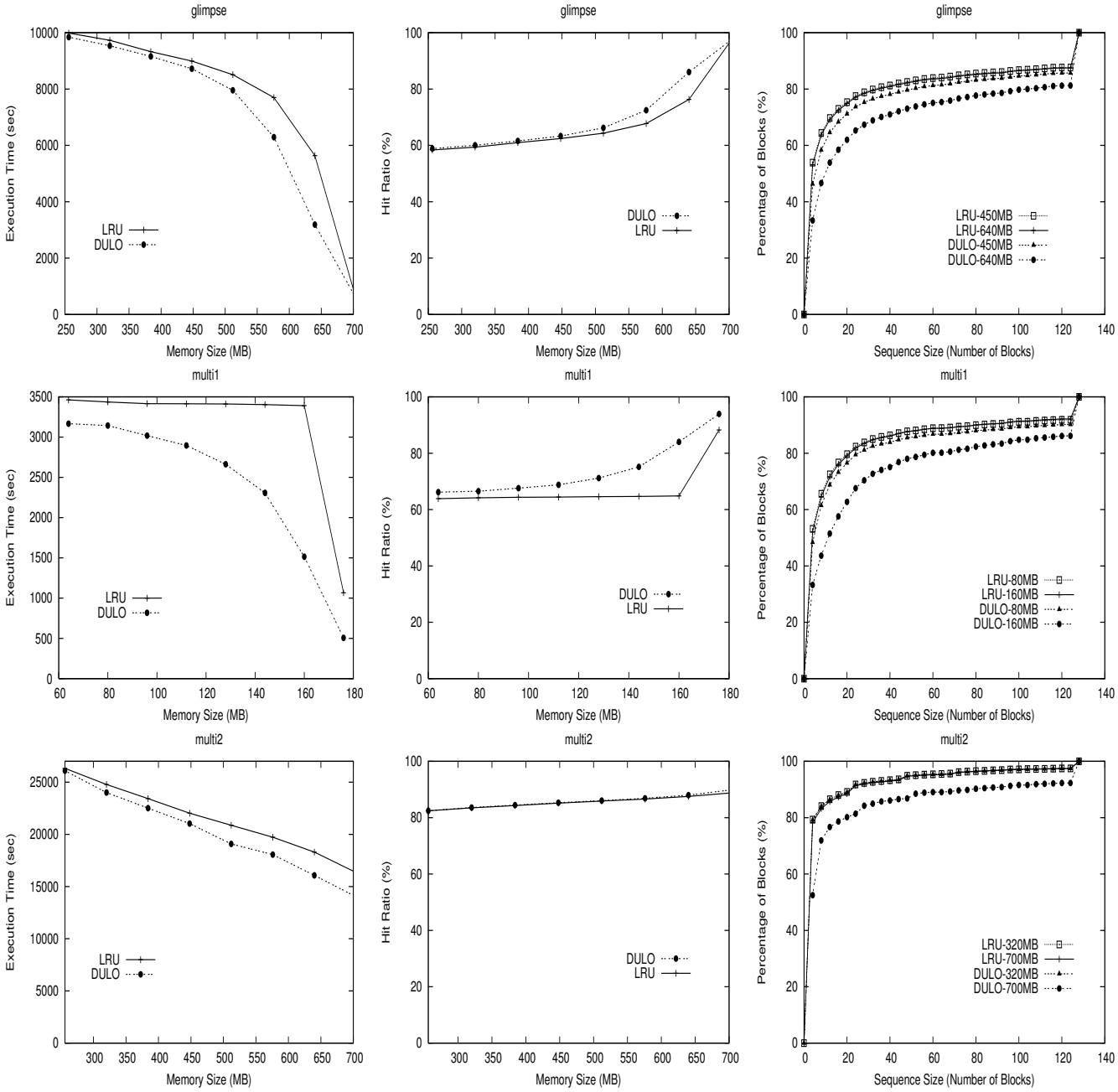


Figure 4: Execution times, hit ratios, and disk access sequence size distributions (CDF curves) of LRU caching and DULO caching schemes for *glimpse*, *multi1*, and *multi2* with mixed request patterns.

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.

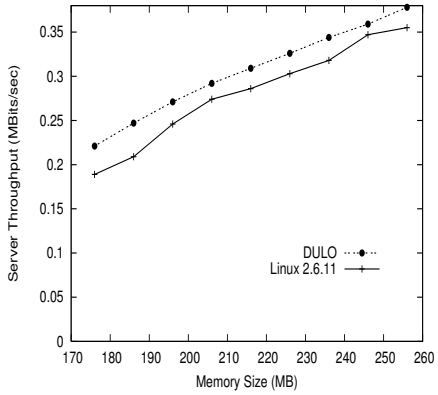


Figure 5: Throughputs of LXR search on the original Linux kernel and DULO instrumented kernel with varying memory sizes.

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.

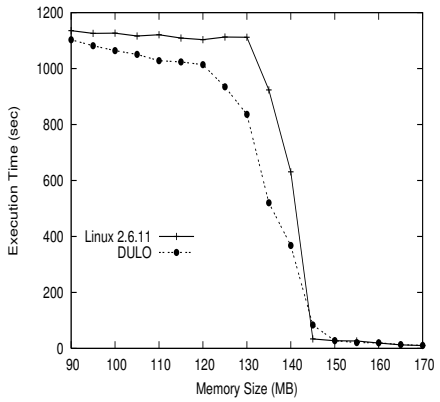


Figure 6: SMM execution times on the original Linux kernel and DULO instrumented kernel with varying memory sizes.

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.

CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives

Feng Chen* Tian Luo Xiaodong Zhang

Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210, USA
{*fchen,luot,zhang*}*@cse.ohio-state.edu*

Abstract

Although Flash Memory based Solid State Drive (SSD) exhibits high performance and low power consumption, a critical concern is its limited lifespan along with the associated reliability issues. In this paper, we propose to build a Content-Aware Flash Translation Layer (CAFTL) to enhance the endurance of SSDs at the device level. With no need of any semantic information from the host, CAFTL can effectively reduce write traffic to flash memory by removing unnecessary duplicate writes and can also substantially extend available free flash memory space by coalescing redundant data in SSDs, which further improves the efficiency of garbage collection and wear-leveling. In order to retain high data access performance, we have also designed a set of acceleration techniques to reduce the runtime overhead and minimize the performance impact caused by extra computational cost. Our experimental results show that our solution can effectively identify up to 86.2% of the duplicate writes, which translates to a write traffic reduction of up to 24.2% and extends the flash space by a factor of up to 31.2%. Meanwhile, CAFTL only incurs a minimized performance overhead by a factor of up to 0.5%.

1 Introduction

The limited lifespan is the *Achilles' heel* of Flash Memory based Solid State Drives (SSDs). On one hand, SSDs built on semiconductor chips without any moving parts have exhibited many unique technical merits compared with hard disk drives (HDDs), particularly high random access performance and low power consumption. On the other hand, the limited lifespan of SSDs, which are built on flash memories with limited program/erase cycles, is still one of the most critical concerns that seriously hinder a wide deployment of SSDs in reliability-sensitive environments, such as data centers [10]. Although SSD manufacturers often claim that SSDs can sustain routine usage for years, the technical concerns about the endurance issues of SSDs still remain high. This is mainly due to three not-so-well-known reasons. First, as bit density increases, flash memory chips become more affordable but, at the same time, less reliable and less durable. In the last two years, for high-density flash devices, we have seen a sharp drop of program/erase cycle ratings from ten thousand to five thousand cycles [7]. As technology scaling continues, this situation could become even worse. Second, traditional redundancy solutions such as RAID, which have been widely used for battling disk failures, are considered less effective for SSDs, because of the high probability of correlated device failures in SSD-based RAID [9]. Finally, although some prior research work [13, 22, 33] has presented empirical and modeling-based studies on the lifespan of flash memories and USB flash drives, both positive and negative results have been reported. In fact, as a recent report from Google® points out, "endurance and retention (of SSDs) not yet proven in the field" [10].

All these aforesaid issues explain why commercial users hesitate to perform a large-scale deployment of SSDs in production systems and why integrating SSDs into commercial systems is proceeding such "painfully slowly" [10]. In order to integrate such a "frustrating technology", which comes with equally outstanding merits and limits, into the existing storage hierarchy timely and reliably, solutions for effectively improving the lifespan of SSDs are highly desirable. In this paper, we propose such a solution from a unique and viable angle.

1.1 Background of SSDs

1.1.1 Flash memory and SSD internals

NAND flash memory is the basic building block of most SSDs on the market. A flash memory package is usually composed of one or multiple *dies* (chips). Each die is segmented into multiple *planes*, and a plane is further divided into thousands (e.g. 2048) of *erase blocks*. An erase block usually consists of 64-128 *pages*. Each page has a data area (e.g. 4KB) and a spare area (a.k.a. metadata area). Flash memories support three major operations. *Read* and *write* (a.k.a. *program*) are performed in units of pages, and *erase*, which clears all the pages in an erase block, must be conducted in erase blocks.

*Currently working at the Intel Labs in Hillsboro, OR.

Flash memory has three critical technical constraints: (1) *No in-place overwrite* – the whole erase block must be erased before writing (programming) any page in this block. (2) *No random writes* – the pages in an erase block must be written sequentially. (3) *Limited program/erase cycles* – an erase block can wear out after a certain number of program/erase cycles (typically 10,000-100,000).

As a critical component in the SSD design, the *Flash Translation Layer* (FTL) is implemented in the SSD controller to emulate a hard disk drive by exposing an array of *logical block addresses* (LBAs) to the host. In order to address the aforesaid three constraints, the FTL designers have developed several sophisticated techniques: (1) *Indirect mapping* – A mapping table is maintained to track the dynamic mapping between logical block addresses (LBAs) and physical block addresses (PBAs). (2) *Log-like write mechanism* – Each write to a logical page only invalidates the previously occupied physical page, and the new content data is appended sequentially in a clean erase block, like a *log*, which is similar to the log-structured file system [41]. (3) *Garbage collection* – A garbage collector (GC) is launched periodically to recycle invalidated physical pages, consolidate the valid pages into a new erase block, and clean the old erase block. (4) *Wear-leveling* – Since writes are often concentrated on a subset of data, which may cause some blocks to wear out earlier than the others, a *wear-leveling* mechanism tracks and shuffles hot/cold data to even out writes in flash memory. (5) *Over-provisioning* – In order to assist garbage collection and wear-leveling, SSD manufacturers usually include a certain amount of over-provisioned spare flash memory space in addition to the host-usable SSD capacity.

1.1.2 The lifespan of SSDs

As flash memory has a limited number of program/erase cycles, the lifespan of SSDs is naturally constrained. In essence, the lifespan of SSDs is a function of three factors: (1) *The amount of incoming write traffic* – The less data written into an SSD, the longer the lifespan would be. In fact, the SSD manufacturers often advise commercial users, whose systems undergo intensive write traffic (e.g. an email server), to purchase more expensive high-end SSDs. (2) *The size of over-provisioned flash space* – A larger over-provisioned flash space provides more available clean flash pages in the allocation pool that can be used without triggering a garbage collection. Aggressive over-provisioning can effectively reduce the average number of writes over all flash pages, which in turn improves the endurance of SSDs. For example, the high-end Intel® X25-E SSD is aggressively over-provisioned with about 8GB flash space, which is 25% of the labeled SSD capacity (32GB) [25]. (3) *The efficiency of garbage collection and wear-leveling mechanisms* – Having been extensively researched, the garbage collection and wear-leveling policies can significantly impact the lifespan of SSDs. For example, *static wear-leveling*, which swaps active blocks with randomly chosen inactive blocks, performs better in endurance than *dynamic wear-leveling*, which only swaps active blocks [13].

Most previous research work [21] focuses on the third factor, garbage collection and wear-leveling policies. A survey [21] summarizes these techniques. In contrast, little study has been conducted on the other two aspects. This may be because incoming write traffic is normally believed to be workload dependent, which cannot be changed at the device level, and the over-provisioning of flash space is designated at the manufacturing process and cannot be excessively large (due to the production cost). In this paper we will show that even at the SSD device level, we can still effectively extend the SSD lifespan by reducing the amount of incoming write traffic and squeezing available flash memory space during runtime, which has not been considered before. This goal can be achieved based on our observation of a widely existing phenomenon – *data duplication*.

1.2 Data Duplication is Common

In file systems data duplication is very common. For example, kernel developers can have multiple versions of Linux source code for different projects. Users can create/delete the same files multiple times. Another example is word editing tools, which often automatically save a copy of documents every few minutes, and the content of these copies can be almost identical.

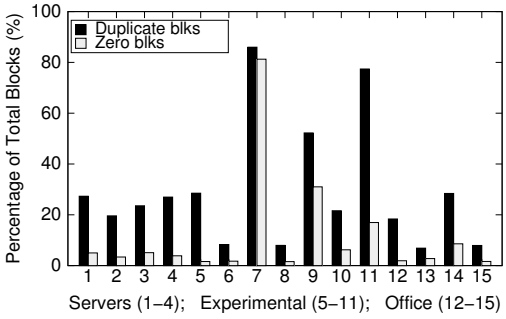


Figure 1: The percentage of redundant data in disks.

To make a case here, we have studied 15 disks installed on 5 machines in the Department of Computer Science and Engineering at the Ohio State University. Three file systems can be found in these disks, namely Ext2, Ext3, and NTFS. The disks are used in different environments, 4 disks from *Database/Web Servers*, 7 disks from *Experimental Systems* for kernel development, and the other 4 disks from *Office Systems*. We slice the disk space into 4KB blocks and use the SHA-1 hash function [1] to calculate a 160-bit hash value for each block. We can identify duplicate blocks by comparing the hash

values. Figure 1 shows the *duplication rates* (i.e. the percentage of duplicate blocks in total blocks).

In Figure 1, we find that the duplication rate ranges from 7.9% to 85.9% across the 15 disks. We also find that in only one disk with NTFS, the duplicate blocks are dominated by 'zero' blocks. The duplicate blocks on the other disks are mostly non-zero blocks, which means that these duplicate blocks contain 'meaningful' data. Considering the fact that a typical SSD has an over-provisioned space of only 6-25% of the flash memory space, removing the duplicate data, which accounts for 7.9-85.9% of the SSD capacity, can substantially extend the available flash space that can be used for garbage collection and wear-leveling. If this effort is successful, we can raise the performance comparable to that of high-end SSDs with no need of extra flash space.

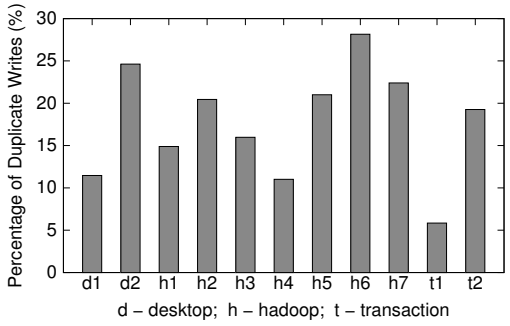


Figure 2: The perc. of duplicate writes in workloads.

Besides the static analysis of the data redundancy in storage, we have also collected I/O traces and analyzed the data accesses of 11 workloads from three categories (see more details in Section 4). For each workload, we modified the Linux kernel by intercepting each I/O request and calculating a hash value for each requested block. We analyzed the I/O traces off-line. Figure 2 shows the percentage of the duplicate writes in each workload. We can find that 5.8-28.1% of the writes are duplicated. This finding suggests that if we remove these duplicate writes, we can effectively reduce the write traffic into flash medium, which directly improves the endurance accordingly, not to mention the indirect effect of reducing the number of extra writes caused by less frequently triggered garbage collections.

1.3 Making FTL Content Aware

Based on the above observations and analysis, we propose a *Content-Aware Flash Translation Layer* (CAFTL) to integrate the functionality of eliminating duplicate writes and redundant data into SSDs to enhance the lifespan at the device level.

CAFTL intercepts incoming write requests at the SSD device level and uses a cryptographic hash function to generate fingerprints summarizing the content of updated data. By querying a fingerprint store, which maintains the fingerprints of resident data in the SSD, CAFTL can accurately and safely eliminate duplicate writes to flash medium. CAFTL also uses a two-level mapping mechanism to coalesce redundant data, which effectively extends available flash space and improves GC efficiency. In order to minimize the performance impact caused by computing hash values, we have also designed a set of acceleration methods to speed up fingerprinting. With these techniques, CAFTL can effectively reduce write traffic to flash, extend available flash space, while retaining high data access performance.

CAFTL is an augmentation, rather than a complete replacement, to the existing FTL designs. Being *content-aware*, CAFTL is orthogonal to the other FTL policies, such as the well researched garbage collection and wear-leveling policies. In fact, the existing mechanisms in the SSDs provide much needed facilities for CAFTL and make it a perfect fit in the existing SSD architecture. For example, the *indirect mapping mechanism* naturally makes associating multiple logical pages to one physical page easy to implement; the *periodic scanning process* for garbage collection and wear-leveling can also carry out an out-of-line deduplication asynchronously; the *log-like write mechanism* makes it possible to re-validate the 'deleted' data without re-writing the same content; and finally, the *semiconductor nature* of flash memory makes reading randomly remapped data free of high latencies.

CAFTL is also backward compatible and portable. Running at the device level as a part of SSD firmware, CAFTL does not need to change the standard host/device interface for passing any extra information from the upper-level components (e.g. file system) to the device. All of the design of CAFTL is isolated at the device level and hidden from users. This guarantees CAFTL as a drop-in solution, which is highly desirable in practice.

1.4 Our Contributions

We have made the following contributions in this paper: (1) We have studied data duplications in file systems and various workloads, and assessed the viability of improving endurance of SSDs through deduplication. (2) We have carefully designed a content-aware FTL to extend the SSD lifespan by removing duplicate writes (up to 24.2%) and redundant data (up to 31.2%) with minimal overhead. To the best of our knowledge, this is the first study using effective deduplication in SSDs. (3) We have also designed a set of techniques to accelerate the in-line deduplication in SSD devices, which are particularly effective with small on-device buffer spaces (e.g. 2MB) and make performance overhead nearly negligible. (4) We have implemented CAFTL in the DiskSim simulator and comprehensively evaluated its performance and shown the effectiveness of improving the SSD lifespan through extensive trace-driven simulations.

The rest of this paper is organized as follows. In Section 2, we discuss the unique challenges in the design of CAFTL. Section 3 introduces the design of CAFTL and our acceleration methods. We present our performance evaluation in Section 4. Section 5 gives the related work. The last section discusses and concludes this paper.

2 Technical Challenges

CAFTL shares the same principle of removing data redundancy with Content-Addressable Storage (CAS), e.g. [11,24,30,45,47], which is designed for backup/archival systems. However, we cannot simply borrow CAS policies in our design due to four unique and unaddressed challenges: (1) *Limited resources* – CAFTL is designed for running in an SSD device with limited memory space and computing power, rather than running on a dedicated powerful enterprise server. (2) *Relatively lower redundancy* – CAFTL mostly handles regular file system workloads, which have an impressive but much lower duplication rate than that of backup streams with high redundancy (often 10 times or even higher). (3) *Lack of semantic hints* – CAFTL works at the device level and only sees a sequence of logical blocks without any semantic hints from host file systems. (4) *Low overhead requirement* – CAFTL must retain high data access performance for regular workloads, while this is a less stringent requirement in backup systems that can run during out-of-office hours.

All of these unique requirements make deduplication particularly challenging in SSDs and it requires non-trivial efforts to address them in the CAFTL design.

3 The Design of CAFTL

The design of CAFTL aims to reach the following three critical objectives.

- *Reducing unnecessary write traffic* – By examining the data of incoming write requests, we can detect and remove duplicate writes in-line, so that we can effectively filter unnecessary writes into flash memory and directly improve the lifespan of SSDs.
- *Extending available flash space* – By leveraging the indirect mapping framework in SSDs, we can map logical pages sharing the same content to the same physical page. The saved space can be used for GC and wear-leveling, which indirectly improves the lifespan.
- *Retaining access performance* – A critical requirement to make CAFTL truly effective in practice is to avoid significant negative performance impacts. We must minimize runtime overhead and retain high data access performance.

3.1 Overview of CAFTL

CAFTL eliminates duplicate writes and redundant data through a combination of both *in-line* and *out-of-line* (a.k.a post-processing or out-of-band) deduplication. Inline deduplication refers to the case where CAFTL proactively examines the incoming data and cancels duplicate writes before committing a write request to flash. As a 'best-effort' solution, CAFTL does not guarantee that all duplicate writes can be examined and removed immediately (e.g. it can be disabled for performance purposes). Thus CAFTL also periodically scans the flash memory and coalesces redundant data out of line.

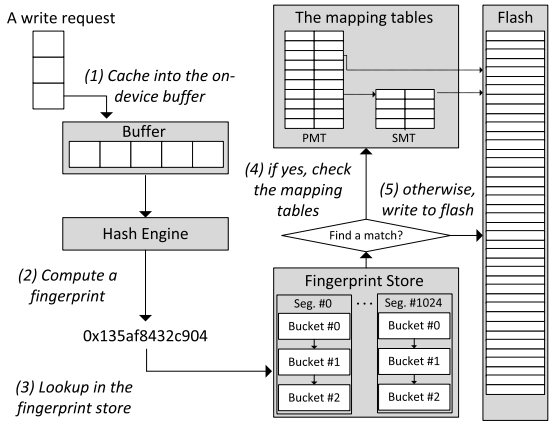


Figure 3: An illustration of CAFTL architecture.

Figure 3 illustrates the process of handling a write request in CAFTL – When a write request is received at the SSD, (1) the incoming data is first temporarily maintained in the *on-device buffer*; (2) each updated page in the buffer is later computed a hash value, also called *fingerprint*, by a *hash engine*, which can be a dedicated processor or simply a part of the controller logic; (3) each fingerprint is looked up against a *fingerprint store*, which maintains the fingerprints of data already stored in the flash memory; (4) if a match is found, which means that a residing data unit holds the same content, the *mapping tables*, which translate the host-viewable logical addresses to the physical flash addresses, are updated by mapping it to the physical location of the residing data, and correspondingly the write to flash is canceled; (5) if no match is found, the write is performed to the flash memory as a regular write.

3.2 Hashing and Fingerprint Store

CAFTL attempts to identify and remove duplicate writes and redundant data. A byte-by-byte comparison is excessively slow. A common practice is to use a cryptographic hash function, e.g. SHA-1 [1] or MD5 [40], to compute a hash value as a fingerprint. Duplicate data can be determined by comparing fingerprints. Here we explain how we produce and manage fingerprints.

3.2.1 Choosing hashing units

CAFTL uses a chunk-based deduplication approach. Unlike most CAS systems, which often use more complicated *variable-sized* chunking, CAFTL adopts a *fixed-sized* chunking approach for two reasons. First, the variable-sized chunking is designed for segmenting a long I/O stream. In CAFTL, we handle a sequence of individual requests, whose size can be very small (a few kilobytes) and vary significantly. Thus variable-sized chunking is inappropriate for CAFTL. Second, the basic operation unit in flash is a page (e.g. 4KB), and the internal management policies in SSDs, such as the mapping policy, are also designed in units of pages. Thus, using pages as the fixed-sized chunks for hashing is a natural choice and also avoids unnecessary complexity.

3.2.2 Hash function and fingerprints

In order to identify duplicate data, a cryptographic hash function is used for summarizing the content of pages. We use the SHA-1 [1], a widely used hash function, and rely on its practically collision-resistant properties to index and compare pages. For each page, we calculate a 160-bit hash value as its *fingerprint* and store it as the page's metadata in flash. The SHA-1 hash function has been proven computationally infeasible to find two distinct inputs hashing to the same value [32]. We can safely determine if two pages are identical using fingerprints.

3.2.3 The fingerprint store

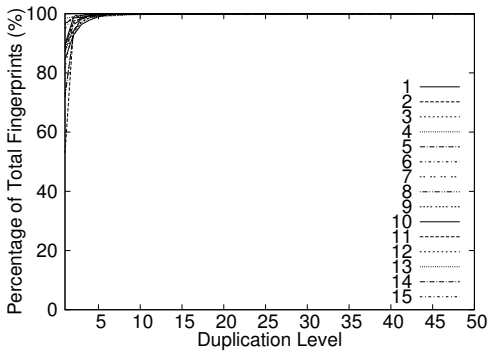


Figure 4: The CDF figure of duplicate fingerprints.

In order to locate quickly the physical page with a specific fingerprint, CAFTL manages an in-memory structure, called *Fingerprint Store*. Apparently, keeping all fingerprints and related information (25 bytes each) in memory is too costly and unnecessary. We have studied the distribution of fingerprints in the 15 disks and we plot a Cumulative Distribution Function (CDF) figure in Figure 4. We can see that the distribution of duplicated fingerprints is skewed – only 10-20% of the fingerprints are highly duplicated (more than 2). This finding provides two implications. First, most fingerprints are unique and never have a chance to match any queried fingerprint. Second, a complete search in the fingerprint store would incur high lookup latencies, and even worse, most lookups eventually turn out to be useless (no match found). Thus, we should only store and search in the most likely-to-be-duplicated fingerprints in memory.

We first logically partition the hash value space into N segments. For a given fingerprint, f, we can map it to segment $(f \ mod \ N)$, and the random nature of the hash function guarantees an even distribution of fingerprints among the segments. Each segment contains a list of *buckets*. Each bucket is a 4KB page in memory and consists of multiple entries, each of which is a key-value pair, $\{$*fingerprint*, (*location*, *reference*)$\}$. The 160-bit *fingerprint* indexes the entry; the 32-bit *location* denotes where we can find the data, either the PBA of a physical flash page or the VBA of a secondary mapping entry (see Section 3.3); the 8-bit *reference* denotes the hotness of this fingerprint (i.e. the number of referencing logical pages). The entries in each bucket are sorted in the ascending order of their fingerprint values to facilitate a fast in-bucket binary search. The total numbers of buckets and segments are designated by the SSD manufacturers.

The fingerprint store maintains the most highly referenced fingerprints in memory. During the SSD startup time, after the mapping tables are built up (to be discussed in Section 3.3), the fingerprint store is also reconstructed by scanning the mapping tables and the metadata in flash to load the key value pairs of $\{$*fingerprint*, (*location*, *reference*)$\}$ into memory. Initially no bucket is allocated in the fingerprint store. Upon inserting a fingerprint, an empty bucket is allocated and linked into a bucket list of the corresponding segment. This bucket holds the fingerprints inserted into the corresponding segment until the bucket is filled up, then we allocate another bucket. We continue to allocate buckets in this way until there are no more free buckets available. If that happens, the newly inserted fingerprint will replace the fingerprint with the smallest reference counter (i.e. the coldest one) in the bucket, unless its reference counter is smaller than any of the resident fingerprints. Note that we choose the inserting bucket in a round-robin manner to ensure a relatively even distribution of hot/cold fingerprints across the buckets in a segment. It is also worth mentioning here that a 8-bit reference counter is sufficiently large for distinguishing the hot fingerprints, because most fingerprints have a reference counter smaller than 255 (see Figure 4). We consider fingerprints with a reference counter larger than 255 as highly referenced and do not further distinguish their difference in hotness. In this way, we can include the most highly referenced fingerprints in memory. Although we may miss some opportunities of identifying the duplicates whose fingerprints are not resident in memory, this probability is considered low (as shown in Figure 4), and we are not pursu-

ing a perfect in-line deduplication. Our out-of-line scanning can still identify these duplicates later.

Searching a fingerprint can be very simple. We compute the mapping segment number and scan the corresponding list of buckets one by one. In each bucket, we use binary search to speed up the in-bucket lookup. However, for a segment with a large set of buckets, this method is still improvable. We have designed three optimization techniques to further accelerate fingerprint lookups. (1) *Range Check* – before performing the binary search in a bucket, we first compare the fingerprint with the smallest and the largest fingerprints in the buckets. If the fingerprint is out of the range, we quickly skip over this bucket. (2) *Hotness-based Reorganization* – the fingerprints in the linked buckets can be reorganized in the descending order of their reference counters. This moves the hot fingerprints closer to the list head and potentially reduces the number of the scanned buckets. (3) *Bucket-level Binary Search* – the fingerprints across the buckets can be reorganized in the ascending order of the fingerprint values by using a merge sort. For each segment we maintain an array of pointers to the buckets in the list. We can perform a binary search at the bucket level by recursively selecting the bucket in the middle to do a Range Check. In this way we can quickly locate the target bucket and skip over most buckets. Although reorganizing the fingerprints requires performing an additional merge sort, our experiments show that these optimizations can significantly reduce the number of comparisons of fingerprint values. In Section 4.3.3 we will show and compare the effectiveness of the three techniques.

3.3 Indirect Mapping

Indirect mapping is a core mechanism in the SSD architecture. SSDs expose an array of logical block addresses (LBAs) to the host, and internally, a *mapping table* is maintained to track the physical block address (PBA) to which each LBA is mapped. For CAFTL, the existing indirect mapping mechanism in SSDs provides a basic framework for deduplication and avoids rebuilding the whole infrastructure from scratch.

On the other hand, the existing 1-to-1 mapping mechanism in SSDs cannot be directly used for CAFTL, which is essentially *N*-to-1 mapping, because of two new challenges. (1) When a physical page is relocated to another place (e.g. in garbage collection), we must be able to identify quickly all the logical pages mapped to this physical page and update their mapping entries to point to the new location. (2) Since a physical page could be shared by multiple logical pages, it cannot be recycled by the garbage collector until all the referencing logical pages are demapped from it, which means that we must track the number of referencing logical pages.

3.3.1 Two-level indirect mapping

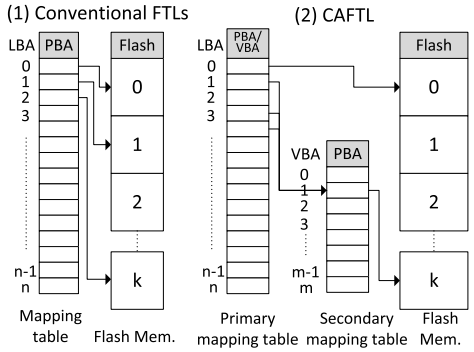


Figure 5: An illustration of the indirect mapping.

We have designed a new indirect mapping mechanism to address these aforementioned issues. As shown in Figure 5, a conventional FTL uses a one-level indirect mapping, from LBAs to PBAs. In CAFTL, we create another indirect mapping level, called *Virtual Block Addresses* (VBAs). A VBA is essentially a pseudo address name to represent a set of LBAs mapped to the same PBA. In this two-level indirect mapping structure, we can locate the physical page for a logical page either through LBA→PBA or LBA→VBA→PBA.

We maintain a *primary mapping table* and a *secondary mapping table* in memory. The *primary mapping table* maps a LBA to either a PBA, if the logical page is unique, or a VBA, if it is a duplicate page. We differentiate PBAs and VBAs by using the most significant bit in the 32-bit page address. For a page size of 4KB, using the remaining 31 bits can address 8,192 GB storage space, which is sufficiently large for an SSD. The *secondary mapping table* maps a VBA to a PBA. Each entry is indexed by the *VBA* and has two fields, $\{PBA, reference\}$. The 32-bit *PBA* denotes the physical flash page, and the 32-bit *reference* tracks the exact number of logical pages mapped to the physical page. Only physical pages without any reference can be recycled for garbage collection.

This two-level indirect mapping mechanism has several merits. First, it significantly simplifies the reverse updates to the mapping of duplicate logical pages. When relocating a physical page during GC, we can use its associated VBA to quickly locate and update the secondary mapping table by mapping the VBA to the new location (PBA), which avoids exhaustively searching for all the referencing LBAs in the huge primary mapping table. Second, the secondary mapping table can be very small. Since CAFTL handles regular file system workloads, most logical pages are unique and directly mapped through the primary table. We can also apply an approach similar to DFTL [23] to further reduce the memory demand by selectively maintaining the most frequently accessed entries of the mapping tables in memory. Finally, this incurs minimal additional lookup

overhead. For unique pages, it performs identically to conventional FTLs; for duplicate pages, only one extra memory access is needed for the lookup operation.

3.3.2 The mapping tables in flash

The mapping relationship is also maintained in flash memory. We keep an in-flash copy of the primary and secondary mapping tables along with a *journal* in dedicated flash space in SSD. Both in-flash structures are organized as a list of linked physical flash pages. When updating the in-memory tables (e.g. remapping a LBA to a new location), the update record is logged into a small in-memory buffer. When the buffer is filled, the log records are appended to the in-flash journal. If power failure happens, a capacitor (e.g. a SuperCap [46]) can provide sufficient current to flush the unwritten logs into the journal and secure the critical mapping structures in persistent storage. Periodically the in-memory tables are synced into flash and the journal is reinitialized. During the startup time, the in-flash tables are first loaded into memory and the logged updates in the journal are applied to reconstruct the mapping tables.

3.3.3 The metadata pages in flash

Unlike much prior work, which writes the metadata (e.g. LBA and fingerprint) in the spare area of physical flash pages, we reserve a dedicated number of flash pages, also called *metadata pages*, to store the metadata, and keep a *metadata page array* for tracking PBAs of the metadata pages. The spare area of a physical page is only used for storing the Error Correction Code (ECC) checksum. If each physical page is associated with 24 bytes of metadata (a 160-bit fingerprint and a 32-bit LBA/VBA), for a 32GB SSD with 4KB flash pages, we need about 0.6% of the flash space for storing metadata and a 192KB metadata page array. In this way, we can detach the data pages and the metadata pages, which allows us to manage flexibly the metadata for physical flash pages.

3.4 Acceleration Methods

Fingerprinting is the key bottleneck of the in-line deduplication in CAFTL, especially when the on-device buffer size is limited. Here we present three effective techniques to reduce its negative performance impact.

3.4.1 Sampling for hashing

In file system workloads, as we discussed previously, duplicate writes are not a 'common case' as in backup systems. This means that most time we spend on fingerprinting is not useful at all. Thus, we selectively pick only one page as a *sample page* for fingerprinting, and we use this sample fingerprint to query the fingerprint store to see if we can find a match there. If this is true, the whole write request is very likely to be a duplicate, and we can further compute fingerprints for the other pages to confirm that. Otherwise, we assume the whole request would not be a duplicate and abort fingerprinting at the earliest time. In this way, we can significantly reduce the hashing cost.

The key issue here is which page should be chosen as the sample page. It is particularly challenging in CAFTL, since CAFTL only sees a sequence of blocks and cannot leverage any file-level semantic hints (e.g. [11]). We propose to use *Content-based Sampling* – We select the first four bytes, called *sample bytes*, from each page in a request, and we concatenate the four bytes into a 32-bit numeric value. We compare these values and the page with the *largest* value is the sample page. The rationale behind this is that if two requests carry similar content, the pages with the largest sample bytes in two requests would be very likely to be the same, too. We deliberately avoid selecting the sample pages based on hash values (e.g. [11, 30]), because in CAFTL, hashing itself incurs high latency. Thus relying on hash values for sampling is undesirable, so we directly pick sample pages based on their *unprocessed* content data. We have also examined choosing other bytes (Figure 6) as the sample bytes and found that using the first four bytes performs constantly well across different workloads.

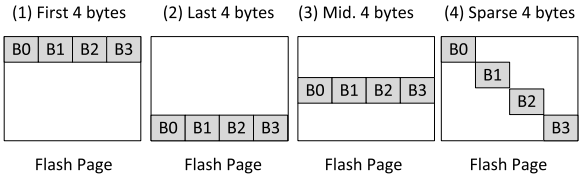


Figure 6: An illustration of four choices of sample bytes.

In our implementation of sampling, we divide the sequence of pages in a write request into several *sampling units* (e.g. 32 pages), and we pick one sample page from each unit. We also note that sampling could affect deduplication – the larger a sampling unit is, the better performance but the lower deduplication rate would be. We will study the effect of unit sizes in Section 4.4.1.

3.4.2 Light-weight pre-hashing

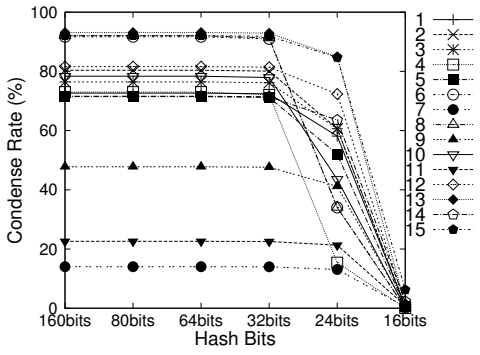


Figure 7: Condense rates vs. hash bits.

Computing a light-weight hash function often incurs lower computational cost. For example, producing a 32-bit CRC32 hash value is over 10 times faster than com-

puting a 160-bit SHA-1 hash value. More importantly, our study shows that reducing the hash strength would *not* incur a significant increase of false positives for a typical SSD capacity. We can see in Figure 7 that using only 32 bits can achieve nearly the same condense rate as using 160 bits. Plus, many SSDs integrate a dedicated ECC engine to compute checksum and detect errors, which can also be leveraged to speed up hashing.

We propose a technique, called *light-weight pre-hashing*. We maintain an extra 32-bit CRC32 hash value for each fingerprint in the fingerprint store. For a page, we first compute a CRC32 hash value and query the fingerprint store. If a match is found, which means the page is very likely to be a duplicate, then we use the SHA-1 hash function to generate a fingerprint and confirm it in the fingerprint store; otherwise, we abort the high-cost SHA-1 fingerprinting immediately and perform the write to flash. Although maintaining CRC32 hash values demands more fingerprint store space, the significant performance benefit well justifies it, as shown in Section 4.4.2. We have also considered using a Bloom filter [12] for pre-screening, like in the DataDomain® file system [47], but found it inapplicable to CAFTL, because it requires multiple hashings and the summary vector cannot be updated when a fingerprint is removed.

3.4.3 Dynamic switches

In some extreme cases, incoming requests may wait for available buffer space to be released by previous requests. CAFTL provides *dynamic switch* as the last line of defense for performance protection in such cases.

We set a *high watermark* and a *low watermark* to turn the in-line deduplication off and on, respectively. If the percentage of the occupied cache space hits a high watermark (95%), we disable the in-line deduplication to flush writes quickly to flash and release buffer space. Once the low watermark (50%) is hit, we re-enable the in-line deduplication. Although this remedy solution would reduce the deduplication rate, we still can perform out-of-line deduplication at a later time, so it is an acceptable tradeoff for retaining high performance.

3.5 Out-of-line Deduplication

As mentioned previously, CAFTL does not pursue a perfect in-line deduplication, and an internal routine is periodically launched to perform *out-of-line fingerprinting* and *out-of-line deduplication* during the device idle time.

Out-of-line fingerprinting is simple. We scan the metadata page array (Section 3.3.3) to find physical pages not yet fingerprinted. If one such a page is found, we read the page out, compute the fingerprint, and update its metadata. To avoid unnecessarily scanning the metadata of pages already fingerprinted, we use one bit in an entry of the metadata page array to denote if all of the fingerprints in the corresponding metadata page have already been computed, and we skip over such pages.

Out-of-line deduplication is more complicated due to the memory space constraint. We adopt a solution similar to the widely used *external merge sort* [39] in database systems. Supposing we have M fingerprints in total and the available memory space can accommodate N fingerprints, where $M > N$. We scan the metadata page array from the beginning, each time N fingerprints are loaded and sorted in memory, and temporarily stored in flash, then we load and sort the next N fingerprints, and so on. This process is repeated for K times ($K = \lceil \frac{M}{N} \rceil$) until all the fingerprints are processed. Then we can merge sort these K blocks of fingerprints in memory and identify the duplicate fingerprints.

Out-of-line fingerprinting and deduplication can be performed together with the GC process or independently. Since there is no harm in leaving duplicate or unfingerprinted pages in flash, these operations can be performed during idle period and immediately aborted upon incoming requests, and the perceivable performance impact to foreground jobs is minimal.

4 Performance Evaluation

4.1 Experimental Systems

We have implemented and evaluated our design of CAFTL based on a comprehensive trace-driven simulation. In this section we will introduce our simulator, trace collection, and system configurations.

4.1.1 SSD Simulator

CAFTL is a device-level design running in the SSD controller. We have implemented it in a sophisticated SSD simulator based on the Microsoft® Research SSD extension [5] for the *DiskSim* simulation environment [14]. This extension was also used in prior work [6].

The Microsoft extension is well modularized and implements the major components of FTL, such as the indirect mapping, garbage collection and wear-leveling policies, and others. Since the current version lacks an on-device buffer, which is becoming a standard component in recent generations of SSDs, we augmented the current implementation and included a shared buffer for handling incoming read and write requests. When a write request is received at the SSD, it is first buffered in the cache, and the SSD immediately reports completion to the host. Data processing and flash operations are conducted asynchronously in the background [16]. A read request returns back to the host once the data is loaded from flash into the buffer. We should note that this simulator follows a general FTL design [6], and the actual implementations of the SSD on the market can have other specific features.

4.1.2 SSD Configurations

Description	Configuration
Flash Page Size	4KB
Pages per Block	64
Blocks per Plane	2048
Planes per Package	8
# of Packages	10
Mapping policy	Full striping
Over-provisioning	15%
Garbage Collection Threshold	5%

Table 1: Configurations of the SSD simulator.

In our experiments, we use the default configurations from the SSD extension, unless denoted otherwise. Table 1 gives a list of the major config parameters.

Description	Latency
Flash Read/Write/Erase	25 µs/200µs/1.5ms
SHA-1 hashing (4KB)	47,548 cycles
CRC32 hashing (4KB)	4,120 cycles

Table 2: Latencies configured in the SSD simulator.

Table 2 gives the parameters of latencies used in our experiments. For the flash memory, we use the default latencies in our experiments. For the hashing latencies, we first cross compile the hash function code to the ARM® platform and run it on the *SimpleScalar-ARM* simulator [4] to extract the total number of cycles for executing a hash function. We assume a processor similar to ARM® Cortex R4 [8] on the device, which is specifically designed for high-performance embedded devices, including storage. Based on its datasheet, the ARM processor has a frequency from 304MHz to 934MHz [8], and we can estimate the latency for hashing a 4KB page by dividing the number of cycles by the processor frequency. It is also worth mentioning here that according to our communications with SSD manufacturer [3], high-frequency (600+ MHz) processors, such as the Cortex processor, are becoming increasingly normal in high-speed storage devices. Leveraging such abundant computing power on storage devices can be a research topic for further investigation.

4.1.3 Workloads and trace collection

We have selected 11 workloads from three representative categories and collected their data access traces.

- *Desktop* (d1,d2) – Typical office workloads, e.g. Internet surfing, emailing, word editing, etc. The workloads run for 12 and 19 hours, respectively, and feature irregular idle intervals and small reads and writes.
- *Hadoop* (h1-h7)– We execute seven TPC-H data warehouse queries (Query 1,6,11,14,15,16,20) with scale factor of 1 on a Hadoop distributed system platform [2]. These workloads run for 2-40 minutes and generate intensive large writes of temp data.
- *Transaction* (t1,t2) – We execute TPC-C workloads (1-3 warehouses, 10 terminals) for transaction processing on PostgreSQL 8.4.3 database system. The two workloads run for 30 minutes and 4 hours, respectively, and feature intensive write operations.

The traces are collected on a DELL® Dimension 3100 workstation with an Intel® Pentium™4 3.0GHz processor, a 3GB main memory, and a 160GB 7,200 RPM Seagate® hard disk drive. We use Ubuntu 9.10 with the Ext3 file system. We modified the Linux kernel 2.6.32 source code to intercept each I/O request and compute a SHA-1 hash value as a fingerprint for each 4KB page of the request. These fingerprints, together with other request information (e.g. offset, type), are transferred to another machine via *netconsole* [35]. This avoids the possible interference caused by tracing. The collected trace files are analyzed offline and used to drive the simulator for our experimental evaluation.

4.2 Effectiveness of Deduplication

CAFTL intends to remove duplicate writes and extend flash space. In this section, we perform two sets of experiments to show the effectiveness of deduplication in CAFTL. In both experiments, we use an SSD with a 934MHz processor and a 16MB buffer.

4.2.1 Removing duplicate writes

CAFTL identifies and removes duplicate writes via inline deduplication. Denoting the total number of pages requested to be written as n, and the total number of pages being actually written into flash medium as m, the *deduplication rate* is defined as $\frac{n-m}{n}$. Figure 8 shows the deduplication rate of the 11 workloads running on CAFTL. In this figure, *offline* refers to the optimal case, where the traces are examined and deduplicated offline. We also show CAFTL without sampling and with a sampling unit size of 128KB (32 pages), denoted as *no-sampling* and *128KB*, respectively.

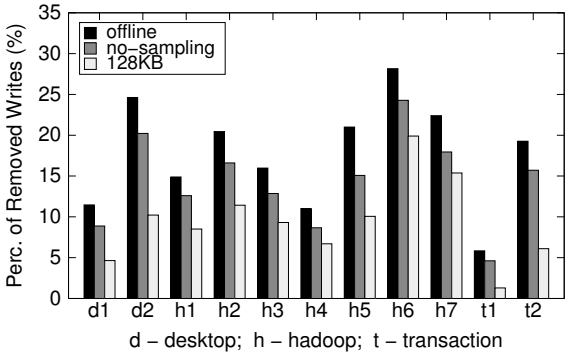


Figure 8: Perc. of removed duplicate writes.

As we see in Figure 8, duplication is highly workload dependent. Across the 11 workloads, the rate of duplicate writes in the workloads ranges from 5.8% (*t1*) to 28.1% (*h6*). CAFTL can achieve deduplication rates from 4.6% (*t1*) to 24.2% (*h6*) with no sampling. Compared with the optimal case (*offline*), CAFTL identifies up to 86.2% of the duplicate writes in *offline*. We also can see that with a larger sampling unit (128KB), CAFTL achieves a lower but reasonable deduplication rate. In Section 4.4.1, we will give more detailed analysis on the effect of sampling unit sizes.

4.2.2 Extending flash space

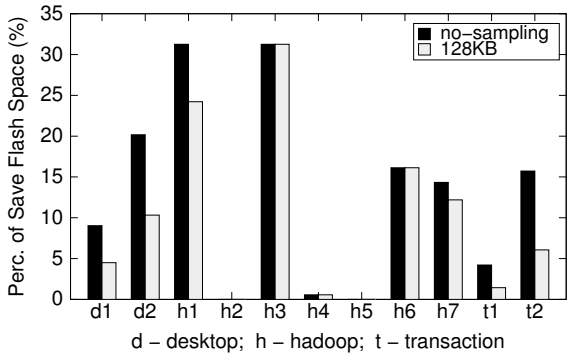


Figure 9: Perc. of extended flash space.

Besides directly removing duplicate writes to the flash memory, CAFTL also reduces the amount of occupied flash memory space and increases the number of available clean erase blocks for garbage collection and wear-leveling. Figure 9 shows the percentage of extended flash space in units of erase blocks, compared to the baseline case (without CAFTL). We show CAFTL without sampling (*no-sampling*) and with sampling (*128KB*).

As shown in Figure 9, CAFTL can save up to 31.2% (*h1*) of the occupied flash blocks for the 11 workloads. The worst cases are *h2* and *h5*, in which no space saving is observed. This is because the two workloads are relatively smaller, the total number of occupied erase blocks is only 176. Although the number of pages being written is reduced by 16.6% (*h2*) and 15% (*h5*), the saved space in units of erase blocks is very small.

4.3 Performance Impact

To make CAFTL truly effective in practice, we must retain high performance and minimize negative impact. Here we study three key factors affecting performance, *cache size*, *hashing speed*, and *fingerprint searching*. The acceleration methods are not applied in experiments.

4.3.1 Cache size

In Figure 10, we show the percentage of the increase of average read/write latencies with various cache sizes (2MB to 16MB). We compare CAFTL with the baseline case (without CAFTL). In the experiments, we configure an SSD with a 934MHz processor. We can see that with a small cache space (2MB), the read and write latencies can increase by a factor of up to 34% (*t1*). With a moderate cache size (8MB), the latency increases are reduced to less than 4.5%. With a 16MB cache, a rather standard size, the latency increases become negligible (less than 0.5%). For some workloads (*d2, h3, h5, h7, t1, t2*), we can even see a slight performance improvement (0.2-0.5%), because CAFTL removes unnecessary writes, which reduces the probability of being blocked by an in-progress flash write operation. In this case we see a negative performance impact with a small cache space, and we will show how to mitigate such a problem through our acceleration methods in Section 4.4.

4.3.2 Hashing speed

Computing fingerprints is time consuming and affects access performance. The hashing speed depends on the capability of processors. Using a more powerful processor can effectively reduce the latency for digesting pages and generating fingerprints. To study the performance impact caused by hashing speed, we vary the processor frequency from 304MHz to 934MHz, based on the Cortex datasheet [8]. We configure an SSD with a 16MB cache space and show the increase of read latencies compared to the baseline case (without CAFTL) in Figure 11. We did not observe an increase of write latencies, since most writes are absorbed in the buffer.

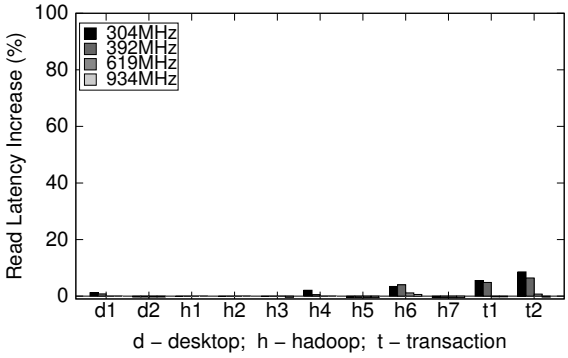


Figure 11: Perf. impact of hashing Speeds.

In Figure 11, we can see that most workloads are insensitive to hashing speed. With a 304MHz processor, the performance overhead is less than 8.5% (*t2*), which has more intensive larger writes. At 934MHz, the performance overhead is merely observable (up to 0.5%). There are two reasons. First, the 16MB on-device buffer absorbs most incoming writes and provides a sufficient space for accommodating incoming reads. Second, the incoming read requests are given a higher priority than writes, which reduces noticeable delays in the critical path. These optimizations make reads insensitive to hashing speed and reduces noticeable latencies. Also

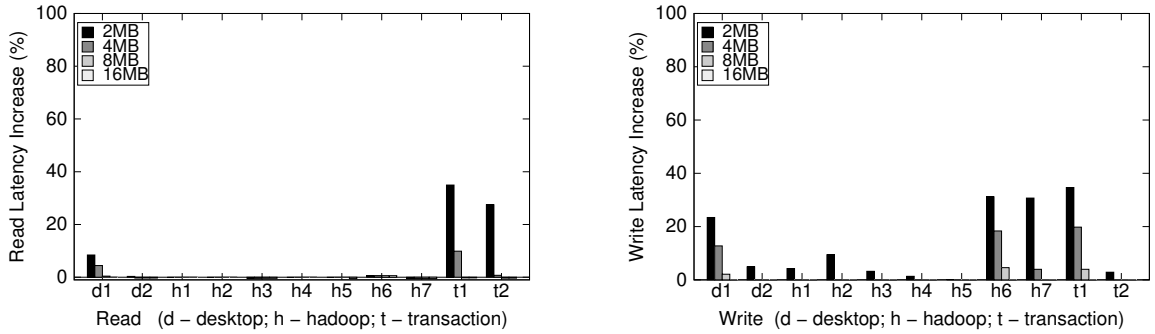


Figure 10: Performance impact of cache sizes (2-16MB).

note that if a dedicated hashing engine is used on the device, the hashing latency could be further reduced.

4.3.3 Fingerprint searching

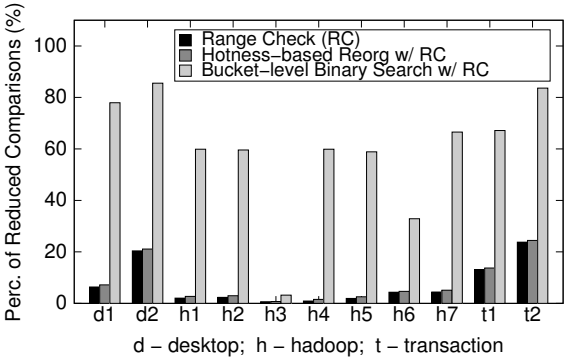


Figure 12: Optimizations on fingerprint searching.

We have proposed three techniques to accelerate fingerprint searching. Figure 12 shows the percentage of reduced fingerprint comparisons compared with the baseline case. We configure the fingerprint store with 256 segments to hold the fingerprints for each workload. We can see that using *Range Check* can effectively reduce the comparisons of fingerprints by up to 23.7% (*t2*). However, *Hotness-based Reorganization* can provide little further improvement (less than 1%), because it essentially accelerates lookups for fingerprints that are duplicated, which is relatively an uncommon case. As expected, *Bucket-level Binary Search* can significantly reduce the average number of comparisons for each lookup. In *d2*, for example, *Bucket-level Binary Search* can effectively reduce the average number of comparisons by a factor of 85.5%. Thus we would suggest applying *Bucket-level Binary Search* and *Range Check* to speed up fingerprint lookups.

4.4 Acceleration Methods

With a small on-device buffer, the high computational latency caused by hashing could be significant and perceived by the users. We have developed three techniques to accelerate fingerprinting. In this section, we will show the effectiveness of each individual technique and then show the effects in aggregate. We configure an SSD with a 934MHz processor and a small 2MB buffer.

4.4.1 Sampling

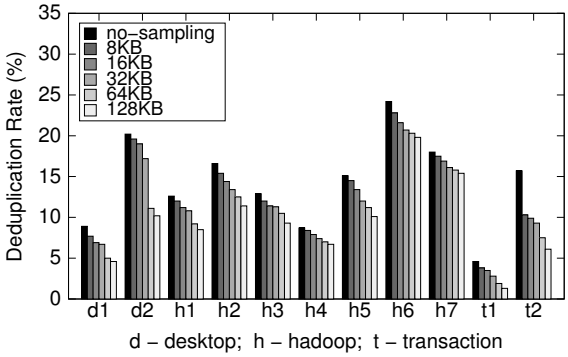
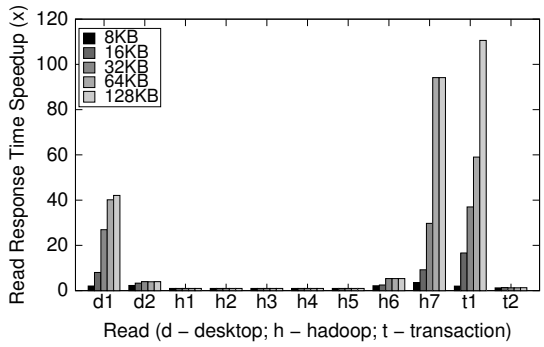


Figure 14: Dedup. with Sampling

As shown in Figure 13 and Figure 14, sampling can significantly improve performance. With the increase of sampling unit size, fewer fingerprints need to be calculated, which translates into a manifold reduction of observed read and write latencies. For example, *h7* achieves a speedup by a factor of 94.1 times for reads and 3.5 times for writes, because of the significantly reduced waiting time for the buffer. Meanwhile, the deduplication rate is only reduced from 18% to 15.4%. Considering such a significant speedup, the minor loss of deduplication rate is acceptable. The maximum speedup, 110.6 times (read), is observed in *t1*, and its deduplication rate drops from 4.6% to 1.3%. This is mostly because for workloads with low duplication rate, the probability of sampling right pages is also relatively low.

4.4.2 Light-weight pre-hashing

Light-weight pre-hashing uses a fast CRC32 hash function to filter most unlikely-to-be-duplicated pages before performing high-cost fingerprinting. Figure 15 shows the speedup of reads and writes by using CRC32 for pre-hashing, compared with CAFTL without pre-hashing. Only pre-hashing is enabled here. We can see that in the best case (*t1*), pre-hashing can reduce the latencies by a factor of up to 148.3 times for reads and 3.9 times for writes. This is because, as mentioned previously,



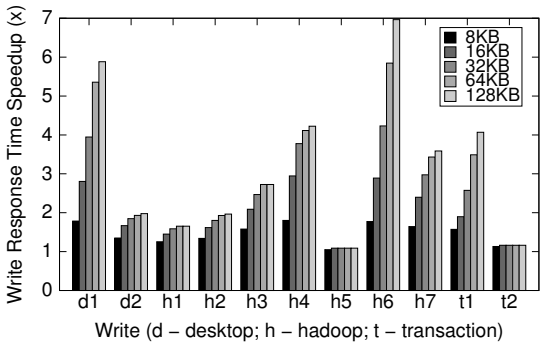


Figure 13: Performance speedup with Sampling (unit size: 8-128KB).

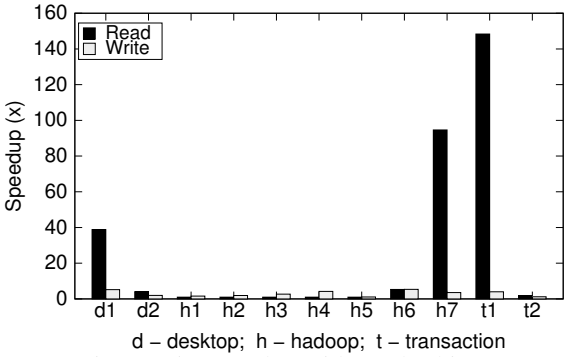


Figure 15: Speedup with pre-hashing.

this workload is write intensive and has a long waiting queue, which makes the queuing effect particularly significant. Similar to sampling, writes receive relatively smaller benefit, because the buffer absorbs the writes with low latency and diminishes the effect of speeding up writes. Meanwhile, we also found negligible difference in deduplication rates, which is consistent with our analysis shown in Figure 7.

4.4.3 Dynamic switch

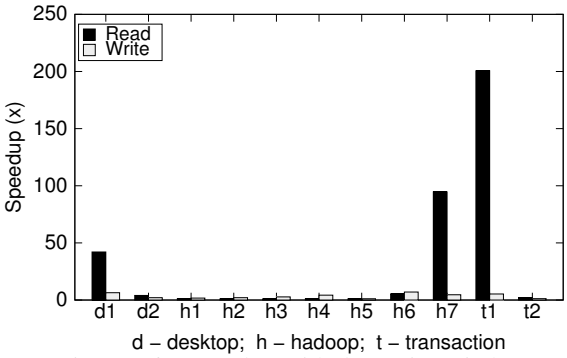


Figure 16: Speedup with dynamic switch.

CAFTL also provides *dynamic switch* to dynamically turn on/off the in-line deduplication, depending on the usage of the on-device buffer. We configure the high watermark as 95% (off) and the low watermark as 50% (on). Figure 16 shows the speedup of reads and writes in the workloads. Again, *t1* receives the most significant performance speedup by a factor of 200.6 times. Some workloads (*h1*-*h5*) receive no benefits, because they are less I/O intensive. For the other workloads, we can observe a speedup of 2.1 times to 94.6 times.

4.4.4 Putting it all together

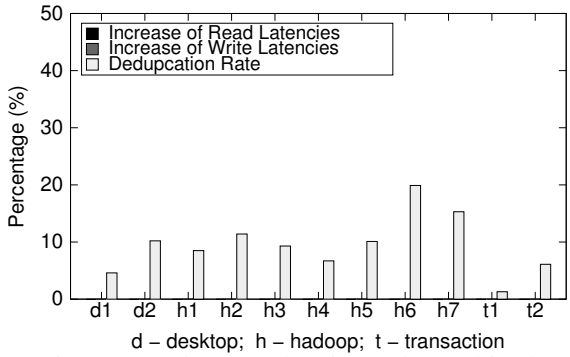


Figure 17: Three acceleration tech. combined

In Figure 17, we enable all the three acceleration techniques and show the increase of read and write latencies, compared with the baseline case (without CAFTL), and the corresponding deduplication rate. We can see that by combining all the three techniques, we can almost completely remove the performance overhead with only a 2MB on-device buffer. In the meantime, we can achieve a deduplication rate of up to 19.9%.

5 Other Related Work

Flash memory based SSDs have received a lot of interest in both academia and industry. There is a large body of research work on flash memory and SSDs (e.g. [6,9,13,15–18,20,23,26–29,31,34,37,38,42,44]). Concerning lifespan issues, most early work focuses on designing garbage collection and wear-leveling policies. A survey [21] summarizes these techniques. Here we only present the papers most related to this work.

Recently Grupp et al. [22] have presented an empirical study on the performance, power, and reliability of flash memories. Their results show that flash memories, particularly MLC devices, exhibit significant error rates after or even before reaching the rated lifetime, which makes using high density SSDs in commercial systems a difficult choice. Another report [13] has studied the write

endurance of USB flash drives with a more optimistic conclusion that the endurance of flash memory chips is better than expected, but whole-device endurance is closely related to the FTL designs. A modeling based study on the endurance issues has also been presented in [33]. These studies provide much needed information about the lifespan of flash memory and small-size flash devices. However, so far the endurance of state-of-the-art SSDs has not yet been proven in the field [10].

Early studies on SSDs mainly focus on performance. Some recent studies have begun to look at reliability issues. Differential RAID [9] tries to improve reliability of an SSD-based RAID storage by distributing parity unevenly across SSDs to reduce the probability of correlated multi-device failure. Griffin [42] extends SSD lifetime by maintaining a log-structured HDD cache and migrating cached data periodically. A recent work [36] considers write cycles in addition to storage space as a constrained resource in depletable storage systems and suggests attribute depletion to users in systems like cloud computing. ChunkStash [19] uses flash memory to speed up index lookups for inline storage deduplication. Another work [43] proposes to integrate phase change memory into SSDs to improve the performance, energy consumption, and also lifetime. Our study has made its unique contributions to enhancing the lifespan of SSDs by removing duplicate writes and coalescing redundant data at the device level, as a more general solution.

6 Conclusion and Discussions

Enhancing the SSD lifespan is crucial to a wide deployment of SSDs in commercial systems. In this paper, we have proposed a solution, called CAFTL, and shown that by removing duplicate writes and coalescing redundant data, we can effectively enhance the lifespan of SSDs while retaining high data access performance.

A potential concern about CAFTL is the volatility of the on-device RAM buffer – the buffered data could be lost upon power failure. However, this concern is not new to SSDs. A hard disk drive also has an on-device buffer, but it provides users an option (e.g. using *sdparm* tool) to flexibly enable/disable the buffer on their needs. Similarly, if needed, the users can choose to disable the in-line deduplication and the buffer in an SSD, and the out-of-line deduplication can still be effective.

Although we have striven to minimize memory usage, CAFTL demands more space for storing fingerprints and the secondary mapping table, compared with traditional FTLs. According to our communications with SSD manufacturer [3], memory actually only accounts for a small percentage of the total production cost, and the most expensive component is flash memory. Thus we consider this tradeoff is worthwhile to extend available flash space, and SSD lifespan. If budget allows, we would suggest maintaining the fingerprint store fully in memory, which not only improves deduplication rate but also simplifies designs.

Further improvements are also possible. One is to relax the stringent "one-time programming" requirement. According to the specification, each flash page in a clean erase block should be programmed (written) only once. In practice, flash chips can allow multiple programs to a page and the risk of "program disturb" is fairly low [7]. We can leverage this feature to simplify many designs. For example, we can write multiple versions of LBA/VBA and fingerprints into the spare area of a physical page, which can largely remove the need for metadata pages. Another consideration is to integrate a byte-addressable persistent memory (e.g. PCM) into the SSDs to maintain the metadata, which can remove much design complexity. We are also considering the addition of on-line compression into SSDs to better utilize the high-speed processor on the device. This can further extend available flash space but may require more changes to the FTL design, which will be our future work.

As SSD technology becomes increasingly mature and delivers satisfactory performance, we believe, the endurance issue of SSDs, particularly high-density MLC SSDs, opens many new research opportunities and should receive more attention from researchers.

Acknowledgments

We are grateful to our shepherd Dr. Christos Karamanolis from VMware® and the anonymous reviewers for their constructive comments. We also thank our colleague Bill Bynum for reading this paper and his suggestions. This research was partially supported by the NSF under grants CCF-0620152, CCF-072380, and CCF-0913150.

References

[1] FIPS 180-1, Secure Hash Standard, April 1995.

[2] Hadoop. http://hadoop.apache.org/, 2010.

[3] Personal communications with an SSD architect, 2010.

[4] SimpleScalar 4.0. http://www.simplescalar.com/v4test.html, 2010.

[5] SSD extension for DiskSim simulation environment. http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/, 2010.

[6] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *Proceedings of USENIX'08* (Boston, MA, June 2008).

[7] ANDERSEN, D. G., AND SWANSON, S. Rethinking flash in the data center. In *IEEE Micro* (July/Aug 2010).

[8] ARM. Cortex R4. http://www.arm.com/products/processors/cortex-r/cortex-r4.php, 2010.

[9] BALAKRISHNAN, M., KADAV, A., PRABHAKARAN, V., AND MALKHI, D. Differential RAID: Rethinking RAID for SSD Reliability. In *Proceedings of EuroSys'10* (Paris, France, April 2010).

[10] BARROSO, L. A. Warehouse-scale computing. In *Keynote in the SIGMOD'10 conference* (Indianapolis, IN, June 2010).

[11] BHAGWAT, D., ESHGHI, K., LONG, D. D. E., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of MASCOTS'09* (London, UK, September 2009).

[12] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM* (1970), vol. 13(7), pp. 422–426.

[13] BOBOILA, S., AND DESNOYERS, P. Write endurance in flash drives: Measurements and analysis. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[14] BUCY, J., SCHINDLER, J., SCHLOSSER, S., AND GANGER, G. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim, 2010.

[15] CHEN, F., JIANG, S., AND ZHANG, X. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proceedings of ISLPED'06* (Tegernsee, Germany, October 2006).

[16] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of SIGMETRICS/Performance'09* (Seattle, WA, June 2009).

[17] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of HPCA'11* (San Antonio, TX, Feb 2011).

[18] CHEN, S. FlashLogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of SIGMOD'09* (Providence, RI, June 2009).

[19] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of USENIX'10* (Boston, MA, June 2010).

[20] DIRIK, C., AND JACOB, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proceedings of ISCA'09* (Austin, TX, June 2009).

[21] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. In *ACM Computing Survey'05* (2005), vol. 37(2), pp. 138–163.

[22] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of MICRO'09* (New York, NY, December 2009).

[23] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of ASPLOS'09* (Washington, D.C., March 2009).

[24] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference Engine: Harnessing memory redundancy in virtual machines. In *Proceedings of OSDI'08* (San Diego, CA, 2008).

[25] INTEL. Intel X25-E extreme SATA solid-state drive. http://www.intel.com/design/flash/nand/extreme, 2008.

[26] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: A file system for virtualized flash storage. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[27] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *Proceedings of USENIX Winter* (New Orleans, LA, Jan 1995), pp. 155–164.

[28] KIM, H., AND AHN, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of FAST'08* (San Jose, CA, February 2008).

[29] LEE, S., AND MOON, B. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of SIGMOD'07* (Beijing, China, June 2007).

[30] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of FAST'09* (San Jose, CA, 2009).

[31] MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of EuroSys'10* (Paris, France, April 2010).

[32] MENEZES, A. J., V. OORSCHOT, P. C., AND VANSTONE, S. A. Handbook of applied cryptography. In *CRC Press* (1996).

[33] MOHAN, V., SIDDIQUA, T., GURUMURTHI, S., AND STAN, M. R. How I learned to stop worrying and love flash endurance. In *Proceedings of HotStorage'10* (Boston, MA, June 2010).

[34] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating enterprise storage to SSDs: analysis of tradeoffs. In *Proceedings of EuroSys'09* (Nuremberg, Germany, March 2009).

[35] NETCONSOLE. http://www.kernel.org/doc/Documentation/ networking/netconsole.txt, 2010.

[36] PRABHAKARAN, V., BALAKRISHNAN, M., DAVIS, J. D., AND WOBBER, T. Depletable storage systems. In *Proceedings of HotStorage'10* (Boston, MA, June 2010).

[37] PRABHAKARAN, V., RODEHEFFEER, T. L., AND ZHOU, L. Transactional flash. In *Proceedings of OSDI'08* (San Diego, CA, December 2008).

[38] PRITCHETT, T., AND THOTTETHODI, M. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of ISCA'10* (Saint-Malo, France, June 2010).

[39] RAMAKRISHNAN, R., AND GEHRKE, J. Database managment systems. McGraw-Hill, 2030.

[40] RIVEST, R. The MD5 message-digest algorithm. http://www.ietf.org/rfc/rfc1321.txt, April 1992.

[41] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems* (1992), vol. 10(1):26-52.

[42] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending SSD lifetimes with disk-based write caches. In *Proceedings of FAST'10* (San Jose, CA, February 2010).

[43] SUN, G., JOO, Y., CHEN, Y., NIU, D., XIE, Y., CHEN, Y., AND LI, H. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of HPCA'10* (Bangalore, India, Jan 2010).

[44] TSIROGIANNIS, D., HARIZOPOULOS, S., AND SHAH, M. A. Query processing techniques for solid state drives. In *Proceedings of SIGMOD'09* (Providence, RI, June 2009).

[45] UNGUREANU, C., ATKIN, B., ARANYA, A., GOKHALE, S., RAGO, S., CALKOWSKI, G., DUBNICKI, C., AND BOHRA, A. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of FAST'10* (San Jose, CA, 2010).

[46] WIKIPEDIA. Battery or supercap. http://en.wikipedia.org/wiki/Solid-state-drive#Battery_or_SuperCap, 2010.

[47] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of FAST'08* (San Jose, CA, 2008).

Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing

Feng Chen[2*] Rubao Lee[1,2] Xiaodong Zhang[2]
[1]Institute of Computing Technology
Chinese Academy of Sciences
liru@cse.ohio-state.edu
[2]Dept. of Computer Science & Engineering
The Ohio State University
{fchen, zhang}@cse.ohio-state.edu

Abstract

Flash memory based solid state drives (SSDs) have shown a great potential to change storage infrastructure fundamentally through their high performance and low power. Most recent studies have mainly focused on addressing the technical limitations caused by special requirements for writes in flash memory. However, a unique merit of an SSD is its rich internal parallelism, which allows us to offset for the most part of the performance loss related to technical limitations by significantly increasing data processing throughput.

In this work we present a comprehensive study of essential roles of internal parallelism of SSDs in high-speed data processing. Besides substantially improving I/O bandwidth (e.g. 7.2x), we show that by exploiting internal parallelism, SSD performance is no longer highly sensitive to access patterns, but rather to other factors, such as data access interferences and physical data layout. Specifically, through extensive experiments and thorough analysis, we obtain the following new findings in the context of concurrent data processing in SSDs. (1) Write performance is largely independent of access patterns (regardless of being sequential or random), and can even outperform reads, which is opposite to the long-existing common understanding about slow writes on SSDs. (2) One performance concern comes from interference between concurrent reads and writes, which causes substantial performance degradation. (3) Parallel I/O performance is sensitive to physical data-layout mapping, which is largely not observed without parallelism. (4) Existing application designs optimized for magnetic disks can be suboptimal for running on SSDs with parallelism. Our study is further supported by a group of case studies in database systems as typical data-intensive applications. With these critical findings, we give a set of recommendations to application designers and system architects for exploiting internal parallelism and maximizing the performance potential of SSDs.

1 Introduction

The I/O performance of Hard Disk Drives (HDDs) has been regarded as a major performance bottleneck for high-speed data processing, due to the excessively *high latency* of HDDs for random data accesses and *low throughput* of HDDs for handling multiple concurrent requests. Recently Flash Memory based Solid State Drives (SSDs) have been incorporated into storage systems. Unlike HDDs, an SSD is built entirely of semiconductor chips with no moving parts. Such an architectural difference provides a great potential to address fundamentally the technical issues of rotating media. In fact, researchers have made extensive efforts to adopt this solid state storage technology in storage systems and proposed solutions for performance optimizations (e.g. [8, 18, 27, 31]). Most of the work has focused on leveraging the high random data access performance of SSDs and addressing their technical limits (e.g. slow random writes). Among these studies, however, an important functionality of SSDs has not received sufficient attention and is rarely discussed in the existing literature, which is the *internal parallelism*, a unique and rich resource provided by SSDs. Parallelism has been reported earlier not to be beneficial to performance [5]. A recent study [20] on the advances of SSD technology reports that with the support of native command queuing (NCQ), the state-of-the-art SSDs are capable of handling multiple I/O jobs simultaneously and achieving a high throughput. In this paper, we will show that the impact of internal parallelism is far beyond the scope of the basic operations of SSDs. We believe that the use of internal parallelism can lead to a fundamental change in the current sequential-access-oriented optimization model adopted in system and application designs, which is particularly important for data-intensive applications.

1.1 Internal Parallelism in SSDs

There are two inherent architectural limitations in the design of SSDs. First, due to current technical constraints, one single flash memory package can only provide limited bandwidth (e.g. 32-40MB/sec [3]). Second, writes in flash memory are often much slower than reads, and many critical operations, such as garbage collection and wear-leveling [3, 6, 10], can incur latencies as high as milliseconds.

To address these limitations, SSD architects have built an ingenious structure to provide internal parallelism – Most SSDs are built on an array of flash memory packages, which are connected through multiple (e.g. 2-10) channels to flash

*Currently working at the Intel Labs in Hillsboro, OR.

memory controllers. SSDs provide logical block addresses (LBA) as a logical interface to the host. Since logical blocks can be striped over multiple flash memory packages, data accesses can be conducted independently in parallel. Such a highly parallelized design yields two benefits: (1) Transferring data from/to multiple flash memory packages in parallel can provide high bandwidth in aggregate. (2) High-latency operations can be effectively hidden behind other concurrent operations. Therefore, the internal parallelism, in essence, is not only an *inherent functionality* but also a *basic requirement* for SSDs to deliver high performance.

Exploiting I/O parallelism has been studied in conventional HDD-based storage, such as RAID [24], a storage based on multiple hard disks. However, there are two fundamental differences between the SSD and RAID architectures, which demand thorough and new investigations: (1) *Different logical/physical mapping mechanisms* – In RAID, a logical block is *statically* mapped to a "fixed" physical location, which is determined by its logical block number (LBN) [24]. In SSD, a logical block can be *dynamically* mapped to any physical location, which changes with writes during runtime. This has called our particular attention to a unique data layout problem in SSDs, and we will show that an ill-mapped data layout can cause significant performance degradation, while such a problem is unlikely to happen in RAID. (2) *Different physical natures* – RAID is built on magnetic disks, whose random access performance is often one order of magnitude lower than sequential access, while SSDs are built on flash memories, in which such a performance gap is much smaller. The difference in physical natures can strongly impact the existing sequentiality-based application designs, because without any moving parts, parallelizing I/O operations on SSDs can make random accesses capable of performing comparably or even slightly better than sequential accesses, while this is difficult to implement in RAID. Therefore, a careful and insightful study of the internal parallelism of this emerging storage technology is highly desirable for storage architects, system designers, and data-intensive application users.

1.2 Research and Technical Challenges

On one hand, the internal parallelism makes a single SSD capable of handling multiple incoming I/O requests in parallel and achieving a high bandwidth. On the other hand, the internal parallelism cannot be effectively exploited unless we address the three following critical challenges.

- **The performance gains from internal parallelism are highly dependent on how the SSD internal structure is insightfully understood and effectively utilized.** Internal parallelism is an architecture dependent resource. For example, the mapping policy directly determines the data layout across flash memory packages, which significantly affects the efficiency of parallelizing data accesses. In our experiments, we find that an ill-mapped data layout can cause up to 4.2 times higher latency for parallel accesses on an SSD. Without knowing the SSD internals, the anticipated performance gains are difficult to achieve. Meanwhile, uncovering such a low-level architectural information without changing the strictly defined interface is very challenging. In this paper we will present a set of simple yet effective experimental techniques to achieve this goal.
- **Parallel data accesses can compete for critical hardware resources in SSDs, and such interference would cause unexpected performance degradation.** Increasing parallelism is a double-edged sword. On one hand, high concurrency would generally improve resource utilization and increase I/O throughput. On the other hand, sharing and competing for critical resources may cause undesirable interference and performance loss. For example, we find mixing reads and writes can cause a throughput decrease as high as 4.5 times for writes. Only by understanding both benefits and side effects of I/O parallelism on an SSD can we effectively exploit its performance potential while avoiding its negative effects.
- **Exploiting internal parallelism in SSDs demands fundamental changes to the existing program design and the sequential-access-oriented optimization model adopted in software systems.** Most existing applications, such as database systems, assume that the underlying storage device is a hard disk drive. As a result, they mostly focus on how to *sequentialize* rather than *parallelize* data accesses for improving storage performance (e.g. [16]). Moreover, many optimization decisions embedded in application designs are implicitly based on such an assumption, which would unfortunately be problematic when being applied to SSDs. In our case studies on the PostgreSQL database system, we not only show that speedups by up to a factor of 5 can be achieved by parallelizing a query with multiple sub-queries, but more importantly, we also show that with I/O parallelism, the *query optimizer*, a key component in database systems, would make an *incorrect* decision when selecting the optimal query plan, which should receive a particular attention from application and system designers.

1.3 Critical Issues for Investigation

In this work, we strive to address the above three challenges by answering several critical questions related to the internal parallelism of SSDs and revealing some untold facts and unexpected dynamics in SSDs.

- Limited by the 'thin' interface between the storage device and the host, how can we effectively uncover the key architectural features of an SSD? Of the most interest, how is the physical data layout determined in an SSD?
- The effectiveness of parallelizing data access depends on many factors, such as workload access patterns, resource redundancy, and others. Can we quantify the benefit of I/O parallelism and its relationship to these factors?

- Reads and writes on SSDs both generate many asynchronous background operations and thus may interfere with each other. Can we quantitatively show such interactive effects between parallel data accesses? Would such interference impair the effectiveness of parallelism?
- Readahead improves read performance on SSDs but it is sensitive to read patterns [6]. Would I/O parallelism affect the readahead? How should we choose between increasing parallelism and retaining effective readahead?
- The physical data layout on an SSD could change on the fly. How does an ill-mapped data layout affect the efficiency of I/O parallelism and the readahead?
- Applications can leverage internal parallelism to optimize the data access performance. How much performance benefit can we achieve in practice?
- Many optimizations in applications are specifically tailored to the properties of hard disk drives and may be ineffective for SSDs. Can we make a case that parallelism-based optimizations would become a critical consideration for maximizing data access performance on SSDs?

In this paper, we will answer these questions, and we hope our experimental analysis and case studies will influence the system and application designers to rethink carefully the current sequential-access-oriented optimization model and treat parallelism as a *top priority* on SSDs.

The rest of this paper is organized as follows. Section 2 provides the background. Section 3 introduces our experimental system and methodology. Section 4 presents how to detect the SSD internals. Section 5 and 6 present our experimental and case studies on SSDs. Section 7 discusses the system implications of our findings. Related work is given in Section 8. The last section concludes this paper.

2 Background of SSD Architecture

2.1 SSD Internals

A typical SSD includes four major components (Figure 1). A *host interface logic* connects to the host through an interface connection (e.g. SATA or IDE bus). An *SSD controller* manages flash memory space, translates incoming requests, and issues commands to flash memory packages via a *flash memory controller*. Some SSDs have a dedicated DRAM *buffer* to hold metadata or data, and some SSDs only use a small integrated SRAM buffer to lower production cost. In most SSDs, multiple (e.g. 2-10) channels are used to connect the controller with flash memory packages. Each channel may be shared by more than one package. Actual implementations may vary across different models, and previous work [3, 10] gives detailed descriptions about the architecture of SSDs.

By examining the internal architectures of SSDs, we can find that parallelism is available at different levels, and operations at each level can be parallelized or interleaved.

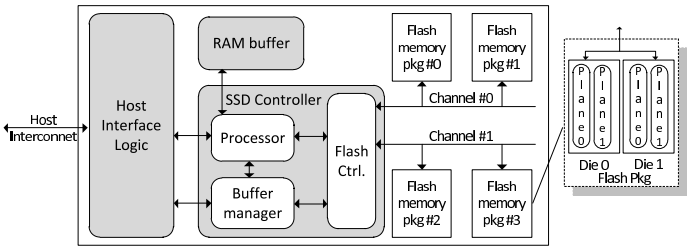


Figure 1. An illustration of SSD architecture [3].

- **Channel-level Parallelism** – In an SSD, flash memory packages are connected to the controller through multiple channels. Each channel can be operated independently and simultaneously. Some SSDs adopt multiple Error Correction Code (ECC) engines and flash controllers, each for a channel, for performance purposes [23].
- **Package-level Parallelism** – In order to optimize resource utilization, a channel is usually shared by multiple flash memory packages. Each flash memory package can be operated independently. Operations on flash memory packages attached to the same channel can be interleaved, so the bus utilization can be optimized [3, 10].
- **Die-level Parallelism** – A flash memory package often includes two or more dies (chips). Each die can be selected individually and execute a command independent of the others, which increases the throughput [28].
- **Plane-level Parallelism** – A flash memory chip is typically composed of two or more planes. Most flash memories (e.g. [2, 28]) support performing the same operation (e.g. read/write/erase) on multiple planes simultaneously. Some flash memories (e.g. [2]) provide `cache mode` to further parallelize medium access and bus transfer.

Such a highly parallelized structure provides rich opportunities for parallelism. In this paper we will present several simple yet effective experimental techniques to uncover the internal parallelism on various levels.

2.2 Native Command Queuing (NCQ)

Native command queuing (NCQ) is a feature introduced by the SATA II standard [1]. With NCQ support, the device can accept multiple incoming commands from the host and schedule the jobs internally. NCQ is especially important to SSDs, because the highly parallelized internal structure of an SSD can be effectively utilized only when the SSD is able to accept multiple concurrent I/O jobs from the host (operating system). Early generations of SSDs do not support NCQ and thus cannot benefit from parallel I/O [5]. The two SSDs used in our experiments can accept up to 32 jobs.

3 Measurement Environment

3.1 Experimental Systems

Our experiments have been conducted on a Dell™ Precision™ T3400. It features an Intel® Core™2Duo E7300 2.66GHz processor and 4GB main memory. A 250GB Seagate 7200RPM hard disk drive is used to hold the OS and

home directories (/home). We use Fedora Core 9 with the Linux Kernel 2.6.27 and Ext3 file system. The storage devices are connected through the on-board SATA connectors.

	SSD-M	SSD-S
Capacity	80GB	32GB
NCQ	32	32
Flash memory	MLC	SLC
Page Size (KB)	4	4
Block Size (KB)	512	256
Read Latency (µs)	50	25
Write Latency (µs)	900	250
Erase Latency(µs)	3500	700

Table 1. Specification data of the SSDs.

We have selected two representative, state-of-the-art SSDs fabricated by a well-known SSD manufacturer for our studies. One is built on multi-level cell (MLC) flash memories and designed for the mass market, and the other is a high-end product built on faster and more durable single-level cell (SLC) flash memories. For commercial reasons, we refer to the two SSDs as *SSD-M* and *SSD-S*, respectively. Both SSDs deliver market-leading performance with full support of NCQ. Their designs are consistent with general-purpose SSD designs (e.g. [3, 10]) and representative in the mainstream technical trend on the market. PCI-E based flash devices (e.g. Fusion-io's ioDrive [11]) are designed for special-purpose systems with a different structure [17]. In this paper we focus on the SATA-based devices. Table 1 shows more details about the two SSDs.

Similar to our prior work [6], we use the *CFQ* (Completely Fair Queuing) scheduler, the default I/O scheduler in the Linux kernel, for the hard disk. We use the *noop* (No-optimization) scheduler for the SSDs to expose the internal behavior of the SSDs for our analysis.

3.2 Experimental Tools and Workloads

To study the internal parallelism of SSDs, two tools are used in our experiments. We use the Intel® Open Storage Toolkit [21] to generate various types of I/O workloads with different configurations, such as read/write ratio, random/sequential ratio, request size, and queue depth (the number of concurrent I/O jobs), and others. It reports bandwidth, IOPS, and latency. We also designed a tool, called *replayer*, which accepts a pr-recorded trace file for input and replays data accesses to a storage device. It facilitates precisely repeating a workload directly at the block device level. We use the two tools to generate three access patterns.

- **Sequential** - Sequential data accesses using specified request size, starting from sector 0.
- **Random** - Random data accesses using specified request size. Blocks are randomly selected from the first 1024MB of the storage space, unless otherwise noted.
- **Stride** - Strided data accesses using specified request size, starting from sector 0 with a stride distance.

In our experiments, each workload runs for 30 seconds in default to limit trace size while collecting sufficient data. No partitions or file systems are created on the SSDs. Unlike previous work [26], which was performed over an Ext3 file system, all workloads in our experiments directly access the SSDs as raw block devices. All requests are issued to the devices synchronously with no think time.

Similar to the methods in [6], before each experiment we fill the storage space using sequential writes with a request size of 256KB and pause for 5 seconds. This re-initializes the SSD status and keeps the physical data layout largely remain constant across experiments [6].

3.3 Trace Collection

To analyze I/O traffic in detail, we use *blktrace* [4] to trace the I/O activities at the block device level. The trace data are first collected in memory and then copied to the hard disk drive to minimize the interference caused by tracing. The collected data are processed using *blkparse* [4] and our post-processing scripts and tools off line.

4 Uncovering SSD Internals

Before introducing our performance studies on the internal parallelism of SSDs, we first present a set of experimental techniques to uncover the SSD internals. Two reasons have motivated us to detect SSD internal structures. First, internal parallelism is an architecture-dependent resource. Knowing the key architectural features of an SSD is required to study and understand the observed device behavior. For example, our findings about the write-order based mapping (see Section 4.4) motivate us to further study the ill-mapped data layout issue, which has not been reported in prior literature. Second, the information detected can also be used for many other purposes in practice. For example, knowing the number of channels in an SSD, we can set a proper concurrency level and avoid over-parallelization.

On the other hand, obtaining such architectural information is particularly challenging for several reasons. (1) Architectural details about the SSD design are often regarded as critical intellectual property of manufacturers. To the best of our knowledge, certain information, such as the mapping policy, is not available in any datasheet or specification, though it is critical for us to understand and exploit the SSD performance potential. (2) Although SSD manufacturers normally provide standard specification data (e.g. peak bandwidth), much important information is absent or obsolete. In fact, across different product batches, hardware/firmware change is common and often cannot be timely reflected in public documents. (3) Most SSDs on the market carefully follow a strictly defined host interface standard (e.g. SATA [1]). Only limited information is allowed to pass through such a 'thin interface'. As a result, it is difficult, if not impossible, to directly get detailed internal information from the hardware. In this section, we present a set of experimental approaches to expose the SSD internals.

4.1 A Generalized Model

Despite various implementations, most SSDs strive to optimize performance essentially in a similar way – evenly distributing data accesses to maximize resource usage. Such a principle is applicable to the parallel structure of SSDs at different levels. Without losing generality, we define an abstract model to characterize such an organization based on open documents (e.g. [3, 10]): A *domain* is a set of flash memories that share a specific set of resources (e.g. channels). A domain can be further partitioned into *sub-domains* (e.g. packages). A *chunk* is a unit of data that is continuously allocated within one domain. Chunks are interleavingly placed over a set of N domains by following a *mapping policy*. A set of chunks across each of N domains are called a *stripe*. One may notice that this model is in principle similar to RAID [24]. In fact, SSD architects often adopt a RAID-0 like striping mechanism [3, 10], some even directly integrate a RAID controller inside an SSD [25]. However, as mentioned previously, the key difference is that SSDs use dynamic mapping, which may cause an ill-mapped data layout as we will see later.

In this work, we are particularly interested in examining three *key factors* that are directly related to internal parallelism. In our future work we will further extend the techniques to uncover other architectural features.

- **Chunk size** – the size of the largest unit of data that is continuously mapped within an individual domain.
- **Interleaving degree** – the number of domains at the same level. The interleaving degree is essentially determined by the redundancy of the resources (e.g. channels).
- **Mapping policy** – the method that determines the domain to which a chunk of logical data is mapped. This policy determines the physical data layout.

In the following, we will present a set of experimental approaches to *infer indirectly* the three key factors. Basically, we treat an SSD as a 'black box' and we assume the mapping follows some repeatable but unknown patterns. By injecting I/O traffic with carefully designed patterns, we observe the 'reactions' of the SSD, measured in several key metrics, e.g. latency and bandwidth. Based on this probing information, we can speculate the internal architectures and policies adopted in the SSD. Our solution shares a similar principle on characterizing RAID [9], but characterizing SSDs needs to explore their unique features (e.g. dynamic mapping). We also note that due to the complexity and diversity of SSD implementations, the retrieved information may not picture all the internal details, and our purpose is not to reverse-engineer the SSD hardware. Instead, we try to characterize the key architectural features of an SSD from the outside in a simple yet effective way. We have applied this technique to both SSDs and we found that it works pleasantly well in serving our performance analysis and optimization purposes. For brevity we only show the results for SSD-S. SSD-M behaves similarly.

4.2 Chunk Size

Program 1 Pseudocode of uncovering SSD internals

```
init_SSD():  sequentially write SSD w/ 256KB req.
rand_pos(A): get a random offset aligned to A sect.
read(P, S):  read S sectors at offset P sect.
stride_read(J,D): read 1 chunk with J jobs from
          offset 0, each read skips over D chunks
plot(X,Y,C): plot a point at (X,Y) for curve C
M:  an estimated max. possible chunk size
D:  an estimated max. possible interleaving degree

(I) detecting chunk size:
  init_SSD();              // initialize SSD space
  for (n = 1 sector; n <= M; n *= 2): //req. size
    for (k = 0 sector; k <= 2*M; k ++):// offset
      for (i = 0, latency=0; i < 100000; i ++):
         pos = rand_pos(M) + k;
         latency += read (pos, n);
      plot (k, latency/100000, n); //plot avg. lat.

(II) detecting interleaving degree:
  init_SSD();              // initialize SSD space
  for (j=2; j <=4; j*=2):  //  num. of jobs
    for (d = 1 chunk; d < 4*D; d ++)://stride dist.
       bw = stride_read (j, d);
       plot (d, bw, j);           //plot bandwidth
```

As a basic mapping unit, a chunk can be mapped in only one domain, and two continuous chunks are mapped in two separate domains. Suppose the chunk size is S. For any read that is aligned to S with a request size no larger than S, only one domain would be involved. With an offset of $\frac{S}{2}$ from an aligned position, a read would split into two domains equally. The latter case would be faster. Based on this feature, we designed an experiment to identify the chunk size. The pseudocode is shown in Program 1(I).

Figure 2 shows an example result on SSD-S. Each curve represents a request size. For brevity, we only show results for request sizes of 1-8 sectors with offsets increasing from 0 to 64 sectors. Except for the case of request size of one sector, a dip periodically appears on the curves as the offset increases. *The chunk size is the interval between the bottoms of two consecutive valleys*. In this case, the detected chunk size is 8 sectors (4KB), the flash page size, but note that a chunk can consist of multiple flash pages in some implementations [10]. We see a flat curve for 1 sector (512 bytes), because the smallest read/write unit in the OS kernel is one sector and cannot be mapped across two domains.

4.3 Interleaving Degree

In our model, chunks are organized into domains based on resource redundancy. Parallel accesses to data in multiple domains without resource sharing can achieve a higher bandwidth than doing that congested in one domain. Based on this feature, we designed an experiment to determine the interleaving degree, as shown in Program 1(II).

Figure 3(I) and (II) show the experimental results with 2 jobs and 4 jobs on SSD-S, respectively. In Figure 3 (I), we observe a periodically appearing dip. *The interleaving degree is the interval between the bottoms of two consecutive valleys, in units of chunks.* In this case, we observe 10 domains, each of which actually corresponds to one channel.

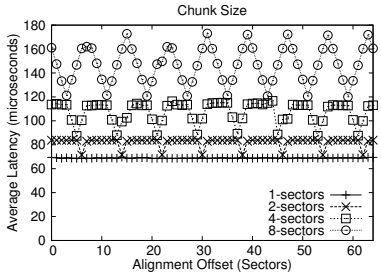


Figure 2. Detecting the chunk size on SSD-S. Each curve corresponds to a specified request size.

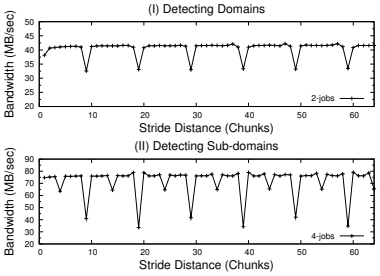


Figure 3. Detecting the interleaving degree on SSD-S. Figure (I) and (II) use 2 jobs and 4 jobs, respectively.

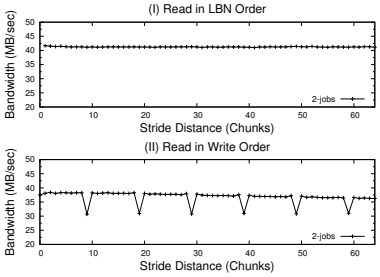


Figure 4. Detecting the mapping policy on SSD-S. Figure (I) and (II) represent two test cases.

The rationale is as follows. Suppose the number of domains is D and the data is interleavingly placed across the domains. When the stride distance, d, is $D \times n - 1$, where $n \geq 1$, every data access would exactly skip over $D - 1$ domains from the previous position and fall into the same domain. Since we issue two I/O jobs simultaneously, the parallel data accesses would compete for the same resource and the bandwidth is only around 33MB/sec. With other stride distances, the two parallel jobs would be distributed across two domains and thus could achieve a higher bandwidth (around 40MB/sec). We can also observe 2 sub-domains using a similar approach, as shown in Figure3 (II).

4.4 The Mapping Policy

The mapping policy determines the physical page to which a logical page is mapped. According to open documents (e.g. [3, 10]), two mapping policies are widely used in practice. (1) *LBA-based mapping* – for a given logical block address (LBA) with an interleaving degree, D, the block is mapped to a domain number (LBA mod D). (2) *Write-order-based mapping* – for the i_{th} write, the block is assigned to a domain number (i mod D). We should note here that write-order-based mapping is *not* the log-structured mapping policy [3], which appends data in a flash block to optimize write performance. Considering the wide adoption of the two mapping policies in SSD products, we will focus on these two policies in our experiments.

To determine which mapping policy is adopted, we first *randomly* overwrite the first 1024MB data with a request size of 4KB, the chunk size. After such a randomization, the blocks are relocated across the domains. If LBA-based mapping is adopted, the mapping should not be affected by such random writes. Thus, we repeat the experiment in Section 4.3, and we find after randomization, the pattern (repeatedly appearing dips) disappears (see Figure 4(I)), which means that the random overwrites have changed the block mapping, and LBA-based mapping is not used.

We then conduct another experiment to confirm that write-order-based mapping is adopted. We first randomly overwrite the first 1024MB space, and each chunk is written *once and only once*. Then we follow the same order in which blocks are written to issue reads to the SSD and repeat the same experiments in Section 4.3. For example, for the LBNs of random writes in the order of (91, 100, 23, 7, ...), we read data in the same order (91, 100, 23, 7, ...). If write-order based mapping is used, the blocks should be interleavingly allocated across domains in the order of writes (e.g. blocks 91, 100, 23, 7 are mapped to domain 0, 1, 2, 3, respectively), and reading data in the same order would repeat the same pattern (dips) we see before. Our experimental results confirm this hypothesis (Figure 4(II)). The physical data layout and the block mapping are *strictly* determined by the order of writes. We have also conducted experiments by mixing reads and writes, and we find that the layout is *only* determined by writes.

5 Performance Studies

Prepared with the knowledge about the internal structures of SSDs, we are now in a position to investigate the performance impact of internal parallelism. We strive to answer several related questions on *the performance benefits of parallelism*, *the interference between parallel jobs*, *the readahead mechanism with I/O parallelism*, and *the performance impact of physical data layout*.

5.1 What are the benefits of parallelism?

In order to quantify the potential benefits of I/O parallelism in SSDs, we run four workloads with different access patterns, namely *sequential read*, *sequential write*, *random read*, and *random write*. For each workload, we increase the request size from 1KB to 256KB[1], and we show the bandwidths with a queue depth increasing from 1 job to 32 jobs. In order to compare workloads with different request sizes, we use bandwidth (MB/sec), instead of IOPS (IO per second), as the performance metric. Figure 5 shows the experimental results for SSD-M and SSD-S. Since write-order-based mapping is used, as we see previously, random and sequential writes on both SSDs show similar patterns. For brevity, we only show the results of sequential writes here.

Differing from prior work [5], our experiments show a great performance improvement from parallelism on SSDs. The significance of performance gains depends on several factors, and we present several key observations here.

[1] Note that increasing request size would weaken the difference between random and sequential workloads. However, in order to give a full picture, we still show the experimental results of changing the request size from 1KB to 256KB for all the workloads here.

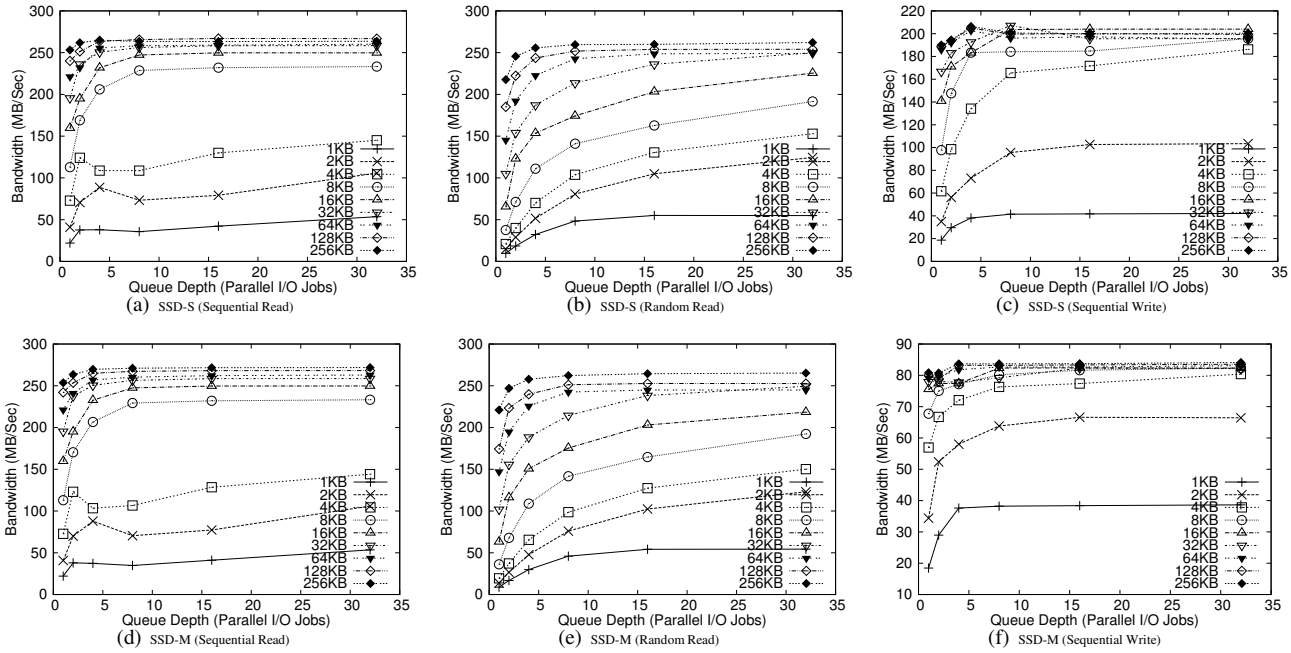


Figure 5. Bandwidths of the SSDs with increasing queue depth (the number of concurrent I/O jobs).

(1) *Workload access patterns determine the performance gains from parallelism.* In particular, *small and random reads* yield the most significant performance gains. For example, increasing queue depth from 1 to 32 jobs for random reads on SSD-S with a request size 4KB achieves a **7.2-fold** bandwidth increase. A large request (e.g. 256KB) benefits relatively less from parallelism, because the continuous logical blocks are often striped across domains and it already benefits from internal parallelism. This means that in order to exploit effectively internal parallelism, we can either increase request sizes or parallelize small requests.

(2) *Highly parallelized small/random accesses can achieve performance comparable to or even slightly better than large/sequential accesses without parallelism.* For example, with only one job, the bandwidth of random reads of 16KB on SSD-S is only 65.5MB/sec, which is 3.3 times lower than that of sequential reads of 64KB (221.3MB/sec). With 32 jobs, however, the same workload (16KB random reads) can reach a bandwidth of 225.5MB/sec, which is even slightly higher than the single-job sequential reads. This indicates that SSDs provide us with an alternative approach for optimizing I/O performance, which is to parallelize small and random accesses. Many new opportunities become possible. For example, database systems traditionally favor a large page size, because hard disk drives perform well with large requests. On SSDs, which are less sensitive to access patterns, a small page size can be a sound choice for optimizing buffer pool usage [13, 20].

(3) *The redundancy of available hardware resources physically limits the performance potential of increasing I/O parallelism.* When the queue depth increases over 8-10 jobs, further increasing parallelism receives diminishing benefits. This actually reflects our finding made in Section 4 that the two SSDs have 10 domains, each of which corresponds to a channel. When the queue depth exceeds 10, a channel has to be shared by more than one job. Further parallelizing I/O jobs can bring additional but smaller benefits.

(4) *Flash memory mediums (MLC/SLC) can provide different performance potential for parallel I/O jobs, especially for writes.* Writes on SSD-M, the MLC-based lower-end SSD, quickly reach the peak bandwidth (only about 80MB/sec) with a small request size at a low queue depth. SSD-S, the SLC-based higher-end SSD, shows much higher peak bandwidth (around 200MB/sec) and more headroom for parallelizing small writes. In contrast, less difference can be observed for reads on the two SSDs, since the main performance bottleneck is transferring data across the serial I/O bus rather than reading the flash medium [3].

(5) *Write performance is insensitive to access patterns, and parallel writes can perform faster than reads.* The uncovered write-order-based mapping policy indicates that the SSDs actually handle incoming writes in the same way, regardless of write patterns. This leads to the observed similar patterns for sequential writes and random writes. Together with the parallelized structure and the on-device buffer, write performance is highly optimized and can even outperform reads in some cases. For example, writes of 4KB with 32 jobs on SSD-S can reach a bandwidth of 186.1MB/sec, which is even **28.3%** higher than reads (145MB/sec). This surprising result is actually *opposite* to our long-existing common understanding about slow writes on SSDs.

On both SSD-S (Figure 5(a)) and SSD-M (Figure 5(d)), we can also observe a slight *dip* for sequential reads with small request sizes at a low concurrency level (e.g. 4 jobs with 4KB requests). This is related to interference in the readahead, and we will give detailed analysis in Section 5.3.

5.2 How do parallel reads and writes interfere with each other and cause performance degradation?

	Seq. Write	Rnd Write	None
Seq. Read	109.2	103.5	72.6
Rnd. read	32.8	33.2	21.3
None	61.4	59.4	

Table 2. Bandwidths (MB/sec) of co-running Reads and Writes.

Reads and writes on SSDs can interfere with each other for many reasons. (1) Both operations share many critical resources, such as the ECC engines and the lock-protected mapping table, etc. Parallel jobs accessing such resources need to be serialized. (2) Both writes and reads can generate background operations internally, such as readahead and asynchronous write-back [6]. (3) Mingled reads and writes can foil certain internal optimizations. For example, flash memory chips often provide a `cache mode` [2] to pipeline a sequence of reads or writes. A flash memory plane has two registers, *data register* and *cache register*. When handling a sequence of reads or writes, data can be transferred between the cache register and the controller, while concurrently moving another page between the flash medium and the data register. However, such pipelined operations must be performed in one direction, so mingling reads and writes would interrupt the pipelining.

To show such an interference, similar to that in the previous section, we use the toolkit to generate a pair of concurrent workloads. The two workloads access data in two separate 1024MB storage spaces. We choose four access patterns, namely *random reads*, *sequential reads*, *random writes* and *sequential writes*, and enumerate the combinations of running two workloads simultaneously. Each workload uses request size of 4KB and one job. We show the aggregate bandwidths (MB/sec) of two workloads co-running on SSD-S in Table 2. Co-running with 'none' means running a workload individually.

We find that *reads and writes have a strong interference with each other, and the significance of this interference highly depends on read access patterns.* If we co-run sequential reads and writes in parallel, the aggregate bandwidth exceeds that of any workload running individually, which means parallelizing workloads obtains benefits, although the aggregate bandwidths cannot reach the optimal results, the sum of the bandwidths of individual workloads. However, when running random reads and writes together, we see a strong negative impact. For example, sequential writes can achieve a bandwidth of 61.4MB/sec when running individually, however when running with random reads, the bandwidth drops by **a factor of 4.5** to 13.4MB/sec. Meanwhile, the bandwidth of random reads also drops from 21.3MB/sec to 19.4MB/sec. Apparently the co-running reads and writes strongly interfere with each other. Although we cannot exactly identify which specific reason causes such an effect, this case shows that we should be careful about mixing reads and writes.

5.3 How does I/O parallelism impact the effectiveness of readahead?

State-of-the-art SSDs implement a readahead mechanism to detect sequential data accesses and prefetch data into the on-device cache [6]. Parallelizing multiple sequential read streams would result in a sequence of mingled reads, which can interfere with the sequential-pattern-based readahead on SSDs. On the other hand, parallelizing reads can improve bandwidths. So there is a tradeoff between *increasing parallelism* and *retaining effective readahead*.

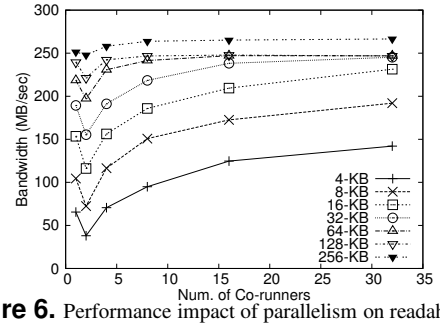


Figure 6. Performance impact of parallelism on readahead.

In order to examine the impact of parallelism on readahead, we generate multiple jobs, each of which sequentially reads an individual 1024MB space (i.e. the i_{th} job reads the i_{th} 1024MB space) simultaneously. We increase the concurrency from 1 to 32 jobs and vary the size from 4KB to 256KB. We compare the aggregate bandwidths of the co-running jobs on SSD-S. SSD-M shows similar results.

Figure 6 shows that in nearly all curves, there exists a *dip* when the queue depth is 2 jobs. For example, for request size of 4KB, the bandwidth drops by **43%** (from 65MB/sec to 37MB/sec). At that point, readahead is strongly inhibited due to mingled reads, and such a negative performance impact cannot be offset at a low concurrency level (2 jobs). When we further increase the concurrency level, the benefits coming from parallelism quickly compensate for the impaired readahead. Similarly, increasing the request size can alleviate the impact of interfered readahead by increasing the aggregate bandwidth. This case indicates that *readahead can be impaired by parallel sequential reads, especially at low concurrency levels and with small request sizes.* SSD architects can consider to include a more sophisticated sequential pattern detection mechanism to identify multiple sequential read streams to avoid this problem.

5.4 How does an ill-mapped data layout impact I/O parallelism?

The write-order-based mapping in SSDs has two advantages. First, high-latency writes can be evenly distributed across domains. This not only guarantees load balance, but also naturally balances available free flash blocks and evens out wear across domains. Second, writes across domains can be overlapped with each other, and the high latency of writes can be effectively hidden behind parallel operations.

On the other hand, such a write-order-based mapping may result in some negative effects. The largest one is that

since the data layout is completely determined by the order in which blocks are written on the fly, logical blocks can be mapped to only a subset of domains and result in an *ill-mapped data layout*. In the worst case, if a set of blocks is mapped into the same domain, data accesses would be congested and have to compete for the shared resources, which impairs the effectiveness of parallel data accesses.

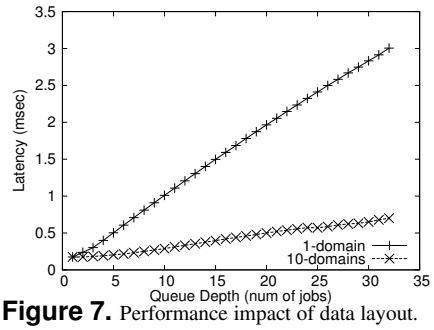


Figure 7. Performance impact of data layout.

To show quantitatively the impact of an ill-mapped physical data layout, we designed an experiment on SSD-S as follows. We first sequentially overwrite the first 1024MB of storage space to map the logical blocks evenly across domains. Knowing the physical data layout, we can create a trace of random reads with a request size of 4KB to access the 10 domains in a round-robin manner. Similarly, we create another trace of random reads to only one domain. Then we use the *replayer* to replay the two traces and we measure the average latencies for the two workloads. We vary the queue depth from 1 job to 32 jobs and compare the average latencies of parallel accesses to data that are concentrated in one domain and that are distributed across 10 domains. Figure 7 shows that when the queue depth is low, accessing data in one domain is comparable to doing it in 10 domains. However, as the I/O concurrency increases, the performance gap quickly widens. In the worst case, with a queue depth of 32 jobs, accessing data in the same domain (3ms) incurs **4.2 times** higher latency than doing that in 10 domains (0.7ms). This case shows that *an ill-mapped data layout can significantly impair the effectiveness of I/O parallelism*. It is also worth mentioning here that we also find the ill-mapped data layout can impair the effectiveness of readahead too.

In summary, our experimental results show that I/O parallelism can indeed provide significant performance improvement, but as the same time, we should also pay attention to the surprising results that come from parallelism, which provides us with a new understanding of SSDs.

6 Case Studies in Database Systems

In this section, we present a set of case studies in database systems as typical data-intensive applications to show the opportunities, challenges, and research issues brought by I/O parallelism on SSDs. Our purpose is not to present a set of well-designed solutions to address specific problems. Rather, we hope that through these case studies, we can show that exploiting the internal parallelism of SSDs can not only yield significant performance improvement in large data-processing systems, but more importantly, it also provides many emerging challenges and new research opportunities.

6.1 Data Accesses in a Database System

Storage performance is crucial to query executions in database management systems (DBMS). A key operation of query execution is *join* between two *relations* (tables). Various *operators* (execution algorithms) of a join can result in completely different data access patterns. For warehouse-style queries, the focus of our case studies, the most important two join operators are *hash join* and *index join* [12]. Hash join sequentially fetches each *tuple* (a line of record) from the driving input relation and probes an in-memory hash table. Index join fetches each tuple from the driving input relation and starts index lookups on B^+-trees of a large relation. In general, hash join is dominated by sequential data accesses on a huge fact table, while index join is dominated by random accesses during index lookups.

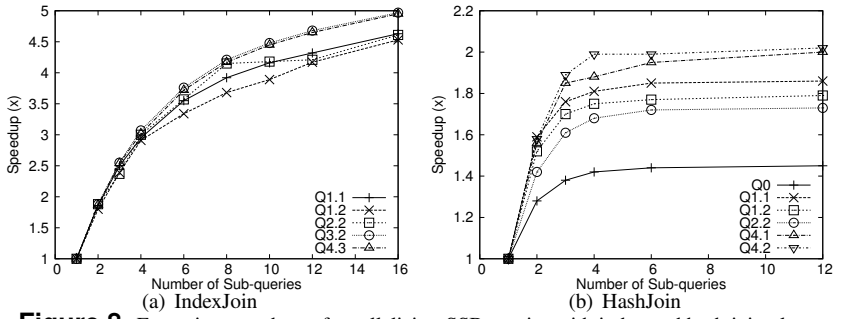
Our case studies are performed on the PostgreSQL 8.3.4. The working directory and the database are located on the SSD-S. We select Star Schema Benchmark (SSB) queries [22] (scale factor 5) as workloads. SSB workloads are considered to be more representative in simulating real warehouse workloads than TPC-H workloads, and they have been used in recent research work, e.g. [19].

6.2 Case 1: Parallelizing Query Execution

In this case, we study the effectiveness of parallelizing query executions on SSDs. Our query-parallelizing approach is similar to prior work [31]. Via data partitioning, a query can be segmented to multiple sub-queries, each of which contains joins on partitioned data sets and pre-aggregation. The sub-queries can be executed in parallel. The final result is obtained by applying a final aggregation over the results of sub-queries.

We study two categories of SSB query executions. One is using the index join operator and dominated by random accesses, and the other is using the hash join operator and dominated by sequential accesses. For index join, we partition the dimension table for the first-level join in the query plan tree. For hash join, we partition the fact table. For index join, we selected query Q1.1, Q1.2, Q2.2, Q3.2, and Q4.3. For hash join, besides Q1.1, Q1.2, Q2.2, Q4.1, Q4.2, we also examined a simple query (Q0), which only scans LINEORDER table with no join operation. Figure 8 shows the speedup (execution time normalized to the baseline case) of parallelizing the SSB queries with sub-queries.

We have the following observations. (1) For index join, which features intensive random data accesses, parallelizing query executions can speed up an index join plan by up to **a factor of 5**. We have also executed the same set of queries on a hard disk drive and observed no speedup, which means the factor of 5 speedup is not due to computational parallelism. This case again shows the importance of parallelizing random accesses (e.g. B^+-tree lookups) on an SSD. In fact, when executing an index lookup dominated query,



(a) IndexJoin (b) HashJoin

Figure 8. Execution speedups of parallelizing SSB queries with index and hash join plans.

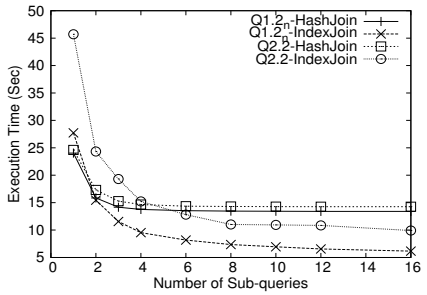


Figure 9. Index join vs. hash join.

the DBMS engine cannot fully utilize the SSD bandwidth, since each access is small and random. Splitting a query into multiple sub-queries effectively reduces the query execution time. (2) For hash join, which features sequential data accesses, parallelizing query executions can speed up a hash join plan by up to **a factor of 2**. Though less significant than that for index join plans, such a speedup is still impressive. (3) Parallelizing query execution with little computation receives relatively less benefits. For example, parallelizing Q0 provides a speedup of **a factor of 1.4**, which is lower than other queries. Since Q0 sequentially scans a big table with little computation, the SSD is kept busy and there is less room for overlapping I/O and computation, which limits further speedup through parallelism. This case clearly shows that in database systems, a typical data-processing application, I/O parallelism can provide substantial performance improvement on SSDs, especially for operations like index-tree lookups.

6.3 Case 2: Revisiting Query Optimizer

A critical component in DBMS is the *query optimizer*, which decides the plan and operator used for executing a query. Implicitly assuming the underlying storage device is an HDD, the optimizer estimates the execution times of a hash join plan and an index join plan, and selects an optimal plan for each query. On an HDD, a hash join enjoys an efficient sequential data access but needs to scan the whole table file; an index join suffers random index lookups but only needs to read a table partially. On an SSD, the situation becomes more complicated, since parallelism weakens the performance difference of various access patterns.

We select a standard Q2.2 and a variation of Q1.2 with a new predicate ("d_weeknuminyear=1") in the DATE table (denoted by $Q1.2_n$). We execute them first with no parallelism and then in a parallel way with sub-queries. Figure 9 shows the query execution times of the two plans with 1 to 16 sub-queries. One sub-query means no parallelism.

We find that SSD parallelism can greatly change the *relative strengths* of the two candidate plans. Without parallelism, the hash join plan is more efficient than the index join plan for both queries. For example, the index join for Q2.2 is 1.9 times slower than the hash join. The optimizer should choose the hash join plan. However, with parallelized sub-queries, the index join outperforms the hash join for both queries. For example, the index join for Q2.2 is 1.4 times faster than the hash join. This implies that the query optimizer *cannot* make an optimal decision if it does not take parallelism into account when estimating the execution costs of candidate plans on SSDs.

This case strongly indicates that when switching to an SSD-based storage, applications designed and optimized for magnetic disks must be carefully reconsidered, otherwise, the achieved performance can be suboptimal.

7 System Implications

Having presented our experimental and case studies, we present several important implications to system and application designers. We hope that our new findings can provide effective guidance and enhance our understanding of the properties of SSDs in the context of I/O parallelism. This section also summarizes our answers to the questions raised at the beginning of this paper.

Benefits of parallelism on SSD: Parallelizing data accesses can provide substantial performance improvement, and the significance of such benefits depends on workload access patterns, resource redundancy, and flash memory mediums. In particular, small random reads benefit the most from parallelism. Large sequential reads achieve less significant but still impressive improvement. Such a property of SSDs provides many opportunities for performance optimization in application designs. For example, many critical database operations, such as index-tree search, feature intensive random reads, which is exactly a best fit in SSDs. Application designers can focus on parallelizing these operations. In our experiments, we did not observe obvious negative effects of over-parallelization, however setting the concurrency level slightly over the number of channels is a reasonable choice. Finally, for practitioners who want to leverage the high parallel write performance, we highly recommend adopting high-end SLC-based SSDs to provide more headroom for serving workloads with intensive writes.

Random reads: A surprising result is that with parallelism, random reads can perform comparably and sometimes even slightly better than simply sequentializing reads without parallelism. Such a counter-intuitive finding indicates that we cannot continue to assume that sequential reads are always better than random reads. Traditionally, operating systems often make trade-offs for the purpose of organizing large sequential reads. For example, the anticipatory I/O scheduler [15] intentionally pauses issuing I/O

requests for a while and anticipates to organize a larger request in the future. In SSDs, such optimization may become less effective and sometimes may be even harmful, because of the unnecessarily introduced delays. On the other hand, some application designs become more complicated. For example, it becomes more difficult for the database query optimizer to choose an optimal query plan, because simply counting the number of I/Os and using static configuration numbers can no longer satisfy the requirement for accurately estimating the relative performance strengths of different join operators with parallelism.

Random writes: Contrary to our common understanding, parallelized random writes can achieve high performance, sometimes even *better* than reads. Moreover, writes are no longer highly sensitive to patterns (random or sequential) as commonly believed. The uncovered mapping policy explains this surprising result from the architectural level – the SSDs internally handle writes in the same way, regardless of access patterns. The indication is two-fold. On one hand, we can consider how to leverage the high write performance through parallelizing writes, e.g. how to commit synchronous transaction logging in a parallel manner [8]. On the other hand, it means that optimizing random writes specifically for SSDs may not continue to be as rewarding as in early generations of SSDs, and trading off read performance for writes would become a less attractive option. But we should still note that in extreme conditions, such as day-long writes [29] or under serious fragmentation [6], random writes are still a research issue.

Interference between reads and writes: Parallel reads and writes on SSDs interfere strongly with each other and can cause unpredictable performance variance. In OS kernels, I/O schedulers should pay attention to this emerging performance issue and avoid mingling reads and writes. A related research issue is on how to maintain a high throughput while avoiding such interference. At the application level, we should also be careful of the way of generating and scheduling reads and writes. An example is the hybrid-hash joins in database systems, which have clear phases with read-intensive and write-intensive accesses. When scheduling multiple hybrid-hash joins, we can proactively avoid scheduling operations with different patterns. A rule of thumb here is to schedule random reads together and separate random reads and writes whenever possible.

Physical data layout: The physical data layout in SSDs is dynamically determined by the order in which logical blocks are written. An ill-mapped data layout caused by such a write-order-based mapping can significantly impair the effectiveness of parallelism and readahead. In server systems, handling multiple write streams is common. Writes from the same stream can fall in a subset of domains, which would be problematic. We can adopt a simple random selection for scheduling writes to reduce the probability of such a worst case. SSD manufacturers can also consider adding a randomizer in the controller logic to avoid this problem. On the other hand, this mapping policy also provides us a powerful tool to *manipulate* data layout to our needs. For example, we can intentionally isolate a set of data in one domain to cap the usable I/O bandwidth.

Revisiting application designs: Many applications, especially data-processing applications, are often heavily tailored to the properties of hard disk drives and are designed with many implicit assumptions. Unfortunately, these HDD-based optimizations can become sub-optimal on SSDs. This calls our attention to revisiting carefully the application designs to make them fit well in SSD-based storage. On the other hand, the internal parallelism in SSDs also enables many new opportunities. For example, traditionally DBMS designers often assume an HDD-based storage and favor large request sizes. SSDs can extend the scope of using small blocks in DBMS design and bring many desirable benefits, such as improved buffer pool usage, which are highly beneficial in practice.

In essence, SSDs represent a fundamental change of storage architecture, and I/O parallelism is *the key* to exploiting the huge performance potential of such an emerging technology. More importantly, with I/O parallelism a low-end personal computer with an SSD is able to deliver as high an I/O performance as an expensive high-end system with a large disk array. This means that parallelizing I/O operations should be carefully considered in not only high-end systems but even in commodity systems. Such a paradigm shift would greatly challenge the optimizations and assumptions made throughout the existing system and application designs. We believe SSDs present not only challenges but also countless new research opportunities.

8 Other Related Work

Flash memory based storage technology is an active research area. In the research community, flash devices have received strong interest and been extensively studied (e.g. [3, 5–7, 10, 14, 27, 30]). Due to space constraints, here we only present the work most related to internal parallelism studied in this paper.

A detailed description about the hardware internals of flash memory based SSD has been presented in [3] and [10]. An early work has compared the performance of SSDs and HDDs [26]. We have presented experimental studies on SSD performance by using non-parallel workloads, and the main purpose is to reveal the internal behavior of SSDs [6]. This work focus on parallel workloads. A benchmark tool called uFLIP is presented in [5] to assess the flash device performance. This early study has reported that increasing concurrency cannot improve performance on early generations of SSDs. Researchers have studied the advances of SSDs and reported that with NCQ support, the recent generation of SSDs can achieve high throughput [20]. Our study shows that parallelism is a critical element to improve I/O performance and must be carefully considered in system and application designs. A mechanism called *FlashLogging* [8] adopts multiple low-cost flash drives to improve log processing, and parallelizing writes is an important consideration. Our study further shows that with parallelism,

many DBMS components need to be revisited, and the disk-based optimization model would become problematic and even error-prone on SSDs.

9 Conclusions

We have presented a comprehensive study on essential roles of exploiting internal parallelism in SSDs. We would like to conclude the paper with the following three messages, based on our analysis and new findings. First, effectively exploiting internal parallelism indeed can significantly improve I/O performance and largely remove the performance limitations of SSDs. This means that we must treat I/O parallelism as a *top priority* for optimizing I/O performance, even in commodity systems. Second, we must also pay particular attention to some potential side effects related to I/O parallelism on SSDs, such as the strong interference between reads and writes, and minimize their impact. Third, and most importantly, in the scenario of I/O parallelism, many of the existing optimizations specifically tailored to the property of HDDs can become ineffective or even harmful on SSDs, and we must revisit such designs. We hope this work can provide insights into the internal parallelism of SSD architecture and guide the application and system designers to utilize this unique merit of SSDs for achieving high-speed data processing.

Acknowledgments

We are grateful to the anonymous reviewers for their constructive comments. We also thank our colleague Bill Bynum for reading this paper and his suggestions. This research was partially supported by the US National Science Foundation under grants CCF-0620152, CCF-072380, and CCF-0913050, and Ministry of Industry and Information Technology of China under grant 2010ZX03004-003-03.

References

[1] Serial ATA revision 2.6. http://www.sata-io.org.

[2] Micron 8, 16, 32, 64gb SLC NAND flash memory data sheet. http://www.micron.com/, 2007.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. of USENIX'08*, 2008.

[4] Blktrace. http://linux.die.net/man/8/blktrace.

[5] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *CIDR'09*, 2009.

[6] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of SIGMETRICS/Performance'09*, 2009.

[7] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proc. of FAST'11*, 2011.

[8] S. Chen. FlashLogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD'09*, 2009.

[9] T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deconstructing storage arrays. In *ASPLOS'04*.

[10] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proc. of ISCA'09*, 2009.

[11] Fusion-io. ioDRIVE DUO datasheet. http://www.fusionio.com/PDFs/Fusion_ioDriveDuo_datasheet_v3.pdf, 2009.

[12] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[13] G. Graefe. The five-minute rule 20 years later, and how flash memory changes the rules. In *DaMon'07*, 2007.

[14] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of MICRO'09*, New York, NY, December 2009.

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

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

[17] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A file system for virtualized flash storage. In *FAST'10*, 2010.

[18] I. Koltsidas and S. Viglas. Flashing up the storage layer. In *Proc. of VLDB'08*, 2008.

[19] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. In *Proc. of VLDB'09*, 2009.

[20] S. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In *Proc. of SIGMOD'09*, 2009.

[21] M. Mesnier. Intel open storage toolkit. http://www.sourceforge.org/projects/intel-iscsi.

[22] P. O'Neil, B. O'Neil, and X. Chen. http://www.cs.umb.edu/poneil/StarSchemaB.PDF.

[23] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi. A high performance controller for NAND flash-based solid state disk (NSSD). In *NVSMW'06*, 2006.

[24] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of SIGMOD'88*, 1988.

[25] P. Perspective. OCZ Apex series 250GB solid state drive review. http://www.pcper.com/article.php?aid=661.

[26] M. Polte, J. Simsa, and G. Gibson. Comparing performance of solid state devices and mechanical disks. In *Proc. of the 3rd Petascale Data Storage Workshop*, 2008.

[27] T. Pritchett and M. Thottethodi. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of ISCA'10*, 2010.

[28] Samsung Elec. Datasheet (K9LBG08U0M). 2007.

[29] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *DaMon'09*, 2009.

[30] G. Sun, Y. Joo, Y. Chen, D. Niu, Y. Xie, Y. Chen, and H. Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of HPCA'10*, Jan 2010.

[31] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Query processing techniques for solid state drives. In *Proc. of SIGMOD'09*, 2009.

Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems

Feng Chen David Koufaty
Circuits and Systems Research
Intel Labs
Hillsboro, OR 97124, USA
{feng.a.chen,david.a.koufaty}@intel.com

Xiaodong Zhang
Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210, USA
zhang@cse.ohio-state.edu

ABSTRACT

With the fast technical improvement, flash memory based Solid State Drives (SSDs) are becoming an important part of the computer storage hierarchy to significantly improve performance and energy efficiency. However, due to its relatively high price and low capacity, a major system research issue to address is on how to make SSDs play their most effective roles in a high-performance storage system in cost- and performance-effective ways.

In this paper, we will answer several related questions with insights based on the design and implementation of a high performance hybrid storage system, called *Hystor*. We make the best use of SSDs in storage systems by achieving a set of optimization objectives from both system deployment and algorithm design perspectives. Hystor manages both SSDs and hard disk drives (HDDs) as one single block device with minimal changes to existing OS kernels. By monitoring I/O access patterns at runtime, Hystor can effectively identify blocks that (1) can result in long latencies or (2) are semantically critical (e.g. file system metadata), and stores them in SSDs for future accesses to achieve a significant performance improvement. In order to further leverage the exceptionally high performance of writes in the state-of-the-art SSDs, Hystor also serves as a write-back buffer to speed up write requests. Our measurements on Hystor implemented in the Linux kernel 2.6.25.8 show that it can take advantage of the performance merits of SSDs with only a few lines of changes to the stock Linux kernel. Our system study shows that in a highly effective hybrid storage system, SSDs should play a major role as an independent storage where the best suitable data are adaptively and timely migrated in and retained, and it can also be effective to serve as a write-back buffer.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Secondary Storage

General Terms

Design, Experimentation, Performance

Keywords

Solid State Drive, Hard Disk Drive, Hybrid Storage System

ICS'11, May 31–June 4, 2011, Tuscon, Arizona, USA.

1. INTRODUCTION

High-performance storage systems are in an unprecedented high demand for data-intensive computing. However, most storage systems, even those specifically designed for high-speed data processing, are still built on conventional hard disk drives (HDDs) with several long-existing technical limitations, such as low random access performance and high power consumption. Unfortunately, these problems essentially stem from the mechanic nature of HDDs and thus are difficult to be addressed via technology evolutions.

Flash memory based Solid State Drive (SSD), an emerging storage technology, plays a critical role in revolutionizing the storage system design. Different from HDDs, SSDs are completely built on semiconductor chips without any moving parts. Such a fundamental difference makes SSD capable of providing one order of magnitude higher performance than rotating media, and makes it an ideal storage medium for building high performance storage systems. For example, San Diego Supercomputer Center (SDSC) has built a large flash-based cluster, called *Gordon*, for high-performance and data-intensive computing [3]. In order to improve storage performance, Gordon adopts 256TB of flash memory as its storage [24]. However, such a design, which is backed by a $20 million funding from the National Science Foundation (NSF), may not be a typical SSD-based storage solution for widespread adoption, because the high cost and relatively small capacity of SSDs will continue to be a concern for a long time [11], and HDDs are still regarded as indispensable in the storage hierarchy due to the merits of low cost, huge capacity, and fast sequential access speed. In fact, building a storage system completely based on SSDs is often above the acceptable threshold in most commercial and daily operated systems, such as data centers. For example, a 32GB Intel® X25-E SSD costs around $12 per GB, which is nearly 100 times more expensive than a typical commodity HDD. To build a server with only 1TB storage, 32 SSDs are needed and as much as $12,000 has to be invested in storage solely. Even considering the price-drop trend, the average cost per GB of SSDs is still unlikely to reach the level of rotating media in the near future [11]. Thus, we believe that in most systems, SSDs should not be simply viewed as a replacement for the existing HDD-based storage, but instead SSDs should be a means to enhance it. Only by finding the fittest position of SSDs in storage systems, we can strike a right balance between performance and cost. Unquestionably, to achieve this goal, it is much more challenging than simply replacing HDDs with fast but expensive SSDs.

1.1 Critical Issues

A straightforward consideration of integrating SSD in the existing memory hierarchy is to treat the state-of-the-art SSDs, whose cost and performance are right in between of DRAM memory and

HDDs, as a *secondary-level cache*, and apply caching policies, such as LRU or its variants, to maintain the most likely-to-be-accessed data for future reuse. However, the SSD performance potential could not be fully exploited unless the following related important issues, from both policy design and system deployment perspectives, be well addressed. In this paper, we present a unique solution that can best fit SSDs in the storage hierarchy and achieve these optimization goals.

1. Effectively identifying the most performance-critical blocks and fully exploiting the unique performance potential of SSDs – Most existing caching policies are temporal locality based and strive to identify the most likely-to-be-reused data. Our experimental studies show that the performance gains of using SSDs over HDDs is highly dependent on workload access patterns. For example, random reads (4KB) on an Intel® X25-E SSD can achieve up to 7.7 times higher bandwidth than that on an HDD, while the speedup for sequential reads (256KB) is only about 2 times. Besides identifying the most likely-to-be-reused blocks as done in most previous studies, we must further identify the blocks that can receive the most significant performance benefits from SSDs. We have systematically analyzed various workloads and identified a simple yet effective metric based on extensive experimental studies. Rather than being randomly selected, this metric considers both *temporal locality* and *data access patterns*, which well meets our goal of distinguishing the most performance-critical blocks.

2. Efficiently maintaining data access history with low overhead for accurately characterizing access patterns – A major weakness of many LRU-based policies is the lack of knowledge about deep data access history (i.e. recency only). As a result, they cannot identify critical blocks for a long-term optimization and thus suffer the well-known cache pollution problem (workloads, such as reading a streaming file, can easily evict all valuable data from the cache [14]). As a key difference from previous studies, we profile and maintain data access history as an important part of our hybrid storage. This avoids the cache pollution problem and facilitates an effective reorganization of data layout across devices. A critical challenge here is how to efficiently maintain such data access history for a large-scale storage system, which is often in Terabytes. In this paper, a special data structure, called *block table*, is used to meet this need efficiently.

3. Avoiding major kernel changes in existing systems while effectively implementing the hybrid storage management policies – Residing at the bottom of the storage hierarchy, a hybrid storage system should improve system performance without intrusively changing upper-level components (e.g. file systems) or radically modifying the common interfaces shared by other components. Some previously proposed solutions attempt to change the existing memory hierarchy design by inserting non-volatile memory as a new layer in the OS kernels (e.g. [18, 19]); some require that the entire file system be redesigned [34], which may not be viable in practice. Our design carefully isolates complex details behind a standard block interface, which minimizes changes to existing systems and guarantees compatibility and portability, which are both critical in practice.

In our solution, compared to prior studies and practices, SSD plays a different role. We treat the high-capacity SSD as a part of storage, instead of a caching place. Correspondingly, different from the conventional caching-based policies, which frequently update the cache content on each data access, we only periodically and asynchronously reorganize the layout of blocks across devices for a long-term optimization. In this paper, we show that this arrangement makes SSDs the best fit in storage hierarchy.

1.2 Hystor: A Hybrid Storage Solution

In this paper, we address the aforesaid four issues by presenting the design and implementation of a practical hybrid storage system, called *Hystor*. Hystor integrates both low-cost HDDs and high-speed SSDs as a *single* block device and isolates complicated details from other system components. This avoids undesirable significant changes to existing OS kernels (e.g. file systems and buffer cache) and applications.

Hystor achieves its optimization objectives of data management through three major components. First, by monitoring I/O traffic on the fly, Hystor automatically learns workload access patterns and identifies performance-critical blocks. Only the blocks that can bring the most performance benefits would be gradually remapped from HDDs to high-speed SSDs. Second, by effectively exploiting high-level information available in existing interfaces, Hystor identifies semantically-critical blocks (e.g. file system metadata) and timely offers them a high priority to stay in the SSD, which further improves system performance. Third, incoming writes are buffered into the low-latency SSD for improving performance of write-intensive workloads. We have prototyped Hystor in the Linux Kernel 2.6.25.8 as a stand-alone kernel module with only a few lines of codes added to the stock OS kernel. Our experimental results show that Hystor can effectively exploit SSD performance potential and improve performance for various workloads

1.3 Our Contributions

The contribution of this work is threefold. (1) We have identified an effective metric to represent the performance-critical blocks by considering both temporal locality and data access patterns. (2) We have designed an efficient mechanism to profile and maintain detailed data access history for a long-term optimization. (3) We present a comprehensive design and implementation of a high performance hybrid storage system, which improves performance for accesses to the high-cost data blocks, semantically-critical (file system metadata) blocks, and write-intensive workloads with minimal changes to existing systems. While we have prototyped Hystor as a kernel module in software, a hardware implementation (e.g. in a RAID controller card) of this scheme is possible, which can further reduce system deployment difficulty as a drop-in solution.

In the rest of this paper, we will first examine the SSD performance advantages in Section 2. We study how to identify the most valuable data blocks and efficiently maintain data access history in Section 3 and 4. Then we present the design and implementation of Hystor in Section 5 and 6. Section 7 presents our experimental results. Related work is presented in Section 8. The last section concludes this paper.

2. SSD PERFORMANCE ADVANTAGES

Understanding the relative performance strengths of SSDs over HDDs is critical to efficiently leverage limited SSD space for the most performance gains. In this section, we evaluate an Intel® X25-E 32GB SSD [13], a representative high-performance SSD, and compare its performance with a 15,000 RPM Seagate® Cheetah® 15.5k SAS hard disk drive, a typical high-end HDD. Details about the two storage devices and experiment system setup are available in Section 7.

In general, a workload can be characterized by its read/write ratio, random/sequential ratio, request size, and think time, etc. We use the Intel® Open Storage Toolkit [21] to generate four typical workloads, namely *random read*, *random write*, *sequential read*, and *sequential write*. For each workload, we set the queue depth of 32 jobs and vary the request size from 1KB to 256KB. All workloads directly access raw block devices to bypass the buffer cache

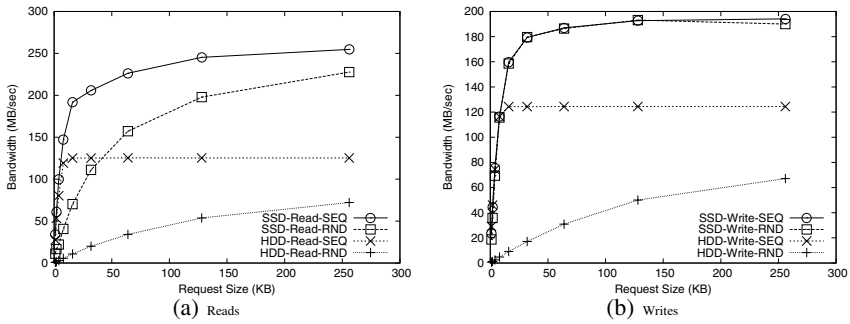


Figure 1: I/O Bandwidths for reads and writes on the Intel® X25-E SSD and the Seagate® Cheetah® HDD.

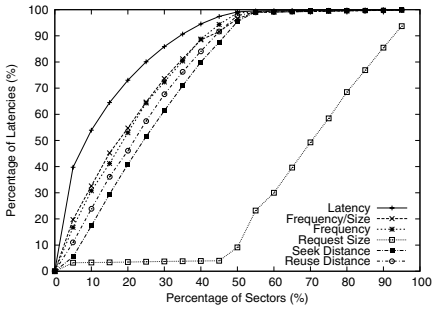


Figure 2: Accumulated HDD latency of sectors sorted using different metrics.

and file system. All reads and writes are synchronous I/O with no think time. Although real-life workloads can be a mix of various access patterns, we use the four synthetic microbenchmarks to qualitatively characterize the SSD. Figure 1 shows the experimental results. We made several findings to guide our system designs.

First, as expected, the most significant performance gain of running workloads on the SSD appears in random data accesses with small request sizes, for both reads and writes. For example, with a request size of 4KB, random reads and random writes on the SSD achieve more than 7.7 times and 28.5 times higher bandwidths than on the HDD, respectively. As request size increases to 256KB, the relative performance gains of sequential reads and writes diminish to 2 times and 1.5 times, respectively. It clearly shows that achievable performance benefits are highly dependent on workload access patterns, and we must identify the blocks that can bring the most performance benefits by migrating them into SSDs.

Second, contrary to the long-existing understanding about low write performance on SSDs, we have observed an exceptionally high write performance on the SSD (up to 194MB/sec). Similar findings have been made in recent performance studies on the state-of-the-art SSDs [5, 6]. As a high-end product, the Intel® X25-E SSD is designed for commercial environments with a sophisticated FTL design [13]. The highly optimized SSD internal designs significantly improve write performance and make it possible to use an SSD as a write-back buffer for speeding up write-intensive workloads, such as email servers.

Third, we can see that write performance on the SSD is largely independent of access patterns, and random writes can achieve almost identical performance as sequential writes. This indicates that it is unnecessary to specially treat random writes for performance purposes like in some prior work. This allows us to remove much unnecessary design complexity. In addition, we also find that writes on the SSD can quickly reach a rather high bandwidth (around 180MB/sec) with a relatively small request size (32KB) for both random and sequential workloads. This means that we can achieve the peak bandwidth on SSDs without need of intentionally organizing large requests as we usually do on HDDs.

Based on these observations, we summarize two key issues that must be considered in the design of Hystor as follows.

1. We need to recognize workload access patterns to identify the most *high-cost* data blocks, especially those blocks being randomly accessed by small requests, which cause the worst performance for HDDs.

2. We can leverage the SSD as a write-back buffer to handle writes, which often raise high latencies in HDDs. Meanwhile, we do not have to treat random writes specifically, since random writes on SSD can perform as fast as sequential writes.

3. HIGH-COST DATA BLOCKS

Many workloads have a *small* data set contributing a *large* percentage of the aggregate latency in data accesses. A critical task for Hystor is to identify the most performance-critical blocks.

3.1 Identifying High-Cost Blocks

A simple way to identify the high-cost blocks is to observe I/O latency of accessing each block and directly use the accumulated latency as an indicator to label the 'cost' of each block. In a hybrid storage, however, once we remap blocks to the SSD, we cannot observe their access latency on HDD any more. Continuing to use the previously observed latency would be misleading, if the access pattern changes after migration. Thus, directly using I/O latency to identify high-cost blocks is infeasible.

Some prior work (e.g. [12, 27]) maintains an on-line hard disk model to predict the latency for each incoming request. Such a solution heavily relies on precise hard disk modeling based on detailed specification data, which is often unavailable in practice. More importantly, as stated in prior work [12], as the HDD internals become increasingly more complicated (e.g. disk cache), it is difficult, if not impossible, to accurately model a modern hard disk and precisely predict the I/O latency for each disk access.

We propose another approach – using a pattern-related metric as an *indicator* to indirectly *infer* access cost without need of knowing the exact latencies. We associate each block with a selected metric and update the metric value by observing accesses to the block. The key issue here is that the selected metric should have a strong correlation to access latency, so that by comparing the metric values, we can effectively estimate the *relative* access latencies associated to blocks and identify the relatively high-cost ones. Since the selected metric is device independent, it also frees us from unnecessary burdens of considering specific hardware details (e.g. disk cache size), which can vary greatly across devices.

3.2 Indicator Metrics

In order to determine an effective indicator metric that is highly correlated to access latencies, we first identify four candidate metrics, namely *request size*, *frequency*, *seek distance*, *reuse distance*, and also consider their combinations. We use the *blktrace* tool [2] to collect I/O traces on an HDD for a variety of workloads. In the off-line analysis, we calculate the accumulated latency for each accessed block, as well as the associated candidate metric values. Then we rank the blocks in the order of their metric values. For example, concerning the metric *frequency*, we sort the blocks from the most frequently accessed one to the least frequently accessed one, and plot the accumulated latency in that order.

Figure 2 shows an example of TPC-H workload (other workloads are not shown due to space constraints). The X axis shows the top

percentage of blocks, sorted in a specific metric value, and the Y axis shows the percentage of aggregate latency of these blocks. Directly using latency as the metric represents the ideal case. Thus, the closer a curve is to the *latency* curve, the better the corresponding metric is. Besides the selected four metrics, we have also examined various combinations of them, among which **frequency/request size** is found to be the most effective one. For brevity, we only show the combination of *frequency/request size* in the figure. In our experiments, we found that *frequency/request size* emulates latency consistently better across a variety of workloads, the other metrics and combinations, such as seek distance, work well for some cases but unsatisfactorily for the others.

The metric **frequency/request size** is selected with a strong basis – it essentially describes both **temporal locality** and **access pattern**. In particular, *frequency* describes the temporal locality and *request size* represents the access pattern for a given workload. In contrast to the widely used recency-based policies (e.g. LRU), we use frequency to represent the temporal locality to avoid the well-recognized cache pollution problem for handling weak-locality workloads (e.g. scanning a large file would evict valuable data from the cache) [14]. It is also worth noting here that there is an *intrinsic correlation* between request size and access latency. First of all, the average access latency per block is highly correlated to request size, since a large request can effectively amortize the seek and rotational latency over many blocks. Second, the request size also reflects workload access patterns. As the storage system sits at the bottom of the storage hierarchy, the sequence of data accesses observed at the block device level is an optimized result of multiple upper-level components. For example, the I/O scheduler attempts to merge consecutive small requests into a large one. Thus, a small request observed at the block device often means that either the upper-level components cannot further optimize data accesses, or the application accesses data in such a non-sequential pattern. Finally, small requests also tend to incur high latency, since they are more likely to be intervened by other requests, which would cause high latencies from disk head seeks and rotations. Although this metric cannot perfectly emulate the ideal curve (latency), as we see in the figure, it performs consistently the best in various workloads and works well in our experiments.

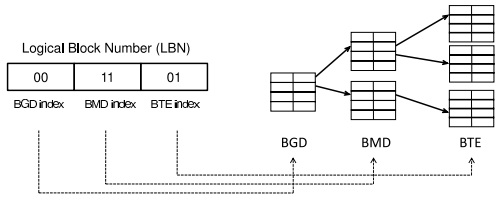
4. MAINTAINING DATA ACCESS HISTORY

To use the metric values to profile data access history, we must address two critical challenges – (1) how to represent the metric values in a compact and efficient way, and (2) how to maintain such history information for each block of a large-scale storage space (e.g. Terabytes). In short, we need an efficient mechanism to profile and maintain data access history at a low cost.

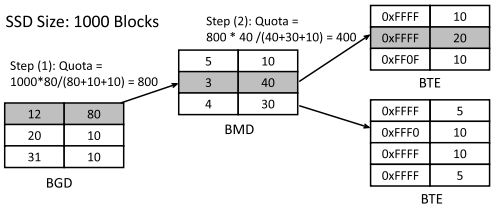
4.1 The Block Table

We use the *block table*, which was initially introduced in our previous work [15], to maintain data access history. Akin to the page table used in virtual memory management, the block table has three levels, *Block Global Directory* (BGD), *Block Middle Directory* (BMD), and *Block Table Entry* (BTE), as shown in Figure 3(a). The three levels, namely BGD, BMD, and BTE, of this structure essentially describe the storage space segmented in units of regions, sub-regions, and blocks, accordingly.

In the block table, each level is composed of multiple 4KB pages, each of which consists of multiple entries. A block's logical block number (LBN) is broken into three components, each of which is an index to an entry in the page at the corresponding level. Each BGD or BMD entry has a 32-bit *pointer* field pointing to a (BMD or BTE) page in the next level, a 16-bit *counter* field recording data



(a) *The block table structure*



(b) *Traversing the block table*

Figure 3: The Block Table. Each box represents an entry page. In BGD and BMD pages, left and right columns represent *unique* and *counter* fields. In BTE pages, two columns represent *flag* and *counter* fields. The two steps show the order of entries being traversed from BGD to BTE entries.

access information, and a 16-bit *unique* field tracking the number of BTE entries belonging to it. Each BTE entry has a 16-bit *counter* field and a 16-bit *flag* field to record other properties of a block (e.g. whether a block is a metadata block). This three-level tree structure is a very efficient vehicle to maintain storage access information. For a given block, we only need three memory accesses to traverse the block table and locate its corresponding information stored in the BTE entry.

4.2 Representing Indicator Metric

We have developed a technique, called *inverse bitmap*, to encode the *request size* and *frequency* in the block table. When a block is accessed by a request of *N* sectors, an *inverse bitmap*, *b*, is calculated using the following equation:

$$b = 2^{max(0,7-\lfloor log_2 N \rfloor)} \tag{4.1}$$

As shown above, inverse bitmap encodes request size into a single byte. The smaller a request is, the bigger the inverse bitmap is.

Each entry at each level of the block table maintains a *counter*. The values of the counters in the BGD, BMD, and BTE entries represent the 'hotness' of the regions, sub-regions, and blocks, respectively. Upon an incoming request, we use the block's LBN as an index to traverse the block table through the three levels (BGD→BMD→BTE). At each level, we increment the counter of the corresponding entry by *b*. So the more frequently a block is accessed, the more often the corresponding counter is incremented. In this way, we use the inverse bitmap to represent the size for a given request, and the counter value, which is updated upon each request, to represent the indicator metric *frequency/request size*. A block with a large counter value is regarded as a high-cost (i.e. hot) block. By comparing the counters associated with blocks, we can identify the blocks that should be relocated to the SSD.

As time elapses, a counter (16 bits) may overflow. In such a case, we right shift all the counters of the entries in the *same* entry page by one bit, so that we can still preserve the information about the relative importance that the counter values represent. Since such a right shift operation is needed for a minimum of 512 updates to a single LBN, this operation would cause little overhead. Also note

that we do not need to right shift counters in other pages, because we only need to keep track of the *relative* hotness for entries in a page, and the relative hotness among the pages is represented by the entries in the upper level.

The block table is a very efficient and flexible data structure to maintain the block-level information. For example, the full block table can be maintained in persistent storage (e.g. SSD). During the periodic update of the block table, we can load only the relevant table pages that need to be updated into memory (but at least one page at each level). Also note that the block table is a sparse data structure – we only need to maintain history for *accessed* blocks. This means that the spatial overhead in persistent storage is only proportional to the *working-set size* of workloads. In the worst case, e.g. scanning the whole storage space, the maximum spatial overhead is approximately 0.1% of the storage space (a 32-bit BTE entry per 4KB chunk). In practice, however, since most workloads only access partial storage space, the spatial overhead would be much lower. If needed, we can further release storage space by trimming the rarely updated table pages. This flexibility of the block table provides high scalability when handling a large storage space.

5. THE DESIGN OF HYSTOR

After introducing the indicator metric and the block table, we are now in a position to present the design of Hystor. Our goal is to best fit the SSD in the storage systems and effectively exploit its unique performance potential with minimal system changes.

5.1 Main Architecture

Hystor works as a pseudo block device at the block layer, as shown in Figure 4(a). The upper-level components, such as file systems, view it simply as a *single* block device, despite the complicated internals. Users can create partitions and file systems on it, similar to any directly attached drive. With minimal system changes, Hystor is easy to integrate into existing systems.

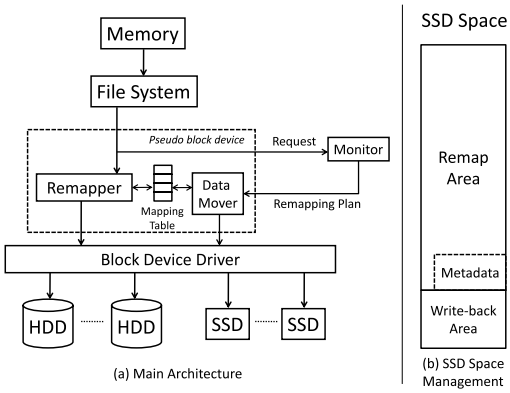


Figure 4: Architecture of Hystor.

Hystor has three major components, namely *remapper*, *monitor*, and *data mover*. The remapper maintains a mapping table to track the original location of blocks on the SSD. When an incoming request arrives at the remapper, the mapping table is first looked up. If the requested block is resident in the SSD, the request is redirected to the SSD, otherwise, it is serviced from the HDD. This remapping process is similar to the software RAID controller. The remapper also intercepts and forwards I/O requests to the monitor, which collects I/O requests and updates the block table to profile workload access patterns. The monitor periodically analyzes the data access history, identifies the blocks that should be remapped to the SSD, and requests the data mover to relocate data blocks across storage devices. The monitor can run in either kernel mode or user mode. The data mover is responsible for issuing I/O commands to the block devices and updating the mapping table accordingly to reflect the most recent changes.

5.2 Logical Block Mapping

Hystor integrates multiple HDDs and SSDs and exposes a linear array of logical blocks to the upper-level components. Each logical block is directly mapped to a physical block in the HDD and indexed using the logical block number (LBN). A logical block can be selected to remap to the SSD, and its physical location in the SSD is dynamically selected. Hystor maintains a *mapping table* to keep track of the remapped logical blocks. This table is also maintained in a statically specified location in the persistent storage (e.g. the first few MBs of SSD), and it is rebuilt in the volatile memory at startup time. Changes to the mapping table are synchronously written to the storage to survive power failures. In memory, the table is organized as a B-tree to speedup lookups, which only incur minimal overhead with several memory accesses. Since only remapped blocks need to be tracked in the mapping table, the spatial overhead of the mapping table is small and proportional to the SSD size. Techniques, similar to the dynamic mapping table [10], can also be applied to only maintain the most frequently accessed mapping entries to further reduce the in-memory mapping table size.

In essence, Hystor manages remapped blocks in an 'inclusive' manner, which means that, when a block is remapped to the SSD, its original home block in the HDD would not be recycled. We choose such an inclusive design for three reasons. First, the SSD capacity is normally at least one order of magnitude smaller than the HDDs, thus, there is no need to save a small amount of capacity for low-cost HDDs. Also, if we attempt to fully utilize the HDD space, a large mapping table has to be maintained to track every block in the storage space (often in granularity of TBs), which would incur high overhead. Second, when blocks in the SSD need to be moved back to the HDD, extra high-cost I/O operations are required. In contrast, if blocks are duplicated to the SSD, we can simply drop the replicas in the SSD, as long as they are clean. Finally, this design also significantly simplifies the implementation and avoids unnecessary complexity.

5.3 SSD Space Management

In Hystor, the SSD plays *a major role* as a storage to retain the best suitable data, and *a minor role* as a write-back buffer for writes. Accordingly, we logically segment the SSD space into two regions, *remap* area and *write-back* area, as shown in Figure 4(b). The remap area is used to maintain the identified critical blocks, such as the high-cost data blocks and file system metadata blocks. All requests, including both reads and writes, to the blocks in the remap area are directed to the SSD. The write-back area is used as a buffer to temporarily hold dirty data of incoming write requests. All other requests are directed to the HDD. Blocks in the write-back area are periodically synchronized to the HDD and recycled for serving incoming writes. We use a configurable quota to guard the sizes of the two regions, so there is no need to physically segment the two regions on the SSD.

We allocate blocks in the SSD in *chunks*, which is similar in nature to that in RAID [25]. This brings two benefits. First, when moving data into the SSD, each write is organized more efficiently in a reasonably large request. Second, it avoids splitting a request into several excessively small requests. In our prototype, we choose an initial chunk size of 8 sectors (4KB). We will further study the effect of chunk size on performance in Section 7.5. In Hystor, all data allocation and management are performed in chunks.

5.4 Managing the Remap Area

The *remap* area is used to maintain identified critical blocks for a long-term optimization. Two types of blocks can be remapped to the SSD: (1) the high-cost data blocks, which are identified by analyzing data access history using the block table, and (2) file system metadata blocks, which are identified through available semantic information in OS kernels.

5.4.1 Identifying High-Cost Data Blocks

As shown in Section 4, the block table maintains data access history in forms of the *counter* values. By comparing the counter values of entries at the BGD, BMD, or BTE levels, we can easily identify the hot regions, sub-regions, and blocks, accordingly. The rationale guiding our design is that the hottest blocks in the hottest regions should be given the highest priority to stay in the high-speed SSD.

Program 1 Pseudocode of identifying candidate blocks.

```
counter():       the counter value of an entry
total_cnt():     the aggregate value of counters
                 of a block table page
sort_unique_asc(): sort entries by unique values
sort_counter_dsc(): sort entries by counter values
quota:           the num. of available SSD blocks

sort_unique_asc(bgd_page); /*sort bgd entries*/
bgd_count = total_cnt(bgd_page);
for each bgd entry && quota > 0; do
  bmd_quota  = quota*counter(bgd)/bgd_count;
  bgd_count -= counter(bgd);
  quota     -= bmd_quota;

  bmd_page = bgd->bmd;    /*get the bmd page*/
  sort_unique_asc(bmd_page);/*sort bmd entries*/
  bmd_count = total_cnt(bmd_page);
  for each bmd entry && bmd_quota > 0; do
    bte_quota  = bmd_quota*counter(bmd)/bmd_count;
    bmd_count -= counter(bmd);
    bmd_quota -= bte_quota;

    bte_page   = bmd->bte;
    sort_counter_dsc(bte_page);
    for each bte entry && bte_quota > 0; do
      add bte to the update list;
      bte_quota --;
    done
    bmd_quota += bte_quota; /*unused quota*/
  done
  quota += bmd_quota; /*unused quota*/
done
```

Program 1 shows the pseudocode of identifying high-cost blocks. We first proportionally allocate SSD space quota to each BGD entry based on their counter values, since a hot region should be given more chance of being improved. Then we begin from the BGD entry with the least number of BTE entries (with the smallest *unique* value), and repeat this process until reaching the BTE level, where we allocate entries in the descending order of their counter values. The blocks being pointed to by the BTE entries are added into a candidate list until the quota is used up. The unused quota is accumulated to the next step. In this way, we recursively determine the hottest blocks in the region and allocate SSD space to the regions correspondingly. Figure 3(b) illustrates this process, and it is repeated until the available space is allocated.

5.4.2 Reorganizing Data Layout across Devices

Workload access pattern changes over time. In order to adapt to the most recent workload access patterns, Hystor periodically wakes up the monitor, updates the block table, and recommends a list of candidate blocks that should be put in SSD, called *updates*, to update the remap area. Directly replacing all the blocks in the SSD with the updates would be over-sensitive to workload dynamics. Thus we take a 'smooth update' approach as follows.

We manage the blocks in the remap area in a list, called the *resident list*. When a block is added to the resident list or accessed, it is put at the top of the list. Periodically the monitor wakes up and sends a list of updates as described in Section 5.4.1 to the data mover. For each update, the data mover checks whether the block is already in the resident list. If true, it informs the monitor that the block is present. Otherwise, it reclaims the block at the bottom of the resident list and reassign its space for the update. In both cases, the new block (update) is placed at the top of the resident list. The monitor repeats this process until a certain number (e.g. 5-10% of the SSD size) of blocks in the resident list are updated. So we identify the high-cost blocks based on the most recent workloads and place them at the top of the resident list, and meanwhile, we always evict unimportant blocks, which have become rarely accessed and thus reside at the list bottom. In this way, we *gradually* merge the most recently identified high-cost data set into the old one and avoid aggressively shifting the whole set from one to another.

Once the resident list is updated, the data mover is triggered to perform I/O operations to relocate blocks across devices asynchronously in the background. Since the data mover can monitor the I/O traffic online and only reorganize data layout during idle periods (e.g. during low-load hours), the possible interference to foreground jobs can be minimized.

5.4.3 User-level Monitor

As a core engine of Hystor, the monitor receives intercepted requests from the remapper, updates the block table, and generates a list of updates to relocate blocks across devices. The monitor can work in either kernel mode or user mode with the same policy. We implemented both in our prototype.

Our user-level monitor functions similarly to *blktrace* [2]. Requests are temporarily maintained in a small log buffer in the kernel memory and periodically passed over to the monitor, a user-level daemon thread. Our prototype integrates the user-level monitor into *blktrace*, which allows us to efficiently use the existing infrastructure to record I/O trace by periodically passing requests to the monitor. The kernel-level monitor directly conducts the same work in the OS kernel.

Compared to the kernel-level monitor, the user-level monitor incurs lower overhead. Memory allocation in the user-level monitor is only needed when it is woken up, and its memory can even be paged out when not in use. Since each time we only update the data structures partially, this significantly reduces the overhead.

5.4.4 Identifying Metadata Blocks

File system metadata blocks are critical to system performance. Before accessing a file, its metadata blocks must be loaded into memory. With only a small amount of SSD space, relocating file system metadata blocks into SSD can effectively improve I/O performance, especially for metadata-intensive workloads during a cold start. Hystor attempts to identify these *semantically critical* blocks and proactively remap them to the SSD to speed up file accesses at an early stage, which avoids high-cost cold misses at a later time. In order to avoid intrusive system changes, we take a conservative approach to leverage the information that is already available in the existing OS kernels.

In the Linux kernel, metadata blocks are tagged such that an I/O scheduler can improve metadata read performance. So far, this mechanism is used by some file systems (e.g. Ext2/Ext3) for metadata reads. In our prototype, we modified a single line at the block

layer to leverage this available information by tagging incoming requests for metadata blocks. No other changes to file systems or applications are needed. Similar tagging technique is also used in Differentiated Storage Services [22]. When the remapper receives a request, we check the incoming request's tags and mark the requested blocks in the block table (using the *flag* field of BTE entries). The identified metadata blocks are remapped to the SSD. Currently, our implementation is effective for Ext2/Ext3, the default file system in Linux. Extending this approach to other file systems needs additional minor changes.

Another optional method, which can identify metadata blocks without any kernel change, is to *statically* infer the property of blocks by examining their logical block numbers (LBN). For example, the Ext2/Ext3 file system segments storage space into 128MB block groups, and the first few blocks of each group are always reserved for storing metadata, such as inode bitmap, etc. Since the location of these blocks is statically determined, we can mark them as metadata blocks. As such a solution assumes certain file systems and default configurations, it has not been adopted in our prototype.

5.5 Managing the Write-back Area

The most recent generation of SSDs has shown an exceptionally good write performance, even for random writes (50-75µs for a 4KB write [5]). This makes the SSD a suitable place for buffering dirty data and reducing latency for write-intensive workloads. As a configurable option, Hystor can leverage the high-speed SSD as a buffer to speed up write-intensive workloads.

The blocks in the write-back area are managed in two lists, a *clean list* and a *dirty list*. When a write request arrives, we first allocate SSD blocks from the *clean list*. The new dirty blocks are written into the SSD and added onto the *dirty list*. We maintain a counter to track the number of dirty blocks in the write-back area. If this number reaches a *high watermark*, a *scrubber* is waken up to write dirty blocks back to the HDD until reaching a *low watermark*. Cleaned blocks are placed onto the clean list for reuse. Since writes can return immediately once the data is written to the SSD, the synchronous write latency observed by foreground jobs is very low. We will examine the scrubbing effect in Section 7.4. Another optional optimization is to only buffer small write requests in the SSD, which further improves the use of the write-back area

As mentioned previously, we do not specifically optimize random writes, since the state-of-the-art SSDs provide high random write performance [5, 6]. One might also be concerned about the potential reliability issues of using SSD as a write-back buffer, since flash memory cells can wear out after a certain number of program/erase cycles. Fortunately, unlike early generations of SSDs, the current high-end SSDs can provide a reasonably high reliability. For example, the Mean Time Before Failure (MTBF) rating of the Intel® X25-E SSDs is as high as 2 million hours [13], which is comparable to that of typical HDDs. In this paper we do not consider the low-end SSDs with poor write performance and low reliability, which are not suitable for our system design goals.

6. IMPLEMENTATION ISSUES

We have prototyped Hystor in the Linux kernel 2.6.25.8 as a stand-alone kernel module with about 2,500 lines of code. The user-level monitor is implemented as a user-level daemon thread with about 2,400 lines of code. Neither one requires any modifications in the Linux kernel. The alternative kernel implementation of the monitor module consists of about 4,800 lines of code and only about 50 lines of code are inserted in the stock Linux kernel.

In our prototype, the remapper is implemented based on the software RAID. When the kernel module is activated, we use `dmsetup` to create a new block device with appointed HDD and SSD devices. By integrating the Hystor functionality on the block layer, we can avoid dealing with some complex issues, such as splitting and merging requests to different devices, since the block layer already handles these issues. The downside of this design is that requests observed at this layer may be further merged into larger requests later, so we track the LBN of the last request to estimate the mergeable request size. Another merit of this design is that Hystor can work seamlessly with other storage, such as RAID and SAN storage. For example, a RAID device can be built upon Hystor virtual devices, similarly, Hystor can utilize RAID devices too. Such flexibility is highly desirable in commercial systems.

As a core engine of Hystor, the monitor can work in either kernel mode or user mode. In both cases, the monitor is implemented as a daemon thread. Periodically it is triggered to process the collected I/O requests, update the block table, and generate updates to drive the data mover to perform data relocation. In kernel mode, the observed requests are held in two log buffers. If one buffer is full, we swap to the other to accept incoming requests, and the requests in the full buffer are updated to the block table in parallel. In user mode, requests are directly passed to the user-level daemon. The analysis of data access history can also be done offline. In our current prototype, we maintain the block table and the mapping table full in memory, and in our future work we plan to further optimize memory usage by only loading partial tables into memory as discussed previously.

7. EVALUATION

7.1 Experimental System

Our experimental system is an Intel® D975BX system with a 2.66GHz Intel® Core™ 2 Quad CPU and 4GB main memory on board. Our prototype system consists of a Seagate® Cheetah® 15k.5 SAS hard drive and a 32GB Intel® X25-E SSD, both of which are high-end storage devices on the market. Also note that we only use *partial* SSD space in our experiments to avoid overestimating the performance. Table 1 lists the detailed specification of the two devices. The HDDs are connected through an LSI® MegaRaid® 8704 SAS card and the SSD uses a SATA 3.0Gb/s connector.

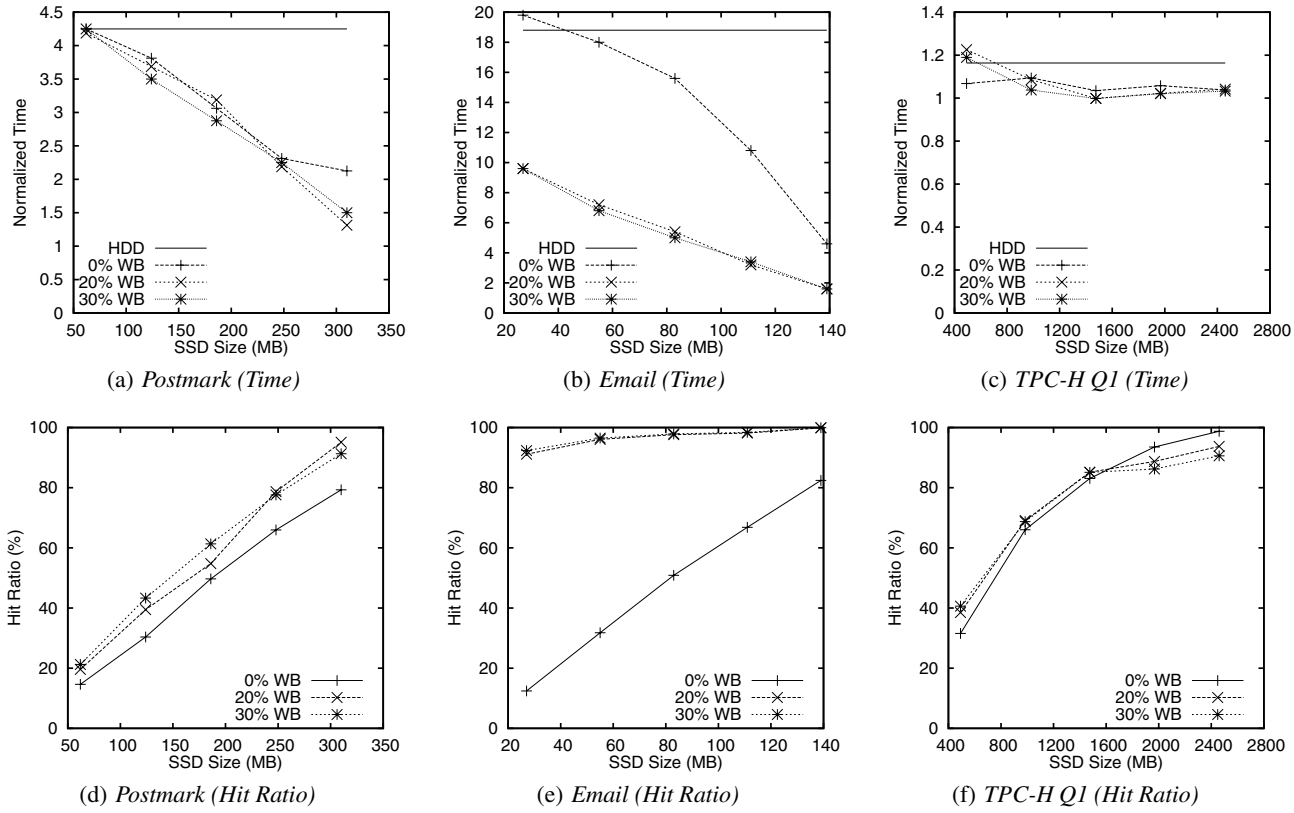
	X25-E SSD	Cheetah HDD
Capacity	32GB	73GB
Interface	SATA2	SAS
Read Bandwidth	250 MB/Sec	125 MB/Sec
Write Bandwidth	180 MB/Sec	125 MB/Sec

Table 1: Specifications of the SSD and the HDD.

We use Fedora™ Core 8 with the Linux kernel 2.6.25.8 and Ext3 file system with default configurations. In order to minimize the interference, the operating system and home directory are stored in a separate hard disk drive. We use the *noop* (No-op) I/O scheduler, which is suitable for non-HDD devices [5,6], for the SSD. The hard disk drives use the *CFQ* (Completely Fair Queuing) scheduler, the default scheduler in the Linux kernel, to optimize the HDD performance. The on-device caches of all the storage devices are enabled. The other system configurations use the default values.

7.2 Performance of Hystor

In general, the larger the SSD size is, the better the Hystor's performance is. In order to avoid overestimating the performance improvement, we first estimate the working-set size (the number of blocks being accessed during execution) of each workload by



(a) *Postmark (Time)* (b) *Email (Time)* (c) *TPC-H Q1 (Time)*

(d) *Postmark (Hit Ratio)* (e) *Email (Hit Ratio)* (f) *TPC-H Q1 (Hit Ratio)*

Figure 5: Normalized execution times and hit ratios of workloads. The horizontal line represents the time of running on HDD.

examining the collected I/O traces off line, then we conduct experiments with five configurations of the SSD size, namely 20%, 40%, 60%, 80%, and 100% of the working-set size. Since we only conservatively use *partial* SSD space in our experiments, the performance of Hystor can be even better in practice. To examine the effectiveness of the write-back area, we configure three write-back area sizes, namely 0%, 20%, and 30% of the available SSD space.

For each workload, we perform the baseline experiments on the SSD. We rerun the experiments with various configurations of the SSD space. We show the execution times normalized to that of running on the SSD-only system. Therefore, the normalized execution time '1.0' represents the ideal case. In order to compare with the worst case, running on the HDD-only system, we plot a horizontal line in the figures to denote the case of running on the HDD. Besides the normalized execution times, we also present the hit ratios of I/O requests observed at the remapper. A request to blocks resident in the SSD is considered a *hit*, otherwise, it is a *miss*. The hit ratio describes what percentage of requests are serviced from the SSD. Due to space constraints, we only present results for the user-mode monitor here, and the kernel monitor shows similar results. Figure 5 shows the normalized execution times and hit ratios.

7.2.1 Postmark

Postmark is a widely used file system benchmark [28]. It creates 100 directories and 20,000 files, then performs 100,000 transactions (reads and writes) to stress the file system, and finally deletes files. This workload features intensive small random data accesses.

Figure 5(a) shows that as the worst case, postmark on the HDD-only system runs 4.2 times slower than on the SSD-only system. Hystor effectively improves performance for this workload. With the increase of SSD space, the execution time is reduced till close to an SSD-only system, shown as a linear curve in the figure. In this case, most data blocks are accessed with similar patterns, which is challenging for Hystor to identify high-cost data blocks based on access patterns. However, Hystor still provides performance gains proportional to available SSD space.

Since this workload features many small writes, allocating a large write-back area helps improve hit ratios as well as execution times. Figure 5(d) shows that with the SSD size of 310MB, allocating 30% of the SSD space for write-back can improve hit ratio from 79% to 91%, compared to without write-back area. Accordingly, the execution time is reduced from 34 seconds to 24 seconds, which is a 29% reduction.

Also note that multiple writes to the same block would cause synchronization issues. With a smaller write-back area, dirty blocks have to be more frequently flushed back to the HDD due to capacity limit. When such an operation is in progress, incoming write requests to the same blocks have to be suspended to maintain consistency, which further artificially increases the request latency. For example, when the cache size grows to 310MB, the amount of hits to the lock-protected blocks decreases by a factor of 4, which translates into a decrease of execution time.

7.2.2 Email

Email was developed by University of Michigan based on Postmark for emulating an email server [31]. It is configured with 500 directories, 500 files, and 5,000 transactions. This workload has intensive synchronous writes with different append sizes and locations based on realistic mail distribution function, and it features a more skewed distribution of latencies. Most data accesses are small random writes, which are significantly faster on the SSD.

Figure 5(b) shows that running *email* exclusively on the SSD is 18.8 times faster than on the HDD. With no write-back area, the performance of Hystor is suboptimal, especially for a small

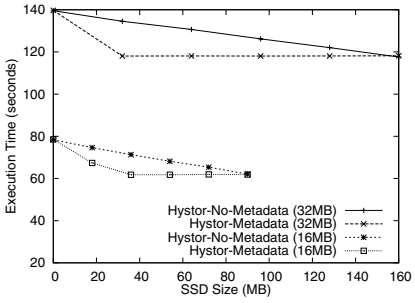


Figure 6: Optimization for metadata blocks. 32MB and 16MB refer to the two workloads. Hystor with and without optimizing metadata are referred to as *Hystor-Metadata* and *Hystor-No-Metadata*.

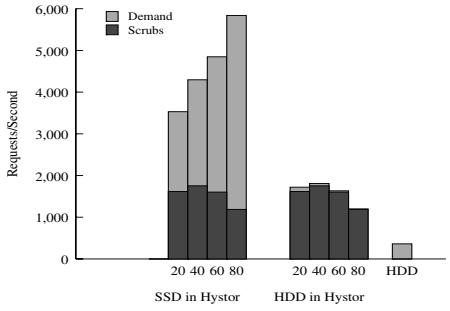


Figure 7: Request arrival rate in *email*. The numbers on X axis for each bar refer to various configurations of the SSD size (% of the working-set size). HDD refers to the HDD-only system.

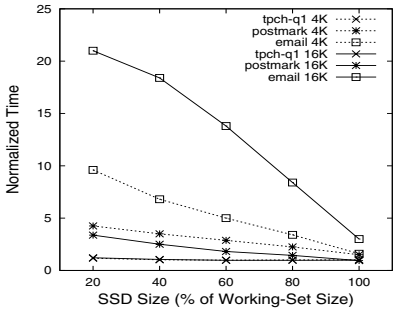


Figure 8: Effect of chunk size on performance. 4K and 16K refer to the chunk sizes, respectively. The numbers on X axis refer to various configurations of the SSD size (% of the working-set size).

SSD size. As we see in the figure, with SSD size of 27MB (20% of the working-set size), Hystor may even be slightly worse than the HDD-only system, due to additional I/O operations and increased probability of split requests. Without the write-back area, data blocks remapped in the SSD may not be necessarily to be reaccessed in the next run. This leads to a hit ratio of only 12.4% as shown in Figure 5(e). In contrast, with the write-back area, the hit ratio quickly increases to over 90%. This shows that the write-back area also behaves like a small cache to capture some short-term data reuse. As a result, the execution time is reduced by a half.

7.2.3 TPC-H Query 1

TPC-H Q1 is the query 1 from the TPC-H database benchmark suite [33]. It runs against a database (scale factor 1) managed by PostgreSQL 8.1.4 database server. Different from the other workloads, this workload does not benefit much from running on the SSD, since its data accesses are more sequential and less I/O intensive. As shown in Figure 5(c), running this workload on the HDD-only system is only 16% slower than running on the SSD. However, since the reuse of data blocks is more significant, the hit ratio in this case is higher. Figure 5(f) shows that with SSD size of 492MB, the hit ratio of incoming requests is about 30% to 40%. When the SSD size is small, the write-back area may introduce extra traffic, which leads to a 2-5% slowdown compared to running on HDD. As the write-back area size increases, the number of write-back operations is reduced dramatically, which improves the I/O performance.

7.3 Metadata Blocks

Hystor also identifies metadata blocks of file systems and remaps them to the SSD. We have designed an experiment to show how such an optimization improves performance.

In Ext2/Ext3 file systems, a large file is composed of many data chunks, and *indirect blocks* are used to locate and link these chunks together. As a type of metadata, indirect blocks do not contain file content but are crucial to accessing files. We create a 32GB file and use the Intel® Open Storage Toolkit [21] to generate two workloads, which randomly read 4KB data each time until 16MB and 32MB of data are read. This workload emulates data accesses in files with complex internal structures, such as virtual disk file used by virtual machines. In such random workloads, the accessed file data are unlikely to be reused, while indirect blocks would be reaccessed, thus holding metadata blocks in the SSD would be beneficial. We use this example to compare the performance of Hystor with and without optimization for file system metadata blocks, denoted as *Hystor-Metadata* and *Hystor-No-Metadata* respectively.

Figure 6 shows the experimental results of Hystor-No-Metadata and Hystor-Metadata. Both approaches eventually can speed up the two workloads by about 20 seconds. However, Hystor-Metadata is able to achieve that performance with a much smaller SSD space. For the workload reading 32MB data, Hystor-Metadata identifies and remaps nearly all indirect blocks to the SSD with just 32MB of SSD space. In contrast, Hystor-No-Metadata lacks the capability of identifying metadata blocks. Since only around 20% of the blocks being accessed are metadata blocks, most blocks remapped to the SSD are file content data blocks, which are unfortunately almost never reused. Therefore Hystor-No-Metadata requires about 160MB of SSD space to cover the whole working-set, while Hystor-Metadata needs only 32MB SSD space. A similar pattern can be observed in the case of reading 16MB data.

This experiment shows that optimization for metadata blocks can effectively improve system performance with only a small amount of SSD space, especially for metadata-intensive workloads. More importantly, different from identifying high-cost data blocks by observing workload access patterns, we can proactively identify these *semantically critical* blocks at an early stage, so high-cost cold misses can be avoided. It is also worth noting that the three major components in Hystor are complementary to each other. For example, although the current implementation of Hystor identifies metadata blocks only for read requests, writes to these metadata blocks still can benefit from being buffered in the write-back area.

7.4 Scrubbing

Dirty blocks buffered in the write-back area have to be written back to the HDD in the background, called *scrubbing*. Each scrub operation can cause two additional I/O operations – a read from the SSD and a write to the HDD. Here we use *email*, the worst case for scrubs, to study how scrubbing affects system performance. Figure 7 shows the request arrival rate (number of requests per second) for *email* configured with four SSD sizes (20-80% of the working-set size) and a 20% write-back fraction . The requests are broken down by the source, internal scrubbing daemon or the upper-layer components, denoted as *scrubs* and *demand* in the figure, respectively.

As shown in Figure 7, the request arrival rate in Hystor is much higher than that in the HDD-only system. This is due to two reasons. First, the average request size for HDD-only system is 2.5 times larger than that in Hystor, since a large request in Hystor may split into several small ones to different devices. Second, two additional I/O operations are needed for each scrub. We can see that, as the SSD size increases to 80% of the working-set size, the arrival rate of scrub requests drops by nearly 25% on the SSD, due to less frequent scrubbing. The arrival rate of on-demand requests in-

creases as the SSD size increases, because the execution time is reduced and the number of on-demand requests remains unchanged.

An increase of request arrival rate may not necessarily lead to an increase of latency. In the case with 80% of the working-set size, as many as 5,800 requests arrive on the SSD every second. However, we do not observe a corresponding increase of execution time (see Figure 5(b)) and the SSD I/O queue still remains very short. This is mainly because the high bandwidth of the SSD (up to 250MB/sec) can easily absorb the extra traffic. On the HDD in Hystor, the request arrival rate reaches over 1,800 requests per second. However, since these requests happen in the background, the performance impact on the foreground jobs is minimal.

This case shows that although a considerable increase of request arrival rate is resident on both storage devices, conducting background scrubbing causes minimal performance impact, even for write-intensive workloads.

7.5 Chunk Size

Chunk size is an important parameter in Hystor. A large chunk size is desirable for reducing memory overhead of the mapping table and the block table. On the other hand, a small chunk size can effectively improve utilization of the SSD space, since a large chunk may contain both hot and cold data.

Figure 8 compares performance of using a chunk size of 8 sectors (4 KB) and 32 sectors (16 KB). We only present data for the cache with a 20% write-back fraction here. We can see that with a large chunk size (16KB), the performance of *email* degrades significantly due to the underutilized SSD space. Recall that most of the requests in *email* are small, hot and cold data could co-exist in a large chunk, which causes the miss rate to increase by four-fold. With the increase of SSD size, such a performance gap is reduced, but it is still much worse than using 4KB chunks. The other workloads are less sensitive to chunk size.

This experiment shows that choosing a proper chunk size should consider the SSD size. For a small-capacity SSD, a small chunk size should be used to avoid wasting precious SSD space. A large SSD can use a large chunk size and afford the luxury of increased internal fragmentation in order to reduce overhead. In general, a small chunk size (e.g. 4KB) is normally sufficient for optimizing performance. Our prototype uses a chunk size of 4KB in default.

8. RELATED WORK

Flash memory and SSDs have been actively studied recently. There is a large body of research work on SSDs (e.g. [1, 5–8]). A survey [9] summarizes the key techniques in flash memory based SSDs. Here we present the work most related to this paper.

The first set of work is generally cache-based solutions. An early work [19] uses flash memory as a secondary-level file system buffer cache to reduce power consumption and access latencies for mobile computers. SmartSaver [4] uses a small-factor flash drive to cache and prefetch data for saving disk energy. A hybrid file system, called Conquest [34], merges the persistent RAM storage into the HDD-based storage system. Conquest caches small files, metadata, executables, shared libraries into the RAM storage and it demands a substantial change to file system designs. AutoRAID [35] migrates data inside the HDD-based RAID storage to improve performance and cost-efficiency based on patterns. Sun® Solaris™ [18] can set a high-speed device as a secondary-level buffer cache between main memory and hard disk drives. Microsoft® Windows® ReadyBoost [23] takes a similar approach to use a flash device as an extension of main memory. Intel® TurboMemory [20] uses a small amount of flash memory as a cache to buffer disk data and uses a threshold size to filter large requests. Kgil et al. [16] propose to use flash memory and DRAM as a disk cache and adopt an LRU-based wear-level aware replacement policy [16]. SieveStore [29] uses a selective caching approach by tracking the access counts and caching the most popular blocks in solid state storage. Hystor views and places SSDs in the storage hierarchy in another way – the high-speed SSD is used as a part of storage rather than an additional caching tier. As such, Hystor only reorganizes data layout across devices periodically and asynchronously, rather than make caching decision on each data access. In addition, recognizing the non-uniform performance gains on SSDs, Hystor not only adopts frequency, rather than recency that has been commonly used in LRU-based caching policies, to better describe the temporal locality, and it also further differentiates various workload access patterns and attempts to make the best use of the SSD space with minimized system changes.

Some other prior work proposes to integrate SSD and HDD together and form a hybrid storage system. Differentiated Storage Services [22] attempts to classify I/O requests and passes information to storage systems for QoS purposes. The upper-level components (e.g. file systems) classify the blocks and the storage system enforces the policy by assigning blocks to different devices. ComboDrive [26] concatenates SSD and HDD into one single address space, and certain selected data and files can be moved into the faster SSD space. As a block-level solution, Hystor hides details from the upper-level components and does not require any modification to applications. Considering the disparity of handling reads and writes in SSDs, Koltsidas and Viglas propose to organize SSD and HDD together and place read-intensive data in SSD and write-intensive data in HDD for performance optimization [17]. Soundararajan et al. propose a solution to utilize HDD as a log buffer to reduce writes and improve the longevity of SSDs [32]. Recently, I-CASH [30] has been proposed to use SSD to store seldom-changed reference data blocks and HDD to store a log of deltas, so that random write traffic to SSD can be reduced. Our experimental studies show that the state-of-the-art SSDs have exceptionally high write performance. Specifically optimizing write performance for SSDs can yield limited benefits on these advanced hardware. In fact, Hystor attempts to leverage the high write performance of SSDs and our experimental results show that such a practice can effectively speed up write-intensive workloads.

9. CONCLUSION

Compared with DRAM and HDD, the cost and performance of SSDs are nicely placed in between. We need to find the fittest position of SSDs in the existing systems to strike a right balance between performance and cost. In this study, through comprehensive experiments and analysis, we show that we can identify the data that are best suitable to be held in SSD by using a simple yet effective metric, and such information can be efficiently maintained in the block table at a low cost. We also show that SSDs should play a major role in the storage hierarchy by adaptively and timely retaining performance- and semantically-critical data, and it can also be effective as a write-back buffer for incoming write requests. By best fitting the SSD into the storage hierarchy and forming a hybrid storage system with HDDs, our hybrid storage prototype, Hystor, can effectively leverage the performance merits of SSDs with minimized system changes. We believe that Hystor lays out a system framework for high-performance storage systems.

10. ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive comments. We also thank our colleagues at Intel® Labs, espe-

cially Scott Hahn and Michael Mesnier, for their help and support through this work. We also would like to thank Xiaoning Ding at Intel® Labs Pittsburgh, Rubao Lee at the Ohio State University, and Shuang Liang at EMC® DataDomain for our interesting discussions. This work was partially supported by the National Science Foundation (NSF) under grants CCF-0620152, CCF-072380, and CCF-0913150.

11. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of USENIX'08*, Boston, MA, June 2008.

[2] Blktrace. http://linux.die.net/man/8/blktrace.

[3] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of ASPLOS'09*, Washington, D.C., March 2009.

[4] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proceedings of ISLPED'06*, Tegernsee, Germany, Oct. 2006.

[5] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of SIGMETRICS/Performance'09*, Seattle, WA, June 2009.

[6] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of HPCA'11*, San Antonio, Texas, Feb 12-16 2011.

[7] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of FAST'11*, San Jose, CA, Feb 15-17 2011.

[8] C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proceedings of ISCA'09*, Austin, TX, June 2009.

[9] E. Gal and S. Toledo. Algorithms and data structures for flash memories. In *ACM Computing Survey'05*, volume 37(2), pages 138–163, 2005.

[10] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of ASPLOS'09*, Washington, D.C., March 2009.

[11] J. Handy. Flash memory vs. hard disk drives - which will win? http://www.storagesearch.com/semico-art1.html.

[12] L. Huang and T. Chieuh. Experiences in building a software-based SATF scheduler. In *Tech. Rep. ECSL-TR81*, 2001.

[13] Intel. Intel X25-E extreme SATA solid-state drive. http://www.intel.com/design/flash/nand/extreme, 2008.

[14] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of USENIX'05*, Anaheim, CA, April 2005.

[15] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proceedings of FAST'05*, San Francisco, CA, December 2005.

[16] T. Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *Proceedings of ISCA'08*, Beijing, China, June 2008.

[17] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. In *Proceedings of VLDB'08*, Auckland, New Zealand, August 2008.

[18] A. Leventhal. Flash storage memory. In *Communications of the ACM*, volume 51(7), pages 47–51, July 2008.

[19] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Science*, Wailea, HI, Jan 1994.

[20] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. In *ACM Transactions on Storage*, volume 4, May 2008.

[21] M. P. Mesnier. Intel open storage toolkit. http://www.sourceforge.org/projects/intel-iscsi.

[22] M. P. Mesnier and J. B. Akers. Differentiated storage services. *SIGOPS Oper. Syst. Rev.*, 45:45–53, February 2011.

[23] Microsoft. Microsoft Windows Readyboost. http://www.microsoft.com/windows/windows-vista/features/readyboost.aspx, 2008.

[24] A. Patrizio. UCSD plans first flash-based supercomputer. http://www.internetnews.com/hardware/article.php/3847456, November 2009.

[25] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of SIGMOD'88*, Chicago, IL, June 1988.

[26] H. Payer, M. A. Sanvido, Z. Z. Bandic, and C. M. Kirsch. Combo Drive: Optimizing cost and performance in a heterogeneous storage device. In *Proceedings of the 1st Workshop on integrating solid-state memory into the storage hierarchy (WISH'09)*, 2009.

[27] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, portable I/O scheduling with the disk mimic. In *Proceedings of USENIX'03*, San Antonio, TX, June 2003.

[28] Postmark. A new file system benchmark. http://www.netapp.com/tech_library/3022.html, 1997.

[29] T. Pritchett and M. Thottethodi. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of ISCA'10*, Saint-Malo, France, June 2010.

[30] J. Ren and Q. Yang. I-CASH: Intelligently coupled array of ssd and hdd. In *Proceedings of HPCA'11*, San Antonio, Texas, Feb 2011.

[31] S. Shah and B. D. Noble. A study of e-mail patterns. In *Software Practice and Experience*, volume 37(14), 2007.

[32] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of FAST'10*, San Jose, CA, February 2010.

[33] Transaction Processing Performance Council. TPC Benchmark H. http://www.tpc. org/tpch, 2008.

[34] A. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a Disk/Persistent-RAM hybrid file system. In *Proceedings of the USENIX'02*, Monterey, CA, June 2002.

[35] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *ACM Tran. on Computer Systems*, volume 14, pages 108–136, Feb 1996.

Client-aware Cloud Storage

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

Michael P. Mesnier
Intel Labs
michael.mesnier@intel.com

Scott Hahn
Intel Labs
scott.hahn@intel.com

***Abstract*—Cloud storage is receiving high interest in both academia and industry. As a new storage model, it provides many attractive features, such as high availability, resilience, and cost efficiency. Yet, cloud storage also brings many new challenges. In particular, it widens the already-significant semantic gap between applications, which generate data, and storage systems, which manage data. This widening semantic gap makes end-to-end differentiated services extremely difficult. In this paper, we present a client-aware cloud storage framework, which allows semantic information to flow from clients, across multiple intermediate layers, to the cloud storage system. In turn, the storage system can differentiate various data classes and enforce predefined policies. We showcase the effectiveness of enabling such client awareness by using Intel's Differentiated Storage Services (DSS) to enhance persistent disk caching and to control I/O traffic to different storage devices. We find that we can significantly outperform LRU-style caching, improving upload bandwidth by 5x and download bandwidth by 1.6x. Further, we can achieve 85% of the performance of a full-SSD solution at only a fraction (14%) of the cost.**

***Index Terms*—Storage, Operating systems, Cloud computing, Cloud storage**

I. Introduction

Cloud storage is becoming increasingly popular among enterprise and consumer users. According to an IHS report, personal cloud storage subscriptions have reached 500 million in 2012 for major providers such as DropBox and iCloud [10]. The combined public and private cloud storage market is predicted to be $22.6 billion by 2015 worldwide [24].

Unlike conventional storage, such as local disk drives or NFS servers, public cloud storage users store, access, and manage files (objects) stored in the data centers of service providers and pay for the service based on a monthly rate, or actual usage. For its unique technical merits, such as high availability, resilience, and cost efficiency, cloud storage is quickly changing the way people store and manage data.

A. The Challenge of Widening Semantic Gap

Benefits aside, cloud storage introduces many new challenges (e.g., [16], [17], [20], [26]). In particular, it further widens the already-significant *semantic gap* between applications and storage systems, making it especially difficult to realize end-to-end differentiated services, often referred to as Class of Service (CoS).

Cloud storage is not a file system that users can directly interact through conventional `read`/`write` commands. Instead, an HTTP-based REST interface (e.g., GET/PUT) is used for transmitting data from client to server. During this process,

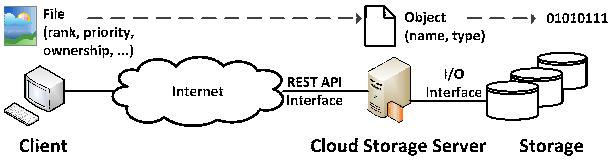


Fig. 1. Data flow in the existing cloud storage framework

valuable semantic information, which is only available on the client, is lost. Figure 1 illustrates such a process – (1) Before uploading a file, the user has rich semantic knowledge about the data, such as ownership, priority, data importance, interest-based ranking, etc. (2) When the data is transmitted to the cloud storage server, the server sees an object with limited information (e.g., object name), and most semantic information is stripped off. (3) When data is written to the storage system, the storage system only sees a stream of bytes. As so, realizing end-to-end (i.e., client-to-server-storage) differentiated services becomes extremely difficult. For example, how can cloud storage tell the difference between 1-star songs and 5-star songs and store them differently?

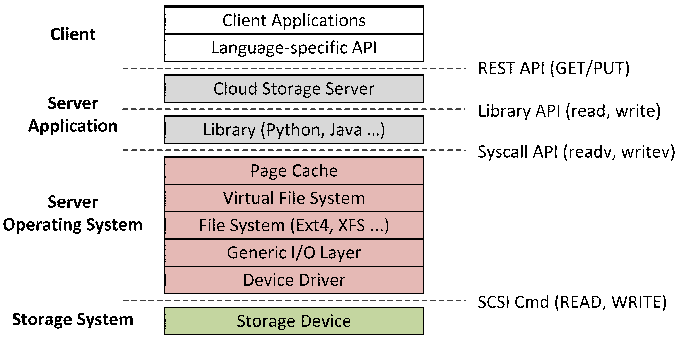


Fig. 2. Logical/physical interfaces in cloud storage

To enable client awareness in cloud storage, semantic information must flow along with the data across multiple logical and physical layers (See Figure 2). Unfortunately, with existing cloud storage infrastructures, we lack the appropriate mechanisms to enable such semantic information flow, from collecting the semantic information on the client side to utilizing such information on the server side. We need to systematically reconsider the entire cloud storage stack and build a semantic information channel from the client to the storage, which involves multiple intermediate layers that need to be tailored to make cloud storage truly *client-aware*.

978-1-4799-5671-5/14/$31.00 © 2014 IEEE Published by the IEEE Computer Society

B. Potential Use Cases

Client-aware cloud storage can be used in many scenarios.

- **Semantic importance-based caching** - Emerging storage technologies, such as flash SSDs, can be used for persistent storage caching in cloud storage systems. Lacking semantic hints from clients, traditional caching schemes, such as LRU, cannot reflect the semantic importance of data from users' perspective. With client-awareness, we can selectively choose semantically critical data for caching.
- **Selective on-line compression** - Cloud storage can apply on-line compression to reduce capacity and cost, however, the efficiency of compression is highly dependent on object types. For example, multimedia objects, such as video and audio files, are already heavily compressed. Compressing such objects cannot provide any further benefits, but only unnecessary overhead. With our client-aware cloud storage framework, these difficult-to-compress objects can be labeled in advance to bypass the compression.
- **Differentiated encryption** - Privacy and security are always important in cloud services. Encryption provides protection but incurs a potentially high performance penalty, which degrades user experience. Giving users the capability to selectively encrypt certain data can help achieve a good balance between performance, cost, and security. For example, videos can be uploaded without being encrypted, while emails can be encrypted.

To enable each of these possibilities, the cloud storage must have the capability of differentiating requests and applying different service policies to user generated data.

C. Client-Aware Cloud Storage

We present a design for building client-aware cloud storage. In principle, our design is based on *data classification*. Specifically, the client classifies data and requests different classes of data to be handled with different storage policies (e.g., low latency versus high throughput). The classification information, as well as the data, is transmitted over the network to the cloud storage server through an augmented REST interface (PUT/GET). The cloud storage is responsible for extracting the classifiers from the HTTP requests and translating them into storage system policies. The storage system, in turn, enforces the associated policies. In other words, this framework enables clients to label data and pass the label along with data to the storage, where data is managed based on labels.

We have prototyped a complete stack of the proposed client-aware cloud storage based on OpenStack Object Storage (Swift) [1]. This prototype system includes (1) a cloud storage client emulator that simulates hundreds of clients that are concurrently performing cloud storage operations based on specified object type/size distributions, (2) a classification-enabled cloud storage server, which handles labeled requests and interacts with a policy-based storage system, and (3) a tiered storage system based on DSS [23], which enforces certain service policies. We demonstrate the strength of this framework with two practical applications, persistent disk caching and fine-grained I/O traffic control, to enhance cloud storage performance and manageability. As mentioned, the proposed mechanism can also enable a variety of other optimization opportunities, such as differentiated security and reliability.

D. Our Contributions

In this paper, we make the following contributions: (1) We present a client-aware cloud storage framework designed to improve end-to-end class of service (CoS). (2) We propose a backward compatible and easy-to-standardize REST API interface for transmitting semantic hints to the server. (3) We present a complete prototype of the proposed framework. (4) We demonstrate the effectiveness of the proposed framework by using persistent disk caching and an I/O traffic control.

The paper is organized as follows. Section II presents background on cloud storage. Sections III and IV describe our design and implementation. Section V presents the experimental evaluation. Section VI presents the related work. Section VII discusses our future work.

II. Background

In this section, we introduce the storage model, the architecture, and the API of cloud storage. Our description is based on OpenStack Swift [1], which is similar to Amazon S3. More details can be found in the on-line documentation [1].

A. The Swift Cloud Storage Model

An *object* is the basic entity of user data in cloud storage. Conceptually, an object is akin to a file in file system. Each object can be associated with optional metadata in the form of a key/value pair. An object can be specified as a URL consisting of a service address, container, and object name (e.g., `http://localhost:8080/v1/AUTH_test/c1/foo`). The maximum object size is 5 GB, which is limited by the HTTP protocol. Objects larger than 5GB must be segmented into smaller chunks, and a manifest is used to locate the related segments of a big object. Objects are organized into logical groups, called *containers* (akin to *buckets* in Amazon S3). A container is analogous to a directory in a file system. Unlike directories, containers cannot be nested, but users can emulate a hierarchical naming structure by arbitrarily inserting "/" in object names.

B. Cloud Storage API

Almost all cloud storage services provide a simple REST (Representational State Transfer) web service interface to allow users to store and retrieve objects. Normally at least five standard HTTP primitives (verbs) are supported – `PUT` (uploading), `GET` (downloading), `POST` (updating object metadata), `HEAD` (retrieving object metadata), and `DELETE` (removing an object). For each operation, a *URL* is presented to specify the target object stored in the cloud storage. Besides the operation and the URL, a number of HTTP *headers* are used to carry extra information to the server. For example, in Swift, the `X-Auth-Token` header carries the authentication

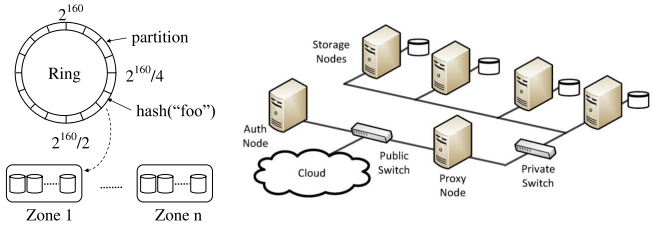
token for access control. We leverage HTTP headers to carry extra hints from the clients.

With the REST API interface, a request can be constructed easily with tools, like `curl` [2]. The below is an example of uploading the file 'foo' to cloud storage in the container 'c1' with an authentication token of 'abc'. Some cloud storage services also provide command-line, GUI (e.g., CyberDuck), or web interfaces. In essence, such tools simply translate actions to REST operations.

```
curl -X PUT -H "X-Auth-Token: abc" -T "foo" \
    http://localhost:8080/v1/AUTH_test/c1/foo
```

C. Swift Object Storage Architecture

Cloud storage is a large-scale distributed storage system carefully designed for availability, resilience, and cost efficiency. A *ring* describes the mapping from the name of storage entities (e.g., an object or a container) to their physical locations. Accounts, containers, and objects have separate rings. Storage entities in a ring are divided into *partitions*. Each partition is replicated (3 times by default) in the cluster, and the mapping (locations for a partition) is maintained by the ring. The ring also determines the fail-over devices. Partitions are guaranteed to be evenly distributed among all devices in the cluster. For reliability, storage devices are logically organized into *zones*, based on their physical location, network connectivity, machines/cabinets, etc. For example, a zone could be a disk drive, a server, or a cluster of servers connected to the same switch. Zones should be isolated from each other as much as possible. When replicating a partition, each replica is guaranteed to reside in a different zone.



(a) Ring Structure **(b) Server Architecture**

Fig. 3. An illustration of cloud storage system

A cloud storage cluster consists of many physical machines (*nodes*), each runs one or more *services* (Figure 3). The *proxy server* is a gateway that exposes APIs to clients and handles incoming requests. The *object server* performs PUT, GET, and DELETE operations on local storage devices. Each object is stored in the host file system as a file and associated metadata, if any, is stored in the file's extended attributes block (XATTR), which requires support from file systems, such as Ext4 or XFS. The *container server* maintains listings of objects in a container. It only tracks whether an object is in a container and disregards its physical location. The *account server* maintains listings of containers for an account, similar to container servers. Swift nodes also run several supporting services, such as the *replicator*, which replicates data in the cluster, the *updater*, which handles delayed updates to storage entities, and the *auditor*, which periodically scans the file system to check object integrity. We normally call a machine running the proxy service a *proxy node*, and a machine running the other services as *storage node*. Since the services and nodes are largely isolated from each other, the storage services can be scaled out.

III. Design

In order to make cloud storage client-aware, we need to consider the following three questions:

- How to describe and express the most valuable semantic information in a compact and general way to satisfy the CoS requirement for applications?
- How to transmit the semantic information across multiple logical and physical interfaces between client and storage, without introducing significant disruptive changes to the existing architecture?
- How to appropriately handle and leverage the client-supplied semantic hints for improving services?

In this section, we answer these questions by following the information flow from the client to the storage – (1) generating semantic information at the client, (2) transmitting the semantic information to the server, (3) handling the semantic information appropriately at the server, and (4) enforcing service policies in the storage system. Figure 4 illustrates this process.

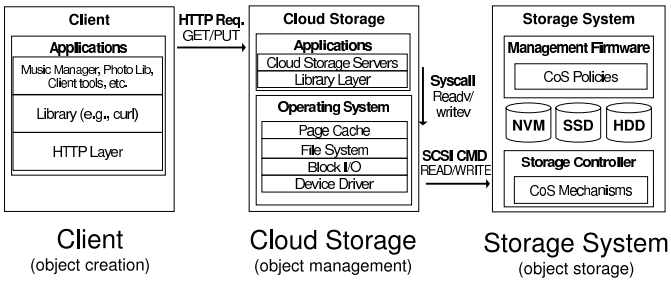


Fig. 4. Architecture of client-aware cloud storage

A. Collecting Semantic Information on the Client

Users possess rich semantic knowledge about data. However, the semantic information that matters the most differs across various applications. For example, ranking information is important for managing a music library (e.g., 5-star songs vs. 1-star songs), while distinguishing I-frames in a video file is important for video analytics. By being able to differentiate distinct classes of data, cloud storage can employ proper storage management policies for various purposes, such as selectively caching important objects in SSDs (e.g., 5-star songs) or encrypting certain objects (e.g., emails).

One principle of our design is to separate *data classification* and *policy enforcement*. Specifically, the client classifies data and the storage system enforces policies. Such a separation enables a dynamic mapping between data classes and storage policies, which can be flexibly defined and configured by administrators.

1) Data classification: A *classifier* is a numerical value differentiating one group of data from another. Classes are used only for the purpose of differentiating data, and the associated numerical values do not necessarily imply any relative priority for storage services. The size of a classifier is defined by the capability of storage system. For example, a 5-bit classifier is used in DSS [23].

In our design, data classification can be performed at three different granularities:

- **Object-based** – The client can associate a classifier with an entire object.
- **Range-based** – The client can associate a classifier with a range of 512-byte sectors in an object.
- **Block-based** – The client can associate a classifier with a given block (e.g., 4096 bytes) in an object.

Applications select the appropriate classification granularity based on their needs. For example, a music library manager can use object-based classification to differentiate 1-star songs and 5-star songs; a video analytics tool can use range-based classification to separate I-frames from the other frames in a video file; a virtual machine manager can use block-based classification to identify metadata and data blocks inside a virtual disk image file. If no class is provided, the object is regarded as a regular object with no classification information and the default management policies are applied. This makes classification an enhancement rather than a requirement, which provides backward compatibility.

2) Classifying data at the client: The client is responsible for creating and transmitting the class information to the server. In our vision, at least three classification mechanisms can be implemented on the client.

- **Manual labeling** – A user can manually assign a classifier to a file either through a command line interface or a GUI interface (e.g., mark a file with colors in a right-click menu). Users often upload data through a dedicated client, which also provides an opportunity for cloud service providers to integrate such a capability.
- **General-purpose classification** – Client systems can provide a default classification. For example, DropBox clients segment objects larger than 4MB into multiple 4MB chunks [14], which makes a cloud storage server lose track of the original file sizes, which is important for caching. A general-purpose classification scheme can directly extract the file size information from the client and send to the server.
- **In-app classification** – Applications with integrated cloud storage support can extract deep semantic information. For example, a music library manager (e.g., iTunes) can use the ranking information (e.g., 1-star songs vs. 5-star songs) to classify music files based on the semantic importance.

B. Transmitting Classification to the Server

We provide both in-band and out-of-band modes to transmit classification information from client to server. In both cases, classifiers can be transmitted either in headers, or in objects.

1) In-band mode: In the in-band mode, the classifiers are transmitted along with the object content to the cloud storage server in one single HTTP request. There are two ways to perform in-band classification.

• **Transmitting classifier in HTTP headers** – When a client sends an HTTP request to the cloud storage server, we create HTTP headers to transmit classifiers for object- and range-based classification. For object-based classification, we create a new header `X-DSS-Object-Class` to contain a classifier, which represents the class of all blocks in the object, in an HTTP request. The following is an example of classifying the object "foo" as class 25.

```
curl -X PUT -H "X-Auth-Token: abc" -T "foo" \
    -H "X-DSS-Object-Class: 25" \
    http://localhost:8080/v1/AUTH_test/c1/foo
```

For range-based classification, we create a new header `X-DSS-Range-Class` to contain a string, which specifies the classes for a sequence of data ranges of the object. Each range is represented as `<offset>-<length>-<class>`, where `offset` and `length` are in units of sectors (512 bytes) and `class` is the classifier for the corresponding data range. The following is an example of classifying two ranges of blocks to class 25 and 26.

```
curl -X PUT -H "X-Auth-Token: abc" -T "foo" \
  -H "X-DSS-Range-Class: 0-64-25,1024-32-26"\
  http://localhost:8080/v1/AUTH_test/c1/foo
```

A constraint of using headers to transmit classifiers is that the header size is limited. For example, the Apache server can accept a header of at most 8190 bytes by default. Thus, it is unsuitable for transmitting a large number of classifiers, especially for block-based classification in a large object.

• **Transmitting classifiers in objects** – To address the header-size limitation, classifiers can be transmitted by crafting a specially formatted object file, containing both classifiers and data. We create a header `X-DSS-Object-File` to use with PUT operations. If the header contains a string 'True', the uploaded file is an object with classifiers embedded. Upon receiving such a request, the cloud storage server knows to extract the embedded classification information.

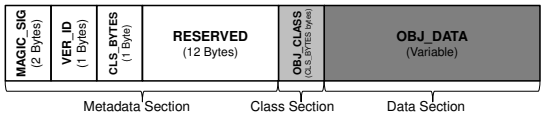


Fig. 5. Object-based Classification Format

An object with embedded classification information contains at least two sections, a *Metadata* and a *Class* section, plus an optional *Data* section. Specifically, (1) the metadata section describes the format of the class section, (2) the class section contains one or multiple fixed-size entries, each of which associates a class with object data, and (3) the data section contains the object data, if the in-band mode is used.

All three classification methods are supported by in-object transmission of classification information. Depending on the classification method used, metadata and class sections are defined differently. Figure 5 shows an example of object-based

classification. Complete descriptions of all three types of data classification can be found in the Appendix (A.1).

2) Out-of-band mode: In the out-of-band mode, the classifiers can be transmitted to the cloud storage after an object is uploaded. This gives users an opportunity to re-classify an object without re-uploading the data. There are two ways to perform out-of-band classification:

• **Transmitting classifiers in headers** – The header can be used with `GET` operations for downloading, or with `POST` operations to update the object's metadata. When the server receives the request, it re-classifies the object. The following example reclassifies object 'foo' to class 26.

```
curl -X POST -H "X-Auth-Token: abc" \
 -H "X-DSS-Object-Class: 26" \
 http://localhost:8080/v1/AUTH_test/c1/foo
```

• **Transmitting classifiers in objects** – We create a header `X-DSS-Class-File` with PUT operations to upload a file only with the metadata and class sections. When using this request, the target URL specifies the object for re-classification, and the header contains a string 'True.' Upon receiving such a request, the cloud storage server knows that the uploaded file contains classification information only. The following is an example of reclassifying object 'foo.'

```
curl -X PUT -H "X-Auth-Token: abc" \
 -T "foo.cls" \
 -H "X-DSS-Class-File: True" \
 http://localhost:8080/v1/AUTH_test/c1/foo
```

C. Handling Requests with Classifiers on the Server

Upon receiving an HTTP request with classifiers, the cloud storage server needs to (1) extract the classification information, and (2) access the local storage system (as per the DSS protocol).

1) Handling requests with classifiers: When a request arrives, the cloud storage server checks to see if it is a request with classification information. If the request carries classifiers in headers (using `X-DSS-Object-Class` or `X-DSS-Range-Class`), the embedded values are extracted. Then the server begins to receive the object data from the socket buffer and writes data into storage using the DSS protocol [23]. To improve storage performance, the object data is written into storage in chunks, each of which is a sequence of contiguous blocks with the same class.

If the incoming request carries classification information in the object (i.e., the `X-DSS-Object-File` header is 'True'), the server first reads the metadata section from the socket buffer to determine the classification format (object-, range-, or block-based), then reads the class section and forms the classifiers, and finally the object data is read out of the socket buffer and written into the storage system with I/Os and associated classifiers.

2) DSS protocol overview: Running at the application level, the cloud storage server needs to deliver the classification information across the application/OS interface. The server code normally interacts with storage using language-specific APIs, which interface to I/O syscalls. The standard I/O syscalls, such as `read()` and `write()`, only pass limited information, such as the file descriptor, length, memory pointer, etc. We use the POSIX scatter/gather I/O interface to transmit extra classification information to local storage [23].

```
unsigned class = 23; /* a class ID */
int fd = open("foo", O_RDWR|O_CLASSIFY);

iov[0].iov_base = &class; /* class ID */
iov[0].iov_len  = 1; /* 1 byte */
iov[1].iov_base = "Hello, world!"; /* data */
iov[1].iov_len  = strlen("Hello, world!");
rc = writev(fd, iov, 2);
close(fd);
```

The POSIX standard provides a scatter/gather I/O interface, namely `readv()` and `writev()`, to perform vectored I/Os to input/output a data stream from/to multiple memory locations in one syscall. We pack the classifier with the data into a multi-element vector and transmit it through the `readv/writev` interface to the OS kernel. As shown in the above example code [23], the file is first opened with a flag `O_CLASSIFY`. When preparing the array of memory buffers, one additional 1-byte scatter/gather element containing a classifier for the I/O is added as the first element. When the OS sees a scatter/gather I/O to a file with the `O_CLASSIFY` flag set, it assumes the first element of the received scatter/gather list points to a classifier (1 byte) and the remaining elements point to data buffers. Upon receiving such an I/O, the OS extracts the classifier and associates it with a kernel-level I/O request, passes it across the VFS layer, the generic I/O layer, and eventually to the device driver. When the request reaches the SCSI device driver, the classifier associated with the I/O is copied into a 5-bit, vendor-specific `Group Number` field in byte 6 of the SCSI CDB. At this point, the I/O with its classifier is given to the storage system.

D. Enforcing Policies in the Storage System

When an I/O with a classifier is received, the storage system enforces the associated policy, which is assigned to predefined classes when the storage system is initialized. A variety of storage system policies can be developed, and here we demonstrate with disk caching and traffic control.

1) Classification-based persistent disk caching: In the tiered storage system, flash SSDs are used as a cache for hard drives. With semantic hints, the limited SSD space can be efficiently used for caching the most "important" data, which improves system throughput, reduces latency, and also improves cost efficiency.

Here we briefly introduce our caching scheme. More details on the caching mechanisms are available in our prior work [23]. We first segment the SSD cache into 4KB entries. Initially all free entries are linked in a *free list*. For each I/O, cache entries are allocated from the free list and added to a class-specific *dirty list*. A hash table tracks the mapping of logical block number (LBN) to the allocated cache entries. A *syncer daemon* tracks the number of free cache entries. If it reaches a low watermark, the syncer initiates a cleaning

process by scanning the dirty list. Whenever an entry is accessed, it is moved to the end, so the syncer always cleans the least recently used (LRU) entry. The cleaned entries are added back to the free list, which is also an LRU list.

As any caching solution, the most important decision is cache admission and eviction. With semantic hints, the tiered storage system can make caching decisions based on the priorities of data classes. Two algorithms are used:

- **Selective allocation** – When the storage cache is under pressure, (i.e., when the syncer daemon is actively cleaning the dirty entries), incoming I/Os that carry a classifier below a specified priority level would bypass the cache, because caching such "less important" data would only increases cache pressure.
- **Selective eviction** – Knowing data classes and their relative priorities, eviction can start with the lowest priority class. When the low watermark is reached, the syncer daemon scans the lowest priority list first, and then the second lowest one, and so on. For each list, the LRU entry is evicted first. This process repeats until reaching a high watermark.

2) Fine-grained I/O traffic control: A client-aware cloud storage framework also enables many other potential optimizations, such as reliability, security, and management. Fine-grained I/O traffic control is one – Cloud storage vendors often desire to have the capability of controlling I/O traffic, e.g., directing emails and videos to different storage pools. Current solutions are very coarse-grained. and inflexible. Our framework provides a fine-grained classification capability to precisely control the location of each uploaded object by labeling objects with different classes. More details will be discussed in Section V.

IV. Implementation

We have implemented a complete stack of the proposed client-aware cloud storage system, from client, server, to storage. Our prototype system includes five major components.

(1) **Cloud Storage Client** – As existing cloud storage benchmarks (e.g., COSbench [4]) do not generate data classification information, we developed a cloud storage client emulator to generate synthesized cloud storage workloads based on specified distributions. With an actual distribution provided by a cloud storage service provider, we can faithfully generate realistic cloud storage traffic. This tool is implemented in Python and consists of about 2,300 lines of code. We use the `pycurl` library for the HTTP communication. Users can specify object type distributions (e.g., 60% for videos), and the object size distribution for each type (e.g., 80% less than 128KB), and assign data classes. When initialized, a pool of files with different types and sizes is created. Then, a pool of connections is created to emulate a specified number of client connections to the cloud storage. This emulator performs experiments in several phases. Each phase of operations follows the user-configured distribution. Details about the workloads will be introduced in the next section.

(2) **Modified the Object Storage Server** – Our client-aware cloud storage server is based on OpenStack Swift 1.4.6 [1]. We added about 600 lines of code in the object server controller, which is a fairly small patch. The main work is to add support to handle requests with classifiers. When we receive a request with classification-related headers (i.e., `X-DSS-<foo>`), the request and the related data are as described in the prior section. For requests with classifiers, we use Python-specific API functions that interface to `writev()` and `readv()` system calls to communicate with the OS kernel. In our current prototype, we have implemented the object-based and range-based classification for both in-band and out-of-band modes. The in-object classification is partially supported in our current prototype.

(3) **Python APIs for Scatter/Gather Syscalls** – OpenStack Swift reads and writes objects through the standard Python APIs for I/Os. Swift 1.4.6 relies on Python 2.x, which does not support the scatter/gather I/O. We wrote a standalone Python library module, called *dssio*, which provides two API functions `dread()` and `dwrite()`, which converts to `readv()` and `writev()` syscalls, respectively. The module is written in C and consists of about 110 lines of code. The library enables the classification information to flow from the cloud storage server to the OS.

(4) **Modified the OS Kernel for Passing I/O Classification** – We modified Linux kernel 3.2.1 to add classification support in the Ext4 file system to allow passing classifiers received from applications via `readv()` and `writev()` to the storage system. We added a classification field (`b_class`) in the `buffer_head` data structure to carry the file system related classifier (e.g., inode). When an I/O request reads or writes the buffers, the classifier is copied to a new classification field (`bi_class`) in a block I/O request (`bio`). When the `bio` reaches the SCSI device driver, the classifier is copied into the 5-bit *Group Number* field in byte 6 of the SCSI CDB. An additional 3 reserved bits can be used, which can further extend 32 classifiers to 256 classifiers.

(5) **Hybrid Storage System** – Our hybrid storage system is implemented as a standalone RAID module in the Linux kernel [23]. We implemented a new RAID level, RAID-9, for users to create and manage a hybrid storage with a heterogeneous set of storage devices (e.g., SSDs and HDDs). Unlike conventional RAID levels, our RAID-9 module dynamically decides the logical-to-physical block mapping across multiple devices based on the data classification information. Also, unlike some RAID levels, such as RAID-5 and RAID-6, we do not need to perform parity calculation. We also modified Linux `mdadm` utility to load the module and create a hybrid storage device (`/dev/md`) with a specified caching device (SSD) and a backend storage device (HDD). The classification scheme and the priority policy are specified during the module loading time. Since the caching device and storage device can both be an another RAID volume, multi-tier (recursive) caching is possible. After the device is initiated, we can create partitions, make file systems, and use it like any block device.

V. Evaluation

We evaluate our prototype with our emulated cloud storage workloads based on real object distributions, including user files, pictures, videos, music, and virtual disk images. The generated workload mimics the cloud storage traffic of a well-known public cloud storage service provider [3].

A. Experimental Setup

All experiments are run on a six-node Linux-based cluster. Two nodes are equipped with two 8-core Xeon Sandy Bridge (E5-2690) 2.9GHz processors (16 cores) and 128GB memory. One node is used as the client, which is responsible for emulating concurrent client connections. The other is used as the cloud storage proxy node, which runs the proxy server to accept incoming requests from the client node. The other four nodes are standard storage servers with two 6-core Xeon Westmere (X5680) 3.3 GHz processors (12 cores) and 24GB memory. The four nodes work as storage nodes providing object, container, account and other supporting services. According to the recommended setup [1], we connect the client and proxy nodes to a 10GbE Switch through two 10GbE links, and the four storage nodes are connected to the switch through 1GbE links.

All nodes are installed with Fedora Core 14 with a DSS-patched Linux 3.2.1 kernel and the Ext4 file system. Each node is equipped with one Intel 710 SSD and two Seagate Constellation ST1000NM0011 1TB hard drives, one of which is used as the system disk and the other is used for experiments. In order to complete the experiments in a reasonable time frame, we keep the working-set around 100GB. We use an SSD and an HDD to organize a hybrid device for each storage node. The actual SSD cache size is configured to a specified percentage (5% and 10%) of the working-set size to reflect the true performance in a real-world setup [8].

Swift cloud storage servers are mostly configured with the recommended default settings. In particular, we set each object to be replicated 3 times. The connection timeout is set to 0.5 second. All services are enabled. We use the built-in authentication method for the proxy server. For each experiment, we create 100 containers and objects are uploaded to randomly selected containers. We configure 32 workers for the proxy server and 8 workers for each object, container, and account server. Each device is configured as an individual zone to evenly distribute the load.

B. Workloads

Our workloads emulate five object categories. For each category, the object size distribution is derived from real files. We use their combination to mimic a real-life public cloud storage service based on their input [3]. Figure 6 shows the Cumulative Distribution Function (CDF). The figure only shows file sizes to 16MB; file sizes larger than 16MB are collapsed. More details are shown in Table I.

- **Files** emulates regular user files (e.g., documents) based on the distribution of files generated by the SPECsfs2008 benchmark, which is also used in prior work [23].

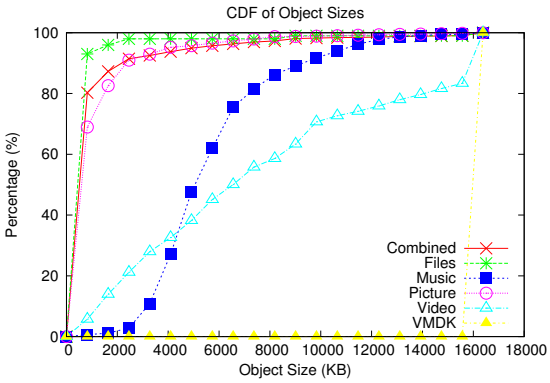


Fig. 6. Distributions of object sizes.

- **Picture** emulates a picture file distribution based on a photo library of 10,711 pictures from flickr.com. The pictures are retrieved with top-ranked search keywords, such as 'wedding,' 'arts,' 'animals,' etc.
- **Music** emulates a music file distribution based on a music library of 2,346 audio files, which consists of different genres, e.g., pop, jazz, and rock, etc.
- **Video** emulates a video file distribution based on a large video library, which contains 319,073 video files crawled from YouTube.com [12].
- **VMDK** emulates a file distribution of large virtual machine disk images in cloud storage. Since the HTTP protocol has an object size limit of 5GB, files larger than that split into smaller ones. So we randomly select the object size between 1GB and 5GB.

File Size	Files	Picture	Music	Video	VMDK
≤64KB	79%	17.1%	0%	0.3%	0%
≤512KB	14%	43.6%	0%	2.8%	0%
≤1MB	3%	14.5%	0.8%	4.9%	0%
≤5MB	2%	20%	52.3%	32.4%	0%
≤10MB	1%	3.5%	39.6%	31.7%	0%
≤50MB	0%	1.3%	7.3%	27.3%	0%
≤100MB	0%	0%	0%	0.4%	0%
>100MB	0%	0%	0%	0.2%	100%
Percentage	60%	35%	4%	0.9%	0.1%

TABLE I
Object Type and Size Distributions

The synthesized workloads perform cloud storage operations in five phases. We first create 100 containers, then perform 26,000 PUT requests to upload 100GB data to the cloud storage, which is replicated 3 times in the cluster. Then we perform 100,000 GET (downloading) requests, and finally DELETE all objects.

C. Case study: disk caching

In cloud storage, SSDs can be used as a cache to speed up I/O accesses. User-specified classification information can be used to guide cloud storage to cache the most important data in the SSD, which significantly improves caching performance.

1) Classification and Storage System Policies: As an example policy, we use object types to classify the objects. We assign the five object types, *Files*, *Pictures*, *Music*, *Video*, and

Description	Class ID	Priority
FS Metadata	1-10	0
Files $\leq$ 256KB	11-14	1
USER0	24	2
...	...	...
USER7	31	9
Files $>$ 256KB	15-22	10
Unclassified	0	11

TABLE II
REFERENCE CLASSES AND CACHING PRIORITY

VMDK with different medium-priority classes (class `USER1-USER5` in DSS [23]), whose caching priorities are high to low in that order. Recognizing that large objects can easily pollute the SSD cache, we assign large objects with the lowest priority (class `USER7` in DSS). For selective allocation, we fence off the lowest priority data when there is cache pressure and let them directly bypass the cache and be self-evicted. As so, we can evict large objects to the backend storage (HDDs) to avoid cache thrash. For selective eviction, we give file system metadata and small files the highest priority, since Swift involves many metadata and small file operations (e.g., SQLite DB files updates). The objects with user-defined classifiers (`USER0-USER7`) are given the second highest priority. The lowest priority is given to large files and unclassified data. Table II gives more details.

2) Performance of Semantic Hint-based Caching: We compare the cloud storage performance on storage with hard drives only (*HDD*), SSDs only (*SSD*), an LRU-based cache (*LRU*), and an enhanced LRU cache that uses semantic hints from the clients (*DSS*). We show the caching effects by setting the cache space proportional (5% and 10%) to the working-set size, which are considered cost-effective in practice [8].

Figure 7 shows the bandwidths, latencies, and failure rates. We can see in the figure that, as we increase the cache space from 5% to 10%, the uploading performance (PUT) is improved for DSS, from 53MB/sec to 76MB/sec. With a 10% cache size, DSS can achieve 87% of the bandwidth of using an SSD-only solution (86MB/sec) and outperform LRU by 5 times. In contrast, LRU remains at 14MB/sec to 15MB/sec, which is even 3.6 times lower than HDD. This is because, knowing data classes, DSS can selectively allocate SSD cache space when the cache is under pressure. In contrast, LRU cannot distinguish and blindly caches everything, which causes data to first flush into cache and soon be evicted out to the hard drives. This doubles the I/O operations and leads to cache thrash.

The downloading (GET) bandwidth difference is also significant. DSS can achieve a bandwidth of 295MB/sec, which is 85% of the performance of the full-SSD solution (347MB/sec) and 1.6 times higher than LRU. LRU, in contrast, shows degraded performance (88MB/sec) with a small 5% cache size, which is 2.1 times lower than HDD. With a 10% cache size, the bandwidth of LRU (178MB/sec) becomes close to HDD.

For latencies, as shown in Figure 7(e), the average downloading latency (i.e., until the first byte is received by the client) for DSS with 10% cache is 89 ms, which is 5.5 times less than LRU (497 ms) and 3 times lower latencies than SSD (275 ms). This is because with DSS, incoming requests can be served from two devices (SSD and HDD), while the SSD-only solution cannot benefit from the device-level parallelism. Also, since the SSD holds mostly small objects, the small requests wait less behind large requests.

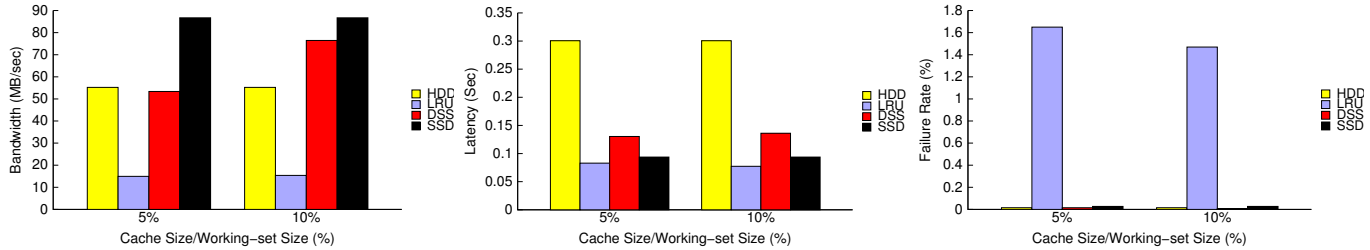
In Figure 7(b), we can find that LRU appears to show a lower latency than DSS for uploading. Figure 7(c) explains this. Due to the HTTP connection timeout, the failure rate of LRU is much higher than DSS, and the failed connections complete earlier than the successful connections by only sending back a failure response (e.g., HTTP 503 code). This makes the average latency of LRU appear lower.

In order to understand why DSS performs better than LRU, we show the content they choose to cache. As shown in Figure 8, LRU uses most space to cache VMDK and other large objects (more than 10MB), and the portion of cache space occupied by these large objects is also roughly constant across different cache sizes (from 5% to 10%). In contrast, DSS caches data based on the user-specified importance. When the cache size is limited, most space is used to cache metadata and the small objects from the "Files" distribution. As cache space increases, more space is used to cache the other classes, such as pictures and music, while VMDK and other large objects remain a small percentage. Since the user-defined classification offloads large objects and VMDK files to the HDD, DSS uses the SSD space more effectively.

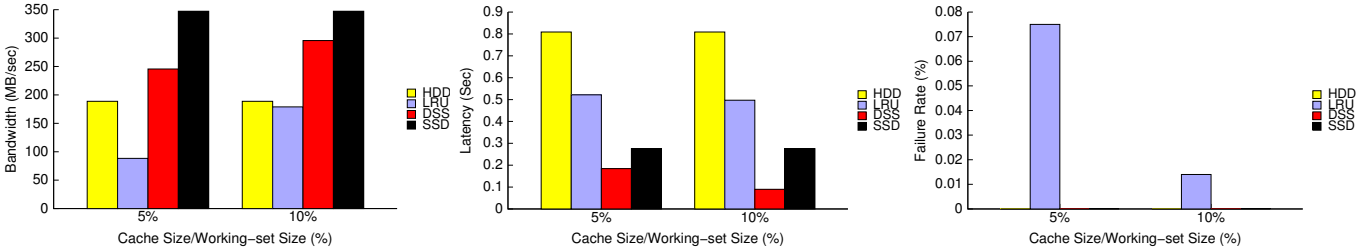
3) Cost Efficiency: We also use "U.S. $/GB/IOPS" as the metric to calculate the cost efficiency. According to Amazon.com, the cost of an Intel 710 series SSD is about $3.95 per GB, which is 26 times more expensive than a Seagate Constellation ST1000NM0011 hard disk ($0.15 per GB). Figure 9 shows the cost efficiency for the four configurations with a 10% cache size. For presentation, we normalize the numbers by using HDD as the baseline. We can see that although the SSD-only solution provides high performance, its cost is 9.9 and 14.3 times higher than HDD for uploading and downloading. DSS is much more cost effective. DSS can achieve a performance comparable to SSD for downloading, but its cost is only 14% of that. LRU is 1.4 times more costly than DSS, due to its less efficient use of SSD space.

D. Case Study: I/O traffic control

Cloud storage service providers often desire to have a flexible control capability to direct I/O traffic to different storage pools for a variety of reasons, such as performance and reliability. In client-aware cloud storage, I/O traffic carries classifiers, which helps achieve this goal easily at a fine level of granularity. In this section, we demonstrate such a case by redirecting I/O traffic to either an SSD or an HDD based on object type. In this experiment, we reuse workloads in the prior section. We set up a non-caching hybrid storage system. As so, when uploading an object, the I/O traffic can be directed either to the SSD or the HDD. This allows us to only speed up accesses for selected data classes.



(a) Uploading B/W (b) Uploading Latency (c) Uploading Failure Rate



(d) Downloading B/W (e) Downloading Latency (f) Downloading Failure Rate

Fig. 7. Performance of caching with semantic hints

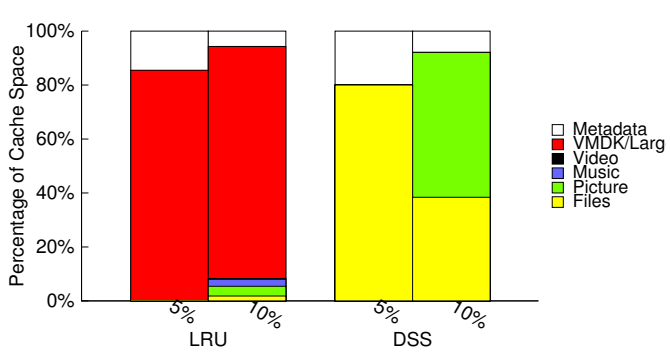


Fig. 8. Cache content breakdown by object types

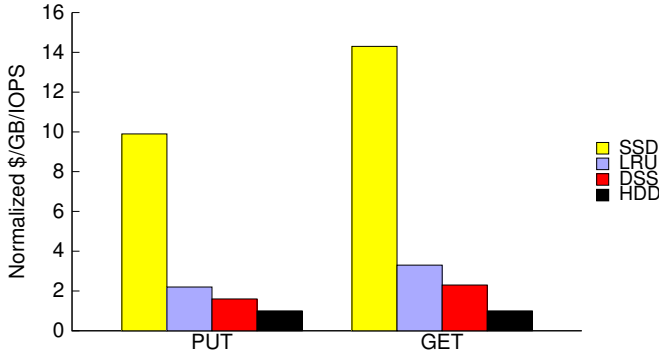


Fig. 9. Normalized cost efficiency (10% cache)

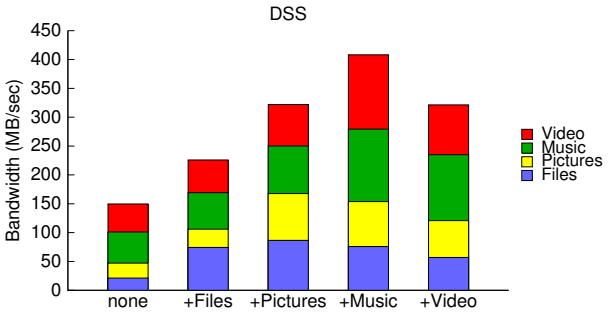
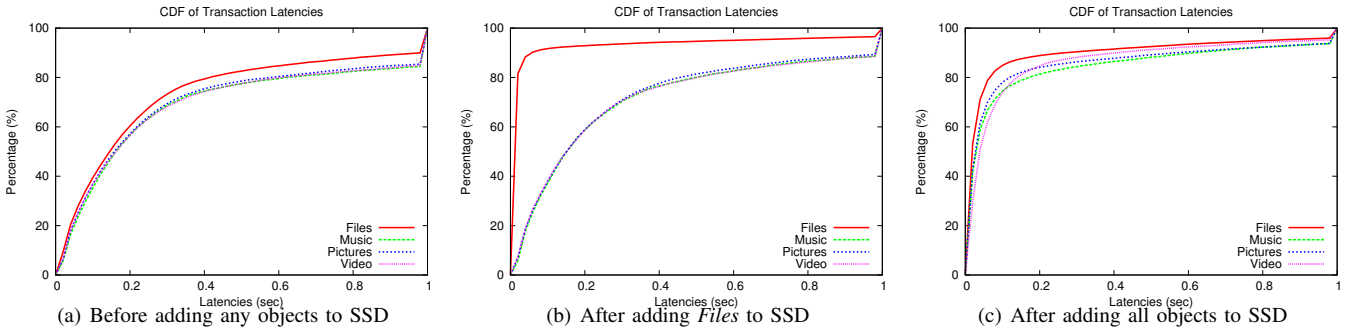


Fig. 10. Aggregate bandwidths with I/O traffic control

For the experiments, we set four concurrent streams, each of which has 25 clients. Each stream uploads one class of objects, namely files, pictures, music, and video. We calculate the bandwidth for each stream individually. For experiments, we performed five test runs with different classification schemes. We first classify all objects as unclassified and send all four streams to the HDD (*none*). In the second run, we send Files and direct its traffic to the SSD (*+Files*). In the third run, we send both Files and Pictures to the SSD (*+Pictures*). In the fourth run, we send Files, Pictures, Music to SSD. In the fifth run, we send all the objects to SSD.

Figure 10 shows the aggregate bandwidths of the five test runs and the bandwidth breakdown of each stream. As we include one additional stream to the SSD, the added stream receives an increase of bandwidth due to the faster device speed. The aggregate bandwidth keeps increasing until it saturates the SSD and network bandwidth. When all four streams are directed to the SSD, we see a decrease in aggregate bandwidth. This is due to two reasons. Firstly, the SSD is over-congested and used to serve all the objects, disregarding the fact that videos can be streamed from the HDD with a good performance. Secondly, the HDD is left unused and the potential I/O parallelism of the two devices is lost.

We also show the Cumulative Distribution Function (CDF) of the transaction latencies for each stream in Figure 11. We can clearly see that after redirecting Files to the SSD (Figure 11(b)), its average access latency can be immediately reduced and its curve differs from other curves. Over 90% of the requests to Files can be finished in less than 100ms. In contrast,



(a) Before adding any objects to SSD (b) After adding *Files* to SSD (c) After adding all objects to SSD

Fig. 11. The effect of traffic control by directing different object types to the SSD

only 40% of the requests can achieve the same latency before this optimization. After we direct all the I/Os to the SSD, their distributions become similar again (Figure 11(c)).

Finally, we would again like to point out that persistent disk caching and the I/O traffic control capability are just two of many possible applications of this client-aware cloud storage framework. Although our evaluation is mostly focused on performance, other optimization goals can be realized. For example, certain objects can be selectively made more reliable through a high-degree replication. Also, selected objects (such as personal emails) can be made more secure by using encryption.

VI. Related Work

Prior cloud storage research has worked on addressing a variety of issues, such as performance, reliability, availability, confidentiality, and lock-in concerns (e.g., [5], [6], [9], [16], [18], [20], [29]). A large body of research has focused on studying the performance of commercial cloud storage services, such as Dropbox and Amazon S3, by passively intercepting and statistically analyzing network traffic on Internet (e.g., [7], [14], [15], [19], [22], [27]). Some prior research attempts to unify the strength of cloud storage and file systems. For example, Vrable et al. have presented a cloud-backed network file system for the enterprise use, called BlueSky [25], to store data in cloud storage and access storage through an on-site proxy, which caches data and provides an NFS and CIFS interface to the clients. Dong et al. presented a similar network file system design, called RFS [13], for mobile devices. Our work is largely orthogonal to these prior efforts. If semantic hints can be provided by the proxy servers when communicating with the cloud storage, potential optimization can be easily achieved with our framework.

Our work is also related to hybrid storage technologies. Caching is important in large-scale storage systems [21]. Mesnier et al. have presented a storage CoS framework, called Differentiated Storage Services, to associate semantic hints with each I/O for optimizing performance in a local storage system [23]. Chen et al. presented a hybrid storage system, called Hystor, by integrating SSDs and HDDs and leveraging SSDs as a cache to hold the small and frequently accessed data [11]. Karma uses hints on database block access patterns to improve multi-level caching [28]. This work aims to use semantic hints in cloud storage scenario, and we find that it can significantly improve performance.

VII. Discussions and Future Work

Exploiting client awareness in cloud storage requires collaboration among clients, servers, and storage. This paper presents a first step in this direction – building a system framework to enable such end-to-end semantic information flow. We demonstrate that this is feasible in practice and can be achieved with relatively small changes to the existing systems. As future research, we will further investigate how to leverage such information to optimize storage systems. In this paper, we have shown two such cases: a priority-based persistent disk caching, and a fine-grained I/O traffic control. Both cases focus on storage management, in which DSS plays an important role for hybrid storage management. In fact, leveraging semantic hints from clients can realize numerous optimization opportunities at various levels, even with non-DSS storage. For example, proxy servers can differentiate uploading traffics to enable a coarse-grained data placement (e.g., different sets of servers). Another potential future work is on the object structure definition and protocol standardization. In this paper, we propose a set of predefined object formats to embed the classification in objects. These definitions, by no means, will be the only possible ones. Other definitions could be developed in the future. However, as any protocol, we need to seek a common agreement between clients and cloud storage service providers. This demands an industry-wide effort to eventually reach a standard to define the ways we describe, transmit, and handle the data and the associated semantic hints in a proper and consistent way. This will be a long-term but important effort in the future.

VIII. Conclusion

Cloud storage is deeply changing the way people store, access, and manage data. In this paper, we present a client-aware cloud storage framework to close the widening semantic gap between client and storage and realize end-to-end differentiated services in cloud storage. With only minor changes to the existing system, we can make semantic information travel together with data from the client end, where data is generated,

to the storage end, where data is stored. We have showcased the effectiveness of enabling client awareness in cloud storage by using semantic hints for persistent disk caching and I/O traffic control. Our experimental results show that we can effectively leverage semantic hints from users to enhance LRU caching, and the cost efficiency ($/GB/IOPS) is 7 times higher than the full-SSD solution for 85% of the performance.

ACKNOWLEDGMENT

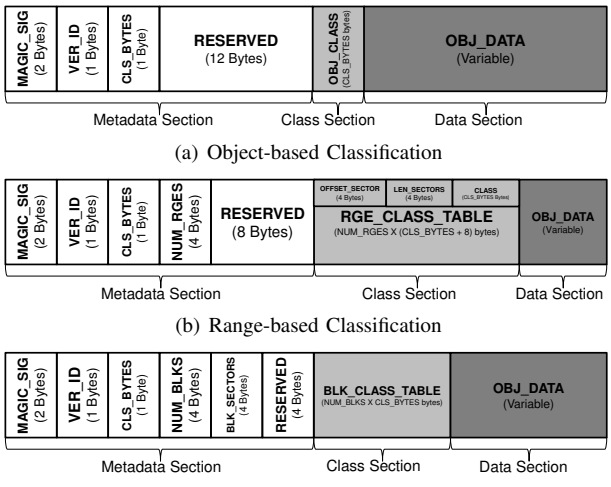
The authors would like to thank reviewers for their constructive comments and the advice. We also thank Paul Brett, Ren Wang, and Pat Stolt for discussions and support during this work. We also thank our industrial partners for providing data and feedback for this research.

REFERENCES

[1] http://www.openstack.org/.
[2] http://curl.haxx.se/.
[3] Anonymized for commercial reasons.
[4] Intel Cloud Object Storage Benchmark. https://github.com/intel-cloud/cosbench.
[5] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC 2010)*, Indianapolis, IN, June 10-11 2010.
[6] S. Bazarbayev, M. Hiltunen, K. Joshi, R. Schlichting, and W. Sanders. PSCloud: A Durable Context-Aware Personal Storage Cloud. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep 2013)*, Farmington, PA, Nov. 3 2013.
[7] I. Bermudez, S. Traverso, M. Mellia, and M. Munafo. Exploring the Cloud from Passive Measurement: the Amazon AWS Case. In *Proceedings of The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, Turin, Italy, April 14-19 2013.
[8] C. Black, M. Mesnier, and T. Yoshii. Solid-State Drive Caching with Differentiated Storage Services. In *IT@Intel White Paper*. Intel Co., July 2012.
[9] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC 2010)*, Indianapolis, Indiana, June 10-11 2010.
[10] B. Butler. Personal Cloud Subscriptions Expected to Reach Half a Billion This Year. In *Network World*, September 7 2012.
[11] F. Chen, D. Koufaty, and X. Zhang. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*, Tucson, AZ, May 31 - June 4 2011.
[12] X. Cheng, C. Dale, and J. Liu. Statistics and Social Network of YouTube Videos. In *Proceedings of the 16th International Workshop on Quality of Service*, Enschede, Netherlands, June 2-4 2008.
[13] Y. Dong, J. Peng, D. Wang, H. Zhu, F. Wang, S. C. Chan, and M. P. Mesnier. RFS - A Network File System for Mobile Devices and the Cloud. In *SIGOPS Operating System Review*, volume 45, pages 101–111, February 2011.
[14] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking Personal Cloud Storage. In *Proceedings of the 2013 ACM conference on Internet measurement conference (IMC 2013)*, Barcelona, Spain, October 23-25 2013.
[15] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 ACM conference on Internet measurement conference (IMC 2012)*, New York, NY, November 14-16 2012.
[16] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, Oct 4-6 2010.
[17] S. L. Garfinkel. An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. In *Tech Report TR-08-07*, Harvard University, 2008.
[18] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *Proceedings of the 2nd ACM symposium on Cloud computing (SoCC 2011)*, Cascais, Portugal, October 27.28 2011.
[19] W. Hu, T. Yang, and J. N. Matthews. The Good, the Bad and the Ugly of Consumer Cloud Storage. In *ACM SIGOPS Operating Systems Review*, volume 44:3, July 2010.
[20] Y. Hu, H. C. H. Chen, P. P. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 2012)*, San Jose, CA, February 14-17 2012.
[21] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP 2013)*, Farmington, PA, November 2013.
[22] T. Mager, E. Biersack, and P. Michiardi. A Measurement Study of the Wuala On-line Storage Service. In *Proceedings of the IEEE 12th International Conference on Peer-to-Peer Computing (P2P 2012)*, Sophia Antipolis, France, Sept 3-5 2012.
[23] M. P. Mesnier, J. Akers, F. Chen, and T. Luo. Differentiated Storage Services. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP 2011)*, Cascais, Portugal, October 23-26 2011.
[24] I. P. Release. Demand from Public Cloud Service Providers and Private Cloud Adopters will Drive Strong Growth for Full Range of Storage Solutions, According to IDC. October 11 2011.
[25] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 2012)*, San Jose, CA, February 14-17 2012.
[26] E. Walker. Benchmarking Amazon EC2 for High-Performance Scientific Computing. In *;Login*, volume 33, October 2008.
[27] H. Wang, R. Shea, F. Wang, and J. Liu. On the Impact of Virtualization on Dropbox-like Cloud File Storage/Synchronization Services. In *Proceedings of International Workshop on Quality of Service (IWQoS 2012)*, Coimbra, Portugal, June 4-5 2012.
[28] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-All Replacement for a Multilevel cAche. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.
[29] R. Zhang, R. Routray, D. Eyers, D. Chambliss, P. Sarkar, D. Willcocks, and P. Pietzuch. IO Tetris: Deep Storage Consolidation for the Cloud via Fine-grained Workload Analysis. In *Proceedings of the 4th International IEEE Conference on Cloud Computing (IEEE CLOUD 2011)*, Washington D.C., July 2011.

APPENDIX

A.1 Data Formats for Embedded Data Classification



(a) Object-based Classification

(b) Range-based Classification

(c) Block-based Classification

Fig. 12. Format of objects with embedded classification

- **Object-based Classification** – The metadata section contains four components, a 2-byte `MAGIC_SIG`, which is a randomly selected magic signature indicating that the object contains self-describing classification information, a

1-byte VER_ID, which is a version ID specifying which classification method is used (0 for object-based, 1 for range-based, and 2 for block-based), a 1-byte CLS_BYTES, which specifies the size (in bytes) of the class value, and a 12-byte RESERVED, which is a reserved space for future extension. The class section contains only 1 classifier, whose size is CLS_BYTES bytes.

- **Range-based Classification** – The metadata section contains five components, a 2-byte MAGIC_SIG, a 1-byte VER_ID, a 1-byte CLS_BYTES, a 4-byte NUM_RGES, which specifies the number of entries in the class section, and an 8-byte RESERVED. The class section contains NUM_RGES entries, each of which contains three components, a 4-byte OFFSET_SECTOR, which is the start offset of the range in sectors, a 4-byte LEN_SECTORS, which is the length of the range in sectors, and a CLS_BYTES-byte classifier, which is the class value of the associated range.
- **Block-based Classification** – The metadata section contains six components, a 2-byte MAGIC_SIG, a 1-byte VER_ID, a 1-byte CLS_BYTES, a 4-byte NUM_BLKS, which specifies the number of entries in the class section, a 4-byte BLK_SECTORS, which specifies the size of each block in sectors, and a 4-byte RESERVED. The class section contains NUM_BLKS entries, each of which contains a CLS_BYTES-byte classifier, which is the class value of the corresponding block of the object.

A Protected Block Device for Persistent Memory

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

Michael P. Mesnier
Intel Labs
michael.mesnier@intel.com

Scott Hahn
Intel Labs
scott.hahn@intel.com

***Abstract*—Persistent Memory (PM) technologies, such as Phase Change Memory, STT-RAM, and memristors, are receiving increasingly high interest in academia and industry. PM provides many attractive features, such as DRAM-like speed and storage-like persistence. Yet, because it draws a blurry line between memory and storage, neither a memory- or storage-based model is a natural fit. Best integrating PM into existing systems has become challenging and is now a top priority for many. In this paper we share our initial approach to integrating PM into computer systems, with minimal impact to the core operating system. By adopting a hybrid storage model, all of our changes are confined to a block storage driver, called *PMBD*, which directly accesses PM attached to the memory bus and exposes a logical block I/O interface to users. We explore the design space by examining a variety of options to achieve performance, protection from stray writes, ordered persistence, and compatibility for legacy file systems and applications. All told, we find that by using a combination of existing OS mechanisms (per-core page table mappings, non-temporal store instructions, memory fences, and I/O barriers), we are able to achieve each of these goals with small performance overhead for both micro-benchmarks and real world applications (e.g., file server and database workloads). Our experience suggests that determining the right combination of existing platform and OS mechanisms is a non-trivial exercise. In this paper, we share both our failed and successful attempts. The final solution that we propose represents an evolution of our initial approach. We have also open-sourced our software prototype with all attempted design options to encourage further research in this area.**

***Index Terms*—Memory, Storage, Persistent memory, Operating system, Device driver**

I. Introduction

Over the years researchers in academia and industry have expended tremendous effort in the long battle with slowly improving storage performance. Countless innovations have been made in almost every corner of computer systems to mitigate the so-called "performance gap" between *volatile* memory and *persistent* storage. Although these innovations have greatly improved storage and I/O performance, the performance gap between delivering data and processing data never stops widening, because it essentially stems from the mechanical nature of hard drives and cannot be completely removed.

A. Persistent Memory

Semiconductor storage (e.g., NAND flash) has begun to change this situation. Beyond flash memory, recent technology breakthroughs may offer us even more relief in this multi-decade struggle. Persistent Memories (PM), such as Phase Change Memory (PCM) [22], STT-RAM [10], and memristors [14], promise many desirable features – access speeds comparable to DRAM, storage-like persistence, reasonably high data endurance and retention, low power consumption, and byte addressability. Table I shows details on the performance and endurance characteristics of PM.

These revolutionary technologies exhibit radically different characteristics, compared to any prior memory and storage technology, and thus raise many critical challenges to computer system designers. A fundamental question must be first answered – what is the appropriate usage model for PM?

	Read	Write	Endurance
PCM	50-85ns	150ns-1µs	10^8-10^{12}
Memristor	100ns	100ns	10^8
STT-RAM	6ns	13ns	10^{15}
DRAM	60ns	60ns	$> 10^{16}$
NAND flash	25 µs	200-500µs	$10^4 - 10^5$

TABLE I
Characteristics of PM technologies

As a new technology, PM draws a blurry line between volatile memory and persistent storage. It cannot be viewed and used simply as non-volatile memory or fast storage. A naive integration of PM into computer systems would introduce either protection or performance problems. In particular, using PM as a byte-addressable memory device opens it up to a wide variety of corruption, such as stray writes in the OS. Yet, using PM behind the protection of an I/O controller leaves considerable performance on the table.

Of course, one approach is to completely redesign the entire system, such as merging memory and storage systems together in the OS. For example, the application programming model can be changed to be aware of volatile and persistent memory and differentiate persistent objects from volatile ones [7], [26]. Also, new file systems could be created for PM to leverage its byte addressability [8], [11], [27].

Unfortunately, such changes are non-trivial in practice. The existing system hierarchy is built on many long-standing assumptions, which have existed explicitly or implicitly for decades – memory is fast, volatile, byte addressable, while storage is slow, persistent, and block addressable. This commonsense understanding forms the foundation of today's computer system architecture but does not readily apply to PM. Directly integrating PM into the computer system requires that we develop a new understanding of the roles of memory and storage. At the same time, we must be mindful of the large amount of intellectual property associated with legacy systems, and the fact that any radical (business-disruptive) changes will

978-1-4799-5671-5/14/$31.00 © 2014 IEEE Published by the IEEE Computer Society

receive strong resistance in industry. Further, convincing users, especially commercial users, to change their heavily tuned systems or rewrite complex applications (e.g., database) is very difficult in practice.

For these reasons, we believe a complete system redesign would be, at least initially, difficult to gain significant traction in practice — and not always necessary, as shown later. From the perspective of cost efficiency, especially for general-purpose systems, our goal is to strike an appropriate balance between system redesign and exploiting the unique features of PM. That is, our design philosophy is to apply an *evolutionary* approach to this revolutionary technology and minimize disruptive changes.

In the rest of this paper, we share our experiences in building a fast, protected, and persistent block storage based on PM. We study a variety of design options and identify the most suitable ones to effectively achieve these goals. We hope the results presented in this paper encourage discussions about other PM usage models.

B. Our Contributions

Several contributions are made in this paper: (1) We present a hybrid model to combine the advantages of a memory-based model and a storage-based model, while making no changes to the core OS and its applications. (2) We implement a prototype of the hybrid model as a kernel module in Linux 2.6.34 and explore a variety of designs to realize performance, protection, persistence, and compatibility goals; we present the advantages and disadvantages of each approach. (3) We show that with existing platform mechanisms (private page table mappings, non-temporal store instructions, memory fences, and OS I/O barriers), both protection and ordered persistence can be effectively achieved with relatively small performance overhead. We also show that, even when compared to the highly efficient RAM-based file systems (tmpfs, ramfs), our solution can provide good performance. (4) We have open-sourced our prototype. Under the GPLv2 license, the source code is available for public downloading [2]. We encourage researchers and developers to take advantage of this software for further research on PM.

A primary contribution of this paper is demonstrating how existing OS mechanisms can be used in bringing memory-bus-attached PM as block devices to the market. Though obvious in hindsight, determining the right combination (e.g., uncachable vs. non-temporal stores, private-mappings vs. page-based protection, etc.) proved to be a non-trivial exercise. To this end, we also share our less successful attempts in integrating PM into the platform, as they each helped lead to the final solution.

The rest of this paper is organized as follows. Section II covers the background of PM and our storage driver model. Section III describes our experimental platform. Section IV investigates various block driver design options, based on the hybrid model approach. Section V presents our performance evaluation. Section VI introduces the related work. Section VII gives additional discussions and introduces the future work.

II. PM Usage Models

Although the usage model of PM has only indirectly been discussed in prior research, many studies [6], [7], [11], [25], [26], [27] have implicitly assumed at least two basic models:

- **Memory-based Model** – PM is integrated as part of the memory system to displace DRAM memory entirely or partially. A memory controller manages PM and connects CPU to PM via a high-speed memory bus [16], [21], [28]. The OS sees PM as memory, which may be marked as non-volatile by BIOS [27]. Aside from persistence, PM is regarded similarly to DRAM and provides byte addressability to the OS and its applications. Data stored in PM is accessible through `load` and `store` instructions.
- **Storage-based model** – PM is simply used as a faster medium to displace NAND flash and encapsulated in the same form as flash SSDs. An I/O controller manages PM and connects to the host. I/O commands and data are transferred via the I/O bus (e.g., SATA or PCI-E) between the device and the host. Data stored in PM is accessible in units of sectors (512 bytes) via the block I/O interface (`read` and `write` commands).

A. Memory-based Model vs. Storage-based Model

Both usage models have their advantages and disadvantages:

- **Performance** – The memory-based model can provide higher performance than the storage-based model. Even compared with high-speed PCI-E bus, main memory bus throughput and latency both are noticeably better. Further considering the internal overhead of I/O controllers, such as interface command decoding and ECC, the storage-based model cannot provide sufficient headroom for PM, whose speed is close to DRAM.
- **Protection** – The memory-based model has much greater risk of data corruption than the storage model. If we directly map PM, which is attached to the memory bus, into an address space (user or kernel), it will be subjected to the types of data corruption typically caused by stray writes (bad memory pointers). Leaving PM exposed to bugs in the kernel code (e.g., device drivers [5]) can easily corrupt a large amount of persistent data, which often leads to catastrophic results.
- **Ordered persistence** – An issue related with the memory-based model is that data written to PM is subject to CPU caching effects, as applications usually store data in volatile CPU caches for performance. As such, a power failure could cause data loss. In contrast, the storage model provides a mechanism (write barriers) to safely persist data.
- **Compatibility** – The storage-based model provides the best compatibility to existing systems and can be used as a 'drop-in' solution. POSIX I/O can be used as it is today, atop a legacy block-based interface. In contrast, the memory-based model requires non-trivial system and application changes in order to exploit the byte addressability of PM.

B. A Hybrid Model

We propose a *hybrid* model. In this model, we separate the physical and logical architectural designs, as shown in Figure 1: (1) **Physical architecture** – similar to the memory model, PM DIMMs are physically attached to the high-speed memory bus and managed by a memory controller. (2) **Logical architecture** – PM is exposed as a block device in the OS. However, relative to the conventional storage-based model, there is no hardware storage controller (e.g., SAS, SATA, or PCI-E). Data accesses are performed through a `read`/`write` block I/O interface and then converted to memory `load`/`store` instructions by a driver in the OS.

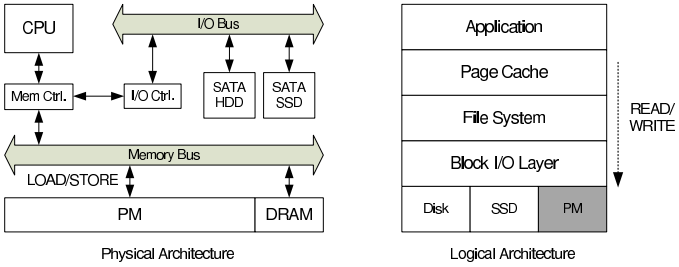


Fig. 1. A hybrid usage model of PM

Such a hybrid design provides several benefits: (1) **High performance** – PM is directly attached to the memory bus, which provides high bandwidth and low latency. (2) **Protection** – We only allow accesses to PM via the `read`/`write` interface. Any unauthorized direct access to PM can be detected and prevented (to be described later). As such, the potential data corruption risk caused by stray writes in the kernel code is minimized. (3) **Ordered persistence** – Since accesses to PM are contained in a single entity (an OS driver), we can enforce ordered persistence through conventional OS write barriers. (4) **Backwards compatibility** – PM is exposed to the rest of the OS kernel in the form of a regular block device, thereby supporting existing OSes and applications. A limitation of the hybrid model is the loss of byte addressability. However, considering that byte addressability demands fundamental code changes to applications, we choose backward compatibility as our top design priority.

In our design, this block device is implemented as a standalone kernel module, called *PMBD*, which requires no change to any other system components. A device driver is loaded and responsible for converting `read`/`write` commands into `load`/`store` instructions. One may note that this design shares the same principle as the RamDisk [24]. However, unlike a RamDisk, our block driver separates page cache and PM and is designed particularly for using PM as *persistent storage*, rather than *volatile memory*. Special considerations, especially protection and ordered persistence, must be carefully explored in the design.

III. Experimental Environment

We first introduce our experimental system and the workloads used in this study. Unless otherwise noted, all the experiments are conducted on the following system setup.

A. System Setup

Our experimental system is a storage server running Fedora Core 14 with Linux kernel 2.6.34 and the Ext4 file system. It has two Intel Xeon X5680 3.3GHz processors, each with 6 cores. In our experiments, we use a 1TB 7,200RPM Seagate hard drive and another 64GB Intel X25-E SLC SSD as the target devices to compare with PM. Another 1TB Seagate hard drive is used as the system disk. All the storage devices are connected through the on-board SAS connectors.

B. PM emulation

As PM devices are unavailable yet, we use DRAM for experiments. We assume that in future systems BIOS will expose PM DIMMs as contiguous physical memory, labeled as non-volatile to the host OS. We also assume hardware handles wear-leveling. We emulate such an architecture by changing the e820 table to reserve the high 16GB of memory space as PM, and the rest as DRAM. Similar to prior work [8], [27], we study the design options with raw DRAM. In Section V-C, we emulate various PM speeds by inserting extra operations to poll the timestamp register (`tsc`), which is similar to [26], and study its end-to-end impact to application performance. To reduce variance, we disable NUMA, HyperThreading, SpeedStep, and lock the memory bus operation speed at 1600MHz.

C. Workloads

We use two sets of workloads. Each workload runs for three iterations, and we show both average values and error ranges.

- **Micro-benchmark** – We use the Intel Open Storage Toolkit [1] to generate various types of micro-benchmark workloads. It can produce I/O workloads with different configurations, such as read/write ratio, random/sequential ratio, request size, and queue depth (the number of concurrent requests). It reports bandwidth, IOPS, and latency.
- **Macro-benchmarks** – We use 8 different workloads as listed in Table II. Five workloads are read intensive, including *sfs*, *tpch*, *glimpseindex*, and *clamav*. TPC-C (*tpcc*) is a database online transaction processing (OLTP) workload and the most write-intensive (63.7%) one. The other two workloads, *tar* and *untar*, have roughly the same amount of reads and writes. We use execution time as the main performance metric. For *tpcc*, which reports the throughput (new order transactions per minute), we convert it to the number of seconds to complete 1 million transactions.

IV. PM Storage Design

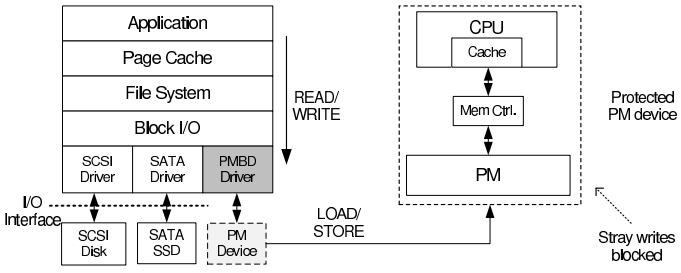


Fig. 2. PM block device architecture

Name	Write	Dataset (MBs)	Amount (MBs)	Description
sfs	7.4%	11,210	146,674	SPECsfs 2008: 10K files, 500K trans., 1K dir.
tpch	9.7%	10,869	78,126	TPC-H (OLAP DB): Scale Factor 4, PostgreSQL 9
tpcc	63.7%	11,298	98-419K	TPC-C (OLTP DB): 80 WH, 20 conn., 60 sec.
tar	46.9%	11,949	11,493	compressing a kernel code tar ball
untar	52.2%	11,970	11,413	uncompressing a kernel code tar ball
devel	38%	2,033	3,470	software development: untar, patch, tar, diff.
glimpseindex	5.5%	12,504	6,019	text index engine: index 12GB Linux kernel code
clamav	0.3%	14,495	5,270	virus scanning: 14GB files generated by sfs

TABLE II
MACRO-BENCHMARK WORKLOAD DESCRIPTION

In our design (Figure 2), the OS views PM as contiguous memory. The PM block driver performs the following tasks: (1) managing the PM space and providing a block device interface for the upper-level components, such as file systems and applications, to access PM via the standard block I/O interface (`read`/`write`), (2) translating incoming `read`/`write` requests into `load`/`store` instructions to access data in PM, (3) providing write protection to prevent PM from being accessed by stray writes (i.e., attempts to modify PM data without going through the PM block device interface), and (4) enforcing ordered persistence through non-temporal store instructions and write barriers.

A. Device Driver Overview

We have implemented the PM block driver (PMBD) as an OS kernel module in Linux 2.6.34. It consists of about 5,000 lines of code with comments. After being loaded into the OS, the driver creates a virtual block device, which provides a standard block I/O interface. From the perspective of other system components and applications, the PM device is no different than any physical block device. Users can create partitions and file systems on it, providing compatibility for legacy applications.

PM is exposed to the OS as a contiguous range of physical memory marked as non-volatile. The PMBD driver is responsible for mapping the PM physical pages (4KB each) into the kernel virtual address space to make it accessible. Upon receiving a request, the driver first computes the physical address of the demanded PM page and then translates `read`/`write` into `load`/`store` instructions. All I/Os are handled synchronously (i.e., no context switch or queuing).

Besides managing PM pages and providing a block I/O interface, the PMBD driver also provides protection from stray writes and ordered persistence. In the following, we will explore a variety of design options to achieve the protection goals without sacrificing performance.

B. Protection from Stray Writes

A fundamental distinction between volatile memory and PM is the requirement for the protection of data. Once written into PM, data is assumed to be *persistent*, as we assume for a hard drive. The challenge is that since PM is physically managed like DRAM and exposed to the OS as memory, reading or writing data requires PM to be mapped into the kernel virtual memory address space, which is shared among all kernel processes. A wild pointer in kernel-level code (e.g., a buggy device driver) can quickly corrupt a large amount of PM data. An example is the famous hardware-destroying bug in Linux 2.6.27-rc8, which overwrites the non-volatile memory on e1000e network adapters [9]. Although it is possible that data could also be corrupted in memory and eventually materialized into storage, it is notably more difficult to pass the sanity checks in multiple layers – page cache, file systems, generic block layer, device I/O, etc. We believe that the same level of protection as a block device must be provided for PM. Also note that our focus is not to prevent malicious attacks but kernel bugs. We have examined several options to provide strong and low-overhead protection as follows.

1) Page Table based Protection: In modern operating systems, *paging* is used to separate the address spaces of processes and share limited physical memory. With paging, physical memory is segmented into multiple fixed-sized frames (4KB, typically). Each process is associated with a *page table* (PT), which is a multi-level tree-like structure translating process-viewable virtual memory addresses to physical memory addresses. The OS is responsible for constructing the page table, allocating and managing physical page frames. The hardware is responsible for automatically translating virtual addresses to physical addresses via the page table or a translation lookaside buffer (TLB) in CPU. The page table is also used for access control. In the last level of the page table, the page table entry (PTE) contains a set of flags. The `R/W` flag controls whether the page is read-only or writable. If the `R/W` flag is 0, the page is read-only, and writing to such a page would be blocked and trigger a page fault.

Based on this page table mechanism, protection from stray writes can be implemented as follows. (1) When the driver is loaded, PM pages are mapped into the kernel virtual address space using `ioremap()` and initialized as read-only by disabling the `R/W` bit. (2) Upon a read, no additional operation is needed. (3) Upon a write, the `R/W` bit is enabled to set the page as writable and perform the write operation, and then disabled to set it back to read-only. (4) When the driver is unloaded, the PM pages are unmapped.

This solution is simple. However, it incurs high performance overhead. To demonstrate, we use the Intel Open Storage Toolkit [1] to generate eight write-only workloads. All workloads use direct I/O to access the PM device (i.e., no page cache) and run for 30 seconds. Figure 3 compares the write

bandwidths of running the micro-benchmarks with and without this protection mechanism, which are denoted as *PT-Naive* and *Baseline* respectively. In the figure, RD and WR denote reads and writes, SEQ and RND denote access patterns (sequential or random), sz4 and sz256 denote the request size (4KB and 256KB), and Q1 and Q32 denote the queue depth (1 or 32 outstanding requests). We can see that write bandwidth is severely reduced by a factor of 18, from 14.8GB/sec to only 808MB/sec.

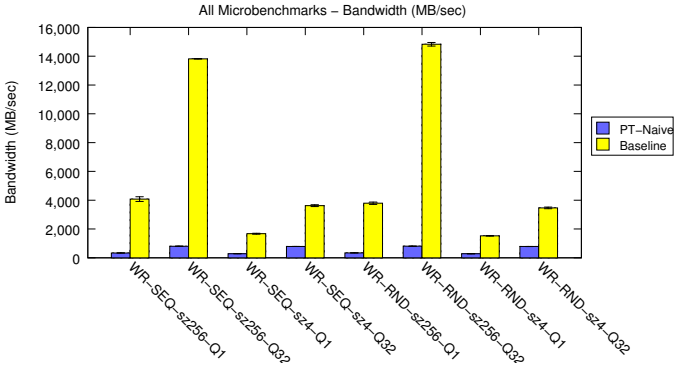


Fig. 3. Performance of page-table based protection

The performance degradation is due to two factors. (1) When writing a page, we need to perform two PTE bit changes (from read-only to writable, and then flip it back). Unfortunately, each PTE attribute change incurs a costly 'TLB shootdown.' Because there is no coherence protocol between TLBs, after a page table entry changes, a high-cost Inter-processor Interrupt (IPI) is needed to flush the TLB entries in all processors. (2) The OS treats changing page table attributes as an infrequent operation, and it is less optimized. For example, the TLB flush happens inside a system-level lock to prevent other processors with stale TLB entries from changing page attributes in parallel, which further lowers the performance.

This protection mechanism, though it can prevent stray writes, violates our performance goal. Although the achievable peak bandwidth is still much higher than most SSDs and HDDs, it only exploits a tiny fraction (5.4%) of its performance potential.

Optimization 1: Protection with Buffer/Batching – Buffering and batching can reduce the cost of manipulating the PTE flags and improve performance, for two reasons. Firstly, a small DRAM space can be used as a temporary buffer to absorb a burst of writes, which acts as an on-device buffer in a hard drive. Secondly, and more importantly, the DRAM buffer can be a staging ground to reorganize pages before updating the page table. Since only one TLB shootdown is needed for updating the `R/W` flags of a sequence of contiguous pages, we can effectively batch up the PTE flag changes and amortize the related overhead.

Buffering – As shown in Figure 4, a circular buffer is used by the PMBD driver to temporarily buffer dirty pages. Upon a write, the dirty page is placed in the buffer. Two pointers, p_{start} and p_{end}, track the first and the last dirty page in the buffer, which segments the buffer into the allocated space and

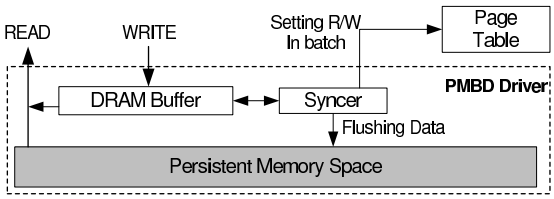


Fig. 4. An illustration of buffering and batching

the free space. We keep track of the number of dirty pages in the buffer. If it exceeds a high watermark, a syncer daemon thread (`nsync`) is woken up to flush the buffer.

Batching – A syncer daemon in the driver can flush the dirty pages in a batch: (1) sort the dirty pages in the order of their virtual memory addresses of the corresponding PM pages; (2) pack contiguous pages into one batch; (3) switch the `R/W` flag of pages in the sequence to be writable; (4) write pages to PM; and (5) switch the PTE flag back to read-only. This process repeats for multiple iterations until all the pages are written to PM.

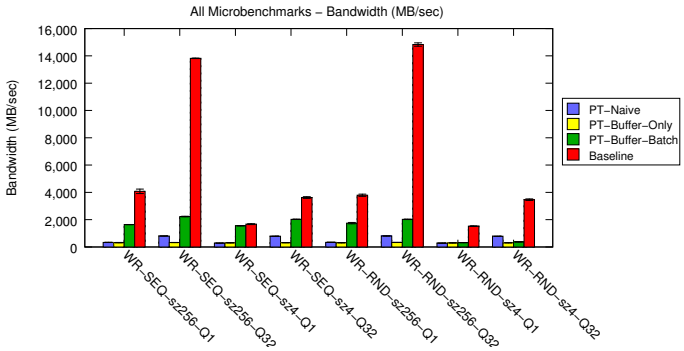


Fig. 5. Performance of buffering and batching

Figure 5 shows the performance comparison between the naive page table based write protection (PT-Naive), the protection with a 16MB buffer (PT-Buffer-Only), the page table protection with buffering and batching (PT-Buffer-Batch), and no protection (Baseline). We have three observations. (1) With buffering only, write performance cannot be improved, sometimes it even gets worse, especially with many concurrent I/O requests. For example, the performance of sequential writes of 4KB with 32 jobs drops from 789 MB/sec to 315 MB/sec. This is because these workloads have no locality and thus benefit little from the buffer, and in the meantime, buffer allocation and flushing become a bottleneck limiting the scalability of handling parallel writes. Without batching, the benefits received from buffering cannot offset this negative impact. (2) With both buffering and batching, sequential write performance can be significantly improved due to the effectively amortized TLB shootdown cost. For example, the performance of sequential writes of 4KB with 32 jobs is improved by 2.5 times to 2028 MB/sec. (3) Random writes receive little performance benefit, as expected, since there is no opportunity for amortizing the cost at all. In fact, with a high queue depth, buffering and batching degrade random write performance. For example, random writes of 4KB with 32 jobs degrade by 2.1 times, from 786 MB/sec to 374 MB/sec.

This case shows that buffering amortizes the cost of page table manipulation but also creates an unexpected new performance bottleneck – For sequential writes, buffering highly benefits from effectively amortized PTE flag change overhead. For random writes, however, the negative impact of losing parallelism dominates and causes substantial performance degradation (2.1 times).

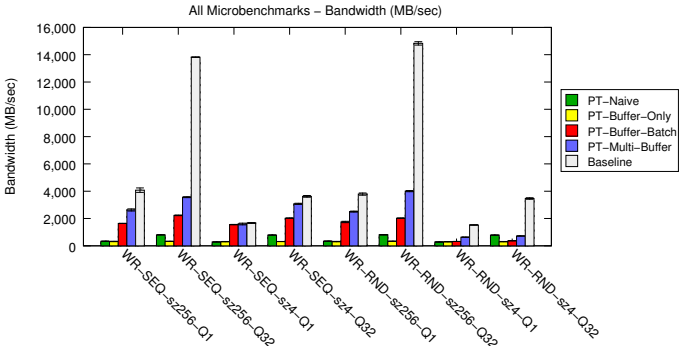


Fig. 6. Performance of multi-buffering

Optimization 2: Protection with Multiple Buffers – In order to improve scalability, multiple buffers can be used, each managed by an individual syncer daemon. Incoming writes are distributed across the buffers in a round-robin manner. Figure 6 shows the performance improvement of using multi-buffering (16 buffers). We see that using multiple buffers can effectively improve system performance. For example, the bandwidth of sequential writes of 4KB with 32 jobs improves from 2028 MB/sec to 3074 MB/sec. For random writes of 4KB with 32 jobs, the worst case for single buffer protection, its performance is improved by 1.9 times, from 374 MB/sec to 724 MB/sec. In these workloads, there is no data access locality, which means adding more buffers cannot bring any caching benefits. However, using multiple buffers enables concurrent buffer accesses in parallel. For workloads with a high queue depth, this optimization removes the buffer bottleneck and substantially improves performance.

In summary, page table based protection incurs a high performance overhead. In the above, we have studied a variety of solutions to address the performance overhead problem. We have improved write performance by nearly 5 times (up to 4,000 MB/sec). Other optimizations, such as disabling write protection bit in CR0 register, are also possible. Besides the performance problem caused by the TLB shootdown, the page table based protection also introduces several other issues. For example, we found that building a page table for a large amount of PM is intolerably slow during the module loading time. Also, for a large capacity of PM, the page table size would become huge and consumes a large amount of memory space. The TLB pollution problems would also emerge [27]. A fundamental reason behind these issues is that the existing page protection mechanism is originally designed for DRAM and not a natural fit to managing hundreds of GBs or even TBs of PM as persistent storage. We need a low-overhead protection for PM.

2) Private Mapping based Protection: We find that *private mapping* can effectively address all the aforesaid issues. Private mappings provide write protection by dynamically mapping PM pages into the kernel space *only when needed*, rather than by controlling the accessibility of each page. When the PM device driver is loaded, the PM pages are not immediately mapped into the kernel virtual address space. Instead, each page is mapped only when the page needs to be accessed (upon a `read`, or a `write`). As so, of most time, the PM space is 'invisible' to the OS kernel code, which prevents the potential damage caused by stray writes.

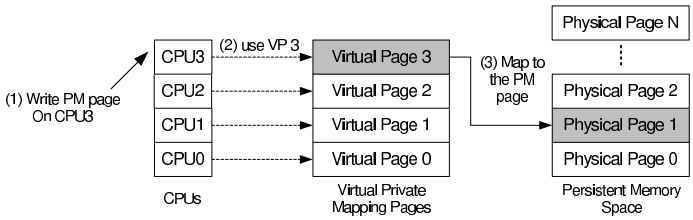


Fig. 7. An illustration of private mapping

For private mappings, we directly manipulate the page table. As shown in Figure 7, assuming a K-core system, we first allocate an array of virtual memory pages, $pmap[K]$, each entry of which is indexed by the CPU ID and private to the process running on it. When a process running on CPU n requests access to PM page p, the physical PM page p is temporarily mapped to the virtual page $pmap[n]$. Then we perform the `memcpy` operation to write data into the specified PM page. After the memory copy is done, we clear the $pmap[n]$ mapping entry. The interrupt is disabled to prevent any context switch. Since no concurrent requests running on other cores will use the same mapping entry, there is no need to shootdown the TLBs on the other cores, and we can completely avoid the related overhead.

Private mappings can provide sufficient write protection. Since we do not enforce the TLB shootdown, if a virtual page happens to be accessed by other cores during the short time window of performing the `memcpy` (typically thousands of cycles), the TLB translation would be loaded into the local TLB. In the worst case, at most $K-1$ TLB entries that are not supposed to be accessible could appear in a TLB, and in total, at most $K \times (K-1)$ entries may be affected at any time. Even so, this is a very small number, compared to the PM storage size, and system events, such as context switches, also flush TLB entries frequently, which further lowers the risk.

Figure 8 shows the performance of using private mappings. We have two key observations. (1) Private mapping can significantly improve write performance close to that of doing no protection (Baseline). For example, random writes of 256KB with 32 jobs can reach 13,389 MB/sec, which is 90% of the optimal case (no protection). (2) For reads, private mappings perform well too. For example, sequential reads of 256KB with 32 jobs reach a bandwidth of 16,076 MB/sec, which is around 85% of the optimal case performance (18,761 MB/sec). The read overhead is mostly due to the fact that when a page

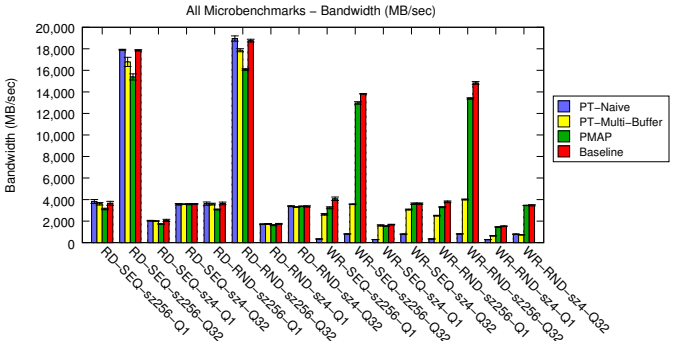


Fig. 8. Performance of private mappings

is to be accessed, it needs to be mapped into the kernel and then the mapping needs to be cleared and invalidated. Such a mapping/unmapping operation would incur additional TLB misses and affect performance, which explains why private mapping slightly underperforms than the others for intensive large reads. However, relative to the huge performance benefit for writes, such a subtle performance loss is acceptable.

Besides performance, private mapping also provides other benefits: (1) **Minimized vulnerable window** – With private mappings, only when an PM page needs to be accessed, is it mapped into the kernel, which minimizes the chance of corruption. The vulnerable window is typically thousands of cycles (while the page is being mapped). (2) **Protection for both reads and writes** – Any attempt to directly access PM data, not only writes but also reads, without going through the driver interface would be immediately blocked. (3) **Scalability with parallel accesses** – With no need for a buffer, the bottleneck for parallel I/O accesses can be removed. It also reduces design complexities and removes the risk of data loss due to the volatile buffer. (4) **Reduced page table size** – Since only one page mapping is needed when being accessed, there is no need to consume a large amount of memory space for building a huge page table for the entire PM storage, which could be as large as multiple Terabytes in the future. (5) **Reduced TLB pollution** – Since we only map the on-access pages, at most one TLB entry is needed on each core at any moment, which removes the risk of TLB pollution [27]. (6) **Fast loading/unloading time** – Since we do not have to build up the page table for all the PM pages, the time of loading and unloading the device driver can significantly be reduced. For all of these reasons, we choose the private mapping as our write protection mechanism.

C. Ordered Persistence

Ensuring ordered data persistence is important for the OS and applications. Storage devices usually provide an on-device buffer and interface ("flush") to write data in two phases – Data is first written into the buffer, and upon an OS write barrier (or the buffer is filled up), data is flushed from the buffer to the medium.

With PM, it is similar but more complicated. Since PM is physically managed by a memory controller and accessed by the `load`/`store` interface, the written data may reside in CPU caches. In other words, the CPU cache acts as a volatile buffer for the PM. Applications, if not changed, may lose data that is supposed to be persistent upon power failures. In our hybrid model, the PMBD driver is the single entity enforcing persistent and ordered writes, which removes the need to change applications. In this section, we first discuss several uncached write schemes for persistence and then how to ensure the write ordering.

1) Uncached Write Schemes: In the existing CPU architecture, we can explicitly make data persistent in several ways: (1) Specifying PM pages as *uncachable* or *write through*. In both modes, writes completely bypass the CPU cache and thus are extremely slow. We do not further consider them in this paper. (2) In write-back mode, use a non-temporal store (e.g., `movntq`), which bypasses the CPU cache and uses the write-combining buffer, and use `sfence` to flush the buffered data to PM. (3) In write-back mode, use regular store instructions (e.g., `mov`) and `clflush` to flush the specified data from the cache, and then use `mfence` to ensure the data is written to PM. (4) In write-back mode, use regular store and `wbinvd` to flush the entire cache. This option is slow and can only be applied during a write barrier, which will be discussed later.

Figure 9 shows the performance of different write schemes: using private mapping in write-back mode, using private mapping with `clflush`/`mfence` or `movntq`/`sfence`, and using no protection in write-back mode (Baseline). When the request size is small (4KB) and the queue depth is high (32 requests), `clflush`/`mfence` outperforms `movntq`/`sfence`. The reason is that `clflush`/`mfence` loads data into the CPU cache, which is much bigger than the buffer used by a non-temporal store, and then flushes the cachelines. This creates more opportunities to pipeline the operations when handling multiple requests in parallel, while `movntq`/`sfence` would be congested in the buffer. However, in all the other cases, `movntq`/`sfence` performs significantly better than `clflush`/`mfence`.

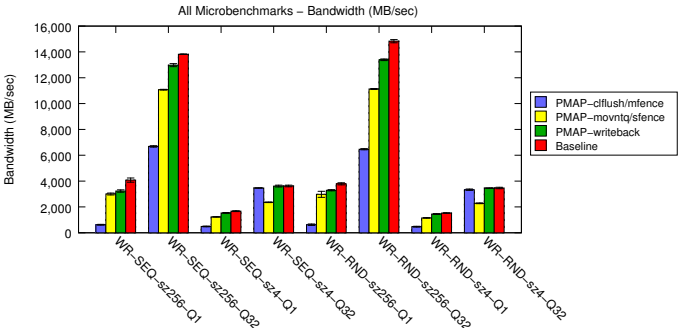


Fig. 9. Micro-benchmark performance of write schemes

We have also examined the write schemes for macro-benchmarks. Figure 10 shows execution times normalized to the baseline case (no protection) of various write schemes without write barriers. As we see, using regular stores with `clflush`/`mfence` performs the worst in most cases, however, the performance difference is smaller than that we see in the micro-benchmarks. Due to the effect of the page cache, most writes are performed asynchronously in the background,

which reduces the impact of write latency. Using a non-temporal store with `sfence` can achieve performance comparable to using private mapping without any persistence guarantees. This conclusion is consistent with that we see in the micro-benchmarks.

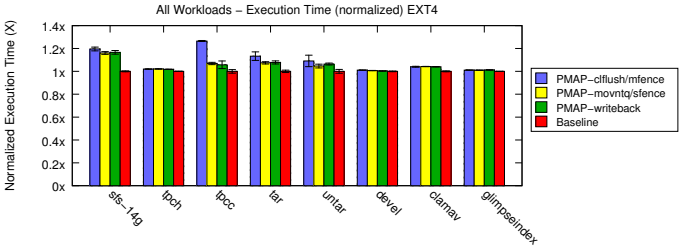


Fig. 10. Macro-benchmark performance results of various write schemes. Normalized to the baseline case.

2) Ordering Writes: Write ordering is needed for applications and systems to safely store data. For example, file system journaling must ensure all the transaction content be materialized to the persistent storage before completing the transaction. Database logging is also similar. With the write schemes described previously, data can be pushed to the memory bus and eventually persistent in PM. However, write ordering is still needed, because multiple parallel writes may exist, and the order of writes being processed may be random.

Write Scheme	Operations in Write Barrier
`movntq/sfence`	N/A
`mov/clflush/mfence`	N/A
`mov`	`wbinvd`

TABLE III
WRITE BARRIER OPERATIONS

Operating systems use *write barriers* to provide such a facility for the system (e.g., file system) and applications (e.g., database) to enforce write ordering and data persistence at a specific time. Upon receiving a write barrier, the OS instructs the storage device to flush its cache before processing any additional incoming writes. We need a similar write ordering mechanism for PM.

In the PM device, the flush command can be implemented as described in Table III. (1) If the non-temporal store (`movntq/sfence`) or (2) the ordered temporal store (`mov/clflush/mfence`) is used, data persistence is provided, and we only need to block the incoming requests (via a spinlock) until all in-progress requests complete for write ordering. (3) If we do not apply `clflush` after each store, we need to flush the entire CPU cache using `wbinvd` during the write barrier.

The abovesaid three schemes have different performance implications. The first two uncached write schemes invalidate the data in the CPU cache immediately after writes, which potentially affects performance due to the lack of cache usage. The third one, using regular store (`mov`) and flushing the entire CPU cache (`wbinvd`) on a write barrier, may provide a potential cache benefit for reads whose data has been written and still in the cache. However, since writebacks performed by the CPU are invisible to the PMBD driver, we cannot exactly track which data (cacheline) is resident in the cache or not. Although a bookkeeping mechanism could be developed to track every updated cacheline, it needs to simulate the CPU cache replacement accurately and is thus infeasible in practice. As so, `wbinvd` has to be used to flush the CPU cache during write barriers. Unfortunately, this could affect performance adversely.

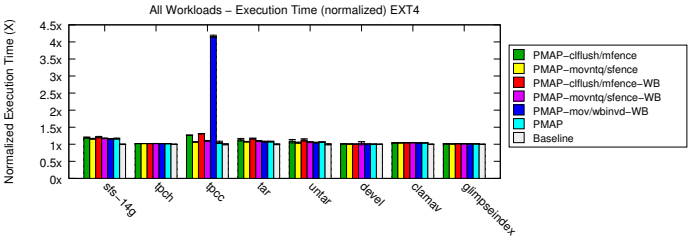


Fig. 12. Performance impact of write barriers

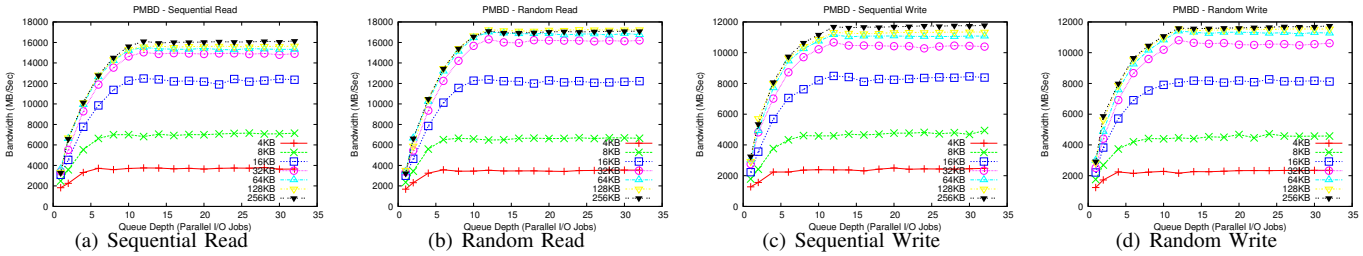
We run 8 macro-benchmarks on the PM device with the Ext4 file system to study the effect of write barriers. Figure 12 shows the performance of using write barriers combined with various write schemes. First, we find that using write barriers for `clflush/mfence` and `movntq/sfence` does not increase the performance overhead, since both write schemes only need to wait for in-progress accesses to finish, if there are any. Secondly, the expected benefit from using the CPU cache is negligible. This is mostly because the CPU cache is too small to effectively hold the working set for typical storage I/Os. In contrast, when handling workloads that generate a large number of write barriers, such as *tpcc*, which is a database workload, the negative impact of frequently flushing CPU cache is significant (a 4.1x slowdown). In almost all workloads, using non-temporal stores (`movntq/sfence`) provides the best performance.

D. Summary

Through the above studies, we have two key conclusions: (1) For write protection, we find that the private mappings are very effective and efficient. Although page table protection can be improved significantly with buffering and batching, it still incurs high performance overhead. (2) For ordered persistence, we find that using a non-temporal store (`movntq`) and `sfence` with write barriers performs the best. Using `clflush/mfence` would result in a performance loss in many cases. Using writeback and flushing the CPU cache with `wbinvd` for write barriers does not improve performance, and in some workloads, it degrades performance significantly. Figure 11 provides a performance overview of our PMBD driver using private mappings, non-temporal stores, and write barriers. With these techniques, reads and writes can achieve bandwidths of about 16GB/sec and 11GB/sec, respectively, which are about 75-85% of the peak (write-back mode with no protection).

V. MACRO-BENCHMARKS

In the above sections, we have investigated the design options to achieve performance, protection from stray writes and ordered persistence. In this section, we focus on the end-to-end performance of applications. We are interested in



(a) Sequential Read (b) Random Read (c) Sequential Write (d) Random Write

Fig. 11. PM performance with various queue depths, request sizes, and patterns

several issues: (1) The impact of file systems running on the PM block device, (2) the performance difference between memory-based file systems and legacy file systems running atop PM, (3) the performance benefit of the PM block device, compared to other block devices (e.g., flash SSDs), and (4) the end-to-end application performance impact of different PM characteristics (read/write speeds).

A. PM Devices vs. Alternatives

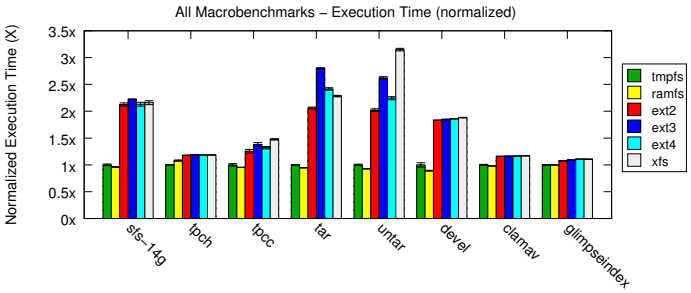


Fig. 13. Performance of tmpfs, ramfs, and PM devices (Ext2/3/4, XFS)

One interesting question is if we adopt the memory based model and design a completely new file system for PM (e.g., BPFS [8]), how much performance difference we would see compared to running a legacy disk-based file system (e.g., Ext4) on PM, which provides necessary protection and ordered persistence transparently. Since BPFS is implemented with FUSE as a user-level file system, its performance is incomparable to a native file system. Thus, we use the two RAM-based file systems in Linux, *tmpfs* and *ramfs*, to estimate the maximum performance difference. Both *tmpfs* and *ramfs* are directly integrated with the page cache and have no protection, so their performance can be regarded as the optimal case for memory-based file systems.

Figure 13 shows the experimental results. We can see that the two RAM-based file systems show higher performance than any stock disk-based file system running on PM. This is not surprising for several reasons. (1) Designed for DRAM, tmpfs and ramfs do not provide special mechanisms (e.g., write ordering and private mappings) for protection and persistence, which means less overhead. (2) Since tmpfs and ramfs are directly integrated with the page cache, there is no extra memory copy between the page cache and the storage, not to mention additional overhead for file system metadata accesses. (3) Since tmpfs and ramfs would not be filled up immediately, the applications have more available memory space for use during execution. In contrast, the PMBD driver would reserve all the memory space at the beginning. However, we should note that, even so, the performance difference between using a stock disk-based file system running on PM and a simple memory based file system (the optimal case) is workload dependent and not as significant as expected in some cases. In workloads, such as *tpch* (19%), *clamav* (16%), and *glimpseindex* (10%), the performance difference is rather small. In the worst case (*untar*), which is completely dominated by I/Os, the difference is about 3 times. This indicates that for most workloads, which are mixed with both computation and I/Os, adopting the block-level solution can deliver reasonable performance without sacrificing compatibility. For extremely I/O intensive workloads, such as *untar*, a customized file system for PM can bring additional performance benefits.

In the figure, we also can see that different file systems show different performance. Ext2 on PM performs the fastest for all workloads, while XFS and Ext3 perform the worst in most cases, since both are journaling file systems, which perform extra I/Os. This indicates that simplifying file systems can bring performance benefits. We also note that Ext4 performs better than Ext3, especially when a large number of small files are involved, such as *tar* and *untar*.

B. PMBD vs. HDD and SSD

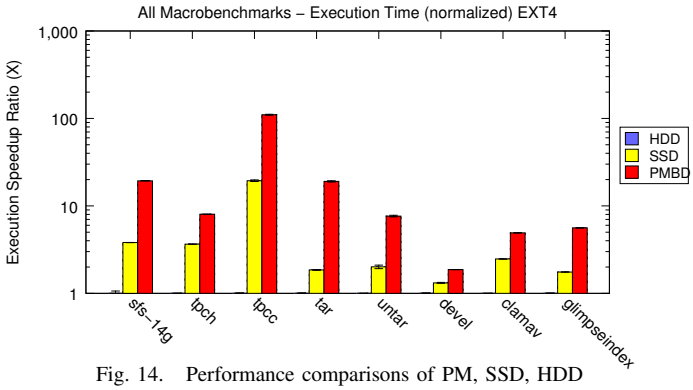


Fig. 14. Performance comparisons of PM, SSD, HDD

One may also ask how much benefit we would see in terms of end-to-end application performance by using PM, compared to a hard drive or an SSD. Figure 14 shows experimental results of running the eight workloads with HDD, SSD, and PMBD. As expected, PMBD clearly shows performance advantages in all cases. The relative performance improvement depends on the I/O intensity of workloads. For example, *devel*

involves much computation, and the speedup by using PMBD is relative smaller (1.8 times faster than HDD). In *tpcc*, the best case, PMBD is 110 times faster than HDD and 5.7 times faster than SSD. This is because *tpcc* generates a large number of small writes with syncs and write barriers, which is the worst case for magnetic disk drives, while PMBD shows superior performance for such workloads. This result also indicates that for databases, PM is a natural fit and can significantly improve system performance.

C. End-to-end Performance Impact of PM Speeds

As a technology in development, there is little device-level specification reported in public literature. Most specification numbers are based on cell-level speed, which cannot reflect the true device-level access speed. Rather than using the absolute time, as used in prior work, we specify the PM device performance by using *relative* speeds to DRAM. In particular, in our PMBD driver, the user can specify two slowdown parameters, `rdsx` and `wrsx`. Accordingly, we track the cycles spent on each memory copy operation and proportionally inject a delay by polling the `tsc` (timestamp counter) register to slowdown DRAM speeds. Using DRAM as a baseline, we can study how the performance would be affected without knowing the exact device specifications.

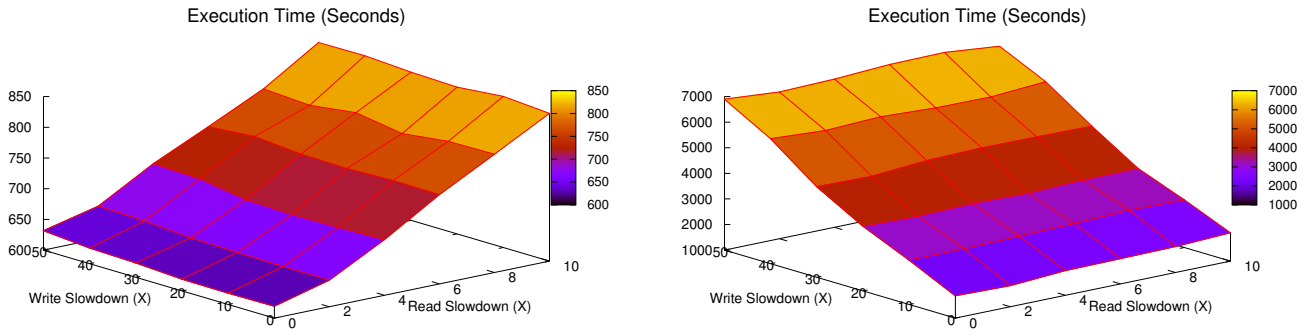
Figure 15 shows the performance impact of PM speed (slowdown factor: 1-10x for reads, 1-50x for writes). We show two representative workloads, *tpch* and *tpcc*. We have several observations: (1) The performance degradation is not proportional to the PM speed. For example, although the read latency of PM has been increased by 10 times, *tpch* performance is only reduced by 36%. (2) The impact to performance varies across applications. For example, the *tpcc* performance degrades by over 3.6 times in the worst case, while *tpch* performance degrades only by 1.4 times. (3) Applications show different sensitivities to read and write speeds. In particular, *tpch* is more sensitive to read performance, while *tpcc* is more sensitive to write performance.

VI. Related Work

Persistent memory has received increasingly high interest in academia and industry. Some suggest using PM to displace DRAM [16], [20], [21], [28]. Some propose to use a storage-based model, such as Onyx [3] and Moneta [6]. Some consider applications of PM, such as providing whole system persistence [18] and unioning the buffer cache and journaling layers [17]. Compared to the impact of flash memory to file and database systems [12], [23], PM is a more disruptive technology and also more deeply changing existing computing practices, from programming models to system designs. For example, Mnemosyne [26] provides a simple interface for programming with persistent memory, such as declaring persistent data with the keyword `pstatic`. CDDS [25] emphasizes more on providing a consistent and durable data structure for programmers to safely exploit the performance and persistence of PM. NV-Heaps [7] provides transactional semantics in an easy-to-use model. SoftPM [13] provides a new memory abstraction to allow `malloc` style allocations. Pâris et al. proposes to use storage-class memory for enhancing reliability of RAID [19]. Kang et al. proposes an object-based model for storage-class memory [15]. Prior work also considers designing new file systems for PM, where PM is directly attached to the memory bus. BPFS [8] is a file system designed specifically for PM. It uses shadow paging techniques to provide fast updates to critical file system structures, and hardware change is needed to provide epoch barrier for write ordering. In contrast, we use PM as a storage device. SCMFS [27] uses the page table based solution to manage PM files. In their results, they also found that TLB pollution can become a problem when handling a large amount of PM. In this paper, we show that using private mapping is efficient to address the TLB cost issue [4]. PMFS [11] provides a file system interface and allows user programs to use memory mapping (mmap) for directly accessing persistent memory. Such an approach can potentially deliver better performance by reducing extra memory copies, however it requires users to migrate to a new file system, and application modification is needed to fully take advantage of memory mapped I/Os. In contrast, we strive to avoid such changes. For write protection, PMFS manipulates the write protection bit in CR0 to avoid the TLB shootdown cost. We uses private mapping to address the same problem. A unique benefit enabled by private mapping is that it also solves the TLB pollution and page table size problems, since it does not require a page table for the entire PM storage. Our work and these prior studies represent distinct but consistent efforts in exploring the potential design space. The final design choices, though different, are simply the results of carefully balancing various factors on how to utilize this new technology with different emphasis.

VII. Discussions and Future Work

Integrating PM into today's computing systems is challenging. It demands a careful consideration of various factors, such as performance, cost, compatibility, etc. In contrast to many prior studies, we strive to reach a balance between exploiting the potential of PM and minimizing system changes. As an important goal of this work, we attempt to minimize disruptive changes to the existing eco-system. By creating a block-addressable device interface atop byte-addressable memory, certain optimization opportunities, such as byte addressability, could be lost. However, such a tradeoff is often worthwhile and necessary in practice. In fact we find that in many cases we can get most of potential performance benefits of PM without any disruptive changes. For example, many workloads running on a memory file system is only 10% to 20% faster than on a legacy disk file system on PM. This is for several reasons. First of all, many workloads access persistent storage in a reasonably large granularity (4KB or larger), and the cost of small I/Os (e.g., file system metadata accesses) are hidden by prefetching and caching effects. As so, enabling sub-page I/Os to PM may bring extra performance gains, but very likely at a limited scale. Also, since many real-life workloads (e.g., *devel*) are fixed with both computation and I/Os, the



(a) TPCH (b) TPCC

Fig. 15. Application performance with various PM specifications (read: 1-10x, write: 1-50x)

overall application performance is not purely dominated by I/O speed. Thus, further speeding up PM accesses may not lead to an extraordinary performance boost as we expect. On the other hand, we also note and confirm that for certain workloads with highly intensive I/Os, especially small I/Os and frequent storage syncs (e.g., *tpcc*), a PM-specific design is beneficial. Leveraging the byte addressability of PM and removing extra memory copies for such workloads can result in noticeable performance benefits. This also inspires us to further improve our solution in the future. For example, we can provide `mmap` support to allow programmers to directly access PM storage at a sub-page granularity. We may also develop other new interfaces to optimize certain applications as well. In short, we believe that as PM technology continues to develop, researchers need to invest more efforts to identify an suitable usage model to adopt this revolutionary technology.

VIII. Conclusion

Persistent memory technologies draw a blurry line between memory and storage. As a result, neither a memory-based model or a storage-based model is a best fit to achieve the goals of performance, protection from stray writes, and ordered persistence. In this paper, we present a hybrid model. We find that with private mappings, non-temporal store, memory fences, and write barriers, we can effectively achieve these goals with little impact to the OS and its applications. With all the lessons learned during this research, it is shown that determining the right combination of existing platform and OS mechanisms for PM is a non-trivial exercise. We have open-sourced our software prototype for public downloading. We hope that this work can encourage researchers to further explore various ways to exploit the benefits of PM.

Acknowledgment

The authors would like to thank anonymous reviewers for their constructive comments to improve this paper. We also thank Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, Jeff Jackson, Paul Brett, Dheeraj Reddy, David Koufaty, and Ross Zwisler for discussions and support during this work.

References

[1] Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[2] Persistent Memory Block Driver. http://www.github.com/linux-pmbd.

[3] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A Prototype Phase Change Memory Storage Array. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2011)*, Portland, OR, June 14 2011.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, Big Sky, MT, October 2009.

[5] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX 2010)*, Boston, MA, June 23-25 2010.

[6] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A High-Performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2010)*, Atlanta, Georgia, Dec 4-8 2010.

[7] J. Coburn, A. M. Caufield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-GEneration, Non-Volatile Memory. In *Proceedings of the 2011 Architectural Support for Programming Languages and Operating Systems (ASLPOS 2011)*, Newport Beach, CA, March 5-11 2011.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. C. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 09)*, Big Sky, MT, October 2009.

[9] J. Corbet. The State of the e1000e Bug. In *Linux Weekly News*, Oct. 1 2009.

[10] B. Dieny, R. S. G. Prenat, and U. Ebels. Spin-dependent Phenomena and Their Implementation in Spintronic Devices. In *Proceedings of 2008 International Symposium onVLSI Technology, Systems and Applications (VLSI-TSA 2008)*, pages 70–71, April 2008.

[11] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, and R. S. J. Jackson. System Software for Persistent Memory. In *Proceedings of the 2014 European Conference on Computer Systems (EuroSys 2014)*, Amsterdam, ST, Netherlands, April 13-16 2014. The ACM.

[12] G. Graefe. The Five-Minute Rule 20 Years Later. In *Communications of ACM*. The ACM, July 2009.

[13] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software Persistent Memory. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, June 13-15 2012.

[14] Y. Ho, G. Huang, , and P. Li. Nonvolatile Memristor Memory: Device Characteristics and Design Implications. In *Proceedings of 2009. IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers (ICCAD 2009)*, pages 485–490, Feb 5 2009.

[15] Y. Kang, J. Yang, and E. L. Miller. Object-based SCM: An Efficient Interface for Storage Class Memories. In *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies: Research Track (MSST 2011)*, Denver, CO, May 23-27 2011.

[16] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, 2009.

[17] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologoies (FAST 2013)*, San Jose, Feb 12-15 2013.

[18] D. Narayanan and O. Hodson. Whole-system Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, March 2012.

[19] J.-F. Pâris, A. Amer, and D. D. E. Long. Using Storage Class Memories to Increase the Reliability of Two-Dimensional RAID Arrays. In *Proceedings of the 17th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*, London, UK, Sept. 2009.

[20] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42th International Symposium on Microarchitecture (MICRO 2009)*, Dec 2009.

[21] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High Performance Main Memory System using Phase-Change Memory Technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, June 2009.

[22] S. Raoux, G. W. Burr, M. J. Breitwisch, C. Rettner, Y.-C. Chen, r. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. Lam. Phase-change Random Access Memory: A Scalable technology. In *IBM Journal of Research and Development*, volume 52(4), pages 465–479, 2008.

[23] J. Ren and Q. Yang. I-CASH: Intelligently Coupled Array of SSD and HDD. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA 2011)*, San Antonio, TX, Feb 2011.

[24] P. Synder. tmpfs: A Virtual Memory File System. In *Proceedings of the Autumn European UNIX User's Group Conference*, September 1990.

[25] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 2011)*, San Jose, CA, February 15-17 2011.

[26] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Light Weight Persistent Memory. In *Proceedings of the 2011 Architectural Support for Programming Languages and Operating Systems (ASLPOS 2011)*, Newport Beach, CA, March 5-11 2011.

[27] X. Wu and A. L. N. Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of Supercomputing (SC 2011)*, Seattle, WA, Nov 12-18 2011.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, June 2009.

Understanding I/O Performance Behaviors of Cloud Storage from a Client's Perspective

Binbing Hou
Louisiana State University
bhou@csc.lsu.edu

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

Zhonghong Ou
Beijing Univ. of Posts & Telecomm.
zhonghong.ou@bupt.edu.cn

Ren Wang
Intel Labs
ren.wang@intel.com

Michael Mesnier
Intel Labs
michael.mesnier@intel.com

***Abstract*—Cloud storage has gained increasing popularity in the past few years. In cloud storage, data is stored in the service provider's data centers, and users access data via the network. For such a new storage model, our prior wisdom about conventional storage may not remain valid nor applicable to the emerging cloud storage. In this paper, we present a comprehensive study and attempt to gain insight into the unique characteristics of cloud storage, primarily from the client's perspective. Through extensive experiments and quantitative analysis, we have acquired several interesting, and in some cases unexpected, findings. (1) Parallelizing I/Os and increasing request sizes are keys to improving the performance, but optimal bandwidth may only be achieved with a proper combination of parallelism and request size. (2) Client capabilities, including CPU, memory, and storage, play an unexpectedly important role in determining the achievable performance. (3) A geographically long distance affects client-perceived performance but does not always result in lower bandwidth and longer latency. Based on our experimental studies, we further present a case study on appropriate chunking and parallelization in a cloud storage client. Our studies show that specific attention should be paid to fully exploiting the capabilities of clients and the great potential of cloud storage services.**

***Index Terms*—Cloud Storage; Storage Systems; Performance Analysis; Measurement.**

I. Introduction

Cloud storage is a quickly growing market. According to a report from Information Handling Services (IHS), personal cloud storage subscriptions increased to 500 million in 2012 and will reach 1.3 billion by 2017 [37]. The global market is expected to grow from $18.87 billion in 2015 to $65.41 billion by 2020 [5]. To date, cloud storage is not only used for archiving personal data, but also plays an indispensable role in various core commercial services, from serving videos on demand to storing unstructured scientific data.

To end users, cloud storage is particularly interesting because it provides a compelling new storage model. In this model, data is stored in the service provider's data centers, and users access data through an HTTP-based REST protocol via the Internet. By physically and logically separating data storage from data consumers, this architecture enables enormous flexibility and elasticity, as well as the highly desirable cross-platform capability. On the other hand, such a model is drastically distinct from conventional direct-attached storage – the "storage medium" is replaced by a large-scale storage cluster, which may consist of thousands of massively parallelized machines; the "I/O bus" is the worldwide Internet, which allows connecting two geographically distant ends; the strictly defined "I/O protocol" is replaced by an HTTP-based protocol; the "host" is not a single computing entity any more but could be any kind of computing devices (e.g., PCs or Smartphones). All these properties, together, form a rather loosely coupled system, which is fundamentally different from its conventional counterpart. A direct impact of such change is that much of our prior wisdom about storage, the basis for our system optimizations, may not continue to be applicable to the emerging cloud-based storage.

This is because of several reasons. First, the massively parallelized storage cluster, where data is stored, potentially allows a large amount of independent parallel I/Os to be processed quickly and efficiently. In contrast, our conventional storage emphasizes how to organize sequential I/O patterns to address the limitation of rotating mediums [25], [39]. Second, compared to the stable and speedy I/O bus, such as the Small Computer System Interface (SCSI), the lengthy Internet connection between the client and the cloud is slow, unstable, and sometimes unreliable. A cloud I/O could travel an excessively long distance (e.g., thousands of miles from coast to coast) to the service provider's data center, which may involve dozens of network components and finally result in an I/O latency of hundreds of milliseconds or even more. Finally, the clients, which consume the data and drive the I/O activities, are highly diverse in all aspects, from CPU, memory, storage, to communication. Certain specifications (e.g., CPU) are directly related to the capability of a client for handling parallel network I/Os.

Unfortunately, our current understanding on storage behaviors, are mostly confined in the conventional storage, which is well-defined and heavily tuned to scale in a limited scope, such as direct attached storage or local Storage Area Network (SAN). Without a thorough and detailed study, it is difficult to obtain key insights on the unique I/O behaviors of cloud storage, a storage solution for cloud stacks, especially from the perspective of data consumers. In this paper, we attempt to answer a set of important questions listed as follows.

978-1-4673-9055-2/16/$31.00 © 2016 IEEE

Successfully answering these questions cannot only help us understand the effect of several conventional key factors (e.g., parallelism and request sizes) on cloud I/O behaviors, but also several new issues (e.g., client capabilities, geo-distances), which are unique to the cloud-based storage model.

- Parallelization and request size are two key factors affecting the performance of storage. What are their effects on the performance of cloud storage? Can we make a proper tradeoff between parallelism degree and request size?
- CPU, memory, and storage are three major components defining the capability of a client. In the scenario of cloud storage, which is the most critical one affecting the performance of cloud storage? What are their effects on the performance under different workloads?
- The geographical distance between the client and the cloud determines the Round Trip Time (RTT), which is assumed to be a critical factor affecting the cloud storage speed. What is the effect of such geographical distance to cloud I/O bandwidths and latencies? Should we always attempt to find a nearby data center of a cloud storage provider?
- Based on our experimental studies on the performance of cloud storage, what are the associated system implications? How can we use them to optimize client applications to efficiently exploit the advantages of cloud storage?

In this paper, we present a comprehensive experimental study on cloud storage and strive to answer these critical questions. Unlike some prior studies that primarily focus on the cloud storage providers (e.g., [27], [28], [29]), we pay specific attention to the client side. In essence, our study regards cloud storage as a type of storage service rather than network service. As such, we are more interested in characterizing the end-to-end performance perceived by the client, rather than the intermediate communications. We believe this approach also echoes the demand for thoroughly understanding cloud storage for a full-system integration as a storage solution [22].

For our experiments, we develop and run a homemade test tool over Amazon Simple Storage Services (S3). By using latencies and bandwidths, which are the two key metrics used in storage studies, we perform a series of experiments with five different client settings to study the effect of clients' capabilities and locations. Based on our experimental studies, we further study several optimization issues on the client side, such as identifying a proper chunk size for caching and parallelization for prefetching. We hope this work can provide a complete picture of cloud storage and inspire the research community, especially cloud storage system and application designers, to further leverage the unique characteristics of cloud storage for effective optimizations.

The rest of the paper is organized as follows. Section II introduces background. Section III describes the methodology for our experimental studies. Section IV and V present the results and case study. Section VI discusses the system implications of our findings. Related work is presented in Section VII and the last section concludes this paper.

II. Background

A. Cloud Storage Model

In cloud storage, the basic entity of user data is an *object*. An object is conceptually similar to a file in file systems. An object is associated with certain metadata in the form of key/value pairs. Typically, an object can be specified by a URL consisting of a service address, bucket, and object name (e.g., https://1.1.1.1:8080/v1/AUTH_test/c1/foo). The maximum object size is typically 5GB, which is the limit of the HTTP protocol [15]. Objects are further organized into logical groups, called *buckets* or *containers*. A bucket/container is akin to a directory in a file system but cannot be nested.

Almost all cloud storage service providers offer an HTTP-based Representational State Transfer (REST) interface to users for accessing cloud storage objects. Some also provide language-specific APIs for programming. Two typical operations are `PUT` (uploading) and `GET` (downloading), which are akin to `write` and `read` in conventional storage. Other operations, such as `DELETE`, `HEAD`, and `POST`, are provided to remove objects, retrieve and change metadata. For each operation, a URL specifies the target object in the cloud storage. Additional HTTP headers may be attached as well.

B. Cloud Storage Services

Cloud storage is designed to offer convenient storage services with high elasticity, reliability, availability, and security guarantees. Amazon S3 [3] is one of the most typical and popular cloud storage services. Other cloud storage services, such as OpenStack Swift [10], share a similar structure. Typically, the cloud storage service is running on a large-scale storage cluster consisting of many servers for different purposes, from handling HTTP requests, accounting, storage, to bucket listings, etc. These servers could be further logically organized into *partitions* or *zones* based on physical locations, machines/cabinets, network connectivity and so on. For reliability, the zones/partitions are isolated with each other, and data replicas should reside separately. In short, the cloud storage services are built on a massively parallelized structure and are highly optimized for throughput.

C. Cloud Storage Applications

Applications can access cloud storage in different ways. Some applications use the vendor-provided APIs to directly program data accesses to the cloud in their software. Such APIs are provided by the service provider and are usually language specific (e.g., Java or Python). Since a cloud storage object can be located via a specified URL, users can also manually generate HTTP requests by using tools like `curl` to access the link.

A more popular category of cloud storage applications is for personal file sharing and backup (e.g., Dropbox). Such applications often provide a filesystem-like interface to allow end users to access cloud storage. From the perspective of data exchange, these clients often use syncing or caching to enhance user experience. With the syncing approach, the client

Client	Instance	Location	Zone	vCPU	Memory	Storage	Network	OS
Baseline	m1.large	Oregon	us-west-2a	2	7.5GB	Magnetic(410GB)	Moderate	Ubuntu 14.04 (PV)
CPU-plus	c3.xlarge	Oregon	us-west-2a	4	7.5GB	Magnetic(410GB)	Moderate	Ubuntu 14.04 (PV)
MEM-minus	m1.large	Oregon	us-west-2a	2	3.5GB	Magnetic(410GB)	Moderate	Ubuntu 14.04 (PV)
STOR-ssd	m1.large	Oregon	us-west-2a	2	7.5GB	SSD(410GB)	Moderate	Ubuntu 14.04 (PV)
GEO-Sydney	m1.large	Sydney	ap-southeast-2a	2	7.5GB	Magnetic(410GB)	Moderate	Ubuntu 14.04 (PV)

TABLE I
CONFIGURATIONS OF AMAZON EC2-BASED CLIENTS. THE SSD IS THE PROVISIONED SSD WITH 3,000 IOPS.

Speed / Size	Magnetic		SSD	
	Read	Write	Read	Write
1KB	2.13 MB/s	0.77 MB/s	2.7 MB/s	1.24 MB/s
4KB	6.70 MB/s	3.13 MB/s	10.57 MB/s	5.67 MB/s
16KB	6.80 MB/s	4.60 MB/s	34.87 MB/s	10.65 MB/s
64KB	7.36 MB/s	10.67 MB/s	62.00 MB/s	28.48 MB/s
256KB	17.36 MB/s	17.46 MB/s	58.24 MB/s	86.63 MB/s
1MB	38.33 MB/s	22.38 MB/s	58.24 MB/s	82.71 MB/s
4MB	61.59 MB/s	23.20 MB/s	58.06 MB/s	82.72 MB/s
16MB	58.12 MB/s	22.66 MB/s	58.12 MB/s	82.92 MB/s

TABLE II
MAGNETIC VS. SSD

Object Size	Object Number	Workload Size
1KB	81920	80MB
4KB	40960	160MB
16KB	40960	640MB
64KB	40960	2560MB
256KB	40960	10240MB
1MB	16384	16384MB
4MB	4096	16384MB
16MB	2048	32768MB

TABLE III
OBJECT-BASED WORKLOADS

maintains a complete copy of the data stored on the cloud-side repository. A syncer daemon monitors the changes and periodically synchronizes the data between the client and the cloud. With the caching approach, the client only maintains the most frequently used data in local, and any cache miss leads to on-demand data fetching from the cloud. In practice, the syncing mode is adopted by almost all personal cloud storage applications, such as Dropbox [6], Google Drive [8], OneDrive [9], etc. The caching mode is adopted by the applications and storage systems that make use of the cloud as a part of the I/O stack, such as RFS [26], S3FS [13], S3backer [12], BlueSky [45], SCFS [18], etc. In general, all the above-mentioned applications essentially convert the POSIX-like file operations into an HTTP-based protocol to communicate with the cloud. For the sake of generality, our study carefully avoids using any specific application techniques but uses the raw HTTP requests.

III. MEASUREMENT METHODOLOGY

As mentioned above, the main purpose of our experimental studies is to characterize the performance behaviors of cloud storage from the client's perspective. In our experiments, we treat the cloud as a "blackbox" storage. In order to avoid interference from client-side optimizations, we carefully generate raw cloud I/O traffic via the HTTP-based REST protocol to directly access the cloud storage and observe the performance on the client side.

Cloud storage services: Our experiments are conducted on Amazon Simple Storage Services (S3). As a representative cloud storage service, Amazon S3 is widely adopted as the basic storage layer in consumer and commercial services (e.g., Netflix and EC2). Some third-party cloud storage services, such as Dropbox, are directly built on S3 [7]. In our experiments, we use the S3 storage system hosted in Amazon's data center in Oregon (s3-us-west-2.amazonaws.com).

Cloud storage clients: In order to run the experiments in a stable and well-contained system, we choose Amazon EC2 as our client platform from which the cloud storage I/O traffic is generated to exercise the target S3 repository. An important reason of choosing Amazon EC2 rather than our own machines is to have a quantitatively standardized client that provides a publicly available baseline for repeatable and meaningful measurement. For analyzing the impact of client variance, we customized five configurations of Amazon EC2 instances which feature different capabilities in terms of CPU, memory, storage, and geographical location. Table I shows these configurations. The *Baseline* client is located in Oregon and equipped with 2 processors, 7.5 GB memory, and 410 GB disk storage (denoted as Magnetic). The speeds of the Magnetic and the SSD are tested and shown in Table II. The other four configurations vary in different aspects, specifically CPU, memory, storage, and geographical location (in Sydney). These instances can properly satisfy our needs of observing cloud storage performance with differing clients.

Test workloads: For our experiments, we develop a home-made tool by using the S3 API [11] to generate raw cloud I/O requests to S3. We purposely avoid using POSIX APIs (e.g., S3FS) because our goal is to gain the direct view of the cloud storage performance from the client side. Certain techniques (e.g., local cache, data deduplication, data compression) used in some client tools will prevent us from observing the cloud I/O behaviors completely or accurately. Our tool allows us to create combinations of various parallelism degrees (1-64), object sizes (1KB to 16MB), and types (PUT or GET). Before each run, we generate objects of the same size with unique keys/names in the client storage and upload to the cloud as the uploading workloads; we then download the objects to the client. Table III lists more details about the workloads.

Accuracy: Considering the possible variance of network services and multi-thread scheduling, we take the following

measures to ensure the accuracy and repeatability of the experiments: (1) As stated above, we customize the instances of Amazon EC2 which can provide stable services as standard clients rather than picking up a random machine. (2) To avoid memory interferences across experiments, the memory is flushed before each run of the experiments. (3) We make the size of the workloads large enough (see Table III) so that each run of an individual experiment lasts for a sufficiently long duration (at least 60 seconds) while still being able to complete the experiments within a reasonable time frame. (4) Each experiment is repeated for five times, and we report the average value while discarding obvious outliers.

IV. Performance Studies

To comprehensively reveal the effects of different factors, our measurement work is composed of two parts. We first conduct a set of general experiments to evaluate the properties of cloud storage, including parallelism degree and request size. We then focus on studying the effects of client capabilities, including CPU, memory, storage, and geographical locations of the clients.

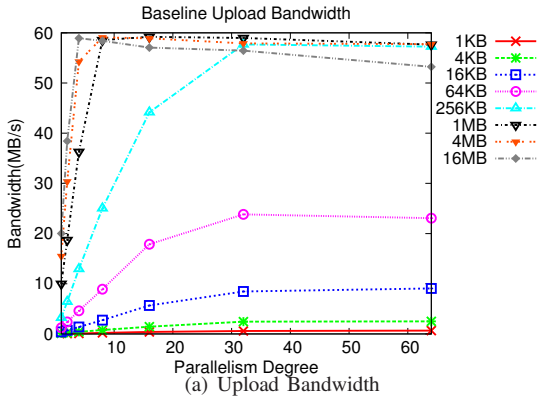
A. Basic Observations

Parallelism and request size are two critical factors that significantly affect the storage performance. Considering the parallelism potential of cloud storage, we set the parallelism degree up to 64. With regard to request size, prior work has found that most user requests are not excessively large [19], typically smaller than 10MB [29]. Also, for transfer over the Internet, most cloud storage clients split large requests into smaller ones. Wuala and Dropbox, for example, adopt 4MB chunks, and Google Drive uses 8MB chunks, while OneDrive uses 4MB for upload and 1MB for download [19]. Therefore, we set the request size up to 16MB to study the size effect.

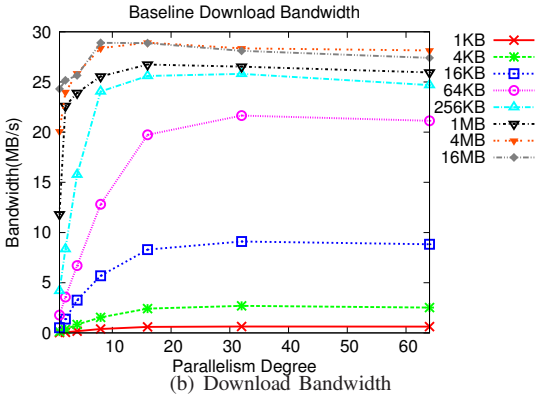
1) The effect of parallelism: In conventional disk drives, I/O parallelism has limited effect due to its mechanic nature. For cloud storage, which stores data in a cluster of massively parallelized storage servers, I/O parallelism has a significant impact to the client-perceivable performance.

Q1: How does parallelism affect the bandwidth?

Generally, proper parallelization can dramatically improve the bandwidth, while over-parallelization may lead to bandwidth degradation to certain degree. As shown in Figure 1, for example, the bandwidth of 1KB upload requests can be improved up to 27-fold (from 0.025MB/s to 0.666MB/s), and the bandwidth of 1KB download requests can be improved up to 21-fold (from 0.03MB/s to 0.634MB/s). There are two reasons for this. One reason is due to the underlying TCP/IP protocol for communication. With TCP/IP, the client and the cloud have to send ACK messages to confirm the success of the transmission of data packets. With a high parallelism degree, multiple flows can continuously transmit data since the time taken by each parallel request to wait for the ACK messages overlaps. Another reason is that smaller requests often require fewer client resources, so the client can support a higher parallelism degree to saturate the pipeline until the effect of parallelization is limited by one of the major client resources.



(a) Upload Bandwidth



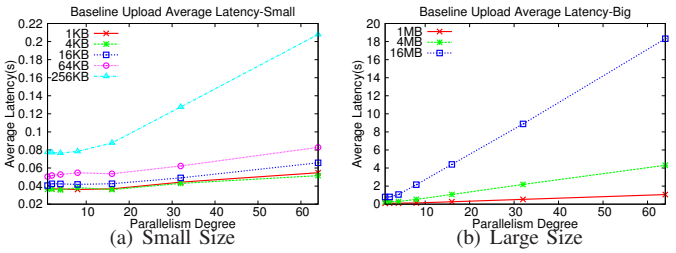
(b) Download Bandwidth

Fig. 1. Bandwidth on Baseline

Over-parallelization brings diminishing benefits and even negative effects. For example, 16MB upload sees a slight performance degradation caused by over-parallelization. This is related to the overhead of maintaining the thread pool when the CPU is overloaded. To confirm this, we use `vmstat` in Linux to investigate the CPU utilization of the 16MB upload request with different parallelism degrees. As shown in Figure 4, when the parallelism degree is 8, the CPU utilization quickly grows close to 100%, indicating that the CPU is overloaded. Under this condition, further increasing the parallelism degree will only increase the overhead of maintaining the thread pool, which will consequently reduce the overall performance. In a later section, we will further study the effect of CPU.

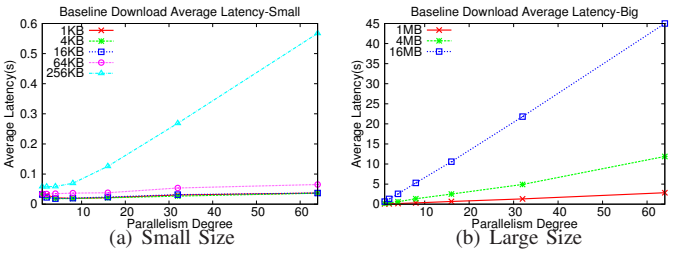
Q2: How does parallelism affect the latency?

In general, proper parallelization does not affect the latency (i.e., end-to-end request completion time) significantly, while over-parallelization leads to a substantial increase of latency. As shown in Figure 2 and Figure 3, this speculation is confirmed by the tendencies of the growing average latencies for both upload and download requests as the parallelism degree increases. For example, for 4KB upload requests, when the parallelism degree increases from 1 to 16, the average latency basically remains the same (about 36ms). When the parallelism degree further increases from 16 to 64, the average latency increases by 43% (from 36.1ms to 51.5ms). For large requests, when the parallelism degree exceeds a threshold, the average



(a) Small Size (b) Large Size

Fig. 2. Average Upload Latency on Baseline



(a) Small Size (b) Large Size

Fig. 3. Average Download Latency on Baseline

latency increases linearly. For example, for 16MB upload requests, when the parallelism degree increases from 4 to 64 (16-fold), the average latency increases from 1.1s to 18.3s (17.3-fold). This implies that for latency-sensitive applications, over-parallelizing large requests should be carefully avoided.

2) The effect of request size: In conventional storage, request size is crucial to organizing large and sequential I/Os and is important in amortizing the disk head seek overhead. A similar effect has also been observed in cloud storage.

Q1: How does request size affect the bandwidth?

As expected, increasing request size (i.e., the size of GET/PUT) can significantly improve bandwidth, but the achieved benefit diminishes as request size exceeds a threshold. As shown in Figure 1(a) and Figure 1(b), the peak bandwidths of large requests and small requests have a significant gap. For example, the peak bandwidth of 4MB upload requests is 23.5 times that of 4KB upload requests (58.9MB/s vs. 2.5MB/s); the peak bandwidth of 4MB download requests is 10.7 times that of 4KB download requests (28.9MB/s vs. 2.7MB/s). There are two reasons for this phenomenon. One reason is that larger I/O requests on client storage generally have higher I/O speeds than small ones. As shown in Table II, the 4MB read speed is 9.2 times that of 4KB (61.6MB/s vs. 6.7MB/s), while the 4MB write speed is 7.4 times that of 4KB (23.2MB/s vs. 3.1MB/s). The other reason is that larger requests have higher efficiency of data transmission via network due to the packet-level parallelism [36].

Similar to parallelization, increasing the request size cannot bring an unlimited bandwidth increase, due to the constraint of other factors. For example, the speed of client storage is limited. Uploaded objects need to be first read from the local device, and downloaded objects need to be written to the local device. As shown in Table II, when the request size grows from 4MB to 16MB, the speed of Magnetic improves slightly, which limits the I/O speed of the client side. Also, the maximum size of the TCP window is limited, though tunable [4], [38]. When the request size exceeds a certain threshold, the benefit brought by increasing the request size diminishes. Our observations have confirmed this speculation. In the scenario of a single thread, as shown in Figure 5, when the request size increases from 16MB to 64MB, the upload bandwidth increases only slightly (20MB/s vs. 21.9MB/s), and the download bandwidth remains the same (24.3MB/s). In addition, other factors, such as the link bandwidth on the route, processing speed on the cloud side, etc., can also limit the achievable bandwidth. All these observations demonstrate that the benefit obtained by increasing request size is significant but is not unlimited.

Q2: How does request size affect the latency?

In general, both larger requests and highly parallelized small requests have longer latencies. For example, as shown in Figure 2 and Figure 3, when the parallelism degree is 1, the average latency of 4MB download requests is 192ms – 5.8 times that of 1MB download requests (33ms). However, when taking parallelism degree into consideration, things become different. For example, the average latency of 4MB download requests at parallelism degree 1 is 192ms, which is 13.8 times lower than the average latency of 1MB download requests at parallelism degree 64 (2.9s). Therefore, *without considering the latency increase caused by over parallelization, it is not safe to say larger requests imply longer latencies*.

As expected, for small requests, even at the same parallelism degree, the latencies do not necessarily increase as the request size increases. Figure 2(a) shows that the average latencies of 1KB and 4KB upload requests are nearly the same. Similarly, in Figure 3(a), we find that the average latencies of 1KB, 4KB, and 16KB download requests are nearly equal. The request latency is mainly composed of three parts: data transmission time via network, client I/O time, and other processing time. For small requests, the data transmission time only accounts for a small portion of the overall latency, while the other two dominant parts remain mostly unchanged, which makes the latencies of small requests similar. Also, since the maximum TCP window is 64KB by default, considering the parallelism of network [36], the transmission time of the data that are smaller than 64KB is supposed to be similar.

3) Parallelism vs. Request size: In prior sections, we find that either increasing the parallelism degree or increasing the request size can effectively improve the bandwidth, but both of them have limitations. Here naturally comes an interesting question: does there exist a combination of parallelism degree and request size to achieve the optimal bandwidth?

Answering this question has a practical value. Consider the following case: if we have a 4MB object to upload, we can choose to upload it by a single thread or split it into four 1MB chunks and upload the them in parallel. Which is faster? Figure 6 shows the performance under different combinations of parallelism degree and request size. Obviously, 256KB$\times$16 has the highest bandwidth (44.2MB/s), which is about 3 times of the lowest (14.5MB/s). This shows that a proper

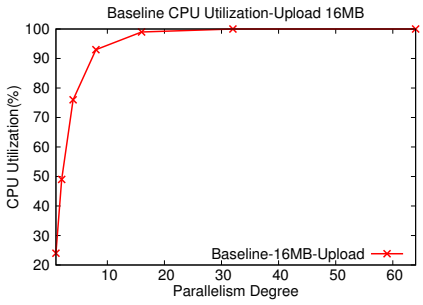
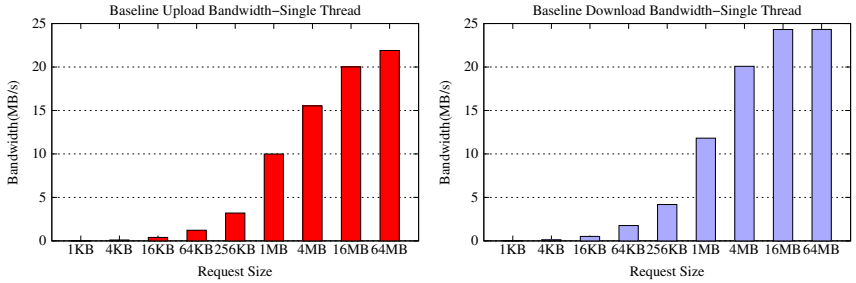


Fig. 4. Baseline CPU Utilization



(a) Upload Bandwidth (b) Download Bandwidth

Fig. 5. Baseline Bandwidth-Single Thread

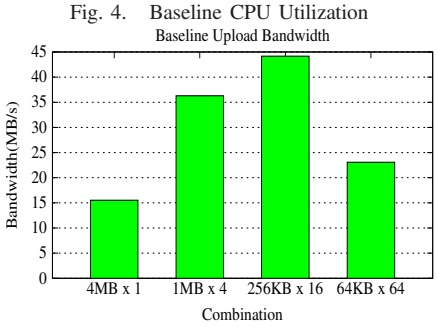
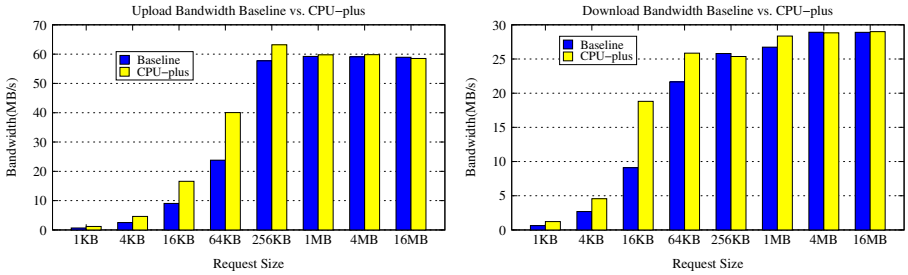


Fig. 6. Request Combinations on Baseline



(a) Upload Bandwidth (b) Download Bandwidth

Fig. 7. Peak Bandwidth (Baseline vs. CPU-plus)

combination exists and can achieve optimal performance. This observation confirms that *appropriately combing request size and parallelism degree can sufficiently improve the bandwidth beyond optimizing only one dimension*.

We also find that, *in some cases, either increasing parallelism degree or increasing request size by the same factor can achieve the same bandwidth improvement*. For example, for upload requests, 1KB×16, 4KB×4, and 16KB×1 have similar bandwidth (0.4MB/s). Here comes another practical question: if we have a set of small files (e.g. 1KB), should we adopt a high parallelism degree (e.g. 16) or bundle the small files to achieve large request size (e.g. 16KB)? From the perspective of improving bandwidth, either high parallelism degree or large request size is feasible. However, from the perspective of the utilization of client resources, we find that a large request size requires less CPU resources. Through `vmstat` in Linux, we find that the CPU utilization of the above three cases are 65%, 15% and 5%, respectively. This indicates that for the combinations that can achieve comparable bandwidth, a larger request size consumes less CPU resources. That is because for a larger request size, fewer threads have to be maintained to achieve the similar bandwidth, which consequently reduces the CPU utilization.

B. Effects of Client Capabilities

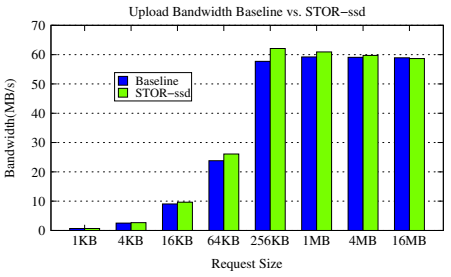
Unlike conventional storage, cloud storage clients are very diverse. In this section, we study different factors affecting the client's capabilities of handling cloud storage I/Os, namely CPU, memory, and storage. We compare the performance of three different clients, including *CPU-plus*, *STOR-ssd* and *MEM-minus*, with the performance of the *Baseline* to reveal the effects of each factor.

1) The effect of the client CPU: In cloud storage I/Os, the client CPU is responsible for both sending/receiving data packets and client I/O. In this section, we try to investigate the effect of client CPU by comparing the performance of Baseline (2 CPUs) and CPU-plus (4 CPUs).

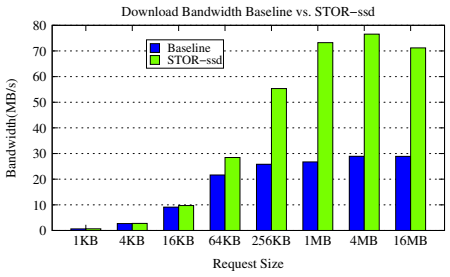
Q1: What is the effect of client CPU on bandwidth?
The client CPU has a strong impact on cloud I/O bandwidth, especially for small requests. Figure 7 shows the peak bandwidth, which is the maximum achievable bandwidth with parallelized requests. We can see that the peak bandwidth of small requests (smaller than 256KB) increases significantly. Interestingly, as shown in Figure 7(b), the peak download bandwidth of 1KB, 4KB and 16KB requests doubles, as the computation capability doubles (2 CPUs vs. 4 CPUs). This vividly demonstrates that small requests are CPU intensive, and as so, small requests receive more benefits from a better CPU.

Large requests, compared to small ones, are relatively less sensitive to CPU resources, as the system bottleneck shifts to some other components. As shown in Figure 7, compared with Baseline, the peak upload and download bandwidth of large requests (256KB to 16MB) increases only slightly. For example, the peak upload bandwidth of 4MB requests increases by 1.4% (59.2MB/s vs. 60MB/s), while the peak download bandwidth of 4MB requests is basically the same (28.9MB/s vs. 28.8MB/s). The system bottleneck may result from the limitation of other factors, such as memory or storage, rather than CPU.

Q2: What is the effect of the client CPU on latency?
In our tests, we find that *the client CPU does not have significant effects on average latency*. For small requests, the data transmission via network dominates the overall latency,



(a) Upload Bandwidth



(b) Download Bandwidth

Fig. 8. Peak Bandwidth (Baseline vs. STOR-ssd)

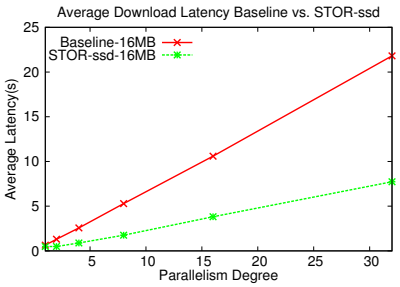


Fig. 9. 16MB Average Download Latency (Baseline vs. STOR-ssd)

while for large requests, the majority of the overall latency is the client I/O time (the I/O waiting time may be significant when client storage becomes the bottleneck) and the cloud response time, In these two cases, a more powerful CPU does not help reduce the latency.

2) The effect of client storage: Client storage plays an important role in data uploading and downloading: For uploading, the data is first read from the client storage; for downloading, the data is written to the client storage. To evaluate the effect of client storage, we set up a comparison client STOR-ssd. The only difference between Baseline and STOR-ssd is storage (Magnetic vs. SSD). Table II shows more details about the two client storage.

Q1: What is the effect of the client storage on bandwidth? We find that *client storage is a critical factor affecting the achievable peak bandwidth*. As shown in Figure 8, on STOR-ssd, the peak download bandwidth increases significantly. For example, the peak download bandwidth of 4MB requests increases by 165% (76.6MB/s vs. 28.9MB/s). On the other hand, we also notice that the upload bandwidth increases slightly. Different from the significant improvement of download bandwidth, for example, the peak upload bandwidth of 4MB requests increases only by 2% (60.3MB/s vs. 59.2MB/s). The reason why STOR-ssd improves the upload bandwidth only slightly is that the Magnetic in our experiments can achieve a similar peak read speed as SSD with a sufficiently large request size and parallelism degree. In contrast, the download bandwidth is limited by the relatively slow speed of Magnetic on the client. We have also tested with a ramdisk on Baseline. The bandwidths can be further improved but to a limited extent (77.2MB/s for uploading and 80.3MB/s for downloading).

Q2: What is the effect of the client storage on latency? *Similar to bandwidth, we did not observe significant effects of client storage to small requests and large upload requests*. For small requests, client I/O is the minority of the overall latency. In this case, client storage is not a critical factor. For large upload requests, since Magnetic and SSD have similar read speed, the latency is comparable; however, for large download requests, STOR-ssd can substantially reduce the latency because STOR-ssd have significantly advantageous write speed. For example, as shown in Figure 9, when the parallelism degree is 1, STOR-ssd can reduce the latency by 24% (0.49s vs. 0.64s); when parallelism degree is 32, the latency can be reduced by 65% (7.7s vs. 21.8s).

3) The effect of client memory: Memory in the clients has two functions. First, memory is responsible for offering running space for parallel requests. Second, memory acts as a buffer for uploading and downloading. In this section, we shrink the memory of Baseline to investigate the performance differences. The only configuration difference between MEM-minus and Baseline is that Baseline has 7.5GB memory while MEM-minus has only 3.5GB.

	1MB	4MB	16MB
Baseline	59.2MB/s	59.1MB/s	58.9MB/s
MEM-minus	58.9MB/s	58.7MB/s	58.7MB/s

TABLE IV
PEAK UPLOAD BANDWIDTH (BASELINE VS. MEM-MINUS)

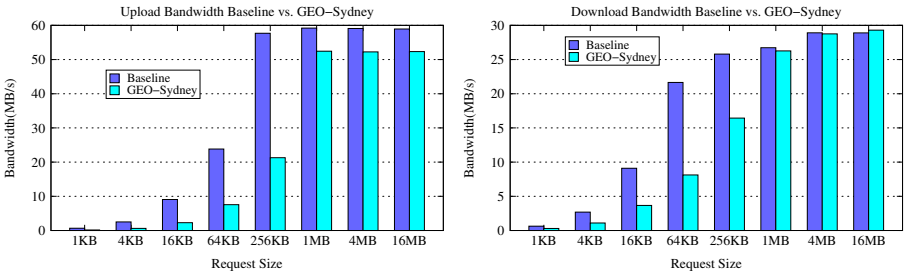
	1MB	4MB	16MB
Baseline	26.7MB/s	28.9MB/s	28.9MB/s
MEM-minus	23.7MB/s	23.8MB/s	20.8MB/s

TABLE V
PEAK DOWNLOAD BANDWIDTH (BASELINE VS. MEM-MINUS)

Since small requests are not memory intensive, the effect of memory is trivial. We only present the bandwidths of large requests. The peak upload bandwidth is basically the same (see Table IV) while the download bandwidth dropped heavily (see Table V). For example, on MEM-minus, the bandwidth of 16MB download is 20.81MB/s, which is 28.0% lower than that on Baseline (28.90MB/s). That is because the write speed of the Magnetic is much lower than read speed and thus more sensitive to the memory space. Therefore, large download requests, especially those involving intensive writes on the client, suffer more from limited memory.

C. Effects of Geographical Distance

Unlike conventional storage, for cloud storage, the geographical distance between the client and the cloud determines the Round-Trip Time (RTT), which accounts for a significant part of the observed I/O latency. The RTT between the Baseline client and the cloud is 0.28ms, as both are in the same Oregon data center. In contrast, the RTT between the GEO-Sydney client and the cloud in Oregon is about 628 times longer (176ms). This section discusses the effects of geographical distance.



(a) Upload Bandwidth (b) Download Bandwidth

Fig. 10. Peak Bandwidth (Baseline vs. GEO-Sydney)

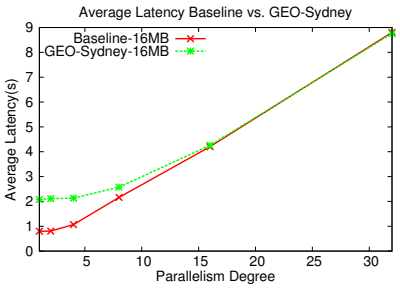


Fig. 11. 16MB Average Upload Latency (Baseline vs. GEO-Sydney)

Q1: What is the effect of geo-distance to bandwidth? *The effect of geographical distance to the achievable peak bandwidth is weaker than expected*. As shown in Figure 10, the peak upload bandwidth of GEO-Sydney is close to that of Baseline. For example, the peak upload bandwidth of 4MB requests of GEO-Sydney is only 10% lower than that of Baseline (53.3MB/s vs. 59.2MB/s) while the peak download bandwidths of 4MB download requests are basically the same (29.3MB/s vs. 28.9MB/s). This means that RTT is not a critical factor affecting the peak bandwidth, which is mostly due to the Bandwidth-Delay Product (BDP) of the network and is also consistent with the conclusion obtained by Burgen et al. [16] that the perceived bandwidth from the client is largely determined by the client's network capabilities and the network performance between the client and the cloud.

At the same time, it is also noticeable that *the achievable peak bandwidth of small requests (smaller than 1MB) is much lower with long geo-distance*. That is because a long RTT needs a high parallelism degree to saturate the pipeline of parallel requests. However, as analyzed in Section IV-B1, small requests with high parallelism are more CPU intensive; therefore, the CPU capability will be a critical bottleneck to sufficiently saturating the pipeline.

Q2: What is the effect of geo-distance to latency? As expected, we also find that *the geo-distance would significantly increase the latency, and its impact to latency makes the client less sensitive to the negative effects caused by over-parallelization to latency*. As shown in Figure 11, when the parallelism degree is 1, the average latency of 16MB upload requests on GEO-Sydney is 2.1s, which is about 2.6 times of the counterpart on Baseline (0.8s); as the parallelism degree increases, the average latencies gradually get closer; when the parallelism degree reaches 16, the average latencies are comparable (4.3s vs. 4.2s). If we compare the two, GEO-Sydney shows a flatter curve than Baseline, because a long RTT needs a high parallelism degree to saturate the pipeline, so the negative effect of over-parallelization appears later.

V. Case Study: Caching and Prefetching

In cloud storage, client-side caching and prefetching are two basic schemes for enhancing the user experience. In this section, we present a case study to show that cloud storage I/O performance could be affected by optimizing caching and prefetching. In specific, we will discuss two key techniques, *chunking* and *parallelization*.

To evaluate the effects of chunking and parallelized prefetching for cloud-based file systems, we build an emulator to implement the basic read/write operations of a typical cloud-based file system with the support of disk caching on the client. To drive this experiment, we use an object-based trace by converting a segment of an NFS trace, which is a mix of email and research workload collected at Harvard University [30]. The size of the workload in our experiments is 4.8 GB, and the average file size is 12.9 MB. For our experiments, we use Amazon S3 (in Oregon) as the cloud storage provider, and a workstation on our campus (in Louisiana) as the client. The client is equipped with a 2-core 1.2 GHZ CPU, 8GB memory, a 450GB disk drive, and installed with Ubuntu 12.04.5 LTS and Ext4 file system.

A. Proper Chunk Size for Caching

Chunking is an important technique used in cloud storage. In S3Backer, for example, the space of the cloud-based block driver is formatted with a fixed block size that can be defined by the user [12]. The choice on chunk sizes can affect caching performance: the smaller the chunk is, the less a cache miss cost would be, but the more cloud I/Os could be generated.

Although it is difficult to accurately determine the optimal chunk size, our findings about the effect of chunk size to the performance of cloud storage can guide us to roughly choose a proper, if not optimal, chunk size. We can identify a relatively small chunk size for reaching an approximately maximum bandwidth by making a reasonable tradeoff between the cache hit ratio and cache miss penalty. Figure 12 shows that, when the chunk size exceeds 4MB, the download bandwidth reaches its peak. Based on this, we speculate that the proper chunk size is possibly around 4MB. This is for two reasons. First, further increasing the chunk size over 4MB (e.g., 8MB or 16MB) cannot deliver a higher bandwidth. For example, on a cache miss of 8MB data, downloading one 8MB chunk takes an almost equal amount of time as downloading two 4MB chunks, while using 8MB chunks increases the risk of downloading irrelevant data. Second, if the chosen chunk size is excessively smaller than 4MB (e.g., 64KB or 1MB), the cache may suffer from a high cache miss ratio and cause too many I/Os.

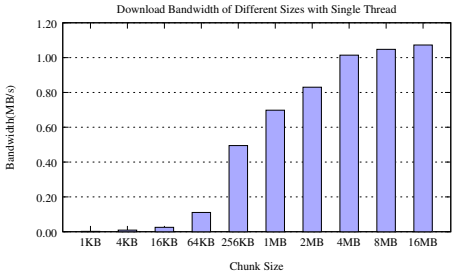


Fig. 12. Download Bandwidth of Different Sizes with Single Thread

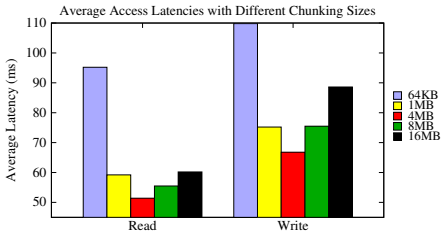


Fig. 13. Avg. Access Latencies

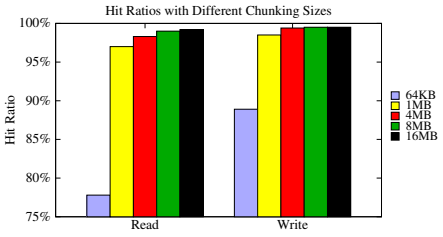


Fig. 14. Hit Ratio

To verify this speculation, we adopt the standard LRU algorithm with asynchronous writeback (for the purpose of generality). Every 30 seconds, we flush dirty data back to the cloud. The cache size is set as 200MB disk space. Besides a 4MB chunk size, for a comparison, we choose two smaller chunk sizes, 64KB and 1MB, and two larger chunk sizes, 8MB and 16MB, to study the effect of the chunk sizes.

The average access latencies with different chunk sizes are shown as Figure 13. It clearly shows that the lowest average read/write latencies are achieved at 4MB, which confirms our speculation. When the chunk size increases from 64KB to 4MB, the average read latency decreases by 47.3% (from 95.2ms to 50.2ms), and the average write latency decreases by 40.4% (from 109.9ms to 65.5ms). This benefit is due to the increase of cache hit ratio: The read hit ratio increases from 77.8% to 98.4%, and the write hit ratio increases from 88.9% to 99.4% (see Figure 14). This is mostly because using a relatively large chunk size allows to pre-load the useful data and consequently improves the cache hit ratio and the overall performance. However, when the chunk size exceeds a certain threshold, further increasing chunk size may cause undesirable negative effects. Figure 14 shows that the cache hit ratios increase slightly with a large chunk size. The increased cache miss penalty with a large chunk size is responsible for the slowdown. Specifically, it takes 4s to load a 4MB chunk, while it needs 14.2s for 16MB. Consequently, the average access latencies increase.

The analysis above has shown how to determine the proper chunk size for a certain client. Specifically, 4MB is the proper chunk size on our client for the testing workload. For the workloads with weak spatial locality, the proper chunk size should be correspondingly smaller. In general, an excessively large chunk size is not desirable, as it increases the risk of unnecessary overhead with no extra benefit.

B. Proper Parallelization for Prefetching

Prefetching is another widely used technique in cloud storage. Since objects can be downloaded (prefetched) in parallel, a proper parallelism degree is important to the performance, while over-parallelization may raise the risk of mis-prefetching and also waste resources.

In order to determine a proper parallelism degree for a certain chunk size, it is critical to ensure that on-demand fetching would not be significantly affected by prefetching. To find the proper parallelism degree that will not significantly increase the average fetching latencies, an exhaustive search on the client is feasible but inefficient. Based on our findings, in fact, we can greatly simplify the process of identifying a proper parallelism degree. To show how to achieve this, we take the chunk sizes 64KB, 1MB and 4MB as examples. We may first choose a 4MB chunk with parallelism degree 1 and then gradually increase the parallelism degree step by step (i.e., 2, 4, 8) for testing. For smaller chunk sizes, we only need to test from a larger parallelism degree, since small chunks are more parallelism friendly and it is unlikely to achieve higher performance at a low parallelism degree as large chunks. Figure 15 gives such an example: 4 parallel jobs for 4MB, 8 parallel jobs for 1MB, and 16 parallel jobs for 64KB are the best choices.

To illustrate the actual effect of parallelization to prefetching, we implement an adaptive prefetching algorithm in our emulator. We adopt the history-based prefetching window to determine the prefetching granularity, which is similar to the file prefetching scheme used in Linux kernel. A prefetching window is maintained to estimate the best prefetching degree. The initial window size is 0 and is enlarged based on the detected sequentiality of observed accesses. Assuming chunk `n` of an object is requested, if chunk `n-i`, chunk `n-i+1`, $\cdots$, chunk `n-1` ($1 \leq i \leq n$) are detected to be sequentially accessed, the size of the prefetching window grows to 2^{i-1}. We set the maximum prefetching window size (i.e., parallelism degree of prefetching) for all chunk sizes (i.e., 64KB, 1MB, 4MB) to 8.

The performance comparison of no-prefetching and prefetching are shown in Figure 16 and Figure 17. We can see that, with prefetching, the optimal chunk size is 1MB. Obviously, small chunk size benefits more from the prefetching (as we see in the prior sections, small objects benefit more from parallelism), and the relative benefits decrease as the chunk size increases (see Table VI).

Chunk Size	Read Lat. Red.	Write Lat. Red.
64KB	70.4%	31.1%
1MB	56.9%	24.6%
4MB	22.6%	-1.2%

TABLE VI
AVERAGE LATENCIES REDUCTION CAUSED BY PREFETCHING

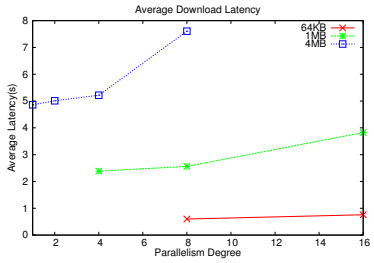


Fig. 15. Avg. Download Latency

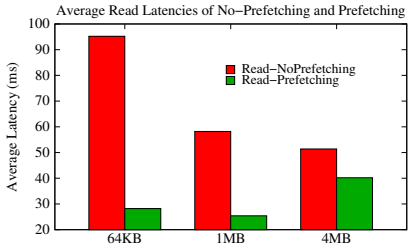


Fig. 16. Avg. Read Latency Comparison

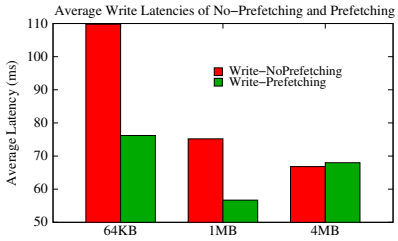


Fig. 17. Avg. Write Latency Comparison

Surprisingly, with prefetching, the average write latency of 4MB increases by 1.2%. This means that the prefetching granularity in our experiment is so aggressive that the negative effects of prefetching overweight the benefits. The negative effects may result from two factors. First, a lot of unnecessary data is prefetched so that the cache efficiency is reduced, leading to a lower cache hit ratio. Second, the competition of parallel prefetching threads may increase the average downloading latency (i.e., the average penalty of cache miss). Specifically for the case of the average write latency of 4MB, the performance degradation is mainly caused by the second factor since the cache hit ratio remains high (close to 98.7%). As a rule of thumb, we should set a small prefetching degree for large chunk sizes (e.g., 4MB) to avoid the intensive competitions of the parallelized downloading threads. For example, we can limit the growing speed of the prefetching window, or cap the maximum prefetching window size. On the contrary, the prefetching granularity of small chunk sizes (e.g., 64KB) can be more aggressive. This also confirms our speculation about the proper parallelism degrees for different chunk sizes (i.e., 16 for 64KB chunks, 4 for 4MB chunks).

In summary, our case studies on prefetching and caching further show that the real-world implementation of client-side management should carefully consider the factors that we have studied in the prior sections, particularly parallelism degree and request size. Other issues, such as client capabilities and geo-distances would also inevitably further complicate the design consideration.

VI. System Implications

With these experimental observations, we are now in a position to present several important system implications. This section also provides an executive summary of our answers to the questions we asked earlier.

Appropriately combining request size and parallelism degree can maximize the achievable performance. This is sometimes a tradeoff between the two factors. By combining the chunking/bundling methods with parallelizing I/Os, the client can enhance bandwidth in two different ways: we can increase the parallelism degree for small requests or increase the request size at low parallelism degree. Both can achieve comparable bandwidth, but interestingly, we also find that compared to increasing parallelism degree, increasing the request size can achieve another side benefit: reduced CPU utilization. This means that for some weak-CPU platforms, such as mobile systems, it is more favorable to create large requests with a low parallelism degree. On the other hand, we should also consider several related side effects of bundling/batching small requests. For instance, if part of a bundled/batched request failed during the transmission, the whole request would have to be re-transmitted. Also, it is difficult to pack a bunch of small requests to different buckets or data centers together. In contrast, parallelizing small requests is easier and more flexible. Therefore, there is no clear winner between the two possible optimization methods (i.e., creating large requests and parallelizing small requests). An optimal way may vary from client to client and from service to service, but we can still use some general principles to guide us in making a decision. For example, as we find that small requests demand a high parallelism degree, if we know the proper parallelism degree on a certain client for 1MB is 8, and 4 for 4MB (we can obtain these combinations via simple measurement or experience), it is reasonable to infer that the proper parallelism degree for 2MB should be between 4 and 8. To avoid the worst situation, a rule of thumb is to make a conservative choice, if uncertain.

The client's capability has a strong impact to the perceived cloud storage I/O performance. CPU, memory, and storage are the three most critical components determining a client's capability. Among the three, CPU plays the most important role in parallelizing small requests, while memory and storage are critical to large requests, especially large download requests. A direct implication is that for optimizing the cloud storage performance, we must also distinguish the capabilities of clients, and one policy will not be effective in all clients. Due to the cross-platform advantage, many personal cloud storage applications can run on multiple platforms (from PCs to Smartphones). Such distinction among clients will inevitably affect our optimization policies. For example, for a mobile client with a weak CPU, we should avoid segmenting objects into excessively small chunks, since it is unable to handle a large number of parallel I/Os, although this is not a constraint for a PC client. Given the diversity of cloud storage clients, we believe that a single optimization policy is unlikely to succeed across all clients.

Geographical distance between the client and the cloud plays an important role in cloud storage I/Os. For cloud storage, the geographical distance determines the RTT. We find that a long RTT has distinct effects to bandwidth and latency. In particular, with a long RTT, we still can achieve a similar peak bandwidth as the case of a short RTT, but

the cloud I/O latency is significantly higher. The implications are two-fold. First, to tackle the long latency issues, it is a must-have to use effective caching and prefetching for latency-sensitive applications. Second, for the clients far from the cloud, we should proactively adopt large request sizes and high parallelism degrees to fully saturate the pipeline and exploit available bandwidth as much as we can. In other words, by sufficiently exploiting the I/O characteristics of cloud storage, if bandwidth is the main requirement (e.g., video streaming), choosing a relatively distant data center of the cloud storage is a viable option and a high bandwidth is still achievable with appropriate client-side optimizations.

In essence, cloud storage represents a drastically different storage model for its diverse clients, network-based I/O connection, and massively parallelized storage structure. Our observations and analysis strongly indicate that fully exploiting the potential of cloud storage demands careful consideration of various factors.

VII. Related Work

Most prior studies focus on addressing various issues of cloud storage, including performance, reliability, and security (e.g., [14], [20], [21], [22], [24], [31], [32], [33], [35], [41], [47], [48]). Some other work studies the design of cloud-based file systems to better integrate cloud storage into current storage systems (e.g., [18], [26], [45]). Our work is orthogonal to these studies and focuses on understanding the behaviors of cloud storage from the client's perspective.

Our work is related to several prior measurement works on cloud storage. Li et al. compared the performance of four major cloud providers: Amazon AWS, Google AppEngine and Rackspace CloudServers [40]. Ou et al. compared a file system client of cloud storage based on CloudFuse with two other IP-based storage, NFS and iSCSI [44]. Bermudez et al. presented a characterization of Amazon's Web Services (AWS) [17]. Copper et al. benchmarked cloud storage systems with YCSB [23]. Meng et al. presented a benchmarking work on cloud-based data management systems to evaluate the effects of different implementation on cloud storage [43]. This work treats cloud storage as a blackbox and focuses on characterizing its I/O behaviors from client's perspective. Several other measurement works focus on the client applications (e.g., [27], [28], [29], [34], [42], [46]). Unlike our study, these prior studies focus on measuring the performance of commercial personal cloud storage clients, such as Dropbox, Wuala, Google Drive, etc. In contrast, we treat the cloud storage as a blackbox and focus on revealing the key factors affecting the interactions of the client and the cloud from the perspective of the client side rather than benchmarking specific cloud storage clients. In fact, we purposely avoid using any specific client tools so that we can minimize the potential interference. Besides object-based storage, some service providers also provide block- and file-level storage services, such as Amazon Elastic Block Store (EBS) [1] and Elastic File System (EFS) [2]. These services are more similar to conventional IP-based storage, such as iSCSI and NFS. In this work, we focus on the HTTP-based object storage, represented by Amazon S3, and we have observed several interesting and unique I/O behaviors.

VIII. Conclusions

We present a comprehensive measurement and quantitative analysis on cloud storage to investigate the critical factors affecting the perceived performance of cloud storage from a client-side perspective. Our experiments show several important characteristics of cloud storage, such as benefits of parallelizing cloud storage I/Os, the latency impact of over-parallelization, the effect of client's capability to achievable performance, and more. Based on these findings, we also present a case study on chunking and parallelization, the two key techniques used for optimizing cloud storage performance on the client. We hope our observations and the associated system implications can provide guidance to help practitioners and application designers exploit various optimization opportunities for cloud storage clients.

Acknowledgment

The authors thank anonymous reviewers for their constructive comments to improve this paper. This work was supported in part by Louisiana Board of Regents under grants LEQSF(2014-17)-RD-A-01 and LEQSF-EPS(2015)-PFUND-391, National Science Foundation under grant CCF-1453705, and generous support from Intel Corporation.

References

[1] Amazon EBS. https://aws.amazon.com/ebs/.
[2] Amazon EFS. https://aws.amazon.com/efs/.
[3] Amazon S3. https://aws.amazon.com/s3/.
[4] Amazon S3 TCP Window Scaling. http://docs.aws.amazon.com/AmazonS3/latest/dev/TCPWindowScaling.html.
[5] Cloud Storage Market by Solutions. http://www.marketsandmarkets.com/Market-Reports/cloud-storage-market-902.html.
[6] Dropbox. https://www.dropbox.com/.
[7] Dropbox Uses Amazon S3 Services for Storage! https://storageservers.wordpress.com/2013/10/25/dropbox-uses-amazon-s3-services-for-storage/.
[8] Google Drive. https://www.google.com/drive/.
[9] OneDrive. https://onedrive.live.com/.
[10] OpenStack Swift. http://www.openstack.org/.
[11] S3 API Reference. https://boto.readthedocs.org/en/latest/ref/s3.html.
[12] S3Backer. https://code.google.com/p/s3backer/.
[13] S3FS. https://code.google.com/p/s3fs/.
[14] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC 2010)*, Indianapolis, IN, June 10-11 2010.
[15] Amazon. Amazon S3 Object Size Limit Now 5 TB. https://aws.amazon.com/blogs/aws/amazon-s3-object-size-limit/.
[16] A. Bergen, Y. Coady, and R. McGeer. Client Bandwidth: The Forgotten Metric of Online Storage Providers. In *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim 2011)*, Victoria, BC, Canada, August 23-26 2011.
[17] I. Bermudez, S. Traverso, M. Mellia, and M. Munafo. Exploring the Cloud from Passive Measurement: the Amazon AWS Case. In *Proceedings of The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, Turin, Italy, April 14-19 2013.
[18] Bessani, Alysson, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, , and P. Verissimo. SCFS: A Shared Cloud-backed File System. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC 2014)*, Philadelphia, PA, June 19-20 2014.

[19] E. Bocchi, I. Drago, and M. Mellia. Personal Cloud Storage Benchmarks and Comparison. *IEEE Transactions on Cloud Computing*, 99:1–14, 2015.

[20] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC 2010)*, Indianapolis, Indiana, June 10-11 2010.

[21] C. Brad, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, and Y. X. et al. Windows Azure Storage: a Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 119–132, New York, NY, October 23-26 2011.

[22] F. Chen, M. P. Mesnier, and S. Hahn. Client-aware Cloud Storage. In *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST'14)*, Santa Clara, CA, June 2-6 2014.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and V. colleagues. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (ACM SoCC 2010)*, Indianapolis, IN, June 10–11 2010. ACM Press.

[24] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao. QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom 2015)*, pages 582–603, Sept 7-11 2015.

[25] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007. The USENIX Association.

[26] Y. Dong, J. Peng, D. Wang, H. Zhu, F. Wang, S. C. Chan, and M. P. Mesnier. RFS - A Network File System for Mobile Devices and the Cloud. In *SIGOPS Operating System Review*, volume 45, pages 101–111, February 2011.

[27] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking Personal Cloud Storage. In *Proceedings of the 2013 ACM conference on Internet measurement conference (IMC 2013)*, Barcelona, Spain, October 23-25 2013.

[28] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Modeling the Dropbox Client Behavior. In *Proceedings of the 2014 IEEE International Conference on Communicationsconference (ICC 2014)*, Sydney, NSW, June 10-14 2014.

[29] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 ACM conference on Internet measurement conference (IMC 2012)*, New York, NY, November 14-16 2012.

[30] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the 2th USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. The USENIX Association.

[31] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, Oct 4-6 2010.

[32] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *Proceedings of the 2nd ACM symposium on Cloud computing (SoCC 2011)*, Cascais, Portugal, October 27.28 2011.

[33] B. D. Higgins, J. Flinn, T. Giuli, B. Noble, C. Peplin, and D. Watson. Informed Mobile Prefetching. In *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys 2012)*, pages 155–158, June 25-29 2012.

[34] W. Hu, T. Yang, and J. N. Matthews. The Good, the Bad and the Ugly of Consumer Cloud Storage. In *ACM SIGOPS Operating Systems Review*, volume 44:3, July 2010.

[35] Y. Hu, H. C. H. Chen, P. P. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 2012)*, San Jose, CA, February 14-17 2012.

[36] L. Huan. A Trace Driven Study of Packet Level Parallelism. *IEEE Communications (ICC 2002)*, 4(1):2191–2195, April 28-May 2 2002.

[37] IHS. Subscriptions to Cloud Storage Services to Reach Half-Billion Level This Year. https://technology.ihs.com/410084/subscriptions-to-cloud-storage-services-to-reach-half-billion-level-this-year.

[38] V. Jacobson, R. Braden, D. Borman, M. Satyanarayanan, J. Kistler, L. Mummert, and M. Ebling. RFC 1323: TCP Extensions for High Performance, 1992.

[39] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. 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 2005)*, San Francisco, CA, December 12-16 2005. The USENIX Association.

[40] A. Li, X. Yang, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (ACM SIGMOD 2010)*, Melbourne, Australia, November 1–3 2010. ACM Press.

[41] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. Efficient Batched Synchronization in Dropbox-like Cloud Storage Services. In *Middleware 2013*, pages 307–327. Springer, 2013.

[42] T. Mager, E. Biersack, and P. Michiardi. A Measurement Study of the Wuala On-line Storage Service. In *Proceedings of the IEEE 12th International Conference on Peer-to-Peer Computing (P2P 2012)*, Sophia Antipolis, France, Sept 3-5 2012.

[43] X. Meng, Y. Chen, J. Xu, and J. Lu. Benchmarking Cloud-based Data Management Systems. In *Proceedings of the 2nd international workshop on Cloud data management (ACM CloudDB 2010)*, Toronto, ON, October 26–30 2010. ACM Press.

[44] Z. Ou, Z.-H. Hwang, A. Ylä-Jääski, F. Chen, and R. Wang. Is Cloud Storage Ready? A Comprehensive Study of IP-based Storage Systems. In *Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'15)*, Limassol, Cyprus, December 7–10 2015.

[45] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 2012)*, San Jose, CA, February 14-17 2012.

[46] H. Wang, R. Shea, F. Wang, and J. Liu. On the Impact of Virtualization on Dropbox-like Cloud File Storage/Synchronization Services. In *Proceedings of International Workshop on Quality of Service (IWQoS 2012)*, Coimbra, Portugal, June 4-5 2012.

[47] R. Zhang, R. Routray, D. Eyers, D. Chambliss, P. Sarkar, D. Willcocks, and P. Pietzuch. IO Tetris: Deep Storage Consolidation for the Cloud via Fine-grained Workload Analysis. In *Proceedings of the 4th International IEEE Conference on Cloud Computing (IEEE CLOUD 2011)*, Washington D.C., July 2011.

[48] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proceedings of the 12th USENIX conference on File and Storage Technologies (FAST 2014)*, pages 119–132, San Jose, CA, February 14-17 2014.

Understanding Storage I/O Behaviors of Mobile Applications

Jace Courville
Louisiana State University
jcourv@csc.lsu.edu

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

***Abstract*—In the past few years, mobile devices quickly gained high popularity in our daily life. Designed for ultra-mobility, these small yet powerful devices are fundamentally distinct from traditional computer systems (e.g., PCs and servers) – from the internal hardware architecture and software stack, to application behaviors. Storage, the slowest component in the I/O stack, plays an important role in mobile systems and can greatly affect user experience. In this paper, we present a set of comprehensive experimental studies on mobile storage and attempt to gain insight on the unique behaviors of mobile applications and characterize the performance properties of underlying mobile storage. In our experiments, we carefully selected 13 representative mobile workloads from 5 different categories. Our studies reveal several unexpected observations on mobile storage. Based on these findings, we further discuss the associated implications to mobile systems and application designers. We hope this work can inspire system architects, application designers, and practitioners to pay specific attention to the high-latency I/O operations, rather than completely relying on the default APIs. We also suggest a further look to new opportunities, such as adopting a faster medium in the mobile system architecture, for future research.**

***Index Terms*—Mobile systems; Storage performance; Flash memory; Measurement.**

I. Introduction

Within the last decade, we have experienced the rise of modern mobile devices. Apple recently sold its 500 millionth iPhone [2], and sales of Android devices exceeded one billion units in 2014 [3]. While mobile devices provide high levels of convenience and enable ubiquitous computing to typical users, these small devices, compared to their traditional computer system counterparts (e.g., PCs and servers), carry a set of fundamentally distinct characteristics, from the hardware architecture and software system stack, to application behaviors. These distinctions demand a careful reconsideration of optimizations for system and application design in various aspects.

Storage, the slowest component in the I/O stack, plays a critical role in overall system performance. Interestingly, storage in mobile devices differentiates itself from conventional platforms in several unique properties. (1) *Mobile devices use a flash-based storage medium*. Unlike PC or server systems, which often adopt large-capacity magnetic disks as storage, mobile devices are almost all reliant on NAND flash memory. As a type of semiconductor device, NAND flash memory delivers high-speed read accesses but is highly sensitive to random writes and may suffer from low performance when such writes are encountered [5]. (2) *Mobile devices require latency-oriented optimization*. For mobile devices, it is of high priority to ensure an optimal user experience. This user experience may be severely impacted by storage performance, or more precisely: *latencies*. A high I/O latency may render the device unresponsive. Even worse, such slight slowness may be easily noticed by users and negatively affect user experience. In contrast, *throughput*, another performance metric widely used in traditional storage benchmarking, is not as important in mobile systems. (3) *Mobile devices have both a distinct software stack and distinct application behaviors*. Mobile device applications (i.e., mobile apps) typically run in a protected environment, or sandbox. For example, Android apps normally run in a Java virtual machine. Privileged operations are encapsulated in a small set of strictly defined API interfaces. As a result, popular APIs such as the SQLite library are heavily used in nearly all mobile apps. Such a development practice results in certain patterns which can be commonly found across various mobile apps. On conventional PC and server systems, this is unlikely to happen. In short, because of its unique physical nature, optimization goal, and software stack, mobile storage, compared to traditional computer systems, inevitably exhibits radically different properties. A more important implication to us is that our prior wisdom about storage and the understanding about its influence to system and application performance may not continue to be applicable to mobile devices. Therefore, a demand of a detailed and thorough study to properly understand the critical issues of mobile storage affecting user experience is necessary. In particular, we desire to answer the following important questions:

- *Do there exist any consistent trends in application performance and behaviors over several different categories of applications?* The presence of those trends may suggest that such behavior is not application specific and may exist across an even more broad spectrum of applications.
- *How much of an impact, if any, do storage I/Os contribute to application performance?* Given the diversity of mobile apps, not all applications may be affected by storage I/Os in the same way. It is necessary to understand how much latency applications experience as a result of these I/Os to ensure that each application can perform as efficiently as possible.
- *Which type of storage I/Os contribute most to latency, and what is the root cause behind such impact?* Only by identifying the critical storage I/Os which affect the performance the most, can we effectively identify the most appropriate solutions to address these issues.

978-1-4673-9055-2/16/$31.00 © 2016 IEEE

- *Does there exist any room for a system level solution to resolve storage I/O latency?* By answering these fundamental questions, we can identify a potential room for optimization of overall mobile app performance through minimizing the total amount of latency contributed by storage I/Os.

In this paper, we present a comprehensive experimental study to explore several important aspects of mobile storage. We carefully select a set of 13 representative mobile workloads from 5 different categories: ranging from games, multimedia, productivity, network, and device utilities. We run these workloads on a Google Nexus 5 mobile phone with a recompiled Linux kernel. By using `blktrace` and `blkparse` tools, we trace the storage I/O activities for each application and perform an offline analysis on the collected experimental data. It is worth noting that our main purpose is not to benchmark these mobile apps or the device itself. Instead, we attempt to observe the I/O activities from the perspective of the lowest storage layer, characterize and understand the storage I/O behaviors of typical mobile apps, and identify the critical issues of mobile storage with a goal of finding the key aspects for potential optimizations in the future. Based on these observations and analysis, we further discuss important implications to mobile system and application designs. We hope this work can inspire the research community, especially mobile OS architects and mobile app designers, to carefully consider the use of many storage I/O related operations and enhance user experience successfully.

This paper is organized as follows. Section II introduces the background about mobile systems and the storage I/O stack. Section III gives the experimental methodology. Section IV and V discuss the experimental results and their system implications. Related work is presented in Section VI, and the last section concludes this paper.

II. Background

Android is a mobile OS developed by Google for mobile devices. Initially released in 2008, Android currently powers a majority of the mobile devices on the market. Our experiments were performed with the stable Android version 5, "Lollipop". In this section, we give a brief overview of the Android architecture, especially the I/O stack. Figure 1 illustrates the basic architecture of Android OS.

At the top level of the Android architecture is the *application layer*. Unlike traditional desktop systems, an application in the Android OS can be considered a different "user". Each mobile app is assigned a user ID and has respective permissions unique to this ID. These applications are written in Java and run in their own virtual machine, meaning that each application runs independently of another – a quality that is not seen on traditional systems [4]. The *framework layer* consists of the various managers which these applications interact with. For example, an application which uses location based services (e.g., Google Maps) interacts with the location manager to get the geographic location of the device.

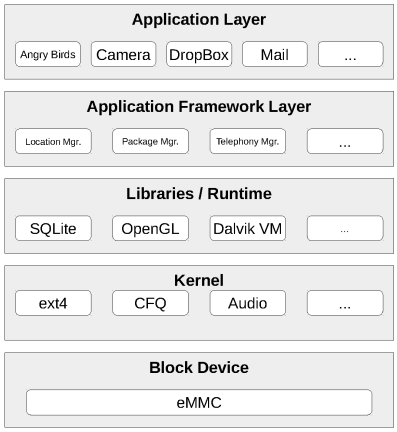


Fig. 1. The Android Architecture

The *library/runtime layer* is responsible for interacting with the OS kernel. These libraries allow application developers to quickly access core system services in a protected manner. For example, SQLite (a journaling based light-weight database) enables application developers to keep data (e.g., user settings) persistent in the form of key/value pairs. Another key component is the Android runtime. Since Android apps are written in Java, the virtual machine runtime, Dalvik (OS versions 4.4 and earlier) or ART, is responsible for application isolation and memory management. The bottom layer is the Linux-based *OS kernel*. The Android OS kernel is a variation of the open-source Linux kernel and contains a set of low-level drivers to control hardware devices, such as the eMMC device and display. The primary file system in Android is the Ext4 file system, which replaced the older YAFFS2 [1]. The CFQ I/O scheduler is responsible for dispatching the I/O requests to the actual eMMC flash block device [4], which completes the whole I/O path.

III. Experimental Methodology

Our experiments were conducted on a Google Nexus 5 device – an Android-powered smartphone. This device is equipped with 32GB of internal eMMC NAND flash based storage and runs an Android Open Source Project (AOSP) version of Android 5. We recompiled the Linux kernel 3.4.0 and ported it to the device to support the capability of block level I/O tracing. We use `blktrace` in Linux to collect the I/O traces of various workloads. The blktrace tool monitors the time stamped events in the I/O path, such as a dispatch of an I/O request and a completion of an I/O request. The traces are first reserved in ramfs and later dumped to persistent storage. We use `blkparse` and our post-processing scripts to process the I/O traces and analyze the traces offline.

For our experiments, we carefully selected 10 mobile apps from both first party and third party sources in order to obtain a true representation of an environment that a typical user may have. When selecting these applications, we chose to prioritize a more real-world set of workloads over choosing workloads that would generate an unrealistically high volume

Workload	Application Type	Read/Write Ratio	Description
Angry Birds	Game	2.03/1	Loading the Angry Birds application
App Removal	Device Utilities	1.35/1	Uninstalling an application from the device
Batch Uninstall	Device Utilities	1/2.79	Using ADB to uninstall several applications at once
Burst Mode Camera	Multimedia	1/204.1	Uses Burst Mode Camera to take a sequence of 100 pictures as a burst
Camera	Multimedia	1/9.12	Uses default camera to take three pictures in quick sequence
Contacts	Productivity	1/2.07	Adding a new contact to the device
Dropbox Sync	Network	1/5.63	Linking an existing Dropbox account to the device and performing an initial sync
E-mail Sync	Network	1/4.25	Linking an existing e-mail account to the device and performing an initial sync
Web Request	Network	1/1.47	Loading the Facebook web site
Route Plotting	Network	1/2.54	Plotting a GPS route using the Google Maps application
MP3 Streaming	Network	1/41.8	Streaming 15 seconds of audio using the Spotify application
Video Playback	Multimedia	1.81/1	Playing back a 5 second recorded video
Video Recording	Multimedia	1/4.25	Recording a 5 second video using the default camera application

TABLE I
WORKLOAD DESCRIPTIONS

of I/Os, at the trade off of code availability in several of the closed-source apps. As one of our goals is to determine the true impact of storage I/Os on users, we felt that data from these apps would better indicate this impact. Using these mobile apps, 13 different use cases from 5 categories (games, multimedia, productivity, network, and device functions) were then designed to create workloads that would best represent a situation that may generate various kinds of I/Os. In order to remove unexpected variance, each test case was completed by first restarting the device. Once booted, we start blktrace and perform the test. Upon completion, we stop blktrace and dump the trace into persistent storage for offline analysis. As overhead resulting from the blktrace operation was of concern, we purposefully stored the output of blktrace within a small amount of DRAM memory to ensure that I/Os directly related to running blktrace would not pollute the collected trace. Each test was run 5 times to ensure that the data was consistent.

All workloads were carefully selected to capture the anticipated largest number of I/Os in critical parts of run time. It is worth noting that our main purpose is to study the impact of storage to user-perceivable performance in practice. As so, we avoid using artificial benchmarks to generate extremely high I/O traffic, which exercises the storage but does not reflect the real-world usage patterns. Also, in our experiments, all workloads were designed to show cases which both involve storage I/Os and practically affect user experience in a typical real-life environment. For example, we were more interested in the process of loading Angry Birds, as the user will be idly waiting for their game to start, over a workload including the user playing Angry Birds, as they will no longer be idling due to storage I/Os. In order to minimize the possibility of latency caused by human interaction, all workloads start at the moment human interaction ends. In all workloads, default configurations were used to get representative results. Table 1 details the type and description of each selected workload.

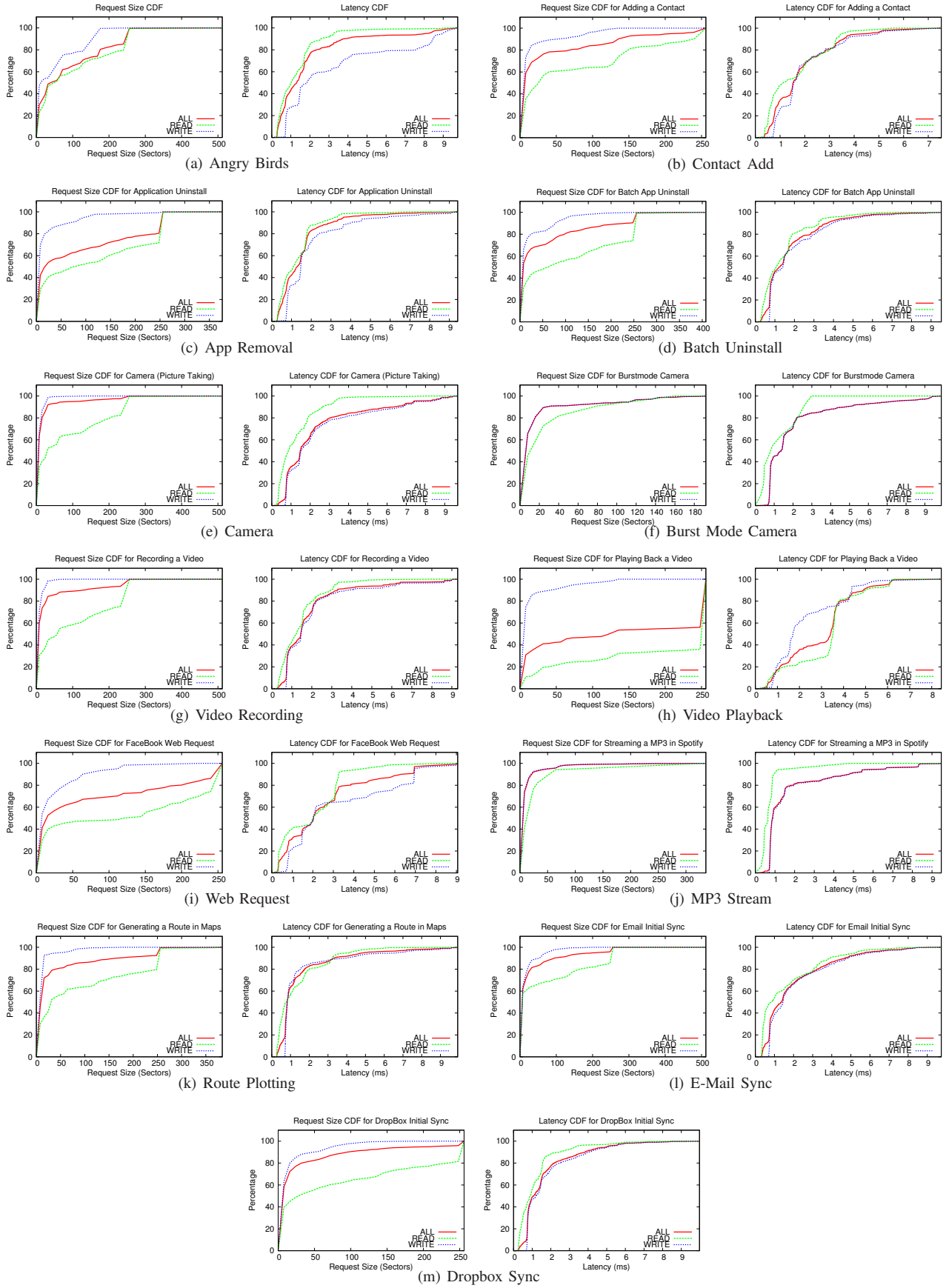
IV. EXPERIMENTAL RESULTS

This section presents our experimental results. We first study the two key factors that describe the basic I/O patterns of a workload, namely *request sizes* and *latencies*. Then, we focus on the *flush* operations, which directly impact the I/O speed on an NAND flash based storage. Next, we consider the data access *locality*, which has a strong implication to cache efficiencies. Finally, we discuss the relative influence of I/O operations to the end-to-end application performance.

A. Request Size and Latency Distribution

Request size and latency are two key factors describing the I/O patterns of a workload. The former determines how large each I/O request is, while the latter measures how long each I/O request takes to the point of completion. These two metrics have a direct but non-linear relationship – a small request is not necessarily equal to a smaller latency, and vice versa. Figure 2 shows the distributions of request sizes and latencies of the 13 workloads. In the following, we examine the workloads based on their categories.

Angry Birds: As a typical mobile game, the Angry Birds workload sees mostly smaller request sizes - 67.8% of all requests are less than 64 KB. Comparatively, however, these write sizes are more variable than the other tests. For instance, in Figure 3(a), we can see two vertical bands of writes at the 36 KB and 88 KB ranges. Write latency for Angry Birds is noticeably longer than reads, as 80% of reads are completed in less than 1.87 ms, while it takes up to 7.50 ms for 80% of the total number of writes to be completed. Of these writes, synchronous writes contribute most to the latency incurred from write I/Os. We also find that reads are more predictable than writes. In Figure 3, we see a nearly linear pattern of reads between latency and request size – smaller reads generally take shorter times to complete while larger reads take longer. In contrast, writes show a much larger variance. There exist distinct patterns of a wide range of latency for a similar request size. For example, latency for a request size of 88 KB ranges from 7-10 ms. Such write latency is surprisingly high, especially considering the eMMC flash device has no mechanical components. This is mostly because writes in flash memory may trigger some high-overhead internal operations, such as block cleaning, which make the I/O latencies more variant [5]. Also, large reads tend to have a relatively higher variance in latencies than small ones, as a large read would take longer to complete and is more likely to affected.



(a) Angry Birds (b) Contact Add

(c) App Removal (d) Batch Uninstall

(e) Camera (f) Burst Mode Camera

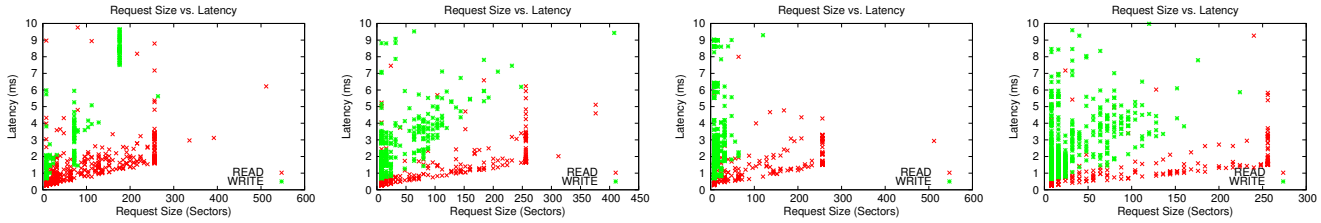
(g) Video Recording (h) Video Playback

(i) Web Request (j) MP3 Stream

(k) Route Plotting (l) E-Mail Sync

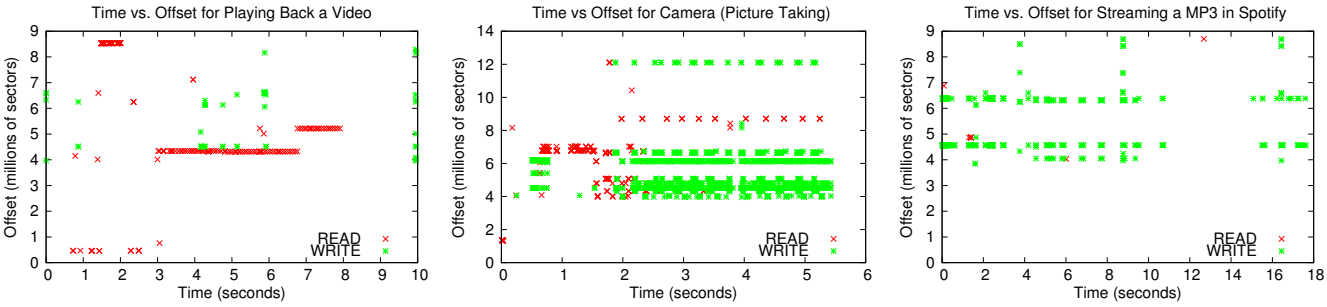
(m) Dropbox Sync

Fig. 2. Request Size and Latency Data by Type for all Workloads (All, Read, Write)



(a) Angry Birds (b) Batch Uninstall (c) Video Recording (d) DropBox Sync

Fig. 3. Selected Request Size vs. Latency Data



(a) Video Playback (b) Camera (c) MP3 Streaming

Fig. 4. Spatial Access Patterns vs. Time. Offset is in units of Sectors.

Uninstall Apps: Removing both a single application and several applications in a batch job often causes noticeable delay. In Figure 2(c) and 2(d), we see similar patterns in both request sizes and latencies. For request size, we find that 82.4% of writes in a single application uninstall and 80.1% of writes in a batch application uninstall are less than 16 KB in size. This is because the two workloads involve intensive file system metadata operations, most of which are rather small (e.g., updating inodes). We also find that writes are slightly slower than reads with 80% of I/Os taking under 2.39 ms for writes and 1.77 ms for reads in a single uninstall and under 3.04 ms for writes and 1.89 ms for reads in a batch uninstall. When comparing latency and request size, we find a similar trend of slow writes of very small size and reads in a linear pattern, as seen in Figure 3(b). The only apparent difference between the single- and batch- uninstall workloads is the quantity of reads and writes being appropriately larger in scale. This indicates that, for such a batch of metadata-intensive workloads, storage I/Os happen mostly in a sequence, and no buffering effect has been observed.

Contact: The Contact Addition experienced several very small writes - about 73.7% of all writes were only 4 KB. Reads, however, were larger than writes - only 64.6% of reads were less than 64 KB. Other data continued to follow typical trends – writes were slightly slower than reads. Compared to other workloads, this workload does not have to load or store data of any significant size, and subsequently it does not present any surprising information.

Multimedia: As shown in Figure 2(e-h), we find several key themes in the four multimedia workloads, Camera, Burst Mode Camera, Video Recording, and Video Playback. First, writes tend to be small. Write sizes of less than 16 KB make up 86.9% of the Camera workload, 81.2% of the Burst Mode Camera workload, and 88.0% of the Video Recording workload. This is because the three workloads involve intensive writes, and frequent flushes create a sequence of small writes. Video Playback is unique. It sees a wider spectrum of write request sizes; however, we still find that most of these writes are small. Second, latency distributions are also similar between the write-intensive multimedia applications. In the Burst Mode Camera, Camera, and Video Recording workloads, 80% of the I/Os are completed in less than 2.20 ms, 3.02 ms, and 2.21 ms respectively. Video Playback shows different patterns - I/Os tended to experience higher latency, with 58.2% of I/Os taking over 3 ms to complete. Third, the latency-vs-request-size trends in multimedia workloads are similar to other workloads – a heavily variable concentration of small writes with a relatively more linear pattern of reads, shown in Figure 3(c). We also find that Burst Mode Camera, Camera, and Video Recording are all quite variable as there are a significant number of writes between 1 and 10 ms of latency. Finally, Video Playback shows a unique pattern. This workload has only 75 total writes, and of these writes, only 5 were larger than 4.5 ms. Unlike other multimedia workloads, Video Playback is read intensive. A large portion of this workload involves retrieving the video from storage to play it back. We see a larger number of slower reads than in other workloads. Also, this workload has the second fewest number of I/Os of any other workload, as it retrieves the video from storage in large (128 KB) chunks. Figure 4(a) illustrates this behavior. We can see a distinct band of I/O reads at the same offset for a large duration of the process of playing back the video. Comparatively, we see the other multimedia applications which save data to storage (e.g., Camera) writing data in small chunks, as shown in Figure 4(b). This process also proves to be extremely costly. Our Camera workload had

the largest percentage of I/O latency than any other workload at nearly 70% of the run time.

Network Apps: The storage I/O behaviors of network-intensive apps follow patterns unique to this category. Of the 5 network applications, shown in Figure 2(i-m), each workload had a majority of small writes. For each network workload, most writes were smaller than 16 KB. For example, the MP3 streaming workload had 92.6% writes smaller than 16 KB, and the Web Request workload had 76.5% writes being less than 16 KB. Reads had some slight variation between each workload and were larger than writes. The MP3 Streaming workload is unique. It has significantly smaller I/O reads, with 76.5% of reads being less than 16 KB, compared to the other Network workloads which, as in the Route Plotting workload which had only 41.0% reads less than 16 KB. This is likely due to the streaming effect, where most reads can be directly satisfied in memory. In this category, we also found that unlike other workloads, I/O request latencies had slower asynchronous writes than synchronous writes. This difference can be most drastically noted in MP3 Streaming, where asynchronous writes are greater than 4 ms in over 74.1% of requests. We continue to see patterns of small slow writes and linear reads when comparing latency and request size. In summary, network workloads are unique in that they experience I/O behavior somewhat similar to other workloads, which is an unexpected finding. An example of this may be seen in the spatial write pattern of MP3 Streaming shown in Figure 3(d) – there are constant I/O writes with very few reads. The reasoning for this may be due to the device having to download and store the data from the network.

B. Flushes

In Android systems, the SQLite library provides a light-weight database for mobile apps to store small pieces of data persistently (e.g., user settings). In order to ensure data consistency, the dirty data in the OS page cache needs to be synchronized to the persistent storage. As a result, the Android operating system frequently uses a flush operation (e.g. FUA and FLUSH) to send buffered data to storage to ensure the persistence. While this is a necessary function to preserve data, too much flushing can result in increased latency, thus degrading application and device performance. We find such a trend of excessive flushing in our analysis of our workloads, characterized by flushes at short intervals, with a only small number of small sized I/O writes between each flush operation. We will examine specific workload categories in greater depth. Figure 5 depicts an average case of flushing in three metrics: the number of I/O requests which occur between flushes, the size of I/O requests between each flush, and the time between successive calls to flush. Table II lists the number of I/O requests, the total I/O request size, and the time interval between two consecutive flushes.

We find in Figure 5 that there are very few I/O requests that take place between successive flushing operations. In 90% of cases, the largest number of I/O requests between two flushes is 74 in the Web Request workload, with all other workloads having less than 49 I/O requests. Workloads such as Angry Birds and Application Remove saw less than 26 requests and 29 requests respectively, and Dropbox Sync had fewer than 14 requests in 90% of cases. We see a varying number of I/O requests between workload categories as well. Network workloads saw as few as less than 10 requests at the 90th percentile (MP3 Streaming, Route Plotting) and as many as less than 74 requests at the 90th percentile (Web Request).

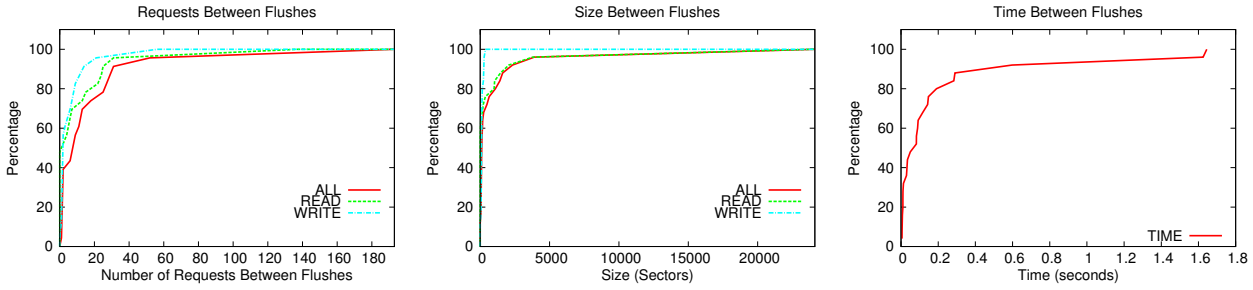
Workload	Requests	Data Size	Time
Angry Birds	26	2028 KB	0.398 sec
App Removal	29	808 KB	0.289 sec
Batch Uninstall	16	332 KB	0.252 sec
Contacts	40	240 KB	1.41 sec
Burst Mode Camera	16	80 KB	0.116 sec
Camera	22	124 KB	0.060 sec
Video Recording	23	204 KB	0.099 sec
Video Playback	49	4196 KB	3.30 sec
Dropbox Sync	14	116 KB	0.216 sec
E-mail Sync	18	180 KB	1.10 sec
Web Request	74	4412 KB	3.13 sec
MP3 Streaming	10	60 KB	0.512 sec
Route Plotting	10	64 KB	0.147 sec

TABLE II
I/OS BETWEEN FLUSHES (90TH PERCENTILE OF CDF)

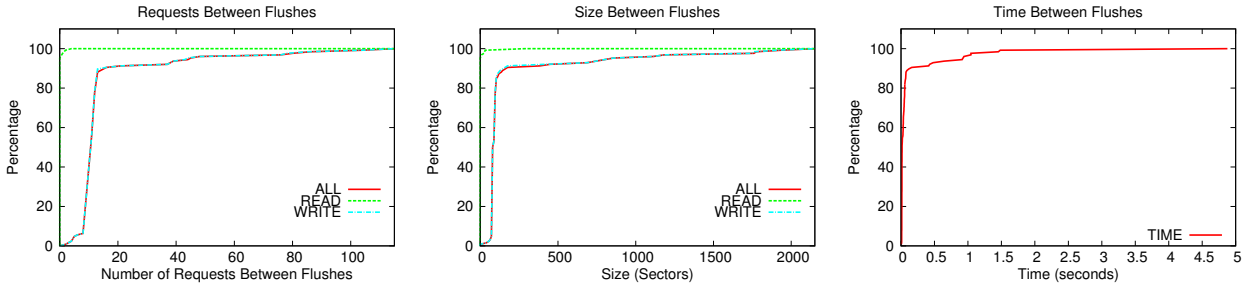
The amount of data of I/O requests between successive flushes is also small but varies between the workloads. Several workloads had total data sizes of less than 128 KB between flushes in 90% of cases, while three outliers are the Angry Birds (2028 KB), Video Playback (4196 KB), and Web Request (4412 KB) workloads.

The time interval between successive flushes is also small. We found a range of typically short intervals. In 8 of the 13 workloads, these intervals were between 0.1 and 0.4 seconds, which means 2-10 flushes happen every second. Workload categories seem to have some influence: three of the five longest intervals occur in the network category, while the three shortest intervals occur in the multimedia category. Since the multimedia apps, such as Camera, involve heavy writes, this suggests that the number of storage I/O writes affects flushing behavior; as more data is written, flushing becomes more frequent.

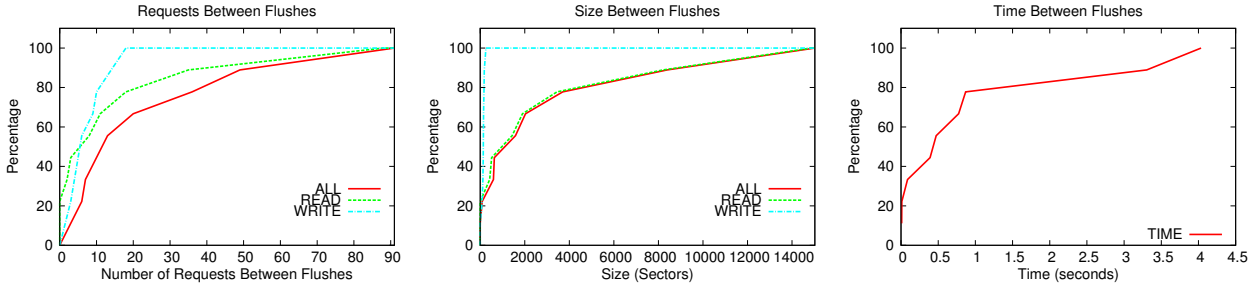
In all, we see very aggressive flushing for the mobile applications. Most have very few requests of small sizes at short intervals between flushes. Application category has influence on the flushing behavior of workloads. Write I/O intensive categories have high frequency, low request numbers, and small request sizes between each flush operation. Less I/O write intensive workloads show less frequent flushes with more requests and larger request sizes between flushes. The biggest reason for variation occurring in flushing between different workloads is the importance of ensuring that the data generated by the respective workload is written to storage. This flushing scheme, however, is problematic because it contributes heavily to the overall latency of storage I/Os.



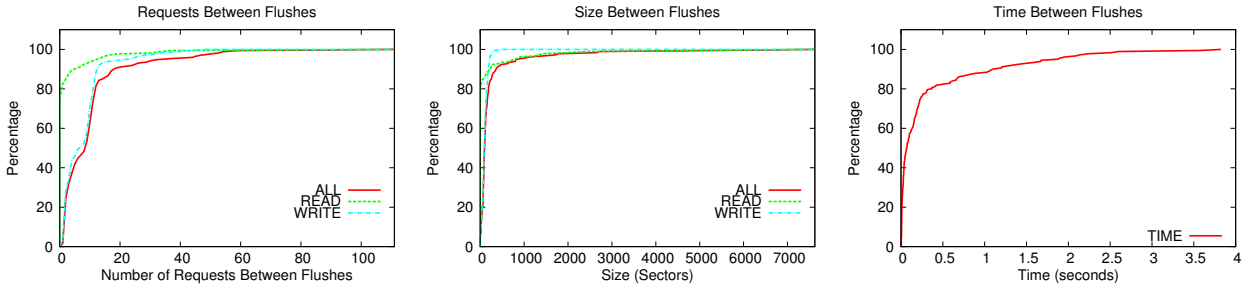
(a) Application Removal



(b) Burst Mode Camera

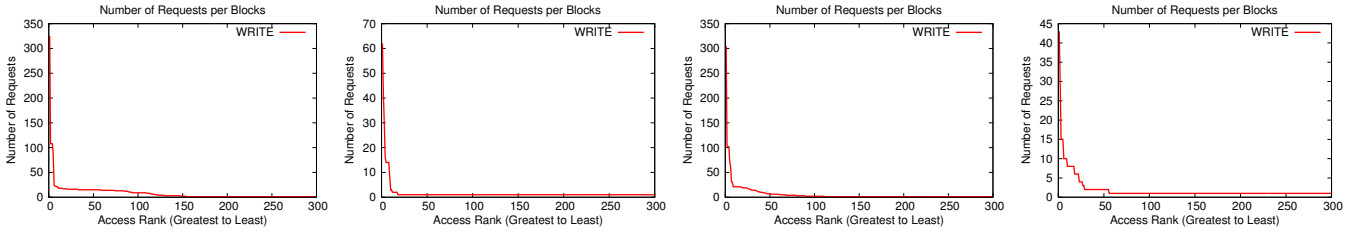


(c) Video Playback



(d) E-mail Sync

Fig. 5. Data Gathered Between Two Flushes



(a) Burst Mode Camera (b) MP3 Streaming (c) Camera (d) Batch Uninstall

Fig. 6. Selected Locality Behavior. All access ranks after 300 have been ignored.

C. Locality

In this section, we briefly analyze spatial locality trends among the workloads in this study. This analysis will refer exclusively to the spatial locality of storage writes, as in each workload, blocks being accessed from read operations were accessed only one time. This is due to the memory being able to fully contain the working set. To confirm this, we repeated the workload under the conditions of reduced available RAM (1 GB) and found that blocks were being accessed twice due to page cache replacement. In general, we have found some localities to be good, while others are heavily skewed.

Workloads with good localities include Burst Mode Camera, Figure 6(a), with 153 blocks covering nearly all of the I/O write requests to 880 total unique blocks, with the 10 most accessed blocks accounting for 25.6% of all accesses. Dropbox Sync also had 187 blocks being re-written. Other workloads with multiple blocks being re-written include Contact Add, MP3 Streaming, and Web Request. Most workloads, however, had heavily skewed localities, where one or very few blocks were re-written in some cases hundreds of times. In an extreme case, Camera saw one block out of 3,293 unique blocks being re-accessed 305 times, as shown in Figure 6(c). Other workloads also saw very few blocks being accessed in a range between about 10 to 150 accesses.

D. End-to-end Impact of Storage I/Os

Although storage I/Os are generally slow, the end-to-end impact of storage I/Os to mobile application performance is complex and depends on various factors, such as relative computing and network speed. In this section, we show the aggregate storage I/O time in the overall workload completion time. Figure 7 and Table III provide the details.

Based on the percentage of I/O latency throughout the duration of the workload, we can distinctly identify 3 major groupings of applications. *(1) Light-I/O Applications:* Four of the five network workloads were found to be lightly affected by I/O latency, with the exception of Dropbox Sync, which stores a lot of data to the device in its initial sync. All other network workloads were found to have less than 9% of I/O latency. This suggests that the network may have a larger contribution in the run time of certain applications. Video Playback was also only lightly affected by storage I/Os, as a large number of sequential reads can be quickly loaded to memory by the OS prefetching. *(2) Moderate-I/O applications:* The two uninstall workloads, Burst Mode Camera, and Dropbox Sync are moderately affected by storage I/Os. This implies that in some cases, even a network based application may be affected in part by storage I/O latency if it is storing a large amount of data. *(3) Heavy-I/O applications:* Three applications are heavily affected by storage I/O latency. The Video Recording application is understandably affected, as it stores a high quantity of data to the storage when the video is completed and saved. Angry Birds was the second worst affected workload. This may be in part due to the complexity of the game itself, as the features of the game may both read the game files and write back progress data to allow for settings or game saving. The Camera workload was the most affected by I/O latency, which accounts for nearly 70% of the run time, as high resolution pictures are written.

In general, our findings indicate that the significance of storage I/Os to the end-to-end performance of mobile apps varies across workloads.

V. System Implications

With these key observations, we are now in a position to present several important system implications. Additionally, this section also provides an executive summary of our answers to the questions we raised at the beginning of this paper.

I/O writes are very small and of varying I/O latency. In all of our mobile applications, we see excessive small write I/Os. The write I/Os exhibit strong locality; some blocks are heavily rewritten while most are only written once. These I/O writes can be of a highly variable latency. Regardless of how large or small the write, we saw a range between 1 ms and 10 ms in nearly every workload. This suggests that write performance, overall, is quite poor. This is mostly because NAND flash does not handle random and small writes well. Consequently, applications which write a lot of data to storage will be the biggest culprits of I/O latency. Because of these small and latent writes, the camera workload saw a near 70% I/O overhead. Even with the Dropbox sync workload's heavy reliance on a network connection to download data, small writes contributed to this workload being the 5th most affected workload from I/O latency. This implies that mobile app designers should pay specific attention to write operations, especially frequent small writes. Buffering and clustering small writes into large ones can effectively reduce the effect.

Aggressive flushing is a common practice in all applications. In nearly every case, a pattern of very short page cache flushes occur (e.g., fsync()). These flushes contain typically less than 40 I/Os of small size and happen very frequently. This is caused by applications which are constantly triggering a flush operation in order to ensure data safety and persistence in the event of some failure. This, in turn, requires the system to stop and wait for the data to be written to the storage. Such a blocking effect further magnifies the effect of slow writes on NAND flash. This aggressive flushing has a large impact to overall system performance: because each interval between flushes is so short, little data is being written per operation, implying that more time is spent waiting than may be necessary. As a result, the user is left waiting for all of the I/Os to complete before they can proceed further. Because these flushes trigger sequences of short and random writes, they inadvertently contribute to the overall problem of storage latency and reduce the possibility of organizing more favorable large writes as well. A potential solution to reducing latency would be to use these flush operations conservatively, thus reducing the frequency of data being written to storage. Also, mobile app developers should understand the impact of such flushes to ensure I/O operations are not issued arbitrarily and should also avoid abusing the SQLite library for randomly

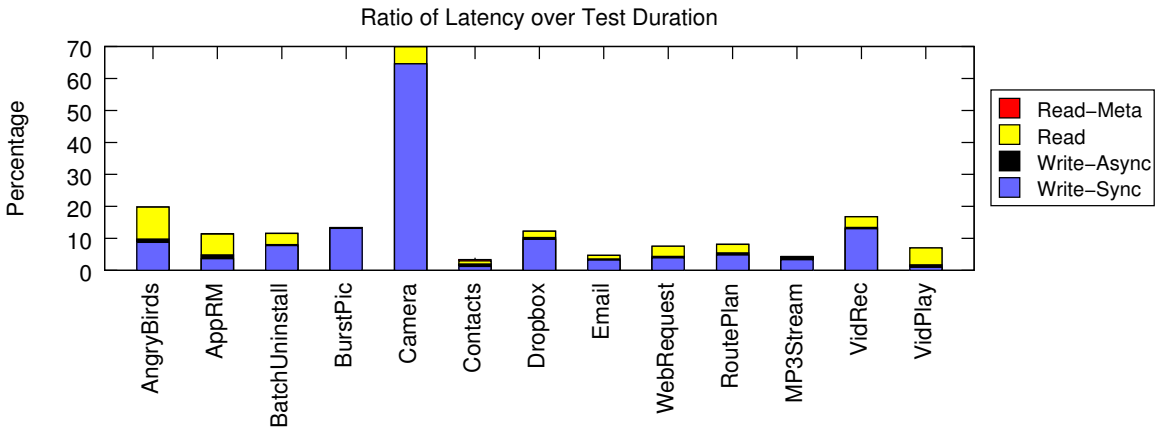


Fig. 7. Ratio of Latency over Test Duration for a given workload

Application Name	Number of I/Os	Number of Reads	Number of Writes	Test Duration (ms)	Percentage of I/O Latency
Angry Birds	846	567	279	7414.26	19.8508
App Removal	511	294	217	5667.04	11.5023
Batch Uninstall	1238	326	912	13346.3	11.5861
Burst Mode Camera	2257	11	2246	17880.6	13.3792
Camera	2319	229	2090	5429.02	69.8771
Contacts	348	113	235	18736.6	2.97313
Dropbox Sync	1983	299	1684	18313.9	12.2474
E-mail Sync	2177	414	1763	61163.4	4.69179
Web Request	173	70	103	4484.59	7.51333
Route Plotting	1950	550	1400	27424.3	8.16327
MP3 Streaming	728	17	711	17606.3	4.33635
Video Playback	256	165	91	7961.2	7.01778
Video Recording	1304	248	1056	10301.7	16.7913

TABLE III
I/Os by Workload

storing small data items - the main source of frequent, small synchronized writes.

I/O reads are mostly one-time access and of relatively predictable latencies. In our experiments, reads exhibit a rather linear behavior between latency and request size in nearly every workload. This finding is consistent with our understanding about reads in NAND flash, in that they are fast and predictable. In addition, these reads are, almost in all cases, one-time access. This is directly a result of the memory being capable of fully containing the workload with a fairly small working-set size. As long as the memory is capable of containing the working-set, the system will not see a significant amount of latency from read I/Os, as it only needs to read storage blocks once. Accordingly, reads do not contribute as much of an impact to I/O related latency. This also is in part due to the read benefits that come from using NAND based flash storage, as these reads will be able to be completed efficiently. This, again, implies that mobile system and app developers need to focus more on optimizing writes.

Synchronous Writes make up a majority of the I/O latency. Of the four different specific types of reads and writes, we see primarily read-aheads and synchronous writes by quantity. By percentage of latency over test duration, however, we find that synchronous writes contribute an overwhelming amount of the total latency from storage I/Os. This latency is compounded by the aggressive flushing experienced in each workload. Because applications are constantly flushing at short intervals (e.g., 200 ms), the device has to spend a large amount of time writing back the data to storage and the applications remain in a state of being blocked. This finding implies that we need to either reduce the amount of synchronous writes or speed them up. The former can be achieved through less use of flushes, while the latter could be realized by adopting a faster storage medium, such as persistent memory [6].

The impact of storage I/Os to the end-to-end application performance is workload dependent. In total, storage I/O based performance degradation appears to be reliant on the type of application being used. The typical percentage of latency due to storage I/Os falls between 7 and 20 percent. We found that network-heavy applications, such as the E-mail application or the MP3 Streaming application, had much less latency due to storage I/Os at just over 4% each. Conversely, multimedia based apps saw typically more latency attributed to storage I/Os, such as the camera workload with nearly 70% latency. These trends are not necessarily strict, as the Dropbox workload indicated that an application may be both heavily reliant on a network and affected by storage I/Os. Comparing Camera and its burst mode version, we can find that leveraging

local buffering can effectively reduce the impact of storage I/O. It also implies that other mobile apps with intensive writes should consider such a simple technique to optimize performance.

Compared to desktop applications, mobile apps show several unique and distinct characteristics. In our experiments, we have observed several interesting properties of mobile apps. First, we see a large volume of synchronous writes and frequent use of flushes in mobile apps. This is related to the mobile apps' heavy reliance on the SQLite library, and we rarely see such patterns in desktop applications. Second, most mobile apps have a more relatively small working set than typical desktop applications. On one hand, it allows most reads to be comfortably accommodated in memory. On the other hand, it makes mobile apps more write I/O intensive and the write performance issues even more obvious. Desktop applications, in contrast, are typically more read intensive and most writes are asynchronous. Thus, the optimization goals for mobile and desktop applications are very different.

In general, through our experimental studies, the implications of the answers show that there is a definite space for optimization with respect to storage I/Os. A solution that could overcome the need to constantly commit data to storage at a short interval, thus only writing small amounts of data, would in turn reduce the overall latency which the user must experience. By reducing the amount of flushing, much of the storage I/O latency can be negated which will consequently optimize application performance on a much larger scale. In the meantime, we should also note that storage I/Os account for a moderate portion of the overall mobile app performance, which indicates that when optimizing mobile app performance, we must consider all system components in a whole package.

VI. Related Work

In recent years, inefficiencies of mobile device optimization have received interests in academia. These prior studies cover various aspects of mobile systems, from power management (e.g., [7], [9], [19], [21]), privacy and security (e.g., [8], [10], [12]), applications (e.g., [11], [22], [24], [25]), and many others. In this section, we will focus on the prior work that is most related to this paper.

Storage I/Os are important to mobile system performance and user experience. Kim et al. have presented benchmarks of Android performance, showing that storage may contribute more of an effect on system performance than previously thought [14]. In contrast to this early work, which focused primarily on SD-card based external storage, our studies are based on the internal eMMC flash storage and show that the impact of storage to the overall user-perceivable performance is moderate for most mobile apps. Lee and Won first noticed several inefficiencies within the various layers of the Android stack [18]. They found that although these layers have been well designed, several issues with journaling still existed and were causing issues with device performance. This anomaly, later known as *Journaling of Journal*, was determined as the result of the innate competition of the journaling actions of the SQLite database which unexpectedly triggered the high-cost journaling in the Ext4 file system [13]. Due to these unexpected interactions between SQLite and Ext4, solutions to address these problems have been proposed, though at this time have not been adopted into Android, as our results suggest that small sync write issues still remain a problem within Android. Jeong et al. investigated the results of changing various features of the Android operating system and discovered that by making changes to the file system and changing the operating mode of the SQLite database, they were able to remove this journaling of journal effect and achieve a 300% performance upgrade from SQLite [13]. Recently, Kim et al. developed an algorithm known as LS-MVBT (multi-version B-tree with lazy split) to reduce the number of `fsync` calls which trigger the journaling in Ext4 [16]. By maintaining the recovery information within the database instead of a journal log, they were able to achieve a 1,220% performance improvement over the default SQLite logging modes. Shen, Park, and Zhu have also identified several implementation limitations of the Android operating system and through applying a custom journaling mode they were able to improve overall database performance by 7% [23]. Additional work in storage areas contributes to the understanding of the uniqueness of mobile storage to traditional desktop systems. Chen et al. performed several tests on flash based SSDs and identified several performance issues that can appear with writes with regards to flash storage, indicating a need to study flash storage uniquely of traditional storage understandings of HDDs [5]. To overcome these innate problems, and those caused by the Journaling of Journal anomaly, Lee et al. show that F2FS, a file system designed to perform for devices using flash storage, outperforms the existing EXT4, effectively removing Journaling of Journal by using append only logging [17]. Recognizing the impact of writes to reads, Nguyen et al. proposed a scheduling scheme to reduce application delay by prioritizing read over write I/Os and grouping them based on priorities [20]. Kim et al. developed a buffer cache replacement scheme to influence the I/O performance on flash [15]. In this paper, our main focus is to characterize the I/O behavior of mobile apps and its interaction with the underlying flash memory. Besides observing frequent flushes, we have also found that the end-to-end impact of these I/O latencies varies depending on applications, which deserves further research.

VII. Conclusions

Mobile devices have become increasingly important in our daily computing. In this paper, we present a comprehensive study on the storage I/O behavior of mobile applications and their interaction with the underlying flash-based storage. By carefully selecting 13 workloads from 5 categories, we perform extensive experimental studies in an attempt to discover the trends that would allow for overall device performance optimization. Our analysis shows that although the number of storage based I/Os comprises a smaller number of the overall I/Os for a running application, the mobile apps exhibit unique I/O patterns. I/O writes, especially those from synchronous

writes and fast flushing, contribute most to latency in these storage I/Os. As a result, a large window of optimization exists at the system level for Android OS design. On the other hand, we have also acknowledged that the end-to-end impact of I/O latencies to application performance depends on workloads, meaning that optimizations must be customized to applications.

ACKNOWLEDGMENT

The authors thank anonymous reviewers for their constructive comments to improve this paper. This work was supported in part by Louisiana Board of Regents under grants LEQSF(2014-17)-RD-A-01 and LEQSF-EPS(2015)-PFUND-391, National Science Foundation under grant CCF-1453705, and generous support from Intel Corporation.

REFERENCES

[1] Yet Another Flash File System. http://www.yaffs.net.

[2] Without Much Fanfare, Apple Has Sold Its 500 Millionth iPhone. http://www.forbes.com/sites/markrogowsky/2014/03/25/without-much-fanfare-apple-has-sold-its-500-millionth-iphone/, 2014.

[3] Market Share: Devices, All Countries, 4Q14 Update. http://www.gartner.com/newsroom/id/2996817, 2015.

[4] Android. https://source.android.com/source/index.html.

[5] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*, pages 181–192, New York, NY, USA, 2009. ACM.

[6] F. Chen, M. P. Mesnier, and S. Hahn. A Protected Block Device for Persistent Memory. In *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST'14)*, Santa Clara, CA, June 2-6 2014.

[7] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom'15)*, Paris, France, September 7-11 2015. ACM.

[8] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, Istanbul, Turkey, March 14-18 2015. ACM.

[9] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*, San Francisco, CA, June 15-18 2010. ACM.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, Canada, October 4-6 2010. USENIX.

[11] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu. Reliable, Consistent, and Efficient Data Sync for Mobile Apps. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, Santa Clara, CA, Feburary 16-19 2015. USENIX.

[12] S. Guha, M. Jain, and V. N. Padmanabhan. Koi: A Location-Privacy Platform for Smartphone Apps. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, April 25-27 2012. USENIX.

[13] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 309–320, San Jose, CA, 2013. USENIX.

[14] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, CA, February 14-17 2012.

[15] H. Kim, M. Ryu, and U. Ramachandran. What is a Good Buffer Cache Replacement Scheme for Mobile Flash Storage? In *Proceedings of the 2012 ACM SIGMETRICS Conference (SIGMETRICS'12)*, London, UK, June 11-15 2012. ACM.

[16] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 273–285, Santa Clara, CA, 2014. USENIX.

[17] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 273–286, Santa Clara, CA, Feb. 2015. USENIX Association.

[18] K. Lee and Y. Won. Smart Layers and Dumb Result: IO Characterization of an Android-based Smartphone. In *Proceedings of the Tenth ACM International Conference on Embedded Software (EMSOFT'12)*, pages 23–32, New York, NY, USA, 2012. ACM.

[19] J. Li, A. Badam, R. Chandra, S. Swanson, B. Worthington, and Q. Zhang. On the Energy Overhead of Mobile Storage Systems. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Santa Clara, CA, Feburary 17-20 2014. USENIX.

[20] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang. Reducing Smartphone Application Delay through Read/Write Isolation. In *Proceedings of the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, Florence, Italy, May 18-22 2015. ACM.

[21] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *Proceedings of the 6th Conference on Computer Systems (EuroSys'11)*, Salzburg, Austria, April 10-13 2011. ACM.

[22] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, October 8-10 2012. USENIX.

[23] K. Shen, S. Park, and M. Zhu. Journaling of Journal Is (Almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 287–293, Santa Clara, CA, 2014. USENIX.

[24] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are Web Browsers Slow on Smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile'11)*, Phoenix, AZ, March 1-2 2011. ACM.

[25] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing Background Email Sync on Smartphones. In *Proceedings of the 11th International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*, Taipei, Taiwan, June 25-28 2013. ACM.

Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives

Feng Chen[1], David A. Koufaty[2], and Xiaodong Zhang[1]

[1]Dept. of Computer Science & Engineering
The Ohio State University
Columbus, OH 43210
{fchen, zhang}@cse.ohio-state.edu

[2]System Technology Lab
Intel Corporation
Hillsboro, OR 97124
david.a.koufaty@intel.com

ABSTRACT

Flash Memory based Solid State Drive (SSD) has been called a "pivotal technology" that could revolutionize data storage systems. Since SSD shares a common interface with the traditional hard disk drive (HDD), both physically and logically, an effective integration of SSD into the storage hierarchy is very important. However, details of SSD hardware implementations tend to be hidden behind such narrow interfaces. In fact, since sophisticated algorithms are usually, of necessity, adopted in SSD controller firmware, more complex performance dynamics are to be expected in SSD than in HDD systems. Most existing literature or product specifications on SSD just provide high-level descriptions and standard performance data, such as bandwidth and latency.

In order to gain insight into the unique performance characteristics of SSD, we have conducted intensive experiments and measurements on different types of state-of-the-art SSDs, from low-end to high-end products. We have observed several unexpected performance issues and uncertain behavior of SSDs, which have not been reported in the literature. For example, we found that fragmentation could seriously impact performance – by a factor of over 14 times on a recently announced SSD. Moreover, contrary to the common belief that accesses to SSD are uncorrelated with access patterns, we found a strong correlation between performance and the randomness of data accesses, for both reads and writes. In the worst case, average latency could increase by a factor of 89 and bandwidth could drop to only 0.025MB/sec. Our study reveals several unanticipated aspects in the performance dynamics of SSD technology that must be addressed by system designers and data-intensive application users in order to effectively place it in the storage hierarchy.

Categories and Subject Descriptors

B.3.2 [**Design Styles**]: Mass storage; D.4.2 [**Storage Management**]: Secondary Storage

General Terms

Experimentation, Measurement, Performance

Keywords

Flash Memory, Hard Disk Drive, Solid State Drive

SIGMETRICS/Performance'09, June 15–19, 2009, Seattle, WA, USA.

1. INTRODUCTION

For the last two decades, researchers have made continuous efforts to address several open issues of Hard Disk Drive (HDD), such as long latencies of handling random accesses (e.g. [15, 21, 32]), excessively high power consumption (e.g. [10, 18, 20]), and uncertain reliability (e.g. [14, 42]). These issues are essentially rooted to the hard disk's mechanic nature, and thus difficult to be solved physically by disks themselves.

Recently, flash memory based Solid State Drive (SSD) has become an emerging technology and received strong interest in both academia and industry [3, 5, 25, 28, 29, 35]. Unlike traditional rotating media, SSD is based on semiconductor chips, which provides many strong technical merits, such as low power consumption, compact size, shock resistance, and most importantly, extraordinarily high performance for random data accesses. Thus flash memory based SSD has been called "a pivotal technology" to revolutionize computer storage systems [39]. In fact, two leading on-line search engine service providers, *google.com* and *baidu.com*, both announced their plans to migrate existing hard disk based storage system to a platform built on SSDs [13].

An important reason behind such an optimistic prediction and such actions is that SSD shares a common interface with traditional hard disks physically and logically. Most SSDs on the market support the same host interfaces, such as Serial Advanced Technology Attachment (SATA), used in hard disks. SSD manufacturers also carefully follow the same standard [1] to provide an array of Logical Block Addresses (LBA) to the host. On the one hand, such a common interface guarantees backward compatibility, which avoids tremendous overhead for migrating existing HDD based systems to SSD based platforms. On the other hand, such a thin interface can hide complex internals from the upper layers, such as operating systems. In fact, many sophisticated algorithms, such as cleaning and mapping policies, have been proposed, and various hardware optimizations are adopted in different SSD hardware implementations [3, 16]. The complicated internal design and divergent implementations behind the 'narrow' interface would inevitably lead to many performance dynamics and uncertainties. In other words, despite the same interface, SSD is not just another 'faster' disk. Thus we need to conduct a thorough investigation, particularly for understanding its intrinsic limits and unexpected performance behavior.

Most existing research literature and technical papers, however, only provide limited information beyond high level description and standard performance data (e.g. bandwidth and latency), similar to documented specifications for HDDs. Some specification data provided by SSD manufacturers may

even overrate performance or not provide critical data for certain workloads (e.g. performance data for random writes is often not presented in SSD specifications). Some previous work has reported SSD performance data based on simulation [3]. So far, little research work has been presented to report the first-hand data and analysis on the unique performance features of different types of SSDs.

In this paper, we carefully select three representative, state-of-the-art SSDs built on two types of flash memory chips. Each SSD is designed to aim at different applications and markets, from low-end PCs, middle-level performance desktops, to high-end enterprise servers. By using the Intel Open Storage Toolkit [36] to generate various types of I/O traffic, we conducted intensive experiments on these SSDs and analyzed collected performance data. Our purpose is not to compare the performance of these competing SSDs on the market. Instead, we attempt to get insightful understanding on performance issues and unique behavior of SSDs through micro-benchmarks. We hope to inspire the research community, especially OS designers, to carefully consider many existing optimizations, which were originally designed for hard disks, for the emerging SSD based platform. We will answer the following questions and reveal some untold facts about SSDs through experiments and analysis.

1. A commonly reported feature of SSD is its uniform read access latency, which is independent of access patterns of workloads. Can we confirm this or observe some unexpected results with intensive experiments?

2. Random writes have been considered as the Achilles´ heel of SSDs. Meanwhile, high-end SSDs employ various solutions to address this problem. Has the random write problem been solved for the recent generation of SSDs? Is random write still a research issue?

3. Caching plays a critical role for optimizing performance in HDDs to mitigate expensive disk head seeks. Can we make a case or not for a cache in SSDs, which do not have the problem of long seek latency?

4. Writes are much more expensive than reads in flash memory. What are the interactive effects between these two types of operations in SSDs? Would low-latency reads be negatively impacted by high-latency writes? What's the reason behind such interference?

5. Many elaborate algorithms are adopted in SSD firmware to optimize performance. This leads to many internal operations running in the background. Can we observe a performance impact on foreground jobs by such internal operations? How seriously would such background operations affect overall performance?

6. Adopting a log-structure internally, SSD can map logically continuous pages to non-continuous physical pages in flash memory. Since a flash memory block may contain both valid and invalid pages, internal fragmentation could be a problem. How would internal fragmentation affect performance?

7. Having well documented our findings on SSDs, what are the system implications? How can we use them to guide system designers and practitioners for effectively adding SSDs into the storage hierarchy?

The rest of this paper is organized as follows. Section 2 introduces background about flash memory and SSD design. Section 3 presents the experimental setup environment. Then we present our experimental results in Section 4 and 5. In Section 6 we summarize the key observations from our experiments and discuss the implications to users and OS designers. Related work is presented in Section 7 and the final section concludes this paper.

2. BACKGROUND

2.1 Flash Memory

There are two types of flash memories, NOR and NAND [33]. NOR flash memory supports random accesses in bytes and it is mainly used for storing code. NAND flash memory is designed for data storage with denser capacity and only allows access in units of sectors. Most SSDs available on the market are based on NAND flash memories. In this paper, flash memory refers to NAND flash memory specifically.

NAND flash memory can be classified into two categories, Single-Level Cell (SLC) and Multi-Level Cell (MLC) NAND. A SLC flash memory cell stores only one bit, while a MLC flash memory cell can store two bits or even more. Compared to MLC, SLC NAND usually has a 10 times longer lifetime and lower access latency (see Table 1). However, considering cost and capacity, most low-end and middle-level SSDs tend to use high-density MLC NAND to reduce production cost. In this paper, we examined two MLC-based SSDs and one SLC-based SSD.

For both SLC and MLC NAND, a flash memory package is composed of one or more *dies* (chips). Each die is segmented into multiple *planes*. A typical plane contains thousands (e.g. 2048) of *blocks* and one or two registers of the page size as an I/O buffer. A block usually contains 64 to 128 pages. Each page has a 2KB or 4KB data part and a metadata area (e.g. 128 bytes) for storing Error Correcting Code (ECC) and other information. Exact specification data varies across different flash memory packages.

Flash memory supports three major operations, *read*, *write*, and *erase*. Read is performed in units of pages. Each read operation may take 25µs (SLC) to 60µs (MLC). Writes are normally performed in page granularity, but some NAND flash (e.g. Samsung K9LBG08U0M) supports sub-page operations [41]. Pages in one block must be written sequentially, from the least significant to the most significant page addresses. Each write operation takes 250µs (SLC) to 900µs (MLC). A unique requirement of flash memory is that a block must be erased before being programmed (written). An erase operation can take as long as 3.5ms and must be conducted in block granularity. Thus, a block is also called an *erase block*.

Flash memory blocks have limited erase cycles. A typical MLC flash memory has around 10,000 erase cycles, while a SLC flash memory has around 100,000 erase cycles. After wearing out, a flash memory cell can no longer store data. Thus, flash memory chip manufacturers usually ship with extra flash memory blocks to replace bad blocks.

2.2 SSD Internals

Since an individual flash memory package only provides limited bandwidth (around 40MB/sec [3]), flash memory based SSDs are normally built on an array of flash memory packages. As logical pages can be striped over flash memory chips, similar to a typical RAID-0 storage, high bandwidth can be achieved through parallel access. A serial I/O bus

connects the flash memory package to a controller. The controller receives and processes requests from the host through connection interface, such as SATA, and issues commands and transfers data from/to the flash memory array. When reading a page, the data is first read from flash memory into the register of the plane, then shifted via the serial bus to the controller. A write is performed in the reverse direction. Some SSDs are also equipped with an external RAM buffer to cache data or metadata [9, 19].

A critical component, called the *Flash Translation Layer* (FTL), is implemented in the SSD controller to emulate a hard disk and exposes an array of logical blocks to the upper-level components. The FTL plays a key role in SSD and many sophisticated mechanisms are adopted to optimize SSD performance. We summarize its major roles as follows.

Logical block mapping – As writes in flash cannot be performed in place as in disks, each write of a logical page is actually conducted on a different physical page. Thus some form of mapping mechanism must be employed to map a logical block address (LBA) to a physical block address (PBA). The mapping granularity could be as large as a block or as small as a page [19]. Although a page-level mapping [23] is efficient and flexible, it requires a large amount of RAM space (e.g. 512MB in [9]) to store the mapping table. On the contrary, a block-level mapping [2], although space-wise efficient, requires an expensive read-modify-write operation when writing only part of a block. Many FTLs [12,22,26,30] adopt a hybrid approach by using a block-level mapping to manage most blocks as *data blocks* and using a page-level mapping to manage a small set of *log blocks*, which works as a buffer to accept incoming write requests efficiently. A mapping table is maintained in persistent flash memory and rebuilt in volatile RAM buffer at startup time.

Garbage Collection – Since a block must be erased before it is reused, an SSD is usually over-provisioned with a certain amount of clean blocks as an allocation pool. Each write of a page just needs to invalidate the previously occupied physical page by updating the metadata, and the new data can be quickly appended into a clean block allocated from the pool, like a *log*, without incurring a costly erase operation synchronously. When running out of clean blocks, a *garbage collector* scans flash memory blocks and recycles invalidated pages. If a page-level mapping is used, the valid pages in the scanned block are copied out and condensed into a new block; otherwise, the valid pages need to be merged together with the updated pages in the same block. Such a cleaning process is similar to the Log-Structured File System [40] and can be conducted in the background.

Wear Leveling – Due to the locality in most workloads, writes are often performed over a subset of blocks (e.g. file system metadata blocks). Thus some flash memory blocks may be frequently overwritten and tend to wear out earlier than other blocks. FTLs usually employ some wear-leveling mechanism to 'shuffle' cold blocks with hot blocks to even out writes over flash memory blocks.

A previous work [3] has an extensive description about the broad design space of flash memory based SSD. Their work indicates that many design trade-offs can be made in SSD, which lead to very different SSD performance. Our experimental results confirmed that diverse performance is resident on various types of SSDs. Since details of SSD hardware design, especially their FTL algorithms, are regarded as the intellectual property of SSD manufacturers and remain highly confidential, we can only speculate about detailed SSD internals and we may not be able to explain all performance behavior observed on SSD hardware. On the other hand, our intention is not to reverse engineer the internal design of SSDs. Instead, we attempt to reveal performance issues and dynamics observed on SSD hardware and give a reasonable explanation to help explore the implications to system designers and practitioners.

3. MEASUREMENT ENVIRONMENT

3.1 Solid State Drives

Currently, many SSDs are available on the market. Their performance and price is very diverse, depending on many factors, such as flash memory chips (MLC/SLC), RAM buffer size, and complexity of hardware controller. In this work, we selected three representative, state-of-the-art SSDs fabricated by two major SSD manufacturers. Each SSD targets at a different market with various performance guarantees, from low-end, middle-class, to high-end. Among them, two SSDs are based on MLC flash memory, and the high-end one is based on SLC flash memory. Since our intention is not to compare the performance of these competing SSDs, we refer to the three SSDs using *SSD-L*, *SSD-M*, and *SSD-H*, from low-end to high-end. Table 1 shows more details about the SSDs. Listed price is as of October 2008.

	SSD-L	SSD-M	SSD-H
Capacity	32GB	80GB	32GB
Price ($/GB)	$5	$10	$25
Flash memory	MLC	MLC	SLC
Page Size (KB)	4	4	4
Block Size (KB)	512	512	256
Read Latency (µs)	60	50	25
Write Latency (µs)	800	900	250
Erase Latency(µs)	1500	3500	700

Table 1: Specification data of the SSDs.

3.2 Experiment System

Our experiment system is a Dell™ PowerEdge™ 1900 server. It is equipped with two Intel® Core™ 2Quad Xeon X5355 2.66GHz processors and 16GB FB-DIMM memory. Five 146GB 15,000 RPM SCSI hard drives are attached on a SCSI interface card to hold the operating system and home directories. We use RedHat Enterprise Linux Sever 5 with Linux Kernel 2.6.26 and Ext3 file system. The SSDs are connected through the on-board SATA 3.0Gb/s connectors. There is no partition or file system created on the SSDs. All requests are performed to directly access SSDs as raw block devices to avoid interference from the OS kernel, such as the buffer cache and the file system.

In our experiment system, the hard disks use the *CFQ* (Completely Fair Queuing) scheduler, the default I/O scheduler used in the Linux kernel, to optimize the disk performance. For the SSDs, which have sophisticated firmware internally, we use the *noop* (No-Op) scheduler to leave the I/O performance optimization directly handled by the block devices, so that we can expose internal behavior of the SSDs. In our experiments, we also found *noop* usually outperforms the other I/O schedulers on the SSDs.

3.3 Intel® Open Storage Toolkit

We attempt to gain insightful understanding of the behavior of SSDs for handling workloads with clear patterns. Thus, our experiments are mainly conducted through well-controlled micro-benchmarks, whose access patterns are known in advance. Macro-benchmarks, such as TPC-H database

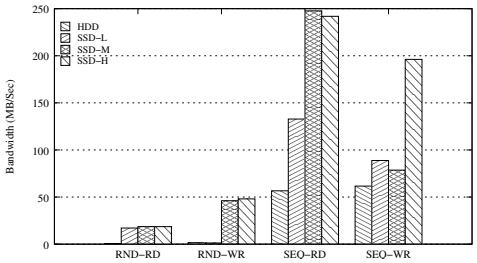


Figure 1: SSD Bandwidths. Four workloads, *Random Read*, *Sequential Read*, *Random Write*, and *Sequential Write*, are denoted as *RND-RD*, *SEQ-RD*, *RND-WR*, and *SEQ-WR* in the figure.

workloads, usually have complicated and variable patterns and thus would not be appropriate for our characteristic analysis of the SSDs. The Intel® Open Storage Toolkit [36] is designed and used for storage research at Intel. It can generate various types of I/O workloads to directly access block devices with different configurations, such as read/write ratio, random/sequential ratio, request size, and think time, etc. It reports bandwidth, IOPS, and latency.

In order to analyze I/O traffic to storage devices in detail, we use *blktrace* [7], which comes with the Linux 2.6 kernel, to trace the IO activities at the block device level. The *blktrace* tool captures various I/O activities, including queuing, dispatching, completion, etc. The most interesting event to us is the *completion* event, which reports the latency for processing each individual request. In our experiments, the trace data is collected in memory during test and then copied to the hard disks that are connected through an RAID interface card to minimize the interference from tracing. The collected data is further processed using *blkparse* and our post-processing scripts and tools off line.

4. GENERAL TESTS

Before we proceeded to the detailed analysis, we first made some general performance measurements on the SSDs. We used the toolkit to generate four distinct workloads, *Random Read*, *Random Write*, *Sequential Read*, and *Sequential Write*. The random workloads used a request size of 4KB to randomly access data in the first 1024MB storage space. The sequential workloads used a request size of 256KB. All the workloads created 32 parallel jobs to maximize the bandwidth usage, and we used direct I/O to bypass the buffer cache and access the raw block devices synchronously. Each test ran for 30 seconds. In order to compare with hard disks, a Western Digital WD1600JS Caviar 7200RPM hard disk was used as a reference HDD.

Figure 1 shows the bandwidths of the four workloads running on the devices. As expected, when handling *random read*, the SSDs exhibit clear performance advantages compared to the HDD. Specifically, all the three SSDs significantly outperform the HDD (0.54MB/sec) by over 31 times higher bandwidths. For *random write*, SSD-M and SSD-H achieve bandwidths of 46MB/sec and 48MB/sec, respectively. SSD-L, the low-end device, has a much lower bandwidth (1.14MB/ sec), which is only comparable to the HDD (1.49MB/sec). When handling *sequential read*, the bandwidth of SSD-L reaches 133MB/sec, and both SSD-M and SSD-H achieve an adorable bandwidth of over 240MB/sec, which is nearly 2/3 of the full bandwidth supported by the SATA bus. In contrast, the HDD has a bandwidth of only 56.5MB/sec. For *sequential write*, SSD-L and SSD-M have bandwidths of 88MB/sec and 78MB/sec, respectively. SSD-H, the enterprise SSD, achieves a much higher bandwidth, 196MB/sec. Considering the long-held belief that SSD has poor write performance, this number is surprisingly high.

As we see in the figure, the performance of the SSDs is better than or comparable to the HDD for all workloads. Compared to SSD-L, the two recently announced higher-end SSDs do show better performance, especially for *random write*.[1] In the next few sections, we will further examine other performance details about the SSDs.

5. EXPERIMENTAL EXAMINATIONS

5.1 Micro-benchmark Workloads

In general, an I/O workload can be characterized by its access patterns (random or sequential), read/write ratio, request size, and concurrency. In order to examine how an SSD handles various types of workloads, we use the toolkit to generate I/O traffic using various combinations of these factors. Three access patterns are used in our experiments.

1. **Sequential** - Sequential data accesses using specified request size, starting from sector 0.

2. **Random** - Random data accesses using specified request size. Blocks are randomly selected from the first 1024MB data blocks of the storage space.

3. **Stride** - Strided data accesses using specified request size. The distance between the end and the beginning of two consecutive accesses is the request size.

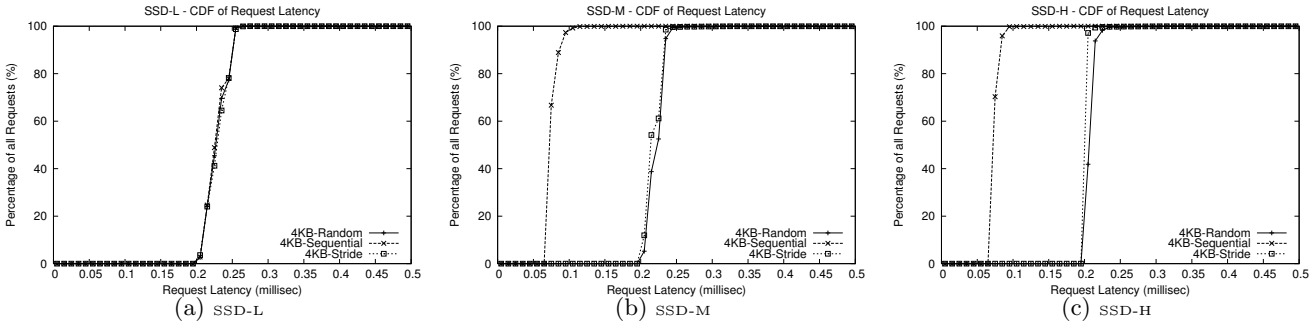
In default, each micro-benchmark runs for 10 seconds to limit trace size while collecting enough data. In order to simplify analysis, we set only one job for all experiments but vary the other three factors, access pattern, read/write ratio, and request size, according to various experimental needs. All workloads directly access the raw block devices. Requests are issued to devices synchronously.

Before our experiments, we first filled the whole storage space using sequential writes with request size of 256KB. After such an initial overwrite, the SSDs in our experiments are regarded as 'full', and the status would remain stable.[2] Note that such a full status remains unchanged thereafter, even if users delete files or format the device, since the liveliness of blocks is unknown at the device level.

Writes into SSDs may change the page mapping dynamically. In order to guarantee that the status of the SSDs remains largely constant across experiments, before each experiment we reinitialize the SSD status by sequentially overwriting the SSD with request size of 256KB. In this way, we make sure that the logical page mapping would be reorganized into an expected continuous manner before each run.

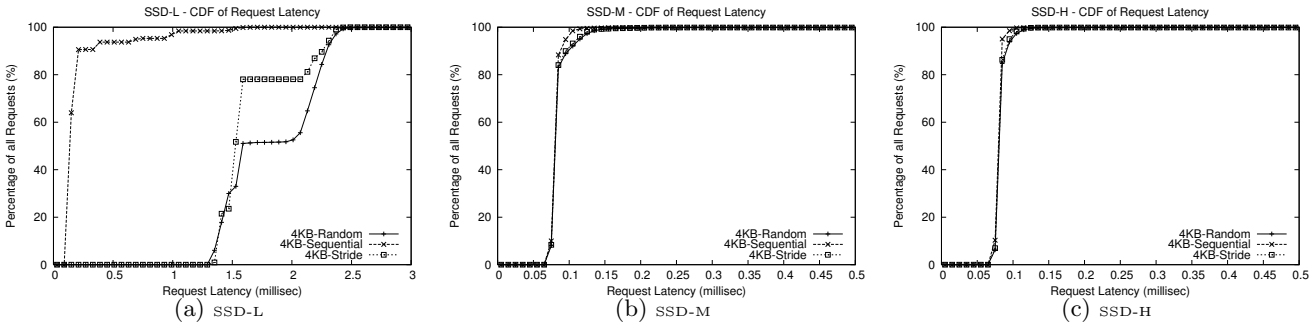
5.2 Do reads on SSD have a uniform latency?

Due to its mechanical nature, the hard disk has a *non-uniform* access latency, and its performance is strongly correlated with workload access patterns. Sequential accesses on hard disks are much more efficient than random accesses. An SSD, without expensive disk head seeks, is normally believed to have a *uniform* distribution of access latencies, which is independent of access patterns. In our experiments, we found this is not always true, for both reads and writes. In this section, we will first examine read operations.



(a) SSD-L (b) SSD-M (c) SSD-H

Figure 2: CDF of request latencies for Read operations on the SSDs.



(a) SSD-L (b) SSD-M (c) SSD-H

Figure 3: CDF of request latencies for Write operations on the SSDs.

In order to examine the distribution of read latencies on SSDs, we run *sequential*, *random*, and *stride* workloads using 4KB read requests on the three SSDs. Figure 2 shows the Cumulative Distribution Function (CDF) of request latencies for the three workloads on the SSDs.

SSD-L does not distinguish random and sequential reads, which is an expected *uniform* distribution. Nearly all reads of the three workloads have latencies of 200-250µs on SSD-L. However, SSD-M and SSD-H, the two higher-end SSDs, do show a *non-uniform* distribution of latencies. Specifically, *sequential* has latencies of only 75-90µs, which is nearly 65% less than *stride* and *random* (200-230µs).

Several reasons may contribute to the unexpected non-uniform distribution of latencies in SSD-M and SSD-H. First, a sequential read in flash memory is three orders of magnitude faster than a random read. For example, in the Samsung K9LBG08U0M flash memory, a random read needs 60µs, while a sequential read needs only 25ns [41]. Second, similar to hard disks, a readahead mechanism can be adopted in an SSD controller to prefetch data for sequential reads with low cost. Most flash memory packages support two-plane operations to read multiple pages from two planes in parallel. Also, operations across dies can be interleaved further [41]. Since logical pages are normally striped over the flash memory array, reading multiple logically continuous pages in parallel for readahead can be performed efficiently.

Concerning our measurement results, we believe readahead is the main reason for the observed non-uniform distribution of latencies. On SSD-M and SSD-H, each sequential read accounts for only 75-90µs, which is even smaller than the time (100µs) to shift a 4KB page out over the serial bus, not to mention the time to transfer data across the SATA interface. Thus, the data is very likely to be fed directly from an internal buffer rather than from flash memory. If data accesses become non-sequential (*stride* or *random*), readahead would be ineffective and the latency increases to around 200-230µs. We also found that the first four requests in *sequential* have latencies of around 220µs, which appears to be an initial period for detecting a sequential pattern. In contrast, as a low-end product, SSD-L does not have a large buffer for readahead, thus each read has to load data directly from flash memory with a similarly long latency for different access patterns, which leads to a uniform distribution of latencies. Another interesting finding is that when handling random reads, the three SSDs have comparable latencies, though SSD-H uses expensive SLC flash.

5.3 Would random writes be the worst case?

A random write on flash memory is normally believed less efficient than a sequential write. Considering an erase block of N pages, a random write may incur up to $N-1$ page reads from the old block, N page writes to the new block, and one block erase. In contrast, a sequential write only incurs one page write and 1/N erase operations on the average. In practice, a log-structured approach [3,5,19] is adopted to avoid the high-cost *copy-erase-write* for random writes. Akin to the Log-Structured File System [40], each write only appends data to a clean block, and garbage collection is conducted in the background. Using such log-structured FTLs, the write is expected to be largely insensitive to access patterns. In this section, we will examine the relationship between writes on SSDs and workload access patterns.

Similar to the previous section, we run three workloads, *sequential*, *random*, and *stride* on the SSDs, except that all requests are write-only. Figure 3 plots the CDF of request latencies of the three workloads.

SSD-L shows a *non-uniform* distribution of request latencies for workloads with different access patterns. Sequential accesses are the most efficient. As shown in Figure 3(a), 88% of the writes in *sequential* have latencies of only 140-160µs. In contrast, *stride* and *random* are much worse. Over 90% of the requests in both workloads have latencies higher than 1.4ms. This seems to confirm the common perception that writes on SSD are highly correlated with access patterns.

In order to investigate why such a huge difference exists

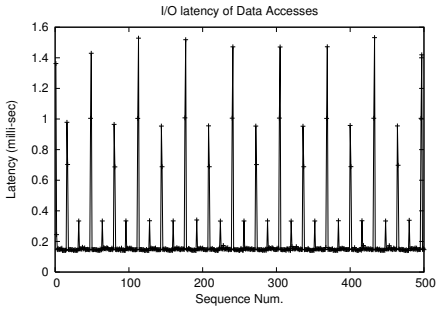
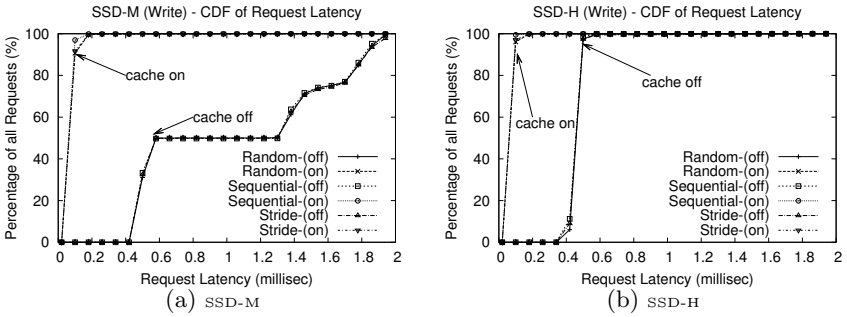


Figure 4: The first 500 requests of *sequential write* on SSD-L.



(a) SSD-M (b) SSD-H

Figure 5: The effect of a disk cache to writes on SSDs. Disk cache state is denoted as *off* and *on*.

between sequential and non-sequential writes on SSD-L, we plot the first 500 write requests of *sequential* in Figure 4 and we see an interesting pattern. As shown in the figure, most requests have a latency of around 140-160µs. A spike of around 1.4-1.5ms appears every 64 requests, and three spikes of around 1ms, 700µs, and 300µs periodically appear in between. The average latency is 209µs.

Our explanation is as follows. From the figure, SSD-L seems to use a small buffer (e.g. 32 pages) to hold data of incoming write requests, because most writes have a latency much lower than the programming time (800µs). If write requests are coming sequentially, data can be temporarily held in the buffer. Since the logically continuous pages can be striped over multiple flash memory chips, the buffered pages can be written into flash memory in parallel later. If incoming requests are non-sequential, such optimization cannot be done, thus data has to be directly written into flash memory, which leads to a much higher latency for each individual write. However, since SSD-L does not employ a sufficiently large buffer (only 16KB) to hold 32 pages, we suspect that SSD-L actually takes advantage of the 4KB registers of the 32 planes, which are organized in 16 flash memory packages, as a temporary buffer. As the data has to be transferred to the register through the serial bus, each write would cause a latency of around 140-160µs, which is slightly lower than a read (200-230µs) that needs extra time to read data from flash memory. When the registers are full, a page program confirm command is issued to initiate the programming process and data can be written into flash memory in parallel, which results in the observed high spikes.

Writes on SSD-M and SSD-H, in contrast, are nearly *independent* of workload access patterns. On both SSDs, over 90% of the requests of all the three workloads have latencies of 75-90µs. Since the observed latency is much lower than the latency for writing a page to flash memory, writes should fall into an on-drive cache instead of being actually written to flash memory medium. Actually, the observed write latency is nearly equal to the latency of sequential reads, which transfer data across SATA bus in the reverse direction. This also confirms our previous hypothesis about readahead on SSDs. As write requests can return to the host as soon as data reaches on-drive cache, distinct access patterns would not make any difference on the write latencies.

5.4 Is a disk cache effective for SSDs?

Modern hard disks are usually equipped with an on-drive RAM cache for two purposes. First, when the disk platter rotates, blocks under the disk head can be prefetched into the cache so that future requests to these blocks can be satisfied quickly. Second, write requests can immediately return as soon as data is transferred into the disk cache, which helps reduce write latency. Similar to disks, SSDs can also benefit from a large RAM cache, especially for relatively high-cost writes. For brevity, we still use the term, *disk cache*, to refer to the RAM buffer on SSDs.

In practice, many low-end SSDs omit the disk cache to lower production cost. For example, SSD-L has no external RAM buffer, instead, it only has a 16KB buffer in the controller, which cannot be disabled. In order to assess the value of the disk cache, we run *random*, *sequential*, and *stride* workloads with requests of 4KB on SSD-M and SSD-H. We use *hdparm* tool to enable and disable the disk cache and compare the performance data to examine the impact of the disk cache. In our experiments we found that disabling the disk cache would not affect read-only workloads. Thus we only present experimental results for write operations.

A disk cache has a significant impact on write performance. Figure 5(a) shows that with disk cache disabled, both SSD-M and SSD-H experienced a significant increase of latencies. On SSD-M, more than 50% of the requests suffer latencies of over 1.3ms, and all requests have latencies of over 400µs. Even with a disabled disk cache, the three workloads share the same distribution of latencies on both SSDs. This indicates that they adopt a log-structured approach and thus are insensitive to workload access patterns.

As a high-end SSD, SSD-H performs substantially better than SSD-M, when the disk cache is disabled. Without the disk cache, the performance strength of SLC over MLC is outstanding. Nearly all requests have latencies of 400-500µs. However, we should note that these latencies are still over 5 times higher than the latencies (75-90µs) observed when the disk cache is enabled. This means that the disk cache is critical to both MLC and SLC based SSDs.

5.5 Do reads and writes interfere with each other?

Read and write in SSDs can interfere with each other. Writes on SSDs often bring many high-cost internal operations running in the background, such as cleaning and asynchronous write-back of dirty data from the disk cache. These internal operations may negatively affect foreground read operations. On the other hand, reads may have negative impact on writes too. For example, reads may compete for buffer space with writes. Moreover, mingled read/write requests can break detected sequential patterns and affect pattern-based optimization, such as readahead.

Since requests submitted in parallel apparently would interfere with each other, we only examine requests submitted in sequence. We especially designed a tool to generate four

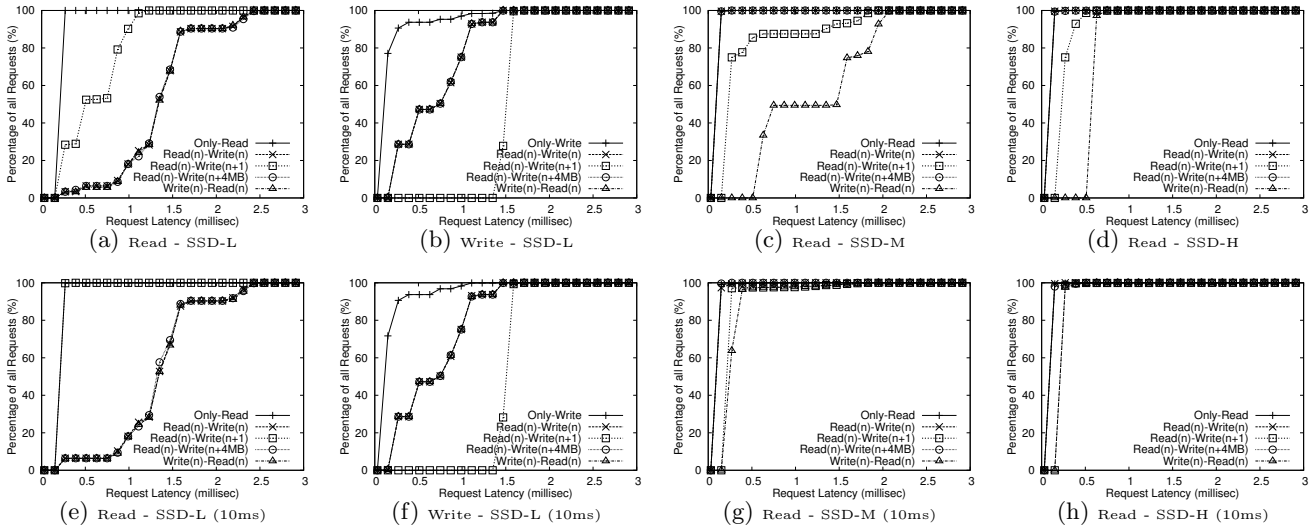


Figure 6: Interference between reads and writes. The second row shows workloads with 10ms interval.

workloads to see how read and write running in sequence interfere with each other. Similar to the Intel® Open Storage Toolkit, this tool directly accesses storage devices as raw block devices, and the other system configurations remain the same. Since sequential workloads are the most efficient operations on SSDs, as shown in previous sections, we use *sequential read* and *sequential write* as two baseline cases, denoted as *only-read* and *only-write*, respectively. We create four types of read/write sequences as follows. Each workload is composed of 1,000 requests for 4KB data.

1. **Read(n)+Write(n)** - sequentially read a page and then write the same page.

2. **Write(n)+Read(n)** - sequentially write a page and then read the same page.

3. **Read(n)+Write(n+1)** - sequentially read a page n and write page $n+1$, then read page $n+2$ and write paper $n+3$, and so on.

4. **Read(n)+Write(n+4MB)** - sequentially read page 0,1,2..N. Simultaneously, we sequentially write pages 4MB apart. Thus, there are two sequential streams.

Figure 6 shows the CDF of read and write latencies of four workloads running on the SSDs. Since writes on SSD-M and SSD-H in this experiment show the same distribution as in Section 5.3, we only show CDF of read latencies here.

Compared to *only-read* and *only-write*, both reads and writes in the four workloads running on SSD-L experienced a substantial degradation (Figure 6(a) and 6(b)). Even when sequentially reading and writing the same page (*read(n) + write(n)* and *write(n) + read(n)*), performance is seriously affected. This indicates that SSD-L does not use a shared buffer for read and write, since a read that immediately follows a write to the same page still encounters a high latency.

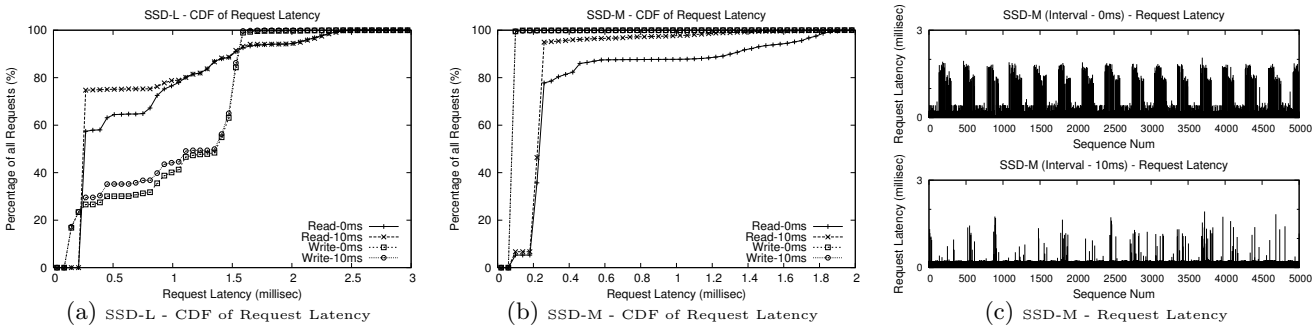
As mentioned in Section 5.3, SSD-L optimizes performance for sequential writes. It seems to detect a sequential write pattern by recording the previous write request's LBN, instead of tracking the sequence of mingled read and write requests. Among the four workloads, *read(n) + write(n+1)* is the only one whose write requests are non-continuous. As expected, it shows the worst write performance, and nearly all requests have a latency of over 1.3ms. Its read operations are also interfered with mingled writes, but it outperforms the other three workloads. We found that inserting an interval of 10ms between two consecutive requests (Figure 6(e)) improves its read performance close to that when running without interleaved with writes. This suggests that the increase of read latencies is a result of asynchronous operations triggered by the previous write.

The other three workloads on SSD-L show nearly identical curves (overlapped in the figures). We can see that writes are apparently affected by mingled reads, but much less severely. Reads, however, are significantly affected by writes, and inserting a delay would not improve performance. We believe that this is because writes just transfer data to the flash memory plane registers, and reads trigger the programing process and cause a synchronous delay (800µs). This also explains the fact that over 80% of the reads in these workloads have a latency of higher than 1ms.

SSD-M and SSD-H both have a disk cache, thus only reads are sensitive to the interleaved read and write accesses. For *read(n)+write(n)* and *read(n)+write(n+4MB)*, read latencies (around 75µs) do not change, even when being interleaved with writes, just like in *only-read*. This indicates that readahead is effective for both cases. However, *read(n) + write(n+1)* and *write(n) + read(n)* are negatively impacted by mingled writes. In the former workload, since each incoming read request is not continuous with the previous read request (interrupted by a write), all of the reads have a latency of over 220µs (i.e. random read latency). In the latter case, it is interesting to see in Figure 6(c) that the curve is similar to the curve of writes when the disk cache is disabled (see Figure 5(a)). This is because when the previous write returns, its dirty data held in the disk cache is flushed to flash memory asynchronously. When a read request arrives, it has to wait for the asynchronous write-back to complete, which leads to a delay. To confirm this, we again inserted an interval of 10ms after each write, and this high read latency disappears, as expected (Figure 6(g)). We also found that even with a delay, both cases only can achieve the performance of *random read*, which shows that readahead is not in effect. We have similar findings for SSD-H.

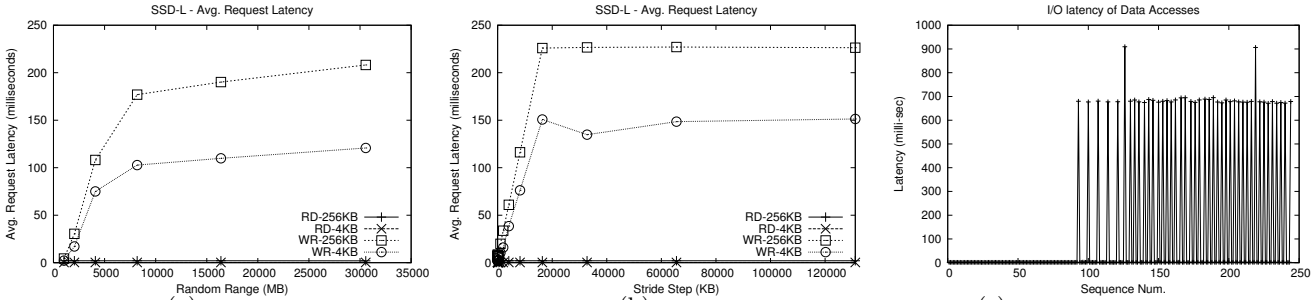
5.6 Do background operations affect performance?

SSDs have many internal operations running in the back-



(a) SSD-L - CDF of Request Latency (b) SSD-M - CDF of Request Latency (c) SSD-M - Request Latency

Figure 7: Background operations on SSD-M. All workloads use 4KB requests.



(a) Random (b) Stride (c) Random Write 4KB (30GB)

Figure 8: Randomness of workloads on SSD-L. Figure 8(c) shows random write of 4KB data in 30GB range.

ground asynchronously, such as delayed writes and cleaning. Though these internal operations create opportunities to leverage idle time, they may also compete for resources with incoming foreground jobs and cause increased latencies. Since write operations are most likely to trigger internal operations, we use the toolkit to run a *sequential* workload using request size of 4KB. The type (read/write) of each request is randomly determined and 50% of the requests are writes. In order to show the performance impact, we intentionally slow down the incoming requests by inserting a 10ms interval between two consecutive requests. Only those *non-stationary* latencies are likely to be caused by background operations. We use this case to *qualitatively* show the impact of background operations on foreground jobs. Due to the space constraints, we only present the data for SSD-L and SSD-M here. SSD-H has similar results.

As shown in Figure 7(a) and 7(b), we can clearly see the performance impact of background operations on foreground jobs, especially on reads. Specifically, after inserting a 10ms interval, the percentage of the reads that have latencies lower than 300µs increases from 57% to 74% on SSD-L, and from 78% to 95% on SSD-M. Writes, in contrast, are barely affected by internal operations, especially on SSD-M. Most writes still fall into the disk cache with low latency. We also examined *sequential* and *random* with write-only requests (not shown here), although we found that background operations can cause some spikes as high as 7-8ms, the overall performance impact on writes is still minimal. To visualize the effect of these background operations, we show the first 5,000 requests of the workloads running on SSD-M in Figure 7(c). We can see that after inserting a 10ms interval, most high spikes disappeared. It is apparent that the background operations are completed during the idle periods such that foreground jobs do not have to compete for resources with them. Though we cannot exactly distinguish various background operations causing these high spikes, we use this case to qualitatively show that such background operations exist and can affect foreground jobs.

5.7 Would increasing workload randomness degrade performance?

In previous sections, we have seen that both read and write on SSDs can be highly correlated to workload access patterns (i.e. sequential or random), similar to hard disks. In this section, we further examine how performance of random writes varies when we increase the randomness of workloads (seek range). We run *random write* and *stride write* on the three SSDs. For *random write*, we vary the random range from 1GB to 30GB. For *stride write*, we vary the stride step from 4KB to 128MB. Each workload uses a request size of 4KB. We did not see an increase of latency with random range and stride steps on SSD-M and SSD-H. Thus, we only report experimental results for SSD-L.

Increasing workload randomness has a significant impact on performance of SSD-L. Figure 8(a) shows that as the random range increases from 1GB to 30GB, the average request latency increases by a factor of 55 (from 2.17ms to 120.7ms), and the average bandwidth drops to as low as 0.03MB/sec. Such a low bandwidth is even 28 times lower than a 7200RPM Western Digit hard disk (0.85MB/sec), and the whole storage system is nearly unusable. The same situation also occurs in *stride write*. As we increase the stride step from 4KB to 128MB, the average latency increases from 1.69ms to 151.2ms (89 times increase) and the bandwidth drops to only 0.025MB/sec. Such a huge performance drop only applies to writes. Increasing request size to 256KB does not help to mitigate such a problem. After investigating workload traces, we found that as the request randomness increases, many requests emerge with a latency of as high as 680ms. The more random the workload is, the more frequently such extraordinarily high-cost requests appear. Figure 8(c) shows the case of randomly writing 4KB data in a 30GB range. The spike of 680ms appears in nearly every three requests in the figure.

Two reasons may cause such high spikes, metadata synchronization and log block merging. An SSD maintains a

mapping table to track the mapping between logical block addresses and physical block addresses. When writing data into SSD, the mapping table needs to be updated in a volatile buffer and synchronized to flash memory periodically. The frequency of metadata synchronization could be sensitive to access patterns. For example, some FTL algorithms employ a B^+ tree-like structure [6,43] to manage the mapping table. Using such a data structure, the more random writes are, the more indexing nodes have to be updated in an in-memory data structure (e.g. *journal tree* in JFFS3). With limited buffer space, SSD-L has to frequently lock the in-memory data structure and flush the metadata in volatile cache to persistent flash memory, which could cause a high spike to appear repeatedly. Some other FTL algorithms (e.g. [19]) can have similar problems with limited RAM space. Interestingly, we also observed similarly high spikes occasionally appearing on SSD-M and SSD-H, and the manufacturer confirmed that it is caused by metadata synchronization.

Another possible reason is log block merging. In order to maintain a small mapping table while achieving reasonable write performance, many FTLs [12, 22, 26, 30] adopt a large mapping granularity (e.g. a block) and use one or multiple *log blocks* to hold data for incoming writes. When running out of log blocks, each page in the log block has to be merged with the other valid pages in the same mapping unit, which causes a high-cost *copy-erase-write* operation. Obviously, if multiple pages in a log block belong to the same mapping unit, only one merge is needed. Thus, the more random the writes are, the more merge operations are needed. In the worst case, each individual page in a log block would belong to a different mapping unit and needs a merge operation correspondingly. This may explain why the performance would not get worse when the stride range exceeds 16MB (Figure 8(b)), because at that time, each page in a log block would belong to an individual mapping unit anyway. Although this speculation still needs confirmation from the SSD manufacturer, it reasonably explains the observed behavior.

5.8 How does fragmentation affect performance?

In order to achieve acceptable performance, many FTL algorithms are based on a log structured approach [3] to handle writes. With such a design, a logical page may be dynamically mapped to any physical flash memory page. When a logical page is overwritten, the data is appended to a clean block, like a *log*. The logical page is mapped to the new position, and the obsolete physical page is only invalidated by updating its metadata. Therefore, if the incoming writes are randomly distributed over the logical block address space, sooner or later all physical flash memory blocks may have an invalid page, which is called *internal fragmentation*.

Such an internal fragmentation may negatively impact performance. First of all, the *cleaning efficiency* drastically drops down. Suppose an erase block has N pages and each block has only one invalid page. In order to get a clean block, N flash memory blocks have to be scanned. During this process, $N \times (N-1)$ page reads, $N \times (N-1)$ page writes, and N block erases are needed. Second, since logically continuous pages are not physically continuous to each other, the readahead mechanism would not be effective any more, which impacts read performance.

We designed a workload, called *striker*, to create fragmentation in SSDs. This workload randomly writes 4GB data into SSD, each request writes only 4KB data, and each overwritten block is randomly selected from the whole SSD space. Under the impact of such extremely intensive random writes, an SSD would be significantly fragmented. In order to show the performance impact of fragmentation, we run *sequential* and *random* workloads using read and write requests of 4KB, and we compare the SSD performance before and after fragmentation. Since *striker* runs excessively slowly on SSD-L, we only report data for SSD-M and SSD-H.

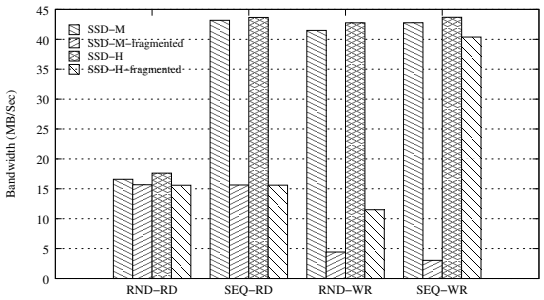


Figure 9: Bandwidth of fragmented SSDs. Four workloads, *Random Read*, *Sequential Read*, *Random Write*, and *Sequential Write*, are denoted as *RND-RD*, *SEQ-RD*, *RND-WR*, and *SEQ-WR*. All workloads use a request size of 4KB.

Figure 9 shows the bandwidths of four workloads running on SSD-M and SSD-H before and after being fragmented. Note that we use request size of 4KB and one job here, thus the sequential workloads do not reach the previously observed bandwidths (Figure 1). We can see that both SSD-M and SSD-H experienced significant performance degradation after being fragmented. Due to the internal fragmentation, logically continuous pages are actually physically non-continuous to each other, which makes readahead ineffective. This causes the bandwidth of *sequential read* to drop from 43MB/sec to only 15MB/sec, which is close to the bandwidth of *random read*. Such a 2.8 times bandwidth decrease is reflected by the corresponding increase of latency from 75µs to 235µs. However, *random read* is not affected much by fragmentation.

Fragmentation has even more significant impact on performance of writes. After fragmentation, the bandwidth of *sequential write* on SSD-M collapses from 42.7MB/sec to 3.02MB/sec (14 times lower). This bandwidth is even much lower than that on a regular laptop disk. Although the disk cache is enabled in this case, the average latency on the fragmented SSD-M increases to as high as 1.27ms. Similar performance degradation is present for *random write*. Compared to that on SSD-M, the write performance drop is less significant on SSD-H, an enterprise-level SSD. Its bandwidth still reaches 40.3MB/sec for *sequential write* and 11.4MB/sec for *random write*. This is attributed to its larger pool of over-provisioned erase blocks (25% of the SSD capacity) and faster erase and write in SLC flash memory.

We believe that the huge performance drop in SSD-M is mainly due to internal fragmentation and exhausted clean blocks. When aggressively writing into SSD-M in a very random manner, the allocation pool of clean blocks is exhausted quickly. Meanwhile, considering the limited life-cycles of MLC flash memory, SSD-M may take a 'lazy' cleaning policy. This causes each write to clean and recycle invalidated pages synchronously. Meanwhile, since most flash memory blocks are significantly fragmented by random writes, each write becomes excessively expensive. The result is that nearly 38% of the writes have a latency higher than 1ms in *sequential write*, and many of them have a latency of 8-9ms. This in turn results in a bandwidth of as low as 3.02MB/sec. On the

other hand, we have to point out that such aggressive random writes only represent a very extreme case, which is unlikely to appear in real-life workloads. However, since most file systems normally generate writes smaller than 256KB, the fragmentation problem still needs to be paid attention, especially in an aged system.

We have attempted to restore SSD performance by inserting a long idle time or mixing read and write requests to create more chance for the firmware to conduct cleaning in the background. However, it seems that a long idle period would not automatically cure a fragmented SSD. Eventually we found a tricky way to restore the SSD performance. We first fill the SSDs with big sequential writes (256KB), then we repeatedly write the first 1024MB data using sequential writes of 4KB many times. The reason this method is effective may be that the cleaning process is triggered by continuous writes to generate clean blocks in the background. Meanwhile, since writes are limited in the first 1024MB, newly generated clean blocks would not be quickly consumed, which helps refill the allocation pool slowly. Other tricks may also be able to recover performance, such as reinitializing the mapping table in the firmware.

6. SYSTEM IMPLICATIONS

Having comprehensively evaluated the performance of three representative SSDs through extensive experiments and analysis, we are now in a position to present some important system implications. We hope that our new findings are insightful to SSD hardware architects, operating system designers, and application users, and hope that our suggestions and guidance are effective for them to further improve efficiencies of SSD hardware, software management of SSDs, and data-intensive applications on SSDs. This section also provides an executive summary of our answers to the questions we raised at the beginning of this paper.

Reads on SSD: Read access latencies on SSDs are *not* always as uniform as commonly believed. Reads on low-end SSDs with no readahead are generally insensitive to access patterns. On higher-end SSDs, however, sequential reads are much more efficient than random reads. This indicates that many existing OS optimizations, such as file-level readahead, will still be effective and beneficial even when migrating to an SSD-based storage. For SSD manufacturers, conducting readahead in SSD firmware is a cost efficient design, which can significantly boost read performance with low cost. More importantly, since most existing file systems have already taken a lot of effort to organize sequential read requests for performance optimization, hardware support on SSDs can be especially effective. For application designers and practitioners, they may believe that high performance can be naturally achieved on SSDs, even without considering data placement. This assumption has to be corrected. Optimizing data placement on SSD is still needed at all levels for improving read performance.

Writes on SSD: As expected, writes on low-end SSDs are strongly correlated to access patterns. Random writes in a large storage space can lead to excessively high latency and make the storage system nearly unusable. However, on higher-end SSDs equipped with a disk cache, write performance is exceptionally good and nearly independent of access patterns. The indication is two-fold. On the one hand, operating system designers should still be careful about random write performance on low-end SSDs. An OS kernel can intentionally organize large and sequential writes to improve write performance. For example, the file system can schedule write-back conservatively with a long interval for clustering writes. Also, flushing out dirty data in a large granularity would be an option. On the other hand, random writes may not continue to be a critical issue on high-end SSDs. Considering the technical trend that a disk cache would become a standard component on SSDs, future research work should not continue to assume that random writes are excessively slow. Instead, avoiding the worst case, such as fragmentation, is worth further research.

Disk Cache: A disk cache can significantly improve performance for both reads and writes in an SSD. With a large cache, high-latency writes can be effectively hidden, and an MLC-based SSD can achieve performance comparable to an SLC-based SSD in most cases. SSD manufacturers should consider a disk cache as a cost-effective configuration and integrate it as a standard component in SSDs, even on low-end ones. With a disk cache, higher-level components, such as OS kernels, can be relieved of optimizing writes specifically for SSDs, and system complexity can be reduced.

Interactions between reads and writes: Reads and writes on SSDs have a surprisingly high interference with each other, even when they are submitted in sequence rather than in parallel. This indicates that upper-level layers should carefully separate read and write streams by clustering different types of requests. Actually, some existing I/O schedulers in OS kernels already manage reads and writes separately, which would be effective on SSDs. Moreover, application designers should attempt to avoid generating interleaved read and write operations. For example, when processing many files in a directory, we should avoid reading and updating only one individual file each time.

Background operations on SSDs: We observed many background operations on SSDs, as well as their performance impact on foreground jobs, especially on reads. Since many operations (e.g. delayed write) must be done sooner or later, if the arrival rate of requests is very high, the performance impact is essentially inevitable. A sophisticated firmware may mitigate the problem. For example, we can estimate the idle period and avoid scheduling long-latency operations (e.g. erase) during a busy time. Moreover, a large write-back buffer can hide the interference more effectively. On the OS and application level, avoiding an interleaved traffic of reads and writes, as aforementioned, is a wise choice.

Internal Fragmentation: On the two higher-end SSDs, we observed significant performance degradation caused by internal fragmentation. In such a condition, the SLC-based SSD obviously outperforms the MLC-based SSD. Although the excessively intensive random writes, which are generated by *striker* in our experiments, rarely happen in practice, it warns that random writes still need to be carefully handled, even on high-end SSDs. Many optimizations can be done in the operating system. For example, some sort of throttling mechanism can be incorporated to intentionally slow down writes into SSD to create more opportunity to cluster consecutive small writes. In addition, spatial locality of writes becomes important. For example, when selecting dirty pages to evict from the buffer cache, intentionally selecting pages that can be grouped sequentially is a wise choice. In order to solve this problem, SSD manufacturers should design more efficient mapping algorithms, and over-provisioning a sufficient number of clean blocks can alleviate the problem. As a short-term solution, a firmware-level defragmentation tool, which can automatically reorganize the page layout inside SSD, is highly desirable. For practitioners, selecting high-end SLC-based SSDs, which show relatively better perfor-

mance under extreme conditions, is a wise choice to handle enterprise workloads with intensive small writes.

Comparing with HDD: If we compare an SSD with a hard disk, the SSD would not necessarily win in all categories. Besides higher price and smaller capacity, even the performance of an SSD is not always better than an HDD. When handling random reads, it is confirmed that SSDs do show much higher bandwidth than a typical 7200RPM hard disk. However, when handling other workloads, the performance gap is much smaller. Under certain workloads, such as random writes in a large area, a low-end SSD could be even substantially slower than a hard disk.

More importantly, we should note that the essential difference between SSD and HDD is not just the performance but the different internal mechanisms to manage data blocks. On HDDs, each logical block is statically mapped on the physical medium, which leads to a simplified management and relatively repeatable performance. On SSDs, the physical location of logical blocks depends greatly on the write order and could change over time (e.g. during cleaning or wear-leveling). This naturally leads to many performance uncertainties and dynamics. In general, we believe it is still too early to conclude that SSD will replace HDD very soon, especially when considering that the 'affordable' SSDs still need significant improvement. System practitioners should also carefully consider many issues exposed in this work before integrating SSDs into the storage hierarchy.

7. RELATED WORK

Flash memory based storage has been actively researched for many years, and a large body of research work is focused on addressing the 'no in-place write' problem of flash memory, either through an flash memory-based file system [4, 6, 17,34] or a flash translation layer (FTL) [12,16,22–24,26,30]. This pioneering research work serves as a concrete foundation in the SSD hardware design.

Recently, some research work has been conducted specifically on flash memory-based SSDs. For example, Agrawal et al. presented a detailed description about the hardware design of flash memory based SSD [3]. In addition to revealing the internal architecture of SSDs, they also reported performance data based on an augmented DiskSim simulator [8]. Similarly, Birrell et al. [5] also presented a hardware design of flash memory based SSD in detail. Concerning the RAM buffer in SSD, Kim and Ahn proposed a write buffer management scheme called BPLRU [25] in SSD hardware to mitigate the problem of high-cost random writes in flash memory. Gupta et al. [19] also proposed a selective caching method to cache a partial mapping table in limited RAM buffer to support efficient page-level mapping. Some papers also suggest to modify the existing interfaces to leverage the unique features of SSDs. For example, the JFTL [11] is designed to remove redundant writes in a journaling file system by directly transferring semantic information from the file system level to the SSD firmware. Transactional Flash [38] further suggests to integrate a transactional interface API to SSDs for supporting transactional operations in file systems and database systems.

Concerning the application of flash memory based SSDs, Sun® has designed a general architecture [31] to leverage the SSD as a secondary-level cache in the standard memory hierarchy. Aiming specifically at peta-scale storage systems, Caulfield et al. [9] proposed a flash memory based cluster to optimize performance and energy consumption for data-intensive applications. Recently, Narayanan et al. [37] presented an interesting model to analyze the potential of merging SSDs into enterprise storage systems from a cost-efficiency perspective. Their study suggests that SSD may not be as appropriate as expected for enterprise workloads.

SSDs are also received strong interest in the database community. Lee et al. [29] examined the performance potential for using SSDs to optimize operations in DBMS, such as hash join and temporary table creation. Lee and Moon proposed an in-page logging approach [28] to optimize write performance for flash-based DBMS. Koltsidas et al. [27] suggested to incorporate both flash and hard disks in a database system by placing read-intensive data on a flash disk and write-intensive data on a hard disk. In general, these SSD related solution papers are based on a common understanding about flash memory based SSD – random writes have poor performance. In our experiments, we confirmed that such assumption still holds on the low-end SSD. However, on the high-end SSDs, we did not see significant performance issues on random writes in normal conditions.

8. CONCLUSION

We have designed a set of comprehensive measurements on three representative, state-of-the-art SSDs fabricated by two major manufacturers. Our experimental results confirmed many well understood features of SSDs, such as the exceptional performance for handling random reads. At the same time, we also observed many unexpected performance issues in the dynamics, most of which are related to random writes. In general, our findings do show that significant advances have been made in SSD hardware design, providing high read access rates combined with reasonable write performance under many regular workloads. However, the substantial performance drop after a stress test showed that it is still too early to draw the firm conclusion that HDD will soon be replaced by SSD. For researchers on storage and operating systems, our work shows that SSD presents many challenges as well as opportunities.

Acknowledgments

We are grateful to our shepherd Prof. Pete Harrison and the anonymous reviewers for their constructive comments to improve the quality of this paper. We also thank our colleague Bill Bynum for reading this paper and his comments. We also thank Shuang Liang and Xiaoning Ding for many interesting and inspiring discussions during this work. This research was partially supported by the National Science Foundation under grants CCF-0620152 and CCF-072380.

9. REFERENCES

[1] Serial ATA revision 2.6. http://www.sata-io.org.

[2] SmartMedia specification. http://www.ssfdc.or.jp.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. of USENIX'08*, 2008.

[4] P. L. Barrett, S. D. Quinn, and R. A. Lipe. System for updating data stored on a flash-erasable, programmable, read-only memory (FEPROM) based upon predetermined bit value of indicating pointers. In *US Patent 5,392,427*, 1995.

[5] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. In *Microsoft Research Technical Report*, 2005.

[6] A. B. Bityutskiy. JFFS3 design issues. http://www.linux-mtd.infradead.org.

[7] Blktrace. http://linux.die.net/man/8/blktrace.

[8] J. Bucy, J. Schindler, S. Schlosser, and G. Ganger. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim.

[9] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proc. of ASPLOS'09*, 2009.

[10] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proc. of ISLPED'06*, 2006.

[11] H. J. Choi, S. Lim, and K. H. Park. JFTL: a flash translation layer based on a journal remapping for flash memory. In *ACM Transactions on Storage*, volume 4, Jan 2009.

[12] T. Chung, D. Park, S. Park, D. Lee, S. Lee, and H. Song. System software for flash memory: a survey. In *Proc. of ICEUC'06*, 2006.

[13] T. Claburn. Google plans to use Intel SSD storage in servers. http://www.informationweek.com/news/ storage/systems/showArticle.jhtml?articleID=207602745.

[14] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided resynchronization for software RAID. In *Proc. of FAST'05*, 2005.

[15] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proc. of USENIX'07*, 2007.

[16] E. Gal and S. Toledo. Algorithms and data structures for flash memories. In *Computing Survey'05*, 2005.

[17] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *Proc. of USENIX'05*, 2005.

[18] C. Gniady, Y. C. Hu, and Y. Lu. Program counter based techniques for dynamic power management. In *Proc. of HPCA'04*, 2004.

[19] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. of ASPLOS'09*, 2009.

[20] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proc. of ISCA'03*, 2003.

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

[22] J. Kang, H. Jo, J. Kim, and J. Lee. A superblock-based flash translation layer for NAND flash memory. In *Proc. of ICES'06*, 2006.

[23] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proc. of USENIX Winter*, 1995.

[24] B. Kim and G. Lee. Method of driving remapping in flash memory and flash memory architecture suitable therefore. In *US Patent No 6,381,176*, 2002.

[25] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proc. of FAST'08*, 2008.

[26] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. In *IEEE Transactions on Consumer Electronics*, volume 48(2):366-375, 2002.

[27] I. Koltsidas and S. Viglas. Flashing up the storage layer. In *Proc. of VLDB'08*, 2008.

[28] S. Lee and B. Moon. Design of flash-based DBMS: An in-page logging approach. In *Proc. of SIGMOD'07*, 2007.

[29] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. A case for flash memory SSD in enterprise database applications. In *Proc. of SIGMOD'08*, 2008.

[30] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. A log buffer based flash translation layer using fully associative sector translation. In *IEEE Tran. on Embedded Computing Systems*, 2007.

[31] A. Leventhal. Flash storage memory. In *Communications of the ACM*, volume 51, July 2008.

[32] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-Miner: Mining block correlations in storage systems. In *Proc. of FAST'04*, 2004.

[33] M-Systems. Two technologies compared: NOR vs NAND. In *White Paper*, 2003.

[34] C. Manning. YAFFS: Yet another flash file system. http://www.aleph1.co.uk/yaffs, 2004.

[35] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. In *ACM Transactions on Storage*, volume 4, May 2008.

[36] M. Mesnier. Intel open storage toolkit. http://www.sourceforge.org/projects/intel-iscsi.

[37] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating enterprise storage to SSDs: analysis of tradeoffs. In *Proc. of EuroSys'09*, 2009.

[38] V. Prabhakaran, T. L. Rodeheffeer, and L. Zhou. Transactional flash. In *Proc. of OSDI'08*, 2008.

[39] D. Robb. Intel sees gold in solid state storage. http://www.enterprisestorageforum.com/technology/ article.php/3782826, 2008.

[40] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems*, volume 10(1):26-52, 1992.

[41] Samsung Elec. Datasheet (K9LBG08U0M). 2007.

[42] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean too you? In *Proc. of FAST'07*, 2007.

[43] C. Wu, L. Chang, and T. Kuo. An efficient B-Tree layer for flash memory storage systems. In *Proc. of Int. Conf. on Real-Time and Embeded Computing Systems and Applications 2003*, 2003.

Notes

[1]When connecting through a SATA controller card supporting parallel I/Os, SSD-M and SSD-H can achieve even higher bandwidths for random workloads and show more significant performance advantages over the low-end SSD. Since handling parallel I/O jobs is out of the scope of this paper, we still use the on-board SATA connectors in our experiments to avoid extra overhead that may be introduced by the controller card.

[2] When filling an SSD for the first time, the mapping table is being created and the number of clean blocks that are available for allocation decreases. During this process, the SSD may experience performance degradation.

Differentiated Storage Services

Michael Mesnier, Feng Chen, Tian Luo*

Intel Labs
Intel Corporation
Hillsboro, OR

Jason B. Akers

Storage Technologies Group
Intel Corporation
Hillsboro, OR

ABSTRACT

We propose an I/O classification architecture to close the widening semantic gap between computer systems and storage systems. By *classifying* I/O, a computer system can request that different classes of data be handled with different storage system *policies*. Specifically, when a storage system is first initialized, we assign performance policies to predefined classes, such as the filesystem journal. Then, online, we include a classifier with each I/O command (e.g., SCSI), thereby allowing the storage system to enforce the associated policy for each I/O that it receives.

Our immediate application is caching. We present filesystem prototypes and a database proof-of-concept that classify all disk I/O — with very little modification to the filesystem, database, and operating system. We associate caching policies with various classes (e.g., large files shall be evicted before metadata and small files), and we show that end-to-end file system performance can be improved by over a factor of two, relative to conventional caches like LRU. And caching is simply one of many possible applications. As part of our ongoing work, we are exploring other classes, policies and storage system mechanisms that can be used to improve end-to-end performance, reliability and security.

Categories and Subject Descriptors

D.4 [**Operating Systems**]; D.4.2 [**Storage Management**]: [Storage hierarchies]; D.4.3 [**File Systems Management**]: [File organization]; H.2 [**Database Management**]

General Terms

Classification, quality of service, caching, solid-state storage

1. INTRODUCTION

The block-based storage interface is arguably the most stable interface in computer systems today. Indeed, the primary read/write functionality is quite similar to that used by the first commercial disk drive (IBM RAMAC, 1956). Such stability has allowed computer and storage systems to evolve in an independent yet interoperable manner, but at at a cost – it is difficult for computer systems to optimize for increasingly complex storage system internals, and storage systems do not have the semantic information (e.g., on-disk FS and DB data structures) to optimize independently.

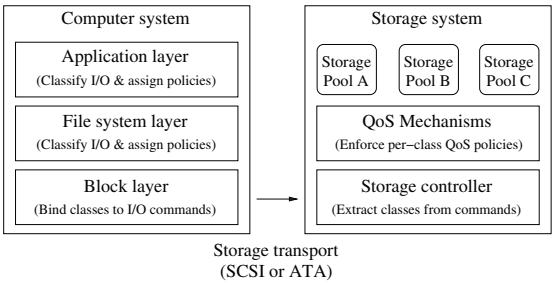


Figure 1: High-level architecture

By way of analogy, shipping companies have long recognized that *classification* is the key to providing differentiated service. Boxes are often classified (kitchen, living room, garage), assigned different policies (deliver-first, overnight, priority, handle-with-care), and thusly treated differently by a shipper (hand-carry, locked van, truck). Separating classification from policy allows customers to pack and classify (label) their boxes once; the handling policies can be assigned on demand, depending on the shipper. And separating policy from mechanism frees customers from managing the internal affairs of the shipper, like which pallets to place their shipments on.

In contrast, modern computer systems expend considerable effort attempting to manage storage system internals, because different classes of data often need different levels of service. As examples, the "middle" of a disk can be used to reduce seek latency, and the "outer tracks" can be used to improve transfer speeds. But, with the increasing complexity of storage systems, these techniques are losing their effectiveness — and storage systems can do very little to help because they lack the semantic information to do so.

We argue that computer and storage systems should operate in the same manner as the shipping industry — by utilizing I/O classification. In turn, this will enable storage systems to enforce per-class QoS policies. See Figure 1.

*The Ohio State University

SOSP '11, October 23-26, 2011, Cascais, Portugal.

Differentiated Storage Services is such a classification framework: I/O is classified in the computer system (e.g., filesystem journal, directory, small file, database log, index, ...), policies are associated with classes (e.g., an FS journal requires low-latency writes, and a database index requires low-latency reads), and mechanisms in the storage system enforce policies (e.g., a cache provides low latency).

Our approach only slightly modifies the existing block interface, so eventual standardization and widespread adoption are practical. Specifically, we modify the OS block layer so that every I/O request carries a classifier. We copy this classifier into the I/O command (e.g., SCSI CDB), and we specify policies on classes through the management interface of the storage system. In this way, a storage system can provide block-level differentiated services (performance, reliability, or security) — and do so on a class-by-class basis. The storage system does not need any knowledge of computer system internals, nor does the computer system need knowledge of storage system internals.

Classifiers describe what the data is, and policies describe how the data is to be managed. Classifiers are handles that the computer system can use to assign policies and, in our SCSI-based prototypes, a classifier is just a number used to distinguish various filesystem classes, like metadata versus data. We also have user-definable classes that, for example, a database can use to classify I/O to specific database structures like an index. Defining the classes (the classification scheme) should be an infrequent operation that happens once for each filesystem or database of interest.

In contrast, we expect that policies will vary across storage systems, and that vendors will differentiate themselves through the policies they offer. As examples, storage system vendors may offer service levels (platinum, gold, silver, bronze), performance levels (bandwidth and latency targets), or relative priority levels (the approach we take in this paper). A computer system must map its classes to the appropriate set of policies, and I/O classification provides a convenient way to do this dynamically when a filesystem or database is created on a new storage system. Table 1 shows a hypothetical mapping of filesystem classes to available performance policies, for three different storage systems.

Beyond performance, there could be numerous other policies that one might associate with a given class, such as replication levels, encryption and integrity policies, perhaps even data retention policies (e.g., secure erase). Rather than attempt to send all of this policy information along with each I/O, we simply send a classifier. This will make efficient use of the limited space in an I/O command (e.g., SCSI has 5 bits that we use as a classifier). In the storage system the classifier can be associated with any number of policies.

We begin with a priority-based performance policy for cache management, specifically for non-volatile caches composed of solid-state drives (SSDs). That is, to each FS and DB class we assign a caching policy (a relative priority level). In practice, we assume that the filesystem or database vendor, perhaps in partnership with the storage system vendor, will provide a default priority assignment that a system administrator may choose to tune.

FS Class	Vendor A: Service levels	Vendor B: Perf. targets	Vendor C: Priorities
Metadata	Platinum	Low lat.	0
Journal	Gold	Low lat.	0
Small file	Silver	Low lat.	1
Large file	Bronze	High BW	2

Table 1: An example showing FS classes mapped to various performance policies. This paper focuses on priorities; lower numbers are higher priority.

We present prototypes for Linux Ext3 and Windows NTFS, where I/O is classified as metadata, journal, directory, or file, and file I/O is further classified by the file size (e.g., ≤4KB ≤16KB, ..., >1GB). We assign a caching priority to each class: metadata, journal, and directory blocks are highest priority, followed by regular file data. For the regular files, we give small files higher priority than large ones.

These priority assignments reflect our goal of reserving cache space for metadata and small files. To this end, we introduce two new block-level caching algorithms: *selective allocation* and *selective eviction*. Selective allocation uses the priority information when allocating I/O in a cache, and selective eviction uses this same information during eviction. The end-to-end performance improvements of selective caching are considerable. Relative to conventional LRU caching, we improve the performance of a file server by 1.8x, an e-mail server by 2x, and metadata-intensive FS utilities (e.g., `find` and `fsck`) by up to 6x. Furthermore, a TCO analysis by Intel IT Research shows that priority-based caching can reduce caching costs by up to 50%, as measured by the acquisition cost of hard drives and SSDs.

It is important to note that in both of our FS prototypes, we do not change which logical blocks are being accessed; we simply classify I/O requests. Our design philosophy is that the computer system continues to see a single logical volume and that the I/O into that volume be classified. In this sense, classes can be considered "hints" to the storage system. Storage systems that know how to interpret the hints can optimize accordingly, otherwise they can be ignored. This makes the solution backward compatible, and therefore suitable for legacy applications.

To further show the flexibility of our approach, we present a proof-of-concept classification scheme for PostgreSQL [33]. Database developers have long recognized the need for intelligent buffer management in the database [10] and in the operating system [45]; buffers are often classified by type (e.g., index vs. table) and access pattern (e.g., random vs. sequential). To share this knowledge with the storage system, we propose a POSIX file flag (`O_CLASSIFIED`). When a file is opened with this flag, the OS extracts classification information from a user-provided data buffer that is sent with each I/O request and, in turn, binds the classifier to the outgoing I/O command. Using this interface, we can easily classify all DB I/O, with only minor modification to the DB and the OS. This same interface can be used by any application. Application-level classes will share the classification space with the filesystem — some of the classifier bits can be reserved for applications, and the rest for the filesystem.

This paper is organized as follows. Section 2 motivates the need for Differentiated Storage Services, highlighting the shortcomings of the block interface and building a case for block-level differentiation. Alternative designs, not based on I/O classification, are discussed. We present our design in Section 3, our FS prototypes and DB proof-of-concept in Section 4, and our evaluation in Section 5. Related work is presented in Section 6, and we conclude in Section 7.

2. BACKGROUND & MOTIVATION

The contemporary challenge motivating Differentiated Storage Services is the integration of SSDs, as caches, into conventional disk-based storage systems. The fundamental limitation imposed by the block layer (lack of semantic information) is what makes effective integration so challenging. Specifically, the block layer abstracts computer systems from the details of the underlying storage system, and *vice versa*.

2.1 Computer system challenges

Computer system performance is often determined by the underlying storage system, so filesystems and databases must be smart in how they allocate on-disk data structures. As examples, the journal (or log) is often allocated in the middle of a disk drive to minimize the average seek distance [37], files are often created close to their parent directories, and file and directory data are allocated contiguously whenever possible. These are all attempts by a computer system to obtain some form differentiated service through intelligent block allocation.

Unfortunately, the increasing complexity of storage systems is making intelligent allocation difficult. Where is the "middle" of the disk, for example, when a filesystem is mounted atop a logical volume with multiple devices, or perhaps a hybrid disk drive composed of NAND and shingled magnetic recording? Or, how do storage system caches influence the latency of individual read/write operations, and how can computer systems reliably manage performance in the context of these caches? One could use models [27, 49, 52] to predict performance, but if the predicted performance is undesirable there is very little a computer system can do to change it.

In general, computer systems have come to expect only best-effort performance from their storage systems. In cases where performance must be guaranteed, dedicated and over-provisioned solutions are deployed.

2.2 Storage system challenges

Storage systems already offer differentiated service, but only at a coarse granularity (logical volumes). Through the management interface of the storage system, administrators can create logical volumes with the desired capacity, reliability, and performance characteristics — by appropriately configuring RAID and caching.

However, before an I/O enters the storage system, valuable semantic information is stripped away at the OS block layer, such as user, group, application, and process information. And, any information regarding on-disk structures is obfuscated. This means that all I/O receives the same treatment within the logical volume.

For a storage system to provide any meaningful optimization within a volume, it must have semantic computer system information. Without help from the computer system, this can be very difficult to get. Consider, for example, that a filename could influence how a file is cached [26], and what would be required for a storage system to simply determine the the name of a file associated with a particular I/O. Not only would the storage system need to understand the on-disk metadata structures of the filesystem, particularly the format of directories and their filenames, but it would have to track all I/O requests that modify these structures. This would be an extremely difficult and potentially fragile process. Expecting storage systems to retain sufficient and up-to-date knowledge of the on-disk structures for each of its attached computer systems may not be practical, or even possible, to realize in practice.

2.3 Attempted solutions & shortcomings

Three schools of thought have emerged to better optimize the I/O between a computer and storage system. Some show that computer systems can obtain more knowledge of storage system internals and use this information to guide block allocation [11, 38]. In some cases, this means managing different storage volumes [36], often foregoing storage system services like RAID and caching. Others show that storage systems can discover more about on-disk data structures and optimize I/O accesses to these structures [9, 41, 42, 43]. Still others show that the I/O interface can evolve and become more expressive; object-based storage and type-safe disks fall into this category [28, 40, 58].

Unfortunately, none of these approaches has gained significant traction in the industry. First, increasing storage system complexity is making it difficult for computer systems to reliably gather information about internal storage structure. Second, increasing computer system complexity (e.g., virtualization, new filesystems) is creating a moving target for semantically-aware storage systems that learn about on-disk data structures. And third, although a more expressive interface could address many of these issues, our industry has developed around a block-based interface, for better or for worse. In particular, filesystem and database vendors have a considerable amount of intellectual property in how blocks are managed and would prefer to keep this functionality in software, rather than offload to the storage system through a new interface.

When a new technology like solid-state storage emerges, computer system vendors prefer to innovate above the block level, and storage system vendors below. But, this tug-of-war has no winner as far as applications are concerned, because considerable optimization is left on the table.

We believe that a new approach is needed. Rather than teach computer systems about storage system internals, or *vice versa*, we can have them agree on shared, block-level goals — and do so through the existing storage interfaces (SCSI and ATA). This will not introduce a disruptive change in the computer and storage systems ecosystem, thereby allowing computer system vendors to innovate above the block level, and storage system vendors below. To accomplish this, we require a means by which block-level goals can be communicated with each I/O request.

3. DESIGN

Differentiated Storage Services closes the semantic gap between computer and storage systems, but does so in a way that is practical in an industry built around blocks. The problem is not the block interface, *per se*, but a lack of information as to how disk blocks are being used.

We must careful, though, to not give a storage system too much information, as this could break interoperability. So, we simply *classify* I/O requests and communicate block-level goals (policies) for each class. This allows storage systems to provide meaningful levels of differentiation, without requiring that detailed semantic information be shared.

3.1 Operating system requirements

We associate a classifier with every block I/O request in the OS. In UNIX and Windows, we add a classification field to the OS data structure for block I/O (the Linux "BIO," and the Windows "IRP") and we copy this field into the actual I/O command (SCSI or ATA) before it is sent to the storage system. The expressiveness of this field is only limited by its size, and in Section 4 we present a SCSI prototype where a 5-bit SCSI field can classify I/O in up to 32 ways.

In addition to adding the classifier, we modify the OS I/O scheduler, which is responsible for coalescing contiguous I/O requests, so that requests with different classifiers are never coalesced. Otherwise, classification information would be lost when two contiguous requests with different classifiers are combined. This does reduce a scheduler's ability to coalesce I/O, but the benefits gained from providing differentiated service to the uncoalesced requests justify the cost, and we quantify these benefits in Section 5.

The OS changes needed to enable filesystem I/O classification are minor. In Linux, we have a small kernel patch. In Windows, we use closed-source filter drivers to provide the same functionality. Section 4 details these changes.

3.2 Filesystem requirements

First, a filesystem must have a *classification scheme* for its I/O, and this is to be designed by a developer that has a good understanding of the on-disk FS data structures and their performance requirements. Classes should represent blocks with similar goals (e.g., journal blocks, directory blocks, or file blocks); each class has a unique ID. In Section 4, we present our prototype classification schemes for Linux Ext3 and Windows NTFS.

Then, the filesystem developer assigns a policy to each class; refer back to the hypothetical examples given in Table 1. How this policy information is communicated to the storage system can be vendor specific, such as through an administrative GUI, or even standardized. The Storage Management Initiative Specification (SMI-S) is one possible avenue for this type of standardization [3]. As a reference policy, also presented in Section 4, we use a priority-based performance policy for storage system cache management.

Once mounted, the filesystem classifies I/O as per the classification scheme. And blocks may be reclassified over time. Indeed, block reuse in the filesystem (e.g., file deletion or defragmentation) may result in frequent reclassification.

3.3 Storage system requirements

Upon receipt of a classified I/O, the storage system must extract the classifier, lookup the policy associated with the class, and enforce the policy using any of its internal mechanisms; legacy systems without differentiated service can ignore the classifier. The mechanisms used to enforce a policy are completely vendor specific, and in Section 4 we present our prototype mechanism (priority-based caching) that enforces the FS-specified performance priorities.

Because each I/O carries a classifier, the storage system does not need to record the class of each block. Once allocated from a particular storage pool, the storage system is free to discard the classification information. So, in this respect, Differentiated Storage Services is a stateless protocol. However, if the storage system wishes to later move blocks across storage pools, or otherwise change their QoS, it must do so in an informed manner. This must be considered, for example, during de-duplication. Blocks from the same allocation pool (hence, same QoS) can be de-duplicated. Blocks from different pools cannot.

If the classification of a block changes due to block re-use in the filesystem, the storage system must reflect that change internally. In some cases, this may mean moving one or more blocks across storage pools. In the case of our cache prototype, a classification change can result in cache allocation, or the eviction of previously cached blocks.

3.4 Application requirements

Applications can also benefit from I/O classification; two good examples are databases and virtual machines. To allow for this, we propose a new file flag `O_CLASSIFIED`. When a file is opened with this flag, we overload the POSIX scatter/gather operations (`readv` and `writev`) to include one extra list element. This extra element points to a 1-byte user buffer that contains the classification ID of the I/O request. Applications not using scatter/gather I/O can easily convert each I/O to a 2-element scatter/gather list. Applications already issuing scatter/gather need only create the additional element.

Next, we modify the OS virtual file system (VFS) in order to extract this classifier from each `readv()` and `writev()` request. Within the VFS, we know to inspect the file flags when processing each scatter/gather operation. If a file handle has the `O_CLASSIFIED` flag set, we extract the I/O classifier and reduce the scatter/gather list by one element. The classifier is then bound to the kernel-level I/O request, as described in Section 3.1. Currently, our user-level classifiers override the FS classifiers. If a user-level class is specified on a file I/O, the filesystem classifiers will be ignored.

Without further modification to POSIX, we can now explore various ways of differentiating user-level I/O. In general, any application with complex, yet structured, block relationships [29] may benefit from user-level classification. In this paper, we begin with the database and, in Section 4, present a proof-of-concept classification scheme for PostgreSQL [33]. By simply classifying database I/O requests (e.g., user tables versus indexes), we provide a simple way for storage systems to optimize access to on-disk database structures.

4. IMPLEMENTATION

We present our implementations of Differentiated Storage Services, including two filesystem prototypes (Linux Ext3 and Windows NTFS), one database proof-of-concept (Linux PostgreSQL), and two storage system prototypes (SW RAID and iSCSI). Our storage systems implement a priority-based performance policy, so we map each class to a priority level (refer back to Table 1 for other possibilities). For the FS, the priorities reflect our goal to reduce small random access in the storage system, by giving small files and metadata higher priority than large files. For the DB, we simply demonstrate the flexibility of our approach by assigning caching policies to common data structures (indexes, tables, and logs).

4.1 OS changes needed for FS classification

The OS must provide in-kernel filesystems with an interface for classifying each of their I/O requests. In Linux, we do this by adding a new classification field to the FS-visible kernel data structure for disk I/O (`struct buffer_head`). This code fragment illustrates how Ext3 can use this interface to classify the OS disk buffers into which an inode (class 5 in this example) will be read:

```
bh->b_class = 5;     /* classify inode buffer */
submit_bh(READ, bh); /* submit read request   */
```

Once the disk buffers associated with an I/O are classified, the OS block layer has the information needed to classify the block I/O request used to read/write the buffers. Specifically, it is in the implementation of `submit_bh` that the generic block I/O request (the BIO) is generated, so it is here that we copy in the FS classifier:

```
int submit_bh(int rw, struct buffer_head * bh) {
    ...
    bio->bi_class = bh->b_class /* copy in class */
    submit_bio(rw, bio);        /* issue read    */
    ...
    return ret;
}
```

Finally, we copy the classifier once again from the BIO into the 5-bit, vendor-specific *Group Number* field in byte 6 of the SCSI CDB. This one-line change is all that is need to enable classification at the SCSI layer:

```
SCpnt->cmnd[6] = SCpnt->request->bio->bi_class;
```

These 5 bits are included with each WRITE and READ command, and we can fill this field in up to 32 different ways (2^5). An additional 3 reserved bits could also be used to classify data, allowing for up to 256 classifiers (2^8), and there are ways to grow even beyond this if necessary (e.g., other reserved bits, or extended SCSI commands).

In general, adding I/O classification to an existing OS is a matter of tracking an I/O as it proceeds from the filesystem, through the block layer, and down to the device drivers. Whenever I/O requests are copied from one representation to another (e.g., from a buffer head to a BIO, or from a BIO to a SCSI command), we must remember to copy the classifier. Beyond this, the only other minor change is to the I/O scheduler which, as previously mentioned, must be modified so that it only coalesces requests that carry the same classifier.

Block layer	LOC	Change made
bio.h	1	Add classifier
blkdev.h	1	Add classifier
buffer_head.h	13	Add classifier
bio.c	2	Copy classifier
buffer.c	26	Copy classifier
mpage.c	23	Copy classifier
bounce.c	1	Copy classifier
blk-merge.c	28	Merge I/O of same class
direct-io.c	60	Classify file sizes
sd.c	1	Insert classifier into CDB

Table 2: Linux 2.6.34 files modified for I/O classification. Modified lines of code (LOC) shown.

Overall, adding classification to the Linux block layer requires that we modify 10 files (156 lines of code), which results in a small kernel patch. Table 2 summarize the changes. In Windows, the changes are confined to closed-source filter drivers. No kernel code needs to be modified because, unlike Linux, Windows provides a stackable filter driver architecture for intercepting and modifying I/O requests.

4.2 Filesystem prototypes

A filesystem developer must devise a classification scheme and assign storage policies to each class. The goals of the filesystem (performance, reliability, or security) will influence how I/O is classified and policies are assigned.

4.2.1 Reference classification scheme

The classification schemes for the Linux Ext3 and Windows NTFS are similar, so we only present Ext3. Any number of schemes could have been chosen, and we begin with one well-suited to minimizing random disk access in the storage system. The classes include metadata blocks, directory blocks, journal blocks, and regular file blocks. File blocks are further classified by the file size ($\leq$4KB, $\leq$16KB, $\leq$64KB, $\leq$256KB, ..., $\leq$1GB, >1GB) — 11 file size classes in total.

The goal of our classification scheme is to provide the storage system with a way of prioritizing which blocks get cached and the eviction order of cached blocks. Considering the fact that metadata and small files can be responsible for the majority of the disk seeks, we classify I/O in such a way that we can separate these random requests from large-file requests that are commonly accessed sequentially. Database I/O is an obvious exception and, in Section 4.3 we introduce a classification scheme better suited for the database.

Table 3 (first two columns) summarizes our classification scheme for Linux Ext3. Every disk block that is written or read falls into exactly one class. Class 0 (unclassified) occurs when I/O bypasses the Ext3 filesystem. In particular, all I/O created during filesystem creation (`mkfs`) is unclassified, as there is no mounted filesystem to classify the I/O. The next 5 classes (superblocks through indirect data blocks) represent filesystem metadata, as classified by Ext3 after it has been mounted. Note, the unclassified metadata blocks will be re-classified as one of these metadata types when they are first accessed by Ext3. Although we differentiate metadata classes 1 through 5, we could have combined them into one class. For example, it is not critical that we

Ext3 Class	Class ID	Priority
Superblock	1	0
Group Descriptor	2	0
Bitmap	3	0
Inode	4	0
Indirect block	5	0
Directory entry	6	0
Journal entry	7	0
File <= 4KB	8	1
File <= 16KB	9	2
...	...	...
File > 1GB	18	11
Unclassified	0	12

Table 3: Reference classes and caching priorities for Ext3. Each class is assigned a unique SCSI Group Number and assigned a priority (0 is highest).

Ext3	LOC	Change made
balloc.c	2	Classify block bitmaps
dir.c	2	Classify inodes tables
ialloc.c	2	Classify inode bitmaps
inode.c	94	Classify indirect blocks, inodes, dirs, and file sizes
super.c	15	Classify superblocks, journal blocks, and group descriptors
commit.c	3	Classify journal I/O
journal.c	6	Classify journal I/O
revoke.c	2	Classify journal I/O

Table 4: Ext3 changes for Linux 2.6.34.

differentiate superblocks and block bitmaps, as these structures consume very little disk (and cache) space. Still, we do this for illustrative purposes and system debugging.

Continuing, class 6 represents directory blocks, class 7 journal blocks, and 8-18 are the file size classes. File size classes are only approximate. As a file is being created, the file size is changing while writes are being issued to the storage system; files can also be truncated. Subsequent I/O to a file will reclassify the blocks with the latest file size.

Approximate file sizes allow the storage system to differentiate small files from large files. For example, a storage system can cache all files 1MB or smaller, by caching all the file blocks with a classification up to 1MB. The first 1MB of files larger than 1MB may also fall into this category until they are later reclassified. This means that small files will fit entirely in cache, and large files may be partially cached with the remainder stored on disk.

We classify Ext3 using 18 of the 32 available classes from a 5-bit classifier. To implement this classification scheme, we modify 8 Ext3 files (126 lines of code). Table 4 summarizes our changes.

The remaining classes (19 through 31) could be used in other ways by the FS (e.g., text vs. binary, media file, bootable, read-mostly, or hot file), and we are exploring these as part of our future work. The remaining classes could also be used by user-level applications, like the database.

4.2.2 Reference policy assignment

Our prototype storage systems implement 16 priorities; to each class we assign a priority (0 is the highest). Metadata, journal, and directory blocks are highest priority, followed by the regular file blocks. 4KB files are higher priority than 16KB files, and so on. Unclassified I/O, or the unused metadata created during file system creation, is assigned the lowest priority. For this mapping, we only require 13 priorities, so 3 of the priority levels (13-15) are unused. See Table 3.

This priority assignment is specifically tuned for a file server workload (e.g., SPECsfs), as we will show in Section 5, and reflects our bias to optimize the filesystem for small files and metadata. Should this goal change, the priorities could be set differently. Should the storage system offer policies other than priority levels (like those in Table 1), the FS classes would need to be mapped accordingly.

4.3 Database proof-of-concept

In addition to the kernel-level I/O classification interface described in Section 4.1, we provide a POSIX interface for classifying user-level I/O. The interface builds on the scatter/gather functionality already present in POSIX.

Using this new interface, we classify all I/O from the PostgreSQL open source database [33]. As with FS classification, user-level classifiers are just numbers used to distinguish the various I/O classes, and it is the responsibility of the application (a DB in this case) to design a classification scheme and associate storage system policies with each class.

4.3.1 A POSIX interface for classifying DB I/O

We add an additional scatter/gather element to the POSIX `readv` and `writev` system calls. This element points to a user buffer that contains a classifier for the given I/O. To use our interface, a file is opened with the flag `O_CLASSIFIED`. When this flag is set, the OS will assume that all scatter/gather operations contain $1 + n$ elements, where the first element points to a classifier buffer and the remaining n elements point to data buffers. The OS can then extract the classifier buffer, bind the classifier to the kernel-level I/O (as described in Section 4.1), reduce the number of scatter gather elements by one, and send the I/O request down to the filesystem. Table 5 summarizes the changes made to the VFS to implement user-level classification. As with kernel-level classification, this is a small kernel patch.

The following code fragment illustrates the concept for a simple program with a 2-element gathered-write operation:

```
unsigned char class = 23;  /* a class ID */
int fd = open("foo", O_RDWR|O_CLASSIFIED);
struct iovec iov[2];       /* an sg list */

iov[0].iov_base = &class; iov[0].iov_len = 1;
iov[1].iov_base = "Hello, world!";
iov[1].iov_len = strlen("Hello, world!");
rc = writev(fd, iov, 2);   /* 2 elements */
close(fd);
```

The filesystem will classify the file size as described in Section 4.2, but we immediately override this classification with the user-level classification, if it exists. Combining user-level and FS-level classifiers is an interesting area of future work.

OS file	LOC	Change made
filemap.c	50	Extract class from sg list
mm.h	4	Add classifier to readahead
readahead.c	22	Add classifier to readahead
mpage.h	1	Add classifier to page read
mpage.c	5	Add classifier to page read
fs.h	1	Add classifier to FS page read
ext3/inode.c	2	Add classifier to FS page read

Table 5: Linux changes for user-level classification.

DB class	Class ID
Unclassified	0
Transaction Log	19
System table	20
Free space map	21
Temporary table	22
Random user table	23
Sequential user table	24
Index file	25
Reserved	26-31

Table 6: A classification scheme PostgreSQL. Each class is assigned a unique number. This number is copied into the 5-bit SCSI Group Number field in the SCSI WRITE and READ commands.

4.3.2 A DB classification scheme

Our proof-of-concept PostgreSQL classification scheme includes the transaction log, system tables, free space maps, temporary tables, user tables, and indexes. And we further classify the user tables by their access pattern, which the PostgreSQL database already identifies, internally, as random or sequential. Passing this access pattern information to the storage system avoids the need for (laborious) sequential stream detection.

Table 6 summarizes our proof-of-concept DB classification scheme, and Table 7 shows the minor changes required of PostgreSQL. We include this database example to demonstrate the flexibility of our approach and the ability to easily classify user-level I/O. How to properly assign block-level caching priorities for the database is part of our current research, but we do share some early results in Section 5 to demonstrate the performance potential.

4.4 Storage system prototypes

With the introduction of solid-state storage, storage system caches have increased in popularity. Examples include LSI's CacheCade and Adaptec's MaxIQ. Each of these systems use solid-state storage as a persistent disk cache in front of a traditional disk-based RAID array.

We create similar storage system caches and apply the necessary modification to take advantage of I/O classification. In particular, we introduce two new caching algorithms: *selective allocation* and *selective eviction*. These algorithms inspect the relative priority of each I/O and, as such, provide a mechanism by which computer system performance policies can be enforced in a storage system. These caching algorithms build upon a baseline cache, such as LRU.

DB file	LOC	Change made
rel.h	6	Pass classifier to storage manager
xlog.c	7	Classify transaction log
bufmgr.c	17	Classify indexes, system tables, and regular tables
freelist.c	7	Classify sequential vs. random
smgr.c/md.c	21	Assign SCSI groups numbers
fd.c	20	Add classifier to scatter/gather and classify temp. tables

Table 7: PostgreSQL changes.

4.4.1 Our baseline storage system cache

Our baseline cache uses a conventional write-back cache with LRU eviction. Recent research shows that solid-state LRU caching solutions are not cost-effective for enterprise workloads [31]. We confirm this result in our evaluation, but also build upon it by demonstrating that a conventional LRU algorithm *can* be cost-effective with Differentiated Storage Services. Algorithms beyond LRU [13, 25] may produce even better results.

A solid-state drive is used as the cache, and we divide the SSD into a configurable number of allocation units. We use 8 sectors (4KB, a common memory page size) as the allocation unit, and we initialize the cache by contiguously adding all of these allocation units to a *free list*. Initially, this free list contains every sector of the SSD.

For new write requests, we allocate cache entries from this free list. Once allocated, the entries are removed from the free list and added to a *dirty list*. We record the entries allocated to each I/O, by saving the mapping in a hash table keyed by the logical block number.

A *syncer daemon* monitors the size of the free list. When the free list drops below a low watermark, the syncer begins cleaning the dirty list. The dirty list is sorted in LRU order. As dirty entries are read or written, they are moved to the end of the dirty list. In this way, the syncer cleans the least recently used entries first. Dirty entries are read from the SSD and written back to the disk. As entries are cleaned, they are added back to the free list. The free list is also sorted in LRU order, so if clean entries are accessed while in the free list, they are moved to the end of the free list.

It is atop this baseline cache that we implement selective allocation and selective eviction.

4.4.2 Conventional allocation

Two heuristics are commonly used by current storage systems when deciding whether to allocate an I/O request in the cache. These relate to the size of the request and its access pattern (random or sequential). For example, a 256KB request in NTFS tells you that the file the I/O is directed to is at least 256KB in size, and multiple contiguous 256KB requests indicate that the file may be larger. It is the small random requests that benefit most from caching, so large requests or requests that appear to be part of a sequential stream will often bypass the cache, as such requests are just as efficiently served from disk. There are at least two fundamental problems with this approach.

First, the block-level request size is only partially correlated with file size. Small files can be accessed with large requests, and large files can be accessed with small requests. It all depends on the application request size and caching model (e.g., buffered or direct). A classic example of this is the NTFS Master File Table (MFT). This key metadata structure is a large, often sequentially written file. Though when read, the requests are often small and random. If a storage system were to bypass the cache when the MFT is being written, subsequent reads would be forced to go to disk. Fixing this problem would require that the MFT be distinguished from other large files and, without an I/O classification mechanism, this would not be easy.

The second problem is that operating systems have a maximum request size (e.g., 512KB). If one were to make a caching decision based on request size, one could not differentiate file sizes that were larger than this maximum request. This has not been a problem with small DRAM caches, but solid-state caches are considerably larger and can hold many files. So, knowing that a file is, say, 1MB as opposed to 1GB is useful when making a caching decision. For example, it can be better to cache more small files than fewer large ones, which is particularly the case for file servers that are seek-limited from small files and their metadata.

4.4.3 Selective allocation

Because of the above problems, we do not make a cache allocation decision based on request size. Instead, for the FS prototypes, we differentiate metadata from regular files, and we further differentiate the regular files by size.

Metadata and small files are always cached. Large files are conditionally cached. Our current implementation checks to see if the syncer daemon is active (cleaning dirty entries), which indicates cache pressure, and we opt to not cache large files in this case (our configurable cut-off is 1MB or larger — such blocks will bypass the cache). However, an idle syncer daemon indicates that there is space in the cache, so we choose to cache even the largest of files.

4.4.4 Selective eviction

Selective eviction is similar to selective allocation in its use of priority information. Rather than evict entries in strict LRU order, we evict the lowest priority entries first. This is accomplished by maintaining a dirty list for each I/O class. When the number of free cache entries reaches a low watermark, the syncer cleans the lowest priority dirty list first. When that list is exhausted, it selects the next lowest priority list, and so on, until a high watermark of free entries is reached and the syncer is put to sleep.

With selective eviction, we can completely fill the cache without the risk of priority inversion. For an FS, this allows the caching of larger files, but not at the expense of evicting smaller files. Large files will evict themselves under cache pressure, leaving the small files and metadata effectively pinned in the cache. High priority I/O will only be evicted after all lower priority data has been evicted. As we illustrate in our evaluation, small files and metadata rarely get evicted in our enterprise workloads, which contain realistic mixes of small and large file size [29].

4.4.5 Linux implementation

We implement a SW cache as RAID level 9 in the Linux RAID stack (MD).[1] The mapping to RAID is a natural one. RAID levels (e.g., 0, 1, 5) and the nested versions (e.g., 10, 50) simply define a static mapping from logical blocks within a volume to physical blocks on storage devices. RAID-0, for example, specifies that logical blocks will be allocated round-robin. A Differentiated Storage Services architecture, in comparison, provides a dynamic mapping. In our implementation, the classification scheme and associated policies provide a mapping to either the cache device or the storage device, though one might also consider a mapping to multiple cache levels or different storage pools.

Managing the cache as a RAID device allows us to build upon existing RAID management utilities. We use the Linux *mdadm* utility to create a cached volume. One simply specifies the storage device and the caching device (devices in `/dev`), both of which may be another RAID volume. For example, the cache device may be a mirrored pair of SSDs, and the storage device a RAID-50 array. Implementing Differentiated Storage Services in this manner makes for easy integration into existing storage management utilities.

Our SW cache is implemented in a kernel RAID module that is loaded when the cached volume is created; information regarding the classification scheme and priority assignment are passed to the module as runtime parameters. Because the module is part of the kernel, I/O requests are terminated in the block layer and never reach the SCSI layer. The I/O classifiers are, therefore, extracted directly from the block I/O requests (BIOs), not the 5-bit classification field in the SCSI request.

4.4.6 iSCSI implementation

Our second storage system prototype is based on iSCSI [12]. Unlike the RAID-9 prototype, iSCSI is OS-independent and can be accessed by both Linux and Windows. In both cases, the I/O classifier is copied into the SCSI request on the host. On the iSCSI target the I/O classifier is extracted from the request, the priority of the I/O class is determined, and a caching decision is made. The caching implementation is identical to the RAID-9 prototype.

5. EVALUATION

We evaluate our filesystem prototypes using a file server workload (based on SPECsfs [44]), an e-mail server workload (modeled after the Swiss Internet Analysis [30]), a set of filesystem utilities (`find`, `tar`, and `fsck`), and a database workload (TPC-H [47]).

We present data from the Linux RAID-9 implementation for the filesystem workloads; NTFS results using our iSCSI prototype are similar. For Linux TPC-H, we use iSCSI.

5.1 Experimental setup

All experiments are run on a single Linux machine. Our Linux system is a 2-way quad-core Xeon server system (8 cores) with 8GB of RAM. We run Fedora 13 with a 2.6.34 kernel modified as described in Section 4. As such, the Ext3

[1]RAID-9 is not a standard RAID level, but simply a way for us to create cached volumes in Linux MD.

File size	File server	E-mail server
1K	17%	0
2K	16%	24%
4K	16%	26%
8K	7%	18%
16K	7%	12%
32K	9%	6%
64K	7%	5%
128K	5%	3%
256K	5%	2%
512K	4%	2%
1M	3%	1%
2M	2%	0%
8M	1%	0%
10M	0%	1%
32M	1%	0%

Table 8: File size distributions.

filesystem is modified to classify all I/O, the block layer copies the classification into the Linux BIO, and the BIO is consumed by our cache prototype (a kernel module running in the Linux RAID (MD) stack).

Our storage device is a 5-disk LSI RAID-1E array. Atop this base device we configure a cache as described in Section 4.4.5, or 4.4.6 (for TPC-H); an Intel®32GB X25-E SSD is used as the cache. For each of our tests, we configure a cache that is a fraction of the used disk capacity (10-30%).

5.2 Workloads

Our file server workload is based on SPECsfs2008 [44]; the file size distributions are shown in Table 8 (File server). The setup phase creates 262,144 files in 8,738 directories (SFS specifies 30 files per directory). The benchmark performs 262,144 transactions against this file pool, where a transaction is reading an existing file or creating a new file. The read/write ratio is 2:1. The total capacity used by this test is 184GB, and we configure an 18GB cache (10% of the file pool size). We preserve the file pool at the end of the file transactions and run a set of filesystem utilities. Specifically, we search for a non-existent file (`find`), archive the filesystem (`tar`), and then check the filesystem for errors (`fsck`).

Our e-mail server workload is based on a study of e-mail server file sizes [30]. We use a request size of 4KB and a read/write ratio of 2:1. The setup phase creates 1 million files in 1,000 directories. We then perform 1 million transactions (reading or creating an e-mail) against this file pool. The file size distribution for this workload is shown in Table 8 (E-mail server). The total disk capacity used by this test is 204GB, and we configure a 20GB cache.

Finally, we run the TPC-H decision support workload [47] atop our modified PostgreSQL [33] database (Section 4.3). Each PostgreSQL file is opened with the flag `O_CLASSIFIED`, thereby enabling user-level classification and disabling file size classification from Ext3. We build a database with a scale factor of 8, resulting in an on-disk footprint of 29GB, and we run the I/O intensive queries (2, 17, 18, and 19) back-to-back. We compare 8GB LRU and LRU-S caches.

5.3 Test methodology

We use an in-house, file-based workload generator for the file and e-mail server workloads. As input, the generator takes a file size distribution, a request size distribution, a read/write ratio, and the number of subdirectories.

For each workload, our generator creates the specified number of subdirectories and, within these subdirectories, creates files using the specified file and write request size distribution. After the pool is created, transactions are performed against the pool, using these same file and request size distributions. We record the number of files written/read per second and, for each file size, the 95th percentile (worst case) latency, or the time to write or read the entire file.

We compare the performance of three storage configurations: no SSD cache, an LRU cache, and an enhanced LRU cache (LRU-S) that uses selective allocation and selective eviction. For the cached tests, we also record the contents of the cache on a class-by-class basis, the read hit rate, and the eviction overhead (percentage of transferred blocks related to cleaning the cache). These three metrics are performance indicators used to explain the performance differences between LRU and LRU-S. Elapsed time is used as the performance metric in all tests.

5.4 File server

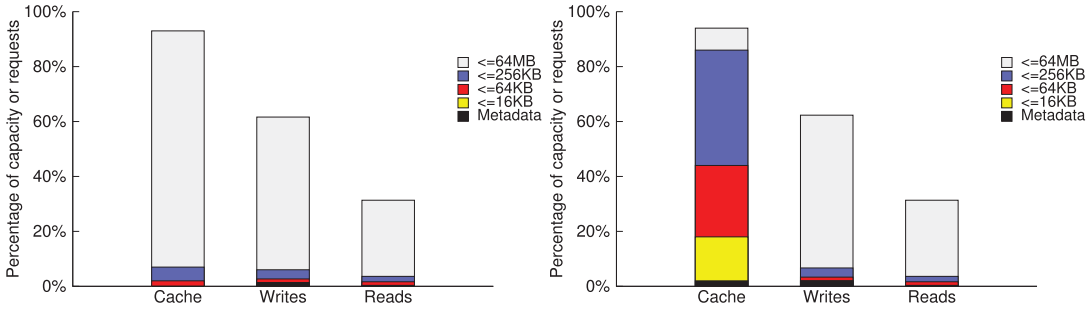
Figure 2a shows the contents of the LRU cache at completion of the benchmark (left bar), the percentage of blocks written (middle bar), and the percentage of blocks read (right bar). The cache bar does not exactly add to 100% due to round-off.[2] Although the cache activity (and contents) will naturally differ across applications, these results are representative for a given benchmark across a range of different cache sizes.

As shown in the figure, the LRU breakdown is similar to the blocks written and read. Most of the blocks belong to large files — a tautology given the file sizes in SPECsfs2008 (most files are small, but most of the data is in large files). Looking again at the leftmost bar, one sees that nearly the entire cache is filled with blocks from large files. The smallest sliver of the graph (bottommost layer of cache bar) represents files up to 64KB in size. Smaller files and metadata consume less than 1% and cannot be seen.

Figure 2b shows the breakdown of the LRU-S cache. The write and read breakdown are identical to Figure 2a, as we are running the same benchmark, but we see a different outcome in terms of cache utilization. Over 40% of the cache is consumed by files 64KB and smaller, and metadata (bottommost layer) is now visible. Unlike LRU eviction alone, selective allocation and selective eviction limit the cache utilization of large files. As utilization increases, large-file blocks are the first to be evicted, thereby preserving small files and metadata.

Figure 3a compares read hit rates. With a 10% cache, the read hit rate is approximately 10%. Given the uniformly random distribution of the SPECsfs2008 workload, this result is expected. However, although the read hit rates are

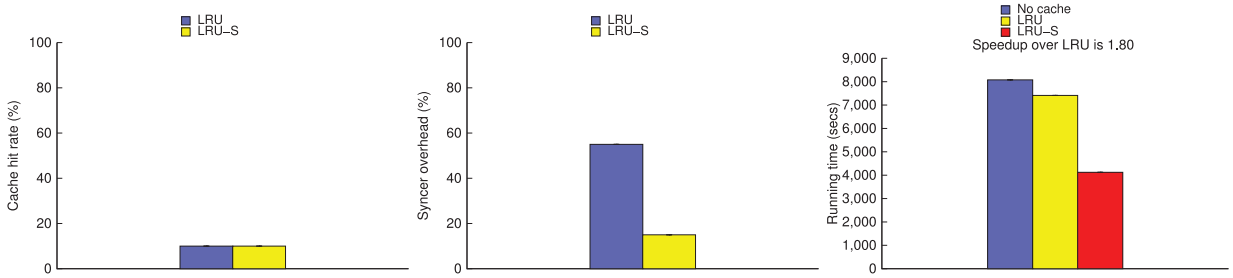
[2]Some of the classes consume less than 1% and round to 0.



(a) LRU cache and I/O breakdown

(b) LRU-S cache and I/O breakdown

Figure 2: SFS results. Cache contents and breakdown of blocks written/read.



(a) Read hit rate

(b) Syncer overhead

(c) SFS performance

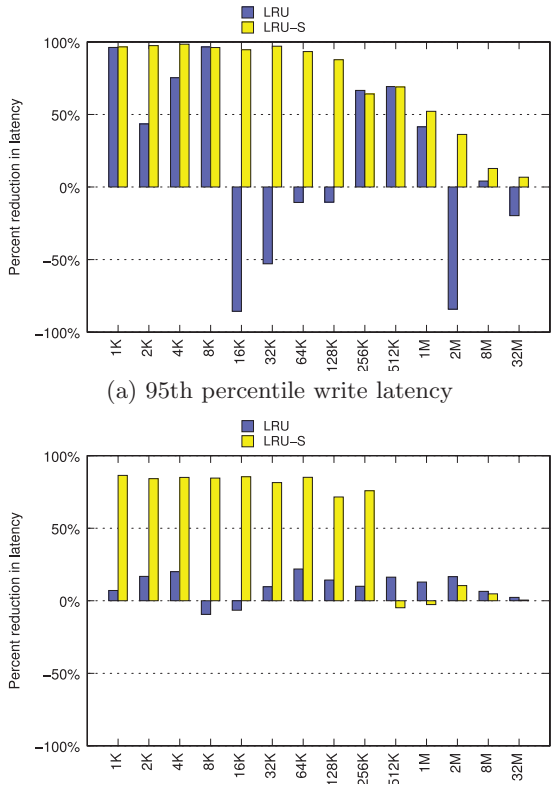
Figure 3: SFS performance indicators

identical, the miss penalties are not. In the case of LRU, most of the hits are to large files. In the case of LRU-S, the hits are to small files and metadata. Given the random seeks associated with small file and metadata, it is better to miss on large sequential files.

Figure 3b compares the overhead of the syncer daemon, where overhead is the percentage of transferred blocks due to cache evictions. When a cache entry is evicted, the syncer must read blocks from the cache device and write them back to the disk device — and this I/O can interfere with application I/O. Selective allocation can reduce the job of the syncer daemon by fencing off large files when there is cache pressure. As a result, we see the percentage of I/O related to evictions drop by more than a factor or 3. This can translate into more available performance for the application workload.

Finally, Figure 3c shows the actual performance of the benchmark. We compare the performance of no cache, an LRU cache, and LRU-S. Performance is measured in running time, so smaller is better. As can be seen in the graph, an LRU cache is only slightly better than no cache at all, and an LRU-S cache is 80% faster than LRU. In terms of running time, the no-cache run completes in 135 minutes, LRU in minutes 124, and LRU-S in 69 minutes.

The large performance difference can also be measured by the improvement in file latencies. Figures 4a and 4b compare the 95th percentile latency of write and read operations, where latency is the time to write or read an entire file. The x-axis represents the file sizes (as per SPECsfs2008) and the y-axis represents the reduction in latency relative to no



(a) 95th percentile write latency

(b) 95th percentile read latency

Figure 4: SFS file latencies

cache at all. Although LRU and LRU-S reduce write latency equally for many of the file sizes (e.g., 1KB, 8KB, 256KB, and 512KB), LRU suffers from outliers that account for the increase in 95th percentile latency. The bars that extend below the x-axis indicate that LRU *increased* write latency relative to no cache, due to cache thrash. And the read latencies show even more improvement with LRU-S. Files 256KB and smaller have latency reductions greater than 50%, compared to the improvements in LRU which are much more modest. Recall, with a 10% cache, only 10% of the working set can be cached. Whereas LRU-S uses this 10% to cache small files and metadata, standard LRU wastes the cache on large, sequentially-accessed files. Stated differently, the cache space we save by evicting large files allows for many more small files to be cached.

5.5 E-mail server

The results from the e-mail server workload are similar to the file server. The read cache hit rate for both LRU and LRU-S is 11%. Again, because the files are accessed with a uniformly random distribution, the hit rate is correlated with the size of the working set that is cached. The miss penalties are again quite different. LRU-S reduces the read latency considerably. In this case, files 32KB and smaller see a large read latency reduction. For example, the read latency for 2KB e-mails is 85ms, LRU reduces this to 21ms, and LRU-S reduces this to 4ms (a reduction of 81% relative to LRU).

As a result of the reduced miss penalty and lower eviction overhead (reduced from 54% to 25%), the e-mail server workload is twice as fast when running with LRU-S. Without any cache, the test completes the 1 million transactions in 341 minutes, LRU completes in 262 minutes, and LRU-S completes in 131 minutes.

Like the file server, an e-mail server is often throughput limited. By giving preference to metadata and small e-mails, significant performance improvements can be realized. This benchmark also demonstrates the flexibility of our FS classification approach. That is, our file size classification is sufficient to handle both file and e-mail server workloads, which have very different file size distributions.

5.6 FS utilities

The FS utilities further demonstrate the advantages of selective caching. Following the file server workload, we search the filesystem for a non-existent file (`find`, a 100% read-only metadata workload), create a tape archive of an SFS subdirectory (`tar`), and check the filesystem (`fsck`).

For the `find` operation, the LRU configuration sees an 80% read hit rate, compared to 100% for LRU-S. As a result, LRU completes the find in 48 seconds, and LRU-S in 13 (a 3.7x speedup). For `tar`, LRU has a 5% read hit rate, compared to 10% for LRU-S. Moreover, nearly 50% of the total I/O for LRU is related to syncer daemon activity, as LRU write-caches the tar file, causing evictions of the existing cache entries and leading to cache thrash. In contrast, the LRU-S fencing algorithm directs the tar file to disk. As as result, LRU-S completes the archive creation in 598 seconds, compared to LRU's 850 seconds (a 42% speedup).

Finally, LRU completes `fsck` in 562 seconds, compared to 94 seconds for LRU-S (a 6x speedup). Unlike LRU, LRU-S retains filesystem metadata in the cache, throughout all of the tests, resulting in a much faster consistency check of the filesystem.

5.7 TPC-H

As one example of how our proof-of-concept DB can prioritize I/O, we give highest priority to filesystem metadata, user tables, log files, and temporary tables; all of these classes are managed as a single class (they share an LRU list). Index files are given lowest priority. Unused indexes can consume a considerable amount of cache space and, in these tests, are served from disk sufficiently fast. We discovered this when we first began analyzing the DB I/O requests in our storage system. That is, the classified I/O both identifies the opportunity for cache optimization, and it provides a means by which the optimization can be realized.

Figure 5 compares the cache contents of LRU and LRU-S. For the LRU test, most of the cache is consumed by index files; user tables and temporary tables consume the remainder. Because index files are created after the DB is created, it is understandable why they consume such a large portion of the cache. In contrast, LRU-S fences off the index files, leaving more cache space for user tables, which are often accessed randomly.

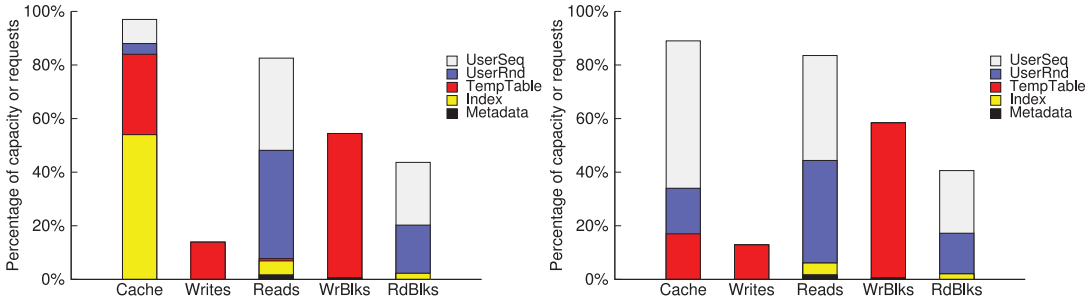
The end result is an improved cache hit rate (Figure 6a), slightly less cache cleaning overhead (Figure 6b), and a 20% improvement in query time (Figure 6c). The non-cached run completes all 4 queries in 680 seconds, LRU in 463 seconds, and LRU-S in 386 seconds. Also, unlike the file and e-mail server runs, we see more variance in TPC-H running time when not using LRU-S. This applies to both the non-cached run and the LRU run. Because of this, we average over three runs and include error bars. As seen in Figure 6c, LRU-S not only runs faster, but it also reduces performance outliers.

6. RELATED WORK

File and storage system QoS is a heavily researched area. Previous work focuses on QoS guarantees for disk I/O [54], QoS guarantees for filesystems [4], configuring storage systems to meet performance goals [55], allocating storage bandwidth to application classes [46], and mapping administrator-specified goals to appropriate storage system designs [48]. In contrast, we approach the QoS problem with I/O classification, which benefits from a coordinated effort between the computer system *and* the storage system.

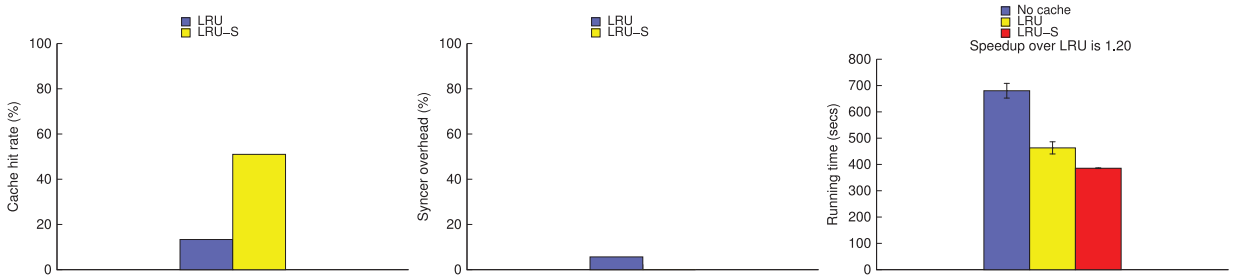
More recently, providing performance differentiation (or isolation) has been an active area of research due to the increasing level in which storage systems are being shared within a data center. Such techniques manage I/O scheduling to achieve fairness within a shared storage system [17, 50, 53]. The work presented in this paper provides a finer granularity of control (classes) for such systems.

Regarding caching, numerous works focus on flash and its integration into storage systems as a conventional cache [20, 23, 24]. However, because enterprise workloads often exhibit such poor locality of reference, it can be difficult to make conventional caches cost-effective [31]. In contrast, we show



(a) LRU cache and I/O breakdown (b) LRU-S cache and I/O breakdown

Figure 5: TPC-H results. Cache contents and breakdown of blocks written/read.



(a) Read hit rate (b) Syncer overhead (c) TPC-H performance

Figure 6: TPC-H performance indicators

that *selective* caching, even when applied to the simplest of caching algorithms (LRU) can be cost effective. Though we introduce selective caching in the context of LRU [39], any of the more advanced caching algorithms could be used, such as LRU-K [32], CLOCK-Pro [13], 2Q [15], ARC [25], LIRS [14], FBR [35], MQ [59], and LRFU [19].

Our block-level selective caching approach is similar to FS-level approaches, such as Conquest [51] and zFS [36], where faster storage pools are reserved for metadata and small files. And there are other block-level caching approaches with similar goals, but different approaches. In particular, Hystor [6] uses data migration to move metadata and other latency sensitive blocks into faster storage, and Karma [57] relies on *a priori* hints on database block access patterns to improve multi-level caching.

The characteristics of flash [7] make it attractive as a medium for persistent transactions [34], or to host flash-based filesystems [16]. Other forms of byte-addressable non-volatile memory introduce additional filesystem opportunities [8].

Data migration [1, 2, 5, 6, 18, 21, 56], in general, is a complement to the work presented in this article. However, migration can be expensive [22], so it is best to allocate storage from the appropriate storage during file creation, whenever possible. Many files have well-known patterns of access, making such allocation possible [26].

And we are not the first to exploit semantic knowledge in the storage system. Most notably, semantically-smart disks [43] and type-safe disks [40, 58] explore how knowledge of on-disk data structures can be used to improve performance, reliability, and security. But we differ, quite fundamentally, in that we send higher-level semantic information with each I/O request, rather than detailed block information (e.g., inode structure) through explicit management commands. Further, unlike this previous work, we do not offload block management to the storage system.

7. CONCLUSION

The inexpressive block interface limits I/O optimization, and it does so in two ways. First, computer systems are having difficulty optimizing around complex storage system internals. RAID, caching, and non-volatile memory are good examples. Second, storage systems, due to a lack of semantic information, experience equal difficulty when trying to optimize I/O requests.

Yet, an entire computer industry has been built around blocks, so major changes to this interface are, today, not practical. Differentiated Storage Services addresses this problem with I/O classification. By adding a small classifier to the block interface, we can associate QoS policies with I/O classes, thereby allowing computer systems and storage system to agree on shared, block-level policies. This will enable continued innovation on both sides of the block interface.

Our filesystem prototypes show significant performance improvements when applied to storage system caching, and our database proof-of-concept suggests similar improvements.

We are extending our work to other realms such as reliability and security. Over time, as applications come to expect differentiated service from their storage systems, additional usage models are likely to evolve.

Acknowledgments

We thank our Intel colleagues who helped contribute to the work presented in this paper, including Terry Yoshii, Mathew Eszenyi, Pat Stolt, Scott Burridge, Thomas Barnes, and Scott Hahn. We also thank Margo Seltzer for her very useful feedback.

8. REFERENCES

[1] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, December 2005. The USENIX Association.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18nd ACM Symposium on Operating Systems Principles (SOSP 01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[3] S. N. I. Association. A Dictionary of Storage Networking Terminology. http://www.snia.org/education/dictionary.

[4] P. R. Barham. A Fresh Approach to File System Quality of Service. In *Proceedings of the IEEE 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 97)*, St. Louis, MO, May 1997.

[5] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 09)*, San Francisco, CA, February 2009. The USENIX Association.

[6] F. Chen, D. Koufaty, and X. Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*, Tucson, AZ, May 31 - June 4 2011.

[7] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2009)*, Seattle, WA, June 2009. ACM Press.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. C. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 09)*, Big Sky, MT, October 2009.

[9] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007. The USENIX Association.

[10] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)*, 9(4):560–595, December 1984.

[11] H. Huang, A. Hung, and K. G. Shin. FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption. In *Proceedings of 20th ACM Symposium on Operating System Principles*, pages 263–276, Brighton, UK, October 2005. ACM Press.

[12] Intel Corporation. Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[13] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC 2005)*, Anaheim, CA, April 10-15 2005. The USENIX Association.

[14] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, Marina Del Rey, CA, June 15-19 2002. ACM Press.

[15] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago Chile, Chile, September 12-15 1994. Morgan Kaufmann.

[16] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST 10)*, San Jose, CA, February 2010. The USENIX Association.

[17] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage*, 1(4):457–480, November 2006.

[18] S. Khuller, Y.-A. Kim, and Y.-C. J. Wan. Algorithms for data migration with cloning. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, San Diego, CA, June 2003. ACM Press.

[19] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.

[20] A. Leventhal. Flash storage memory. In *Communications of the ACM*, volume 51(7), pages 47–51, July 2008.

[21] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: online data migration with performance guarantees. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[22] P. Macko, M. Seltzer, and K. A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST 10)*, San Jose, CA, February 2010. The USENIX Association.

[23] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Science*, Wailea, HI, Jan 1994.

[24] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. In *ACM Transactions on Storage (TOS)*, volume 4, May 2008.

[25] N. Megiddo and D. S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *IEEE Computer Magazine*, 37(4):58–65, April 2004.

[26] M. Mesnier, E. Thereska, G. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC-04)*, New York, NY, May 2004. IEEE Computer Society.

[27] M. Mesnier, M. Wachs, R. R. Sambasivan, A. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2007)*, San Diego, CA, June 2007. ACM Press.

[28] M. P. Mesnier, G. R. Ganger, and E. Riedel. Object-based Storage. *IEEE Communications*, 44(8):84–90, August 2003.

[29] D. T. Meyer and W. J. Bolosky. A Study of Practical Deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 11)*, San Jose, CA, Feb 15-17 2011. The USENIX Association.

[30] O. Muller and D. Graf. Swiss Internet Analysis 2002. http://swiss-internet-analysis.org.

[31] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer systems (EuroSys '09)*, Nuremberg, Germany, March 31 - April 3 2009. ACM Press.

[32] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM International Conference on Management of Data (SIGMOD '93)*, Washington, D.C., May 26-28 1993. ACM Press.

[33] PostgreSQL Global Development Group. Open source database. http://www.postgresql.org.

[34] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, December 2008. The USENIX Association.

[35] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1990)*, Boulder, CO, May 22-25 1990. ACM Press.

[36] O. Rodeh and A. Teperman. zFS - A Scalable Distributed File System Using Object Disks. In *Proceedings of the 20th Goddard Conference on Mass Storage Systems (MSS'03)*, San Diego, CA, April 2003. IEEE.

[37] C. Ruemmler and J. Wilkes. Disk shuffling. Technical Report HPL-91-156, Hewlett-Packard Laboratories, October 2001.

[38] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[39] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating Systems Concepts*. Wiley, 8th edition, 2009.

[40] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe Disks. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. The USENIX Association.

[41] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 04)*, pages 379–394, San Francisco, CA, December 2004. The USENIX Association.

[42] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, pages 15–30, San Francisco, CA, March 2004. The USENIX Association.

[43] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2th USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March-April 2003. The USENIX Association.

[44] Standard Performance Evaluation Corporation. Spec sfs. http://www.storageperformance.org.

[45] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 2(7):412–418, July 1981.

[46] V. Sundaram and P. Shenoy. A Practical Learning-based Approach for Dynamic Storage Bandwidth Allocation. In *Proceedings of the Eleventh International Workshop on Quality of Service (IWQoS 2003)*, Berkeley, CA, June 2003. Springer.

[47] Transaction Processing Performance Council. TPC Benchmark H. http://www.tpc.org/tpch.

[48] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease. Polus: Growing Storage QoS Management Beyond a "4-Year Old Kid". In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. The USENIX Association.

[49] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proceedings of the 9th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2001)*, Cincinnati, OH, August 2001. IEEE/ACM.

[50] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[51] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a Disk/Persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX ATC 2002)*, Monterey, CA, June 2002. The USENIX Association.

[52] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with CART models. In *Proceedings of the 12th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2004)*, Volendam, The Netherlands, October 2004. IEEE.

[53] Y. Wang and A. Merchant. Proportional share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[54] R. Wijayaratne and A. L. N. Reddy. Providing QoS guarantees for disk I/O. *Multimedia Systems*, 8(1):57–68, February 2000.

[55] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proceedings of the 9th International Workshop on Quality of Service (IWQoS 2001)*, Karlsruhe, Germany, June 2001.

[56] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, February 1996.

[57] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-All Replacement for a Multilevel cAche. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[58] C. Yalamanchili, K. Vijayasankar, E. Zadok, and G. Sivathanu. DHIS: discriminating hierarchical storage. In *Proceedings of The Israeli Experimental Systems Conference (SYSTOR 09)*, Haifa, Israel, May 2009. ACM Press.

[59] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 25-30 2001. The USENIX Association.

Internal Parallelism of Flash Memory-Based Solid-State Drives

FENG CHEN and BINBING HOU, Louisiana State University
RUBAO LEE, Ohio State University

A unique merit of a solid-state drive (SSD) is its *internal parallelism*. In this article, we present a set of comprehensive studies on understanding and exploiting internal parallelism of SSDs. Through extensive experiments and thorough analysis, we show that exploiting internal parallelism of SSDs can not only substantially improve input/output (I/O) performance but also may lead to some surprising side effects and dynamics. For example, we find that with parallel I/Os, SSD performance is no longer highly sensitive to access patterns (random or sequential), but rather to other factors, such as data access interferences and physical data layout. Many of our prior understandings about SSDs also need to be reconsidered. For example, we find that with parallel I/Os, write performance could outperform reads and is largely independent of access patterns, which is opposite to our long-existing common understanding about slow random writes on SSDs. We have also observed a strong interference between concurrent reads and writes as well as the impact of physical data layout to parallel I/O performance. Based on these findings, we present a set of case studies in database management systems, a typical data-intensive application. Our case studies show that exploiting internal parallelism is not only the key to enhancing application performance, and more importantly, it also fundamentally changes the equation for optimizing applications. This calls for a careful reconsideration of various aspects in application and system designs. Furthermore, we give a set of experimental studies on new-generation SSDs and the interaction between internal and external parallelism in an SSD-based Redundant Array of Independent Disks (RAID) storage. With these critical findings, we finally make a set of recommendations to system architects and application designers for effectively exploiting internal parallelism.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; D.4.2 [**Operating Systems**]: Performance—*Measurements*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Flash memory, solid state drive, internal parallelism, storage systems

ACM Reference Format:
Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory-based solid-state drives. ACM Trans. Storage 12, 3, Article 13 (April 2016), 39 pages.
DOI: http://dx.doi.org/10.1145/2818376

1. INTRODUCTION

High-speed data processing demands high storage input/output (I/O) performance. Having dominated computer storage systems for decades, magnetic disks, due to their

An earlier version of this article was published in *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA)*, San Antonio, Texas. This work was supported in part by National Science Foundation under grants CCF-1453705, CCF-0913150, OCI-1147522, and CNS-1162165, Louisiana Board of Regents under grants LEQSF(2014-17)-RD-A-01 and LEQSF-EPS(2015)-PFUND-391, as well as generous support from Intel Corporation.
Authors' addresses: F. Chen and B. Hou, Department of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA 70803; email: {fchen, bhou}@csc.lsu.edu; R. Lee, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210; email: liru@cse.ohio-state.edu.

DOI: http://dx.doi.org/10.1145/2818376

mechanical nature, are becoming increasingly unsuitable for handling high-speed data processing jobs. In recent years, flash memory-based solid-state drives (SSDs) have received strong interest in both academia and industry [Chen et al. 2009; Kgil et al. 2008; Dirik and Jacob 2009; Agrawal et al. 2008; Birrell et al. 2005; Park et al. 2006]. As a type of semiconductor devices, SSDs are completely built on flash memory chips with no moving parts. Such an architectural difference enables us to address fundamentally the technical issues of rotating media. Researchers have invested extensive efforts to adopt SSD technology in the existing storage systems and proposed solutions for performance optimizations (e.g., Chen [2009], Tsirogiannis et al. [2009], Canim et al. [2009], Do and Patel [2009], Agrawal et al. [2009], Lee and Moon [2007], Lee et al. [2008], Lee et al. [2009b], Li et al. [2009], Nath and Gibbons [2008], Koltsidas and Viglas [2008], Shah et al. [2008], Graefe [2007], Lee et al. [2009b], Bouganim et al. [2009], Stoica et al. [2009], and Pritchett and Thottethodi [2010]). These prior research has focused mostly on leveraging the high random data access performance of SSDs and addressing the slow random write issues. However, an important factor of SSDs has not received sufficient attention and has been rarely discussed in the existing literature, which is *internal parallelism*, a unique and rich resource embedded inside SSDs. In this article, we show that the impact of internal parallelism is far beyond the scope of the basic operations of SSDs. We believe that fully exploiting internal parallelism resources can lead to a fundamental change in the current sequential-access-oriented optimization model, which has been adopted in system and application designs for decades. This is particularly important for data-intensive applications.

1.1. Internal Parallelism in SSDs

Internal parallelism is essentially a result of optimizations for addressing several architectural challenges in the design of SSD. SSDs have several inherent architectural limitations. (1) Due to current technical constraints, one single flash memory package can only provide limited bandwidth (e.g., 32 to 40MB/sec [Agrawal et al. 2008]), which is unsatisfactory for practical use as a high-performance storage. (2) Writes in flash memory are often much slower than reads. For example, Samsung K9LBG08U0M [Samsung 2007] requires 800μs for writing a page. (3) The Flash Translation Layer (FTL) running at the device firmware level needs to perform several critical operations, such as garbage collection and wear leveling [Chen et al. 2009; Agrawal et al. 2008; Dirik and Jacob 2009]. These internal operations can incur a high overhead (e.g., milliseconds).

In order to address these limitations, SSD architects have built an ingenious structure that enables rich internal parallelism: An array of flash memory packages integrated in an SSD and connected through multiple (e.g., 2–10) channels to flash memory controllers. A sequence of logical block addresses (LBA) is provided by the SSD as a logical interface to the host. Since logical blocks can be striped over multiple flash memory packages, data accesses can be performed independently in parallel. Such a highly parallelized design yields two benefits: (1) Transferring data from/to multiple flash memory packages in parallel can provide high bandwidth in aggregate. (2) High-latency operations, such as *erase*, can be effectively hidden behind other concurrent operations, because when a flash chip is performing a high-latency operation, the other chips can still service incoming requests simultaneously. These design optimizations effectively address the above-mentioned issues. Therefore, internal parallelism, in essence, is not only an *inherent functionality* but also a *basic requirement* for SSDs to deliver high performance.

Exploiting I/O parallelism has been studied in conventional disk-based storage, such as RAID [Patterson et al. 1988], a storage based on multiple hard disks. However, there are two fundamental differences between the SSD and RAID internal structures.

(1) *Different logical / physical mapping mechanisms*: For an RAID storage, logical blocks are *statically* mapped to "fixed" physical locations, which is determined by its logical block number (LBN) [Patterson et al. 1988]. For an SSD, a logical block is *dynamically* mapped to physical flash memory, and this mapping changes during runtime. A particular attention should be paid to a unique data layout problem in SSDs. In a later section, we will show that an ill-mapped data layout can cause undesirable performance degradation, while such a problem would not happen in RAID with its static mapping. (2) *Different physical natures*: RAID is built on magnetic disks, whose random access performance is often one order of magnitude lower than sequential access. In contrast, SSDs are built on flash memories, in which such a performance gap is much smaller, and thus are less sensitive to access patterns. This physical difference has a strong system implication to the existing sequentiality-based application designs: Without any moving parts, parallelizing I/O operations on SSDs could make random accesses capable of performing comparably to or even better than sequential accesses, while this is unlikely to happen in RAID. Considering all these factors, we believe a thorough study on internal parallelism of SSDs is highly desirable and will help us understand its performance impact, both the benefits and side effects, as well as the associated implications to system designers and data-intensive application users. Interestingly, we also note that using multiple SSDs to build a RAID system can form another layer of parallelism. In this article, we will also perform studies on the interaction between the internal parallelism and the RAID-level external parallelism.

1.2. Research and Technical Challenges

Internal parallelism makes a single SSD capable of handling multiple incoming I/O requests in parallel and achieving a high bandwidth. However, internal parallelism cannot be effectively exploited unless we address several critical challenges.

(1) *Performance gains from internal parallelism are highly dependent on how the SSD internal structure is insightfully understood and effectively utilized.* Internal parallelism is an architecture-dependent resource. For example, the mapping policy directly determines the physical data layout, which significantly affects the efficiency of parallelizing data accesses. In our experiments, we find that an ill-mapped data layout can cause up to 4.2 times higher latency for parallel accesses on an SSD. Without knowing the SSD internals, it is difficult to achieve the anticipated performance gains. Meanwhile, uncovering the low-level architectural information without changing the strictly defined device/host interface, such as Small Computer System Interface (SCSI), is highly challenging. In this article we present a set of simple yet effective experimental techniques to achieve this goal.
(2) *Parallel data accesses can compete for critical hardware resources in SSDs, and such interference would cause unexpected performance degradation.* Increasing parallelism is a double-edged sword. On one hand, high concurrency would improve resource utilization and increase I/O throughput. On the other hand, sharing and competing for critical resources may cause undesirable interference and performance loss. For example, we find mixing reads and writes can cause a throughput decrease as high as 4.5 times for writes, which also makes the runtime performance unpredictable. Only by understanding both benefits and side effects of I/O parallelism can we effectively exploit its performance potential while avoiding its negative effects.
(3) *Exploiting internal parallelism in SSDs demands fundamental changes to the existing program design and the sequential-access-oriented optimization model adopted in software systems.* Applications, such as database systems, normally assume that the underlying storage devices are disk drives, which perform well with sequential

I/Os but lack support for parallel I/Os. As such, the top priority for application design is often to *sequentialize* rather than *parallelize* data accesses for improving storage performance (e.g., Jiang et al. [2005]). Moreover, many optimization decisions embedded in application designs are implicitly based on such an assumption, which would unfortunately be problematic when being applied to SSDs. In our case studies on the PostgreSQL database system, we find that parallelizing a query with multiple subqueries can achieve a speedup of up to a factor of 5, and, more importantly, we also find that with I/O parallelism, the *query optimizer*, a key component in database systems, would make an *incorrect* decision on selecting the optimal query plan, which should receive increased attention from application and system designers.

1.3. Our Contributions

In this article, we strive to address the above-mentioned three critical challenges. Our contributions in this work are fivefold. (1) To understand the parallel structure of SSDs, we first present a set of experimental techniques to uncover the key architectural features of SSDs without any change to the interface. (2) Knowing the internal architectures of SSDs, we conduct a set of comprehensive performance studies to show both benefits and side effects of increasing parallelism on SSDs. Our new findings reveal an important opportunity to significantly increase I/O throughput and also provide a scientific basis for performance optimization. (3) Based on our experimental studies and analysis, we present a set of case studies in database systems, a typical data-intensive application, to show the impact of leveraging internal parallelism for real-world applications. Our case studies indicate that the existing optimizations designed for HDDs can be suboptimal for SSDs. (4) To study the interactions between internal and external parallelism, we further conduct experiments on an SSD-based RAID storage to understand the interference between the two levels. Our results show that only by carefully considering both levels can all parallelism resources be fully exploited. (5) Seeing the rapid evolution of flash technologies, we finally perform a set of experiments to revisit our studies on a comprehensive set of new-generation SSDs on the market. Our results show that recent innovations, such as 3D Vertical NAND (V-NAND) and advanced FTL designs, create both opportunities and new challenges.

The rest of this article is organized as follows. We first discuss the critical issues in Section 2. Section 3 provides the background. Section 4 introduces our experimental system and methodology. Section 5 presents how to detect the SSD internals. Sections 6 and 7 present our experimental and case studies on SSDs. Section 8 discusses the interaction with external parallelism. Section 9 gives experimental studies on new-generation SSDs. Section 10 discusses the system implications of our findings. Related work is given in Section 11. The last section concludes this article.

2. CRITICAL ISSUES FOR INVESTIGATION

In order to fully understand internal parallelism of SSDs, we strive to answer several critical questions to gain insight into this unique resource of SSDs and also reveal some untold facts and unexpected dynamics in SSDs.

(1) Limited by the thin device/host interface, how can we effectively uncover the key architectural features of an SSD? In particular, how is the physical data layout determined in an SSD?
(2) The effectiveness of parallelizing data accesses depends on many factors, such as access patterns, available resources, and others. Can we quantify the benefit of I/O parallelism and its relationship to these factors?

(3) Reads and writes on SSDs may interfere with each other. Can we quantitatively illustrate such interactive effects between parallel data accesses? Would such interference impair the effectiveness of parallelism?
(4) Readahead improves read performance, but it is sensitive to read patterns [Chen et al. 2009]. How would I/O parallelism affect the readahead? Can the benefits from parallelism offset the negative impact? How should we trade off between increasing parallelism and retaining effective readahead?
(5) The physical data layout on an SSD could change on the fly. How does an ill-mapped data layout affect the effectiveness of I/O parallelism and the readahead mechanism?
(6) Applications can exploit internal parallelism to optimize the data access performance. How much performance benefit can we achieve in a large storage system for data-intensive applications? Can we see some side effects?
(7) Many optimizations in applications are specifically tailored to the properties of hard drives and may be ineffective for SSDs. Can we make a case that parallelism-based optimizations would be important for maximizing data access performance on SSDs and changing the existing application design?
(8) SSD-based RAID provides another level of parallelism across devices. What are the interactions between internal parallelism of SSDs with such external cross-device parallelism? How would the RAID configurations affect the effectiveness of leveraging internal parallelism?
(9) Flash memory and SSD technologies are continuously evolving. Can we effectively uncover the architectural details for new-generation SSDs, which have highly complex internal structures? To what degree can we expose such low-level information without hardware support?

In this article, we will answer these questions through extensive experiments. We hope our experimental analysis and case studies will influence the system and application designers to carefully rethink the current sequential-access-oriented optimization model and treat parallelism as a *top priority* on SSDs.

3. BACKGROUND OF SSD ARCHITECTURE

3.1. NAND Flash Memory

Today's SSDs are built on NAND flash memory. NAND flash can be classified into two main categories, Single-Level Cell (SLC) and Multi-Level Cell (MLC) NAND. An SLC flash memory cell stores only one bit. An MLC flash memory cell can store two bits or even more. For example, recently Samsung began manufacturing Triple Level Cell (TLC) flash memories, which store 3 bits in one cell. An NAND flash memory package is usually composed of multiple *dies* (chips). A typical die consists of two or more *planes*. Each plane contains thousands (e.g., 2,048) of *blocks* and registers. A block normally consists of dozens to hundreds of pages. Each page has a data part (e.g., 4–8KB) and an associated metadata area (e.g., 128 bytes), which is often used for storing Error Correcting Code (ECC) and other information.

Flash memory has three major operations, *read*, *write*, and *erase*. Reads and writes are normally performed in units of pages. Some support subpage operations. A read normally takes tens of microseconds (e.g., 25μs to 50μs). A write may take hundreds of microseconds (e.g., 250μs to 900μs). Pages in a block must be written sequentially, from the least significant address to the most significant address. Flash memory does not support *in-place overwrite*, meaning that once a page is programmed (written), it cannot be overwritten until the entire block is erased. An erase operation is slow (e.g., 1.5ms) and must be conducted in block granularity. Different flash memories may have various specifications.

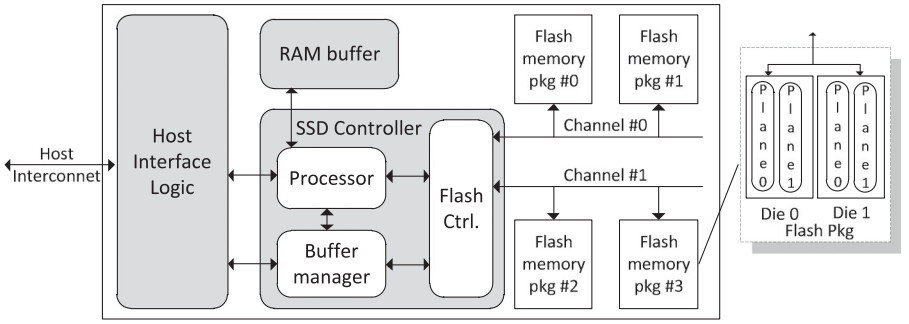


Fig. 1. An illustration of SSD architecture [Agrawal et al. 2008].

3.2. Flash Translation Layer (FTL)

In SSD firmware, FTL is implemented to emulate a hard disk. FTL exposes an array of logical blocks to the host and implements a set of complex mechanisms to manage flash memory blocks and optimize SSD performance. A previous work [Gal and Toledo 2005] provides a detailed survey on the FTL algorithms, and here we briefly introduce three key roles of FTL as follows: (1) *logical block mapping*, which maps logical block addresses (LBA) to *physical block addresses* (PBA); (2) *garbage collection*, which handles the no-in-place-write issue, and asynchronously recycles dirty blocks in a way similar to Log-Structured File System [Rosenblum and Ousterhout 1992]; and (3) *wear leveling*, which shuffles cold blocks (read intensive) with hot blocks (write intensive) to even out writes over flash memory blocks.

3.3. SSD Architecture and Internal Parallelism

A typical SSD includes four major components (see Figure 1). A *host interface logic* connects to the host via an interface connection (e.g., Serial ATA bus). An *SSD controller* manages flash memory, translates I/O requests, and issues I/O commands to flash memory via a *flash memory controller*. A dedicated Dynamic Random-Access Memory (DRAM) *buffer* holds metadata or data. Some SSDs use a small integrated Static Random-Access Memory (SRAM) buffer to lower production cost. In most SSDs, multiple (e.g., 2–10) channels connect the controller with flash memory packages. Each channel may be shared by multiple packages. Actual implementations may vary across different models. Prior research work [Agrawal et al. 2008; Dirik and Jacob 2009] gives detailed descriptions about the architecture of SSDs.

With such an architecture design, we can find that parallelism is available at different levels. Operations at each level can be parallelized or interleaved, which provides rich opportunities of internal parallelism.

(1) *Channel-Level Parallelism*: Since the data bus is the main performance bottleneck, SSDs usually incorporate multiple channels to connect the controller to the flash memory packages. Each channel can operate independently and simultaneously. Resource redundancy is provided. For example, some SSDs adopt an independent ECC engine for each channel to further speedup I/Os [Park et al. 2006], which improves performance but also increases production cost.
(2) *Package-Level Parallelism*: In order to optimize resource utilization, a channel is usually shared among multiple flash memory packages. Since each flash memory package can be operated independently, operations on flash memory packages attached to the same channel can be interleaved. As so, the bus utilization can be optimized [Dirik and Jacob 2009; Agrawal et al. 2008].

(3) *Die-Level Parallelism*: A flash memory package often includes two or more dies (chips). For example, a Samsung K9LBG08U0M flash memory package is composed of two K9GAG08U0M dies. Each die can be selected individually and execute a command independent of the others, which increases the throughput [Samsung 2007].
(4) *Plane-Level Parallelism*: A flash memory chip is typically composed of two or more planes. Flash memories (e.g., Samsung [2007] and MIC [2007]) support performing the same operation (e.g., read/write/erase) on multiple planes simultaneously. Some flash memories (e.g., MIC [2007]) provide `cache mode` to further parallelize medium access and bus transfer.

Such a highly parallelized architecture provides rich parallelism opportunities. In this article we will present several simple yet effective experimental techniques to uncover the internal parallelism at various levels.

3.4. Command Queuing

Native command queuing (NCQ) is a feature introduced by the SATA II standard [SAT 2011]. With NCQ support, the device can accept multiple incoming commands from the host and schedule the jobs internally. For a hard drive, by accepting multiple incoming commands from the host, it can better schedule jobs internally to minimize disk head movement. However, only one request can be executed each time due to its mechanical nature. For SSDs, NCQ is especially important, because the highly parallelized internal structure of an SSD can be effectively utilized only when the SSD is able to accept multiple concurrent I/O jobs from the host (operating system). NCQ enables SSDs to achieve real parallel data accesses internally. Compared to hard drives, NCQ on SSDs not only allows the host to issue multiple commands to the SSD for better scheduling but, more importantly, it also enables multiple data flows to concurrently exist inside SSDs. Early generations of SSDs do not support NCQ and thus cannot benefit from parallel I/Os [Bouganim et al. 2009].

The SATA-based SSDs used in our experiments can accept up to 32 jobs. New interface logic, such as NVMe [NVM Express 2015; Ellefson 2013], can further enable a much deeper queue depth (65,536) than the 32 parallel requests supported by SATA. Based on the PCI-E interface, NVMe technology can provide an extremely high bandwidth to fully exploit internal parallelism in large-capacity SSDs. Although in this article we mostly focus on the SATA-based SSDs, the technical trend is worth further study.

4. MEASUREMENT ENVIRONMENT

4.1. Experimental Systems

Our experimental studies have been conducted on a Dell Precision T3400. It is equipped with an Intel Core 2Duo E7300 2.66GHz processor and 4GB main memory. A 250GB Seagate 7200RPM hard drive is used for maintaining the OS and home directories (`/home`). We use Fedora Core 9 with the Linux Kernel 2.6.27 and Ext3 file system. The storage devices are connected through the on-board SATA connectors.

We select two representative, state-of-the-art SSDs fabricated by a well-known SSD manufacturer for our initial studies (see Table I). The two SSDs target different markets. One is built on multi-level cell (MLC) flash memories and designed for the mass market, and the other is a high-end product built on faster and more durable single-level cell (SLC) flash memories. For commercial reasons, we refer to the two SSDs as *SSD-M* and *SSD-S*, respectively. Delivering market-leading performance, both SSDs provide full support for NCQ and are widely used in commercial and consumer systems. Our experimental results and communication with SSD manufacturers also show that

Table I. SSD Specification

	SSD-M	SSD-S
Capacity	80GB	32GB
NCQ	32	32
Interface	SATA	SATA
Flash memory	MLC	SLC
Page Size (KB)	4	4
Block Size (KB)	512	256
Read Latency (μs)	50	25
Write Latency (μs)	900	250
Erase Latency (μs)	3,500	700

their designs are consistent with general-purpose SSD designs [Agrawal et al. 2008; Dirik and Jacob 2009] and are representative in the mainstream technical trend on the market. Besides the two SSDs, we also use a PCI-E SSD to study the SSD-based RAID in Section 8 and another set of five SSDs to study the new-generation SSDs with various flash chip and FTL technologies in Section 9. The hardware details about these SSDs will be presented in Sections 8 and 9.

In our experiments, we use the Completely Fair Queuing (CFQ) scheduler, the default I/O scheduler in the Linux kernel, for the hard drives. We use the No-optimization (*noop*) scheduler for the SSDs to leave performance optimization handled directly by the SSD firmware, which minimizes the interference from upper-level components and helps expose the internal behavior of the SSDs for our analysis. We also find *noop* outperforms the other I/O schedulers on the SSDs.

4.2. Experimental Tools and Workloads

For studying the internal parallelism of SSDs, we have used two tools in our experiments. We use the Intel Open Storage Toolkit [Mesnier 2011], which has also been used in prior work [Chen et al. 2009; Mesnier et al. 2007], to generate various types of I/O workloads with different configurations, including read/write ratio, random/sequential ratio, request size, and queue depth (the number of concurrent I/O jobs), and others. The toolkit reports bandwidth, IOPS, and latency. The second one is our custom-built tool, called *replayer*. The replayer accepts a pre-recorded trace file and replays I/O requests to a storage device. This tool facilitates us to precisely repeat an I/O workload directly at the block device level. We use the two tools to generate workloads with the following three access patterns.

(1) *Sequential*: Sequential data accesses using a specified request size, starting from sector 0.
(2) *Random*: Random data accesses using a specified request size. Blocks are randomly selected from the first 1,024MB of the storage space, unless otherwise noted.
(3) *Stride*: Strided data accesses using a specified request size, starting from sector 0 with a stride distance between two consecutive data accesses.

Each workload runs for 30 seconds in default to limit trace size while collecting sufficient data. Unlike prior work [Polte et al. 2008], which was performed over an Ext3 file system, all workloads in our experiments directly access the SSDs as raw block devices. We do not create partitions or file systems on the SSDs. This facilitates our studies on the device behavior without being intervened by the OS components, such as page cache and file system (e.g., no file system level readahead). All requests are issued to the devices synchronously with no think time. Since writes to SSD may change the physical data layout dynamically, similarly to prior work [Bouganim et al. 2009; Chen et al. 2009], before each experiment we fill the storage space using sequential writes

with a request size of 256KB and pause for 5 seconds. This re-initializes the SSD status and keeps the physical data layout remaining largely constant across experiments.

4.3. Trace Collection

In order to analyze I/O traffic in detail, we use `blktrace` [Blktrace 2011] to trace the I/O activities at the block device level. The tool intercepts I/O events in the OS kernel, such as queuing, dispatching, and completion, and so on. Among these events, we are most interested in the *completion* event, which reports the latency for each individual request. The trace data are first collected in memory and then copied to the hard disk drive to minimize the interference caused by tracing. The collected data are processed using `blkparse` [Blktrace 2011] and our post-processing scripts off line.

5. UNCOVERING SSD INTERNALS

Before introducing our performance studies on SSD internal parallelism, we first present a set of experimental techniques to uncover the SSD internals. Two reasons have motivated us to detect SSD internal structures. First, internal parallelism is an architecture-dependent resource. Understanding the key architectural characteristics of an SSD is required to study and understand the observed device behavior. For example, our findings about the write-order based mapping (Section 5.4) has motivated us to further study the ill-mapped data layout issue, which has not been reported in prior literature. Second, the information detected can also be used for many other purposes. For example, knowing the number of channels in an SSD, we can set a proper concurrency level and avoid over-parallelization.

Obtaining such architectural information is particularly challenging for several reasons. (1) SSD manufacturers often regard the architectural design details as critical intellectual property and commercial secrets. To the best of our knowledge, certain information, such as the mapping policy, is not available in any datasheet or specification, although it is important for us to understand and exploit the SSD performance potential. (2) Although SSD manufacturers normally provide standard specification data (e.g., peak bandwidth), much important information is absent or obsolete. In fact, hardware/firmware changes across different product batches are common in practice, and, unfortunately, such changes are often not reflected in a timely manner in public documents. (3) SSDs on the market carefully follow a strictly defined host interface standard (e.g., SATA [SAT 2011]). Only limited information is allowed to pass through such a thin interface. As so, it is difficult, if not impossible, to directly get detailed internal information from the hardware. In this section, we present a set of experimental approaches to expose the SSD internals.

5.1. A Generalized Model

Despite various implementations, almost all SSDs strive to optimize performance essentially in a similar way—evenly distributing data accesses to maximize resource usage. Such a principle is implemented through the highly parallelized SSD architecture design and can be found at different levels. For the sake of generality, we define an abstract model to characterize such an organization based on open documents (e.g., Agrawal et al. [2008] and Dirik and Jacob [2009]): A *domain* is a set of flash memories that share a specific type of resources (e.g., channels). A domain can be further partitioned into *subdomains* (e.g., packages). A *chunk* is a unit of data that is continuously allocated within one domain. Chunks are interleavingly placed over a set of N domains, following a *mapping policy*. We call the chunks across each of N domains a *stripe*. One may notice that this model is, in principle, similar to RAID [Patterson et al. 1988]. In fact, SSD architects often adopt a RAID-0-like striping mechanism [Agrawal et al. 2008; Dirik and Jacob 2009]. Some SSD hardware even directly integrates a RAID

controller inside an SSD [PC Perspective 2009]. However, as compared to RAID, the key difference here is that SSDs map data dynamically, which determines the physical data layout on the fly and may cause an ill-mapped data layout, as we will see later.

In an SSD, hardware resources (e.g., pins, ECC engines) are shared at different levels. For example, multiple flash memory packages may share one channel, and two dies in a flash memory package share a set of pins. A general goal is to minimize resource sharing and maximize resource utilization, and, in this process, three *key factors* directly determine the internal parallelism.

(1) *Chunk size*: the size of the largest unit of data that is continuously mapped within an individual domain.
(2) *Interleaving degree*: the number of domains at the same level. The interleaving degree is essentially determined by the resource redundancy (e.g., channels).
(3) *Mapping policy*: the method that determines the domain to which a chunk of logical data is mapped. This policy determines the physical data layout.

We present a set of experimental approaches to *infer indirectly* the three key characteristics. Basically, we treat an SSD as a "black box" and we assume the mapping follows some repeatable but unknown patterns. By injecting I/O traffic with carefully designed patterns to the SSD, we observe the reactions of the device, measured with several key metrics, for example, latency and bandwidth. Based on this probing information, we can speculate the internal architectural features and policies adopted in the SSD. In essence, our solution shares a similar principle with characterizing RAID [Denehy et al. 2014], but characterizing SSDs needs to explore their unique features (e.g., dynamic mapping). We also note that, due to the complexity and diversity of SSD implementations, the retrieved information may not precisely uncover all the internal details. However, our purpose is not to reverse engineer the SSD hardware. We try to characterize the key architectural features of an SSD that are most related to internal parallelism, from the outside in a simple yet effective way. We have applied this technique to both SSDs and we found that it works pleasantly well for serving the purposes of performance analysis and optimization. For brevity, we only show the results for SSD-S. SSD-M behaves similarly. In Section 9, we will further study the effectiveness of our probing techniques on new-generation SSDs.

5.2. Chunk Size

As a basic mapping unit, a chunk can be mapped in only one domain, and two continuous chunks are mapped in two separate domains. Assuming the chunk size is S, for any read that is aligned to S with a request size no larger than S, only one domain would be involved. With an offset of $\frac{S}{2}$ from an aligned position, a read would split into two domains equally. Since many NAND flash memories support subpage access, the latter case would be faster due to the parallel I/Os. Based on this feature, we have designed an experiment to identify the chunk size.

We use read-only requests to probe the SSD. Before experiments, we first initialize the SSD data layout by sequentially overwriting the SSD with request size of 256KB to evenly map logical blocks across domains [Chen et al. 2009]. We estimate a maximum possible chunk size,[1] M (a power of 2) sectors, and we select a request size, n sectors, which must be no larger than M. For a given request size, we conduct a set of tests, as follows: We first set an offset, k sectors (512 bytes each), which is initialized to 0 at the beginning. Then we generate 100,000 reads to the SSD, each of which reads n sectors at a position which is k sectors (offset) from a random position that is aligned to

[1]In practice, we can start with a small M (e.g., 64 sectors) to reduce the number of tests.

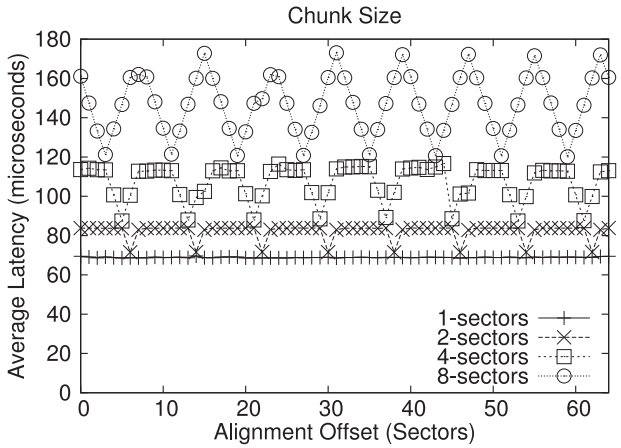


Fig. 2. Detecting the chunk size on SSD-S. Each curve corresponds to a specified request size.

PROGRAM 1: Pseudocode of uncovering SSD internals

```
init_SSD():             sequentially write SSD with 256KB requests
rand_pos(A):            get a random offset aligned to A sector
read(P, S):             read S sectors at offset P sect.
stride_read(J,D):       read 1 chunk with J jobs from offset 0, each read skips over D chunks
plot(X,Y,C):            plot a point at (X,Y) for curve C
M:                      an estimated maximum chunk size
D:                      an estimated maximum interleaving degree
(I) detecting chunk size:
  init_SSD();                                   /* initialize SSD space */
  for (n = 1 sector; n <= M; n *= 2):           /* request size */
    for (k = 0 sector; k <= 2*M; k ++):         /* offset */
      for (i = 0, latency=0; i < 100000; i ++):
        pos = rand_pos(M) + k;
        latency += read (pos, n);
      plot (k, latency/100000, n);              /* plot average latency */

(II) detecting interleaving degree:
  init_SSD();                                   /* initialize SSD space */
  for (j=2; j <=4; j*=2):                       /* number of jobs */
    for (d = 1 chunk; d < 4*D; d ++):           /* stride distance */
      bw = stride_read (j, d);
      plot (d, bw, j);                          /* plot bandwidth */
```

the estimated maximum chunk size M. By selecting a random position, we can reduce the possible interference caused by on-device cache. We calculate the average latency of the 100,000 reads with an offset of k sectors. Then we increment the offset k by one sector and repeat the same test for no less than $2M$ times and plot a curve of latencies with different offsets. Then we double the request size, n, and repeat the whole set of tests again until the request size reaches M. The pseudocode is shown in Program 1(I).

Figure 2 shows an example result on SSD-S. Each curve represents a request size. For brevity, we only show results for request sizes of 1–8 sectors with offsets increasing from 0 to 64 sectors. Except for the case of request size of one sector, a dip periodically appears on the curves as the offset increases. *The chunk size is the interval between the bottoms of two consecutive valleys.* In this case, the detected chunk size is 8 sectors (4KB), the flash page size, but note that a chunk can consist of multiple flash pages in some implementations [Dirik and Jacob 2009]. For a request size of 1 sector (512 bytes),

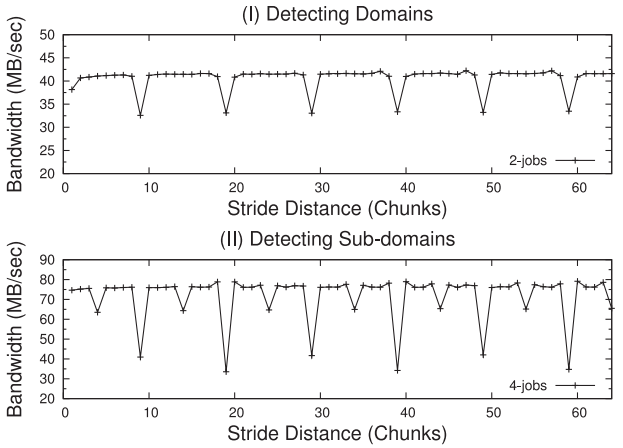


Fig. 3. Detecting the interleaving degree on SSD-S. Figures (I) and (II) use two jobs and four jobs, respectively.

we see a flat curve, because the smallest read/write unit in the OS kernel is one sector and cannot be mapped across two domains. Also note that since this approach relies on the hardware properties, some devices may show a distinct or weak pattern. In Section 9, we will show our studies on new-generation SSDs, and we find that the two SSDs adopting V-NAND technologies do exhibit very weak patterns.

5.3. Interleaving Degree

In our model, chunks are organized into domains based on resource redundancy. Interleaving degree (i.e., the number of domains) represents the degree of resource redundancy. Parallel accesses to multiple domains without resource sharing can achieve a higher bandwidth than doing that congested in one domain. Based on this feature, we have designed an experiment to determine the interleaving degree.

We use read-only requests with size of 4KB (the detected chunk size) to probe the SSD. Similarly to the previous test, we first initialize the SSD data layout and estimate a maximum possible interleaving degree, D. We conduct a set of tests as follows: We first set a stride distance of d chunks, which is initialized to 1 chunk (4KB) at the beginning. Then we use the toolkit to generate a read-only workload with *stride* access pattern to access the SSD with multiple (2–4) jobs. This workload sequentially reads a 4KB chunk each time, and each access skips over d chunks from the preceding access position. We run the test to measure the bandwidth. Then we increment the stride distance d by one chunk and repeat this test for no less than $2D$ times. Program 1(II) shows the pseudocode.

Figure 3(I) and (II) shows the experimental results with two jobs and four jobs on SSD-S, respectively. In Figure 3(I), we observe a periodically appearing dip. *The interleaving degree is the interval between the bottoms of two consecutive valleys, in units of chunks.* For SSD-S, we observe 10 domains, each of which corresponds to one channel.

The rationale behind this experiment is as follows. Suppose the number of domains is D and the data are interleavingly placed across the domains. When the stride distance, d, is $D \times n - 1$, where $n \geq 1$, every data access would exactly skip over $D - 1$ domains from the previous position and fall into the same domain. Since we issue two I/O jobs simultaneously, the parallel data accesses would compete for the same resource and the bandwidth is only around 33MB/sec. With other stride distances, the two parallel jobs

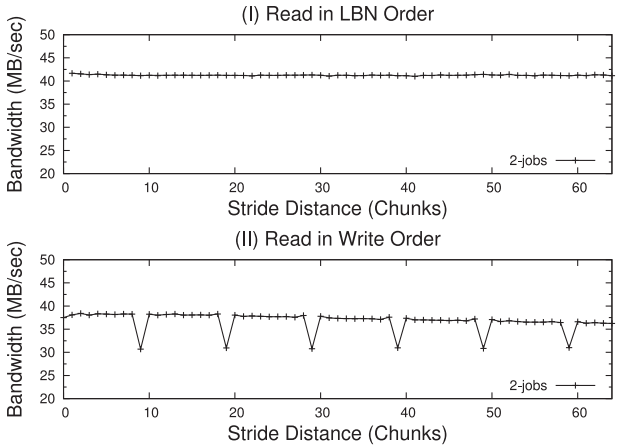


Fig. 4. Detecting the mapping policy on SSD-S. Figures (I) and (II) represent two test cases.

would be distributed across two domains and thus could achieve a higher bandwidth (around 40MB/sec).

By extending the queue depth to four jobs, we can further observe m subdomains in each of D domains. As shown in Figure 3(II), we see two deep dips (33MB/sec) appearing at stride distance of $D \times m \times n - 1$ chunks, where $n, m \geq 1$. At those points, all requests are directed to the same subdomain. A moderate dip (40MB/sec) appears in the middle of two deep dips, because the requests are directed to one domain but in different subdomains. The two shallow dips (64MB/sec) represent the case where two domains are accessed by the four jobs. In this case, we observed two subdomains, each of which corresponds to one package.

5.4. The Mapping Policy

Although all SSDs strive to evenly distribute data accesses across domains, the mapping policies may differ. The mapping policy determines the physical page to which a logical page is mapped. According to open documents (e.g., Dirik and Jacob [2009] and Agrawal et al. [2008]), two mapping policies are widely used in practice. (1) *A LBA-based static mapping*: For a given logical block address (*LBA*) with an interleaving degree, *D*, the block is mapped to a domain number ($LBA \bmod D$). (2) *A write-order-based dynamic mapping*: Differing from static mapping, dynamic mapping is determined by the order in which the blocks are written. That is, for the ith write, the block is assigned to a domain number ($i \bmod D$). We should note here that the write-order-based mapping is *not* the log-structured mapping policy [Agrawal et al. 2008], which appends data in a flash block to optimize write performance. Considering the wide adoption of the two mapping policies in SSD products, we will focus on these two policies in our experiments.

To determine which mapping policy is adopted, we first *randomly* overwrite the first 1024MB data with a request size of 4KB, the chunk size. Each chunk is guaranteed to be overwritten *once and only once*. After such a randomization, the blocks are relocated across the domains. If the LBA-based static mapping is adopted, then the mapping should not be affected by such random writes. Then, we repeat the experiment in Section 5.3 to see if the same pattern (repeatedly appearing dips) repeats. We find, after randomization, that the pattern (repeatedly appearing dips) disappears (see Figure 4(I)), which means that the random overwrites have changed the block mapping, and the LBA-based mapping is not used.

We then conduct another experiment to confirm that the write-order-based dynamic mapping is adopted. We use `blktrace` [Blktrace 2011] to record the order of random writes. We first randomly overwrite the first 1024MB space, and each chunk is written *once and only once*. Then we follow the same order in which blocks are written to issue reads to the SSD and repeat the same experiments in Section 5.3. For example, for the LBNs of random writes in the order of (91, 100, 23, 7, ...), we will read data in the same order (91, 100, 23, 7, ...). If the write-order based mapping is used, then the blocks should be interleavingly allocated across domains in the order of writes (e.g., blocks 91, 100, 23, 7 are mapped to domains 0, 1, 2, 3, respectively), and reading data in the same order would repeat the same pattern (dips) that we saw previously. Our experimental results have confirmed this hypothesis, as shown in Figure 4(II). The physical data layout and the block mapping are *strictly* determined by the order of writes. We have also conducted experiments by mixing reads and writes, and we find that the layout is *only* determined by writes.

5.5. Discussions

Levels of Parallelism: The internal parallelism of SSDs is essentially a result of redundant resources shared at different levels in the SSD. Such a resource redundancy enables the channel-, package-, die-, and plane-level parallelism. Our experimental studies have visualized such potential parallelism opportunities at different levels and also proved that they may impact performance to different extents. For example, Figure 3 shows that there exist at least two levels of parallelism in the SSD, which are corresponding to the channel and package levels. We also can see that leveraging the channel-level parallelism is the most effective (the top points). In contrast, leveraging the package-level parallelism (the middle points) can bring additional benefits but at a limited scale, and congesting all I/O requests in one package would cause the lowest performance. In Section 9, we will further study five new-generation SSDs, and one exhibits three levels of parallelism. In general, due to the interference and overhead in the system, the deeper the level of internal parallelism, the weaker the effect we can see. Nevertheless, we can largely conclude that regarding different types of internal parallelism, the top-level parallelism is the most important resource. We should parallelize our applications to maximize the use of such most shared resources in design optimizations.

Limitations of our approach: Our approach for uncovering the SSD internals has several limitations. First, as a black-box approach, our solution assumes certain general policies adopted in the device. As the hardware implementation becomes more and more complex, it is increasingly challenging to expose all the hardware details. For example, on-device cache in new-generation SSDs could be large (512MB) and interfere our results. In experiments, we attempt to avoid such a problem by flooding the cache with a large amount of data before the test and accessing data at random locations. Second, new flash memory technologies, such as TLC and 3D V-NAND technology, may change the property of storage medium itself, and some observed patterns could be differ. Also, sophisticated FTL designs, such as on-device compression and deduplication [Chen et al. 2011b], may further increase the difficulty. In Section 9, we further study a set of new-generation SSDs, which are equipped with complex FTL and new flash memory chips, to revisit the effectiveness of the hardware probing techniques presented in this section.

6. PERFORMANCE STUDIES

Prepared with the knowledge about the internal structures of SSDs, we are now in a position to investigate the performance impact of internal parallelism. We strive to answer several related questions on *the performance benefits of parallelism*, *the*

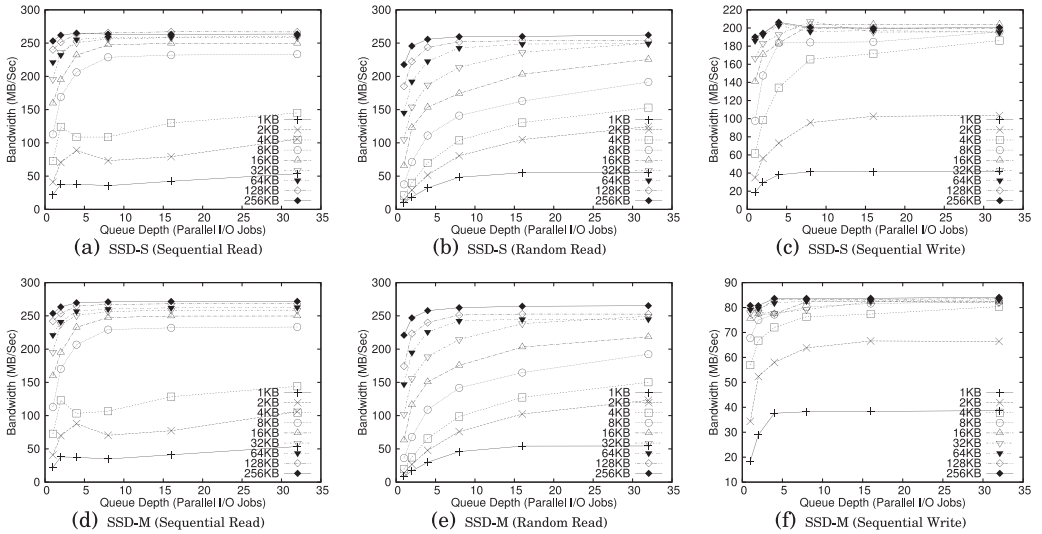


Fig. 5. Bandwidths of the SSDs with increasing queue depth (the number of concurrent I/O jobs).

interference between parallel jobs, *the readahead mechanism with I/O parallelism*, and *the performance impact of physical data layout*.

6.1. What Are the Benefits of Parallelism?

In order to quantitatively measure the potential benefits of I/O parallelism in SSDs, we run four workloads with different access patterns, namely *sequential read*, *sequential write*, *random read*, and *random write*. For each workload, we increase the request size from 1KB to 256KB,[2] and we show the bandwidths with a queue depth (i.e., the number of concurrent I/O jobs) increasing from 1 job to 32 jobs. In order to compare workloads with different request sizes, we use bandwidth (MB/s), instead of I/O per second (IOPS), as the performance metric. Figure 5 shows the experimental results for SSD-M and SSD-S. Since the write-order-based mapping is used, as we see previously, random and sequential writes on both SSDs show similar patterns. For brevity, we only show the results of sequential writes here.

Differing from prior work [Bouganim et al. 2009], our experiments show a great performance benefit from parallelism on SSDs. The significance of performance gains depends on several factors, and we present several key observations here.

(1) *Workload access patterns determine the performance gains from parallelism.* In particular, small and random reads yield the most significant performance gains. In Figure 5(b), for example, increasing queue depth from 1 to 32 jobs for random reads on SSD-S with a request size 4KB achieves a *7.2-fold* bandwidth increase. A large request (e.g., 256KB) benefits relatively less from parallelism, because the continuous logical blocks are often striped across domains and it already benefits from internal parallelism. This implies that in order to exploit effectively internal parallelism, we can either increase request sizes or parallelize small requests.
(2) *Highly parallelized small/random accesses can achieve performance comparable to large sequential accesses without parallelism.* For example, with only 1 job, the

[2]Increasing request size would weaken the difference between random and sequential workloads. However, in order to give a complete picture, we show the experimental results of changing the request size from 1KB to 256KB for all the workloads here.

bandwidth of random reads of 16KB on SSD-S is only 65.5MB/sec, which is 3.3 times lower than that of sequential reads of 64KB (221.3MB/sec). With 32 jobs, however, the same workload (16KB random reads) can reach a bandwidth of 225.5MB/s, which is comparable to the single-job sequential reads. This indicates that SSDs provide us an alternative approach for optimizing I/O performance, *parallelizing small and random accesses*, in addition to simply organizing sequential accesses. This unlocks many new opportunities. For example, database systems traditionally favor a large page size, since hard drives perform well with large requests. On SSDs, which are less sensitive to access patterns, if parallelism can be utilized, a small page size can be a sound choice for optimizing buffer pool usage [Lee et al. 2009b; Graefe 2007].

(3) *The performance potential of increasing I/O parallelism is physically limited by the redundancy of available resources.* We observe that when the queue depth reaches over 8–10 jobs, further increasing parallelism receives diminishing benefits. This observation echoes our finding made in Section 5 that the two SSDs have 10 domains, each of which corresponds to a channel. When the queue depth exceeds 10, a channel has to be shared by more than 1 job. Further parallelizing I/O jobs can bring additional but smaller benefits. On the other hand, we also note that such an over-parallelization does not result in undesirable negative effects.

(4) *The performance benefits of parallelizing I/Os also depend on flash memory mediums.* MLC and SLC flash memories provide noticeable performance difference, especially for writes. In particular, writes on SSD-M, the MLC-based lower-end SSD, quickly reach the peak bandwidth (only about 80MB/sec) with a small request size at a low queue depth. SSD-S, the SLC-based higher-end SSD, shows much higher peak bandwidth (around 200MB/sec) and more headroom for parallelizing small writes. In contrast, less difference can be observed for reads on the two SSDs, because the main performance bottleneck is transferring data across the serial I/O bus rather than reading the flash medium [Agrawal et al. 2008].

(5) *Write performance is insensitive to access patterns, and parallel writes can perform faster than reads.* The uncovered write-order-based mapping policy indicates that the SSDs actually handle incoming writes in the same way, regardless of write patterns (random or sequential). This leads to the observed similar patterns for sequential writes and random writes. Together with the parallelized structure and the on-device buffer, write performance is highly optimized and can even outperform reads in some cases. For example, writes of 4KB with 32 jobs on SSD-S can reach a bandwidth of 186.1MB/sec, which is even 28.3% higher than reads (145MB/sec). Although this surprising result contrasts with our common understanding about slow writes on SSDs, we need to note that for sustained intensive writes, the cost of internal garbage collection will dominate and lower write performance eventually, which we will show in Section 9.

On both SSD-S (Figure 5(a)) and SSD-M (Figure 5(d)), we can also observe a slight *dip* for sequential reads with small request sizes at a low concurrency level (e.g., four jobs with 4KB requests). This is related to interference to the readahead mechanism. We will give a detailed analysis on this in Section 6.3.

6.2. How Do Parallel Reads and Writes Interfere with Each Other and Cause Performance Degradation?

Reads and writes on SSDs can interfere with each other for many reasons. (1) Both operations share many critical resources, such as the ECC engines and the lock-protected mapping table, and so on. Parallel jobs accessing such resources need to be serialized. (2) Both writes and reads can generate background operations internally, such

Table II. Bandwidths (MB/sec) of Co-Running Reads and Writes

	Sequential Write	Random Write	None
Sequential Read	109.2	103.5	72.6
Random read	32.8	33.2	21.3
None	61.4	59.4	

as readahead and asynchronous writeback [Chen et al. 2009]. These internal operations could negatively impact foreground jobs. (3) Mingled reads and writes can foil certain internal optimizations. For example, flash memory chips often provide a `cache mode` [MIC 2007] to pipeline a sequence of reads or writes. A flash memory plane has two registers, *data register* and *cache register*. In the cache mode, when handling a sequence of reads or writes, data can be transferred between the cache register and the controller, while concurrently moving another page between the flash medium and the data register. However, such pipelined operations must be performed in one direction, so mingling reads and writes would interrupt the pipelining and cause performance loss.

In order to illustrate such an interference, we use the toolkit to generate a pair of concurrent workloads, similar to that in the previous section. The two workloads access data in two separate 1,024MB storage spaces. We choose four access patterns, namely *random reads*, *sequential reads*, *random writes*, and *sequential writes*, and enumerate the combinations of running two workloads simultaneously. Each workload uses request size of 4KB and one job. We report the aggregate bandwidths (MB/s) of two co-running workloads on SSD-S in Table II. Co-running with "none" means running a workload individually.

We have observed that for all combinations, the aggregate bandwidths cannot reach the optimal result, the sum of the bandwidths of individual workloads. We also find that *reads and writes have a strong interference with each other, and the significance of this interference highly depends on read access patterns*. If we co-run sequential reads and writes in parallel, the aggregate bandwidth exceeds that of each workload running individually, which means parallelizing workloads obtains benefits, although the aggregate bandwidths cannot reach the optimal (i.e., the sum of the bandwidths of all workloads). However, when running random reads and writes together, we see a strong negative impact. For example, sequential writes can achieve a bandwidth of 61.4MB/s when running individually; however, when running with random reads, the bandwidth drops by a factor of 4.5 to 13.4MB/sec. Meanwhile, the bandwidth of random reads also drops from 21.3MB/sec to 19.4MB/sec. Apparently the co-running reads and writes strongly interfere with each other.

A recent research [Wu and He 2014] by Wu and He has studied the conflict between program/erase operations and read operations in NAND flash and found that the lengthy P/E operations can significantly interfere with the read performance, which well explains our observation here. In their study, they also propose a low-overhead P/E suspension scheme to service pending reads with a priority, which shows significant performance benefits and points to the direction for eventually resolving this challenge.

6.3. How Does I/O Parallelism Impact the Effectiveness of Readahead?

Prior study [Chen et al. 2009] shows that with no parallelism, sequential reads on SSD can substantially outperform random reads, because modern SSDs implement a readahead mechanism at the device level to detect sequential data accesses and prefetch data into the on-device buffer. However, parallelizing multiple sequential read streams could result in a sequence of mingled reads, which can affect the detection of sequential patterns and invalidate readahead. In the meantime, parallelizing reads

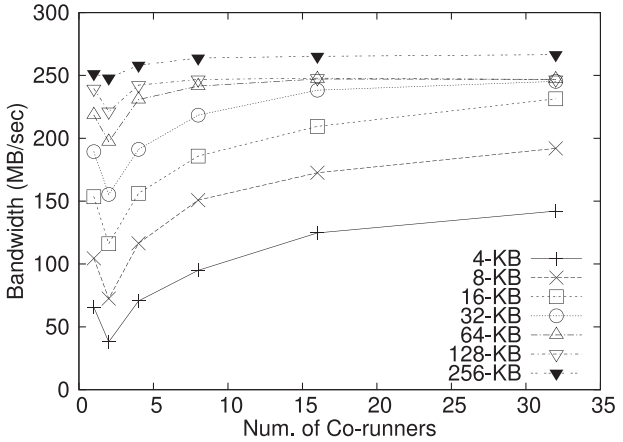


Fig. 6. Performance impact of parallelism on readahead. Each curve represents a request size.

can improve bandwidths, as we see before. So there is a tradeoff between *increasing parallelism* and *retaining effective readahead*.

In order to examine the impact of parallelism on readahead, we have designed a workload to generate multiple jobs, each of which sequentially reads an individual 1024MB space (i.e., the *i*th job reads the *i*th 1,024MB space) simultaneously. Before the experiments, we sequentially overwrite the storage space. We increase the concurrency from 1 to 32 jobs and vary the size from 4KB to 256KB. We compare the aggregate bandwidths of the co-running jobs on SSD-S. SSD-M shows similar results.

Figure 6 shows that in nearly all curves, there exists a *dip* when the queue depth is two jobs. For example, the bandwidth of 4KB reads drops by *43%* from one job to two jobs. At that point, readahead is strongly inhibited due to mingled reads, and such a negative impact cannot be offset at a low concurrency level (two jobs). When we further increase the concurrency level, the benefits coming from parallelism quickly compensate for impaired readahead. It is worth noting that we have verified that this effect is not due to accidental channel conflicts. We added a 4KB shift to the starting offset for each parallel stream, which makes requests mapped to different channels, and we observed the same effect. This case indicates that *readahead can be impaired by parallel sequential reads, especially at low concurrency levels and with small request sizes.* SSD architects can consider including a more sophisticated sequential pattern detection mechanism to identify multiple sequential read streams to avoid this problem.

Another potential hardware limit is the on-device buffer size. For optimizations like readahead, their effectiveness is also limited by the buffer capacity. Aggressively preloading data into the buffer, although it could remove future access cost, may also bring a side effect: buffer pollution. Therefore, a large buffer size in general is beneficial for avoiding such a side effect and accommodating more parallel requests for processing. As flash devices are moving toward supporting very high I/O parallelism, the buffer size must be correspondingly increase. On the other hand, this could lead to another problem: how to keep data safe in such volatile memory. New technologies, such as non-volatile memory (NVM), could be valid solutions [Chen et al. 2014].

6.4. How Does an Ill-Mapped Data Layout Impact I/O Parallelism?

The dynamic write-order-based mapping enables two advantages for optimizing SSD designs. First, high-latency writes can be evenly distributed across domains, which not only guarantees load balance but also naturally balances available free flash blocks and

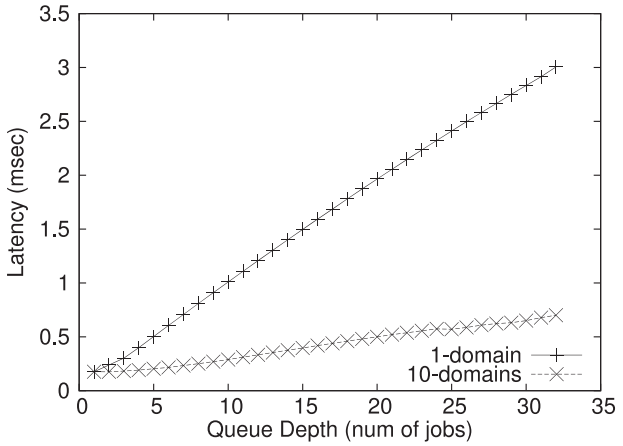


Fig. 7. Performance impact of data layout.

helps even out wears across domains. Second, writes across domains can be overlapped with each other, and the high latency of writes can be effectively hidden behind parallel operations.

On the other hand, such a dynamic write-order-based mapping may result in negative effects in certain conditions. An outstanding one is that since the data layout is completely determined by the order of incoming writes on the fly, logical blocks could be mapped to only a subset of domains and result in an *ill-mapped data layout*. In the worst case, if a set of blocks is mapped to the same domain, then data accesses would be congested and have to compete for the shared resources, which would impair the effectiveness of parallel data accesses.

As discussed before, the effectiveness of increasing I/O parallelism is highly dependent on the physical data layout. Only when the data blocks are physically located in different domains is high I/O concurrency meaningful. An ill-mapped data layout can significantly impair the effectiveness of increasing I/O parallelism. To quantitatively show the impact of an ill-mapped physical data layout, we design an experiment on SSD-S as follows. We first sequentially overwrite the first 1,024MB of storage space to map the logical blocks evenly across domains. Since sequential reads benefit from the readahead, which would lead to unfair comparison, we create a random read trace with request size of 4KB to access the 10 domains in a round-robin manner. Similarly, we create another trace of random reads to only one domain. Then we use the *replayer* to replay the two workloads and measure the average latencies for each. We vary the queue depth from 1 job to 32 jobs and compare the average latencies of parallel accesses to data that are concentrated in one domain and that are distributed across 10 domains. Figure 7 shows that when the queue depth is low, accessing data in one domain is comparable to doing it in 10 domains. However, as the I/O concurrency increases, the performance gap quickly widens. In the worst case, with a queue depth of 32 jobs, accessing data in the same domain (3ms) incurs *4.2 times* higher latency than doing that in 10 domains (0.7ms). This case shows that *an ill-mapped data layout can significantly impair the effectiveness of I/O parallelism*.

6.5. Readahead on Ill-Mapped Data Layout

An ill-mapped data layout may impact readahead, too. To show this impact, we map the blocks in the first 1,024MB storage space into 1, 2, 5, and 10 domains. Manipulating data layout is simple: We sequentially write the first 1,024MB data with request size

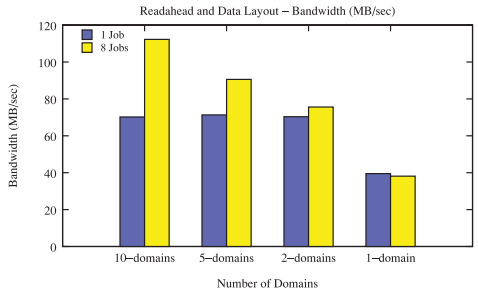


Fig. 8. The impact of ill-mapped data layout for readahead with queue depths of one and eight jobs.

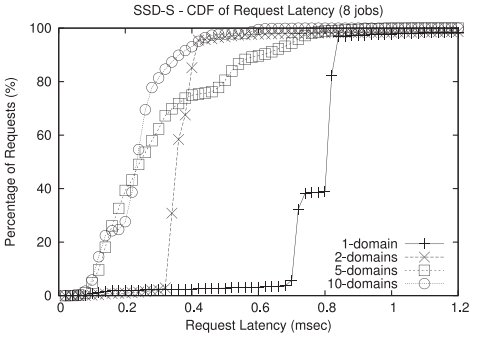


Fig. 9. CDF of latencies for sequential reads with 8 jobs to data mapped in 1–10 domains.

of 4KB, but between each two writes, we insert different numbers of random writes of 4KB data. For example, if we attempt to map all blocks into 1 domain, we insert 9 random writes of 4KB data. Then we use the toolkit to generate sequential reads with request size of 4KB and queue depth of one job andeight jobs on different physical layouts.

Figure 8 shows that as the logical blocks are increasingly concentrated into a subset of domains, the bandwidth decreases significantly (up to 2.9×). It is apparent that the ill-mapped data layout weakens the effectiveness of the readahead mechanism, since asynchronous reads are congested in one domain and data cannot be preloaded into the cache quickly enough to service the next read. We also can see that sequential reads with a high concurrency level (eight jobs in Figure 8) is more sensitive to the ill-mapped physical data layout. We collected the trace for sequential reads with eight jobs and compare the cumulative distribution function (CDF) of request latencies on different data layouts in Figure 9. In the worst case (one domain), a read incurs a latency of over 700μs, which is much worse than accessing data allocated in 10 domains.

6.6. Summary

In this section, we use microbenchmarks to show that parallelism can significantly improve performance, especially for random and small data accesses. In the meantime, our experimental results also show many dynamics and surprising results. We find that with parallelism, random reads can outperform sequential reads, and reads and writes have a strong interferences with each other and cause not only significant performance loss but also unpredictable performance. It is also evident in our study that the ill-mapped data layout can impair the effectiveness of I/O parallelism and readahead mechanism. Our results show that I/O parallelism can indeed provide significant performance improvement, but, in the meantime, we should also pay specific attention to the surprising results that come from parallelism, which provides us with a new understanding of SSDs.

7. CASE STUDIES IN DATABASE SYSTEMS

In this section, we present a set of case studies in database systems as typical data-intensive applications to show the opportunities, challenges, and research issues brought by I/O parallelism on SSDs. Our purpose is not to present a set of well-designed solutions to address specific problems. Rather, we hope that through these case studies, we can show that exploiting internal parallelism of SSDs can not only yield significant

performance improvement in large data-processing systems, but more importantly, it also introduces many emerging challenges.

7.1. Data Accesses in a Database System

Storage performance is crucial to query executions in database management systems (DBMS). A key operation of query execution is *join* between two *relations* (tables). Various *operators* (execution algorithms) of a join can result in completely different data access patterns. For warehouse-style queries, the focus of our case studies, two important join operators are *hash join* and *index join* [Graefe 1993]. Hash join sequentially fetches each *tuple* (a line of record) from the driving input relation and probes an in-memory hash table. Index join, in contrast, fetches each tuple from the driving input relation and starts index lookups on B^+ trees of a large relation. Generally speaking, hash join is dominated by sequential data accesses on a huge fact table, while index join is dominated by random accesses during index lookups.

Our case studies are performed on the PostgreSQL 8.3.4, which is a widely used open-source DBMS in Linux. The working directory and the database are located on SSD-S. We select Star Schema Benchmark (SSB) queries [O'Neil et al. 2009] as workloads (scale factor: 5). SSB workloads are considered to be more representative in simulating real warehouse workloads than TPC-H workloads [Stonebraker et al. 2007], and they have been used in recent research work (e.g., Abadi et al. [2008] and Lee et al. [2009a]). The SSB has one huge fact table (LINEORDER, 3340MB) and four small dimension tables (DATE, SUPPLIER, CUSTOMER, PART, up to 23MB). The query structures of all SSB queries are the same, that is, aggregations on equal joins among the fact table and one or more dimension tables. The join part in each query dominates the query execution. When executing SSB queries in PostgreSQL, hash join plans are dominated by sequential scans on LINEORDER, while index join plans are dominated by random index searches on LINEORDER.

7.2. Case 1: Parallelizing Query Execution

We first study the effectiveness of parallelizing query executions on SSDs. Our query-parallelizing approach is similar to the prior work [Tsirogiannis et al. 2009]. A star schema query with aggregations on multiway joins on the fact table and one or more dimension tables naturally has intraquery parallelism. Via data partitioning, a query can be segmented to multiple subqueries, each of which contains joins on partitioned data sets and preaggregation. The subqueries can be executed in parallel. The final result is obtained by applying a final aggregation over the results of subqueries.

We study two categories of SSB query executions. One is using the index join operator and dominated by random accesses, and the other is using the hash join operator and dominated by sequential accesses. For index join, we partition the dimension table for the first-level join in the query plan tree. For hash join, we partition the fact table. For index join, we select query Q1.1, Q1.2, Q2.2, Q3.2, and Q4.3. For hash join, besides Q1.1, Q1.2, Q2.2, Q4.1, and Q4.2, we have also examined a simple query (Q0), which only scans LINEORDER table with no join operation. We use Q0 as an example of workloads with little computation. Figure 10 shows the speedup (execution time normalized to the baseline case) of parallelizing the SSB queries with subqueries.

We have the following observations. (1) For index join, which features intensive random data accesses, parallelizing query executions can speed up an index join plan by up to *a factor of 5*. We have also executed the same set of queries on a hard disk drive and observed no performance improvement, which means the factor-of-5 speedup is not due to computational parallelism. This case again illustrates the importance of parallelizing random accesses (e.g., B^+-tree lookups) on an SSD. When executing an index lookup dominated query, since each access is small and random, the DBMS

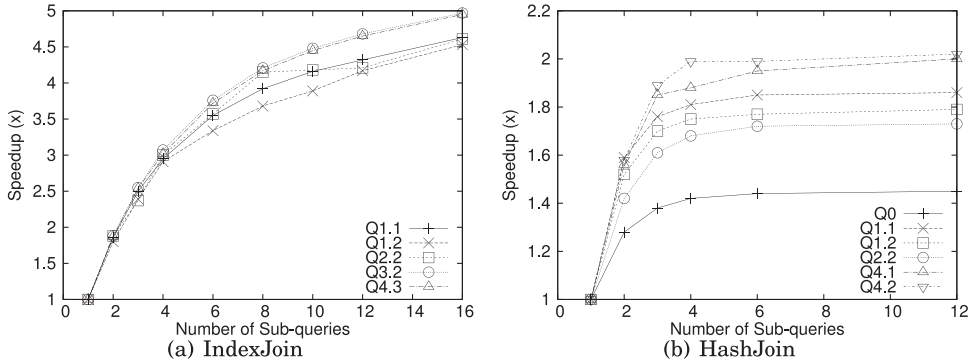


Fig. 10. Execution speedups of parallelizing SSB queries with index and hash join plans.

engine cannot fully utilize the SSD bandwidth. Splitting a query into multiple subqueries effectively reduces the query execution time. (2) For hash join, which features sequential data accesses, parallelizing query executions can speed up a hash join plan by up to *a factor of 2*. This is mainly because hash join is dominated by sequential accesses, which already benefit from internal parallelism to some extent. However, it still cannot fully exploit the SSD bandwidth, and parallelizing queries can achieve a lower but decent performance improvement. (3) Parallelizing query execution with no computation receives relatively less benefits. For example, parallelizing Q0 provides a speedup of *a factor of 1.4*, which is lower than other queries. Since Q0 simply scans a big table with no computation, the workload is very I/O intensive. In such a case, the SSD is fully saturated, which leaves little room for overlapping I/O and computation and limits further speedup through parallelism. In summary, this case clearly shows that in a typical data-processing application like database, I/O parallelism can truly provide substantial performance improvement on SSDs, especially for operations like index-tree lookups.

7.3. Case 2: Database-Informed Prefetching

The first case shows a big advantage of parallelizing a query execution over conventional query execution on SSDs. However, to achieve this goal, the DBMS needs to be extended to support query splitting and merging, which involves extra overhead and effort. Here we present a simpler alternative, *Database-Informed Prefetching*, to achieve the same goal for sequential-access queries, such as hash join.

We designed a user-mode daemon thread (only around 300 lines of C code) to asynchronously prefetch file data on demand. We added a hook function in the PostgreSQL server with only 27 lines of code to send prefetching commands to the daemon thread through the UNIX domain socket. When a query sequentially scans a huge file in hash join, the daemon is notified by the PostgreSQL server to initiate an asynchronous prefetching to load data from user-specified positions (e.g., each 32MB, in our settings). The daemon thread generates 256KB read requests to fully exploit the SSD bandwidth. Prefetched data are loaded into the OS page cache, rather than the database buffer pool, to minimize changes to the DBMS server code.

Figure 11 shows that, with only minor changes to the DBMS, our solution can keep the SSD as busy as possible and reduce query execution times by up to 40% (Q1.1). As expected, since the four queries used in our experiments are all dominated by large sequential reads, the performance improvement is relatively less significant than index joins but still substantial. An important merit of such a database-informed prefetching over the default OS-level prefetching is that with detailed application-level knowledge,

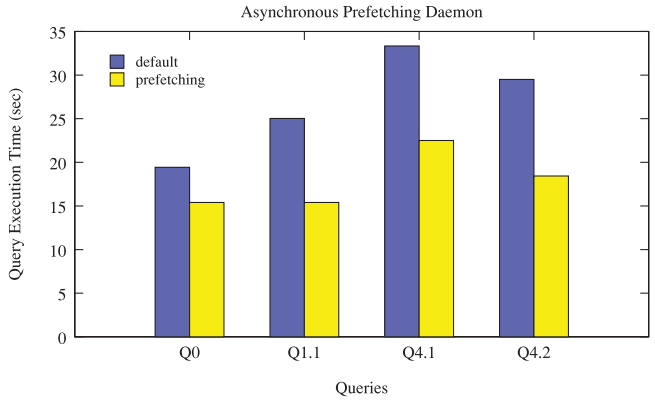


Fig. 11. Asynchronous prefetching in DBMS. *Default* and *prefetching* refer to the cases without and with prefetching, respectively.

the DBMS knows the exact data access patterns before executing a query, and thus aggressive prefetching can be safely applied without worrying about a low hit ratio that may be caused by mis-prefetching. In contrast, the default OS-level prefetching has to use a conservative prefetching, which cannot fully exploit the high bandwidth of SSDs. This case clearly indicates that even with only minor changes to DBMS, we can effectively speedup certain operations through parallelism.

7.4. Case 3: Interference between Parallel Queries

Although parallelizing I/O operations can improve system bandwidth in aggregate, concurrent workloads may interfere with each other. In this case, we give a further study on such an interference in macrobenchmarks by using four workloads as follows.

(1) *Sequential Read*: the SSB query Q1.2 executed using hash join, denoted as QS.
(2) *Random Read*: the SSB query Q1.2 executed using index join, denoted as QR.
(3) *Sequential Write*: *pgbench* [Pgbench 2011], a tool of PostgreSQL for testing OLTP performance. It is configured with 2 clients and generates frequent synchronous log flushing with intensive overwrites and small sequential writes.
(4) *Random Write*: *Postmarkm* [Shah and Noble 2007], a widely used file system benchmark. It emulates an email server, and we configure it with 1,000 files, 10 directories, and 20,000 transactions. It is dominated by synchronous random writes.

We run each combination of a pair of the workloads on SSD, except *pgbench*, which is measured using the reported transaction/second, the other three workloads are measured directly using execution times. When evaluating performance for running a target workload with another workload, we repeatedly execute the co-runner until the target workload completes. We report the performance degradations (%), compared to the baseline of running them alone, as shown in Figure 12.

As expected, Figure 12 shows a strong interference between I/O intensive workloads. Similarly to Section 6.2, we find that running two random database workloads (QR/QR) has the least interference (only 6%). However, we also have observed that the sequential read workload (QS) has a strong performance impact to its co-runners, both reads and writes, by up to 181% (QR/QS), which differs from what we have seen in microbenchmarks. This is because sequential reads in practical systems often benefit from the OS-level prefetching and form large requests. When running with other workloads, it would consume the most SSD bandwidth and interfere its co-runners. Sequential reads can even obtain some performance improvement (20% to 25%). This

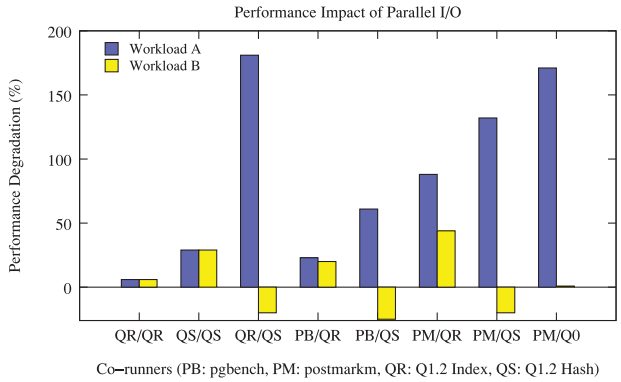


Fig. 12. Interference between workloads. PB, PM, QR, QS refer to pgbench, postmarkm, q1.2-index, and q1.2-hash, respectively.

result is repeatable across many runs and CPU is confirmed not the bottleneck. After examining the I/O trace, we found that without a co-runner, the query Q1.2 scans a large table file in a sequential pattern, which results in a low (14) queue depth with a large request size (512 sectors). With an injection of writes or random reads, the large requests are broken down to several smaller reads (256 sectors), which doubles the queue depth. As a result, reads are further parallelized, which aggressively overlaps computation and I/Os and results in better performance. To confirm this, we run Q0, which only sequentially scans the LINEORDER table without any computation, together with postmarkm (PM/Q0), and we find that without any computation, Q0 cannot benefit from the increased queue depth and becomes 0.9% slower. This case shows that the interference between co-running workloads in practice may lead to complicated results, and we should pay specific attention to minimize such interference.

7.5. Case 4: Revisiting Query Optimizer

A critical component in DBMS is the *query optimizer*, which decides the plan and operators used for executing a query. Traditionally, the query optimizer implicitly assumes the underlying storage device is a hard drive. Based on such an assumption, the optimizer estimates the execution times of various job plans and selects the lowest-cost query plan for execution. On a hard drive, a hash join enjoys an efficient sequential data access but needs to scan the whole table file; an index join suffers random index lookups but only needs to read a table partially. On an SSD, the situation becomes even more complicated, since parallelizing I/Os would weaken the performance difference of various access patterns, as we see previously in Section 6.1.

In this case, we study how the SSD internal parallelism will bring new consideration for the query optimizer. We select a standard Q2.2 and a variation of Q1.2 with a new predicate ("d_weeknuminyear=1") in the DATE table (denoted by $Q1.2_n$). We execute them first with no parallelism and then in a parallel way with subqueries. Figure 13 shows the query execution times of the two plans with 1 to 16 subqueries. One subquery means no parallelism.

In the figure, we find that SSD parallelism can greatly change the *relative strengths* of the two candidate plans. Without parallelism, the hash join plan is more efficient than the index join plan for both queries. For example, the index join for Q2.2 is 1.9 times slower than the hash join. The optimizer should choose the hash join plan. However, with parallelized subqueries, the index join outperforms the hash join for both queries. For example, the index join for Q2.2 is 1.4 times faster than the hash join.

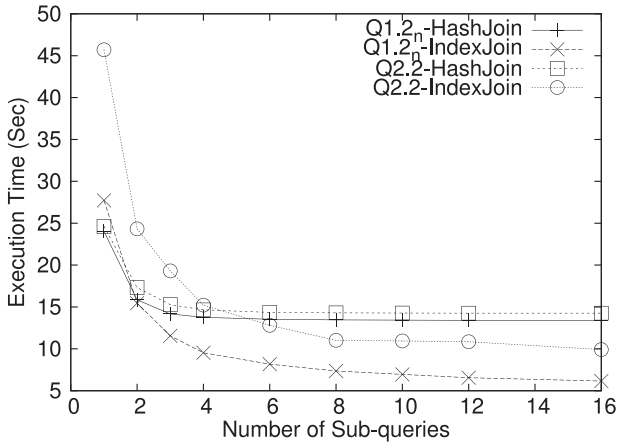


Fig. 13. Hash Join vs. Index Join with parallelism.

This implies that the query optimizer *cannot* make an optimal decision if it does not take parallelism into account when estimating the execution costs of candidate plans on SSDs. This case strongly indicates that when switching to an SSD-based storage, applications designed and optimized for magnetic disks must be carefully reconsidered, otherwise, the achieved performance can be suboptimal.

8. INTERACTION WITH RAID

Internal parallelism of SSD provides great performance potential for optimizing applications. RAID, which organizes an array of storage devices as a single logical device, can provide another level of parallelism to further parallelize I/Os, which we call *external parallelism*. The interaction between the two levels of parallelism is important. For example, if I/O requests are congested in one device, although internal parallelism can be well utilized, cross-device parallelism would be left unexploited. In this section, we study the interaction between the two layers by examining the effect of RAID data layout to the overall performance.

For our experiments, we use a high-end PCI-E SSD manufactured by the same vendor for our prior experiments to study the SSD-based RAID. This 800GB SSD, named SSD-P, is equipped with 25nm MLC NAND flash memories and exposes four independent volumes to the host. Each volume is 200GB and can operate independently. We use the `mdadm` tool in Linux to combine the four volumes and create a single RAID-0 device (`/dev/md0`) for analyzing the interaction between internal parallelism and external parallelism. The high-speed PCI-E (x8) interface allows us to reach a bandwidth of nearly 2GB/s.

8.1. Microbenchmark

We first perform a set of microbenchmarks by using the Intel OpenStorage Toolkit to generate four types of workloads, *random read*, *sequential read*, *random write*, and *sequential write*, with various request sizes (4KB to 256KB), with a queue depth of 32 jobs. We report the aggregate bandwidth as the performance metric.

In a RAID-0 storage, logical blocks are evenly distributed across the participating devices in a round-robin manner, and the chunk size determines the size of the unit that is contiguously allocated in each individual device. In other words, the chunk size determines the data layout of RAID storage. By evenly distributing chunks, all participating devices can operate independently in parallel. Built on SSDs, where each

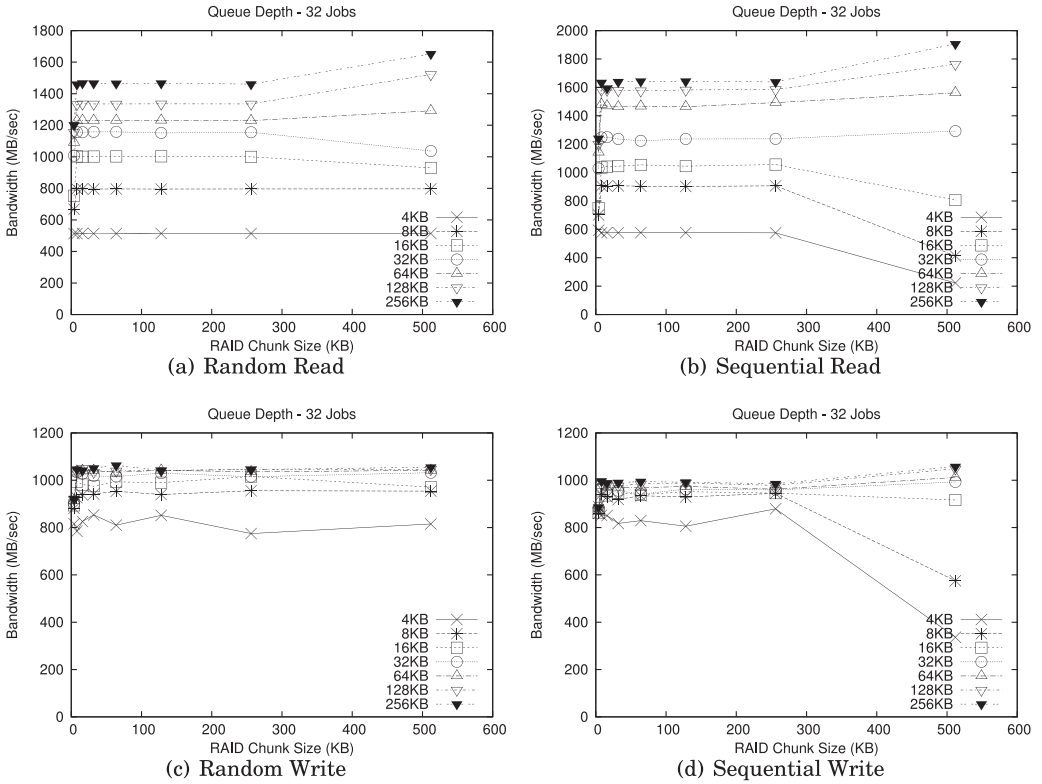


Fig. 14. Interaction of parallelism with RAID (queue depth, 32 jobs).

individual device already shows substantial performance potential for parallel I/Os, various RAID chunk sizes can influence the parallelism across the devices and also the overall performance of an RAID device. Figure 14 presents the results of the four workload patterns. We can see that for random reads with a relatively large request size, increasing chunk size can improve performance. For example, the 256KB workload can be improved from about 1,201MB/sec (chunk size 4KB) to 1,652MB/sec (chunk size 512KB). This is because with a large chunk size, a request is more likely to fall into only one device. For a large request, the internal parallelism of each device can be fully utilized, and a large queue depth ensures that all devices can be fully utilized, which leads to better performance. In contrast, this effect is not observed with small requests (e.g., 4KB), since the small requests cannot fully leverage the internal parallelism of each single device. For sequential reads (Figure 14(b)), as chunk size increases, we see a performance degradation with small requests. The reason is that as we increase the chunk size, despite the parallel I/Os, it is more likely to congest multiple requests in one device, since the access pattern is sequential rather than random. For example, having 32 parallel sequential requests of 4KB, the 32 requests can fall in one 128KB chunk, which leaves the other three devices unused. A similar pattern can be observed with sequential writes (see Figure 14(d)). For random writes, requests experience a performance increase as the chunk size increases, but the trend stops at small chunk size (e.g., 32KB). The reason is similar to random reads: A relatively larger write chunk size can ensure internal parallelism to be fully utilized first. However, writes cannot provide a headroom for performance improvement as large as reads. In general, we

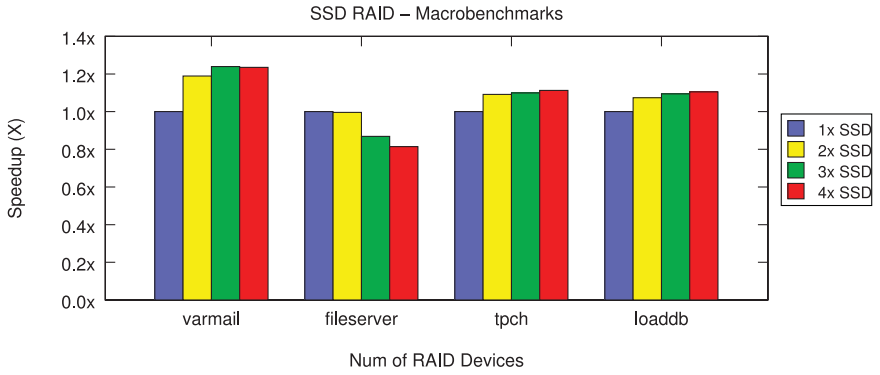


Fig. 15. Macrobenchmarks with SSD-based RAID.

can see that a 32KB chunk size is a sweet spot for this particular case. Comparing random writes and sequential writes, we also can see that both can reach similar peak performance (around 1GB/s), which again indicates that the SSD internally handles random and sequential writes similarly.

8.2. Macrobenchmarks

To show the impact of external parallelism to the end-to-end performance of applications, we select four different workloads as our macrobenchmarks: *Varmail* is a workload in the Filebench suite [Filebench 2015]. This workload simulates a mail server, which generates create-append-sync, read-append-sync, and read-delete file system operations. It has an equal amount of read and write operations. We configure the workload with 1,000,000 files, 16KB mean file size, and 100 threads. This benchmark reports the throughput (operations per second). *Fileserver* is another workload in Filebench. This workload simulates a file server, which generates create, delete, append, read, write, and attribute operations. The read/write ratio is about 1/2. In our experiments, we configure 100,000 files, a mean file size of 128KB, and 50 threads. *TPC-H* is an online analytical processing (OLAP) workload running with PostgreSQL database (v8.4). It is configured with a scale factor of 10, and we use the execution time of completing all 22 queries as our performance metric. *LoadDB* simulates a database loading process. It uses *dbgen* in the TPC-H benchmark to generate raw table files with a scale factor of 10. Then the database is created and data are loaded from the raw table files, and, finally, indexes are created. This workload reports the execution time. For our experiments, we configure four RAID configurations by using one to four devices and we desire to see the performance difference by increasing the opportunities of external parallelism. We use RAID-0 and a moderate chunk size (64KB). In order to show both throughput and execution time data in one figure, we normalize the results to the baseline case, using only one device. Figure 15 shows the results of performance speedups.

As shown in Figure 15, we can see that the impact of creating external parallelism opportunities to applications varies across the four workloads. For highly parallelized workloads, such as *varmail*, although its I/Os are relatively small (mean size is 16KB), increasing the number of RAID devices can effectively distribute the small I/O requests to different devices and bring substantial performance benefit (24%). *Fileserver* is interesting. It shows a decreasing performance as we add in more SSDs into the RAID. Considering our chunk size is 64KB and the mean file size is 128KB, many I/Os will split into multiple devices. In the meantime, since the workload has a mixed and relatively high write ratio, some SSDs could be congested or slowed down for handling write requests due to internal operations, such as garbage collection. As a result, a

Table III. SSD Specification

Name	Vendor	Capacity	Interface	Flash Type	Lithography
SSD-A	#1	240GB	SATA III	TLC NAND	19nm
SSD-B	#2	250GB	SATA III	TLC V-NAND	40nm
SSD-C	#2	256GB	SATA III	MLC V-NAND	40nm
SSD-D	#3	240GB	SATA III	MLC NAND	20nm
SSD-E	#3	400GB	SATA III	MLC NAND	25nm

request that is across multiple devices could be slowed down if any involved device is slow. Prior research, called Harmonia [Kim et al. 2011], has studied this effect in particular by coordinating garbage collections in SSDs. For workloads lacking sufficient parallelism but involving many I/O operations, such as *TPC-H* and *LoadDB*, we find that they only receive relatively moderate performance benefits (10%). For example, *LoadDB* is highly write intensive and most writes are sequential and large (512KB). As a result, increasing the number of devices can help leverage additional device bandwidth, but the low queue depth leaves less space for further improvement. Similarly, *TPC-H* is read intensive but also lacks parallelism. This case indicates that when creating an SSD-based RAID storage, whether a workload can receive benefits depends on the workload patterns. A basic rule is to fully utilize the parallelism delivered at both levels. In general, with sufficiently large and parallel I/Os, a reasonably large chunk size can effectively ensure each device to receive a sufficiently large request to exploit its internal parallelism and keep all device active. On the other hand, we should also be careful about the possible slowdown caused by cross-device dependency.

9. OBSERVATIONS WITH NEW-GENERATION SSDS

SSD technology is rapidly developing. In recent years, the industry has experienced a quick development in fabrication and manufacturing process of NAND flash. Nowadays, the fabrication process is entering a sub-20nm era; the high-density TLC NAND is replacing MLC and SLC NAND flash and becoming the mainstream; the 3D V-NAND flash memory technology further breaks the scale limit of the planar (2D) flash and provides higher reliability and better write speed. Inside SSDs, the FTL design is also becoming more and more sophisticated; most SSDs on the market are now adopting the the third-generation SATA interface, which can provide a peak bandwidth of 6.0Gbps; many devices are integrated with a large on-device DRAM/flash cache for buffering purposes. All these technologies together have enabled a great boost in performance and substantially reduced the average cost per gigabyte. On the other hand, as the SSD internals are increasingly complex, the performance behaviors of SSDs are becoming more and more diverse. In this section, we select five new-generation SSDs of three major brands on the market. These SSDs present a comprehensive set of devices for revisiting our experimental studies. Table III gives more details about these SSDs.

9.1. Performance Benefits of I/O Parallelism

Figure 16 shows the benefits of increasing I/O parallelism with the new SSDs. In comparison to results in Section 6.1, we made the following observations: (1) The new generation of SSDs has exhibited significantly higher performance. Almost all these SSDs can achieve a peak read bandwidth of above 450MB/sec and a peak write bandwidth of more than 400MB/s. Such a performance improvement is a result of both the interface upgrade and other optimizations. (2) Most SSDs show performance benefit from I/O parallelism, similarly to our observations in old-generation SSDs. This indicates that increasing I/O parallelism can enable several times higher speedups. As an exception, SSD-A show interference when increasing the queue depth for sequential reads over four parallel jobs. SSD-A is a product in the low-end range and

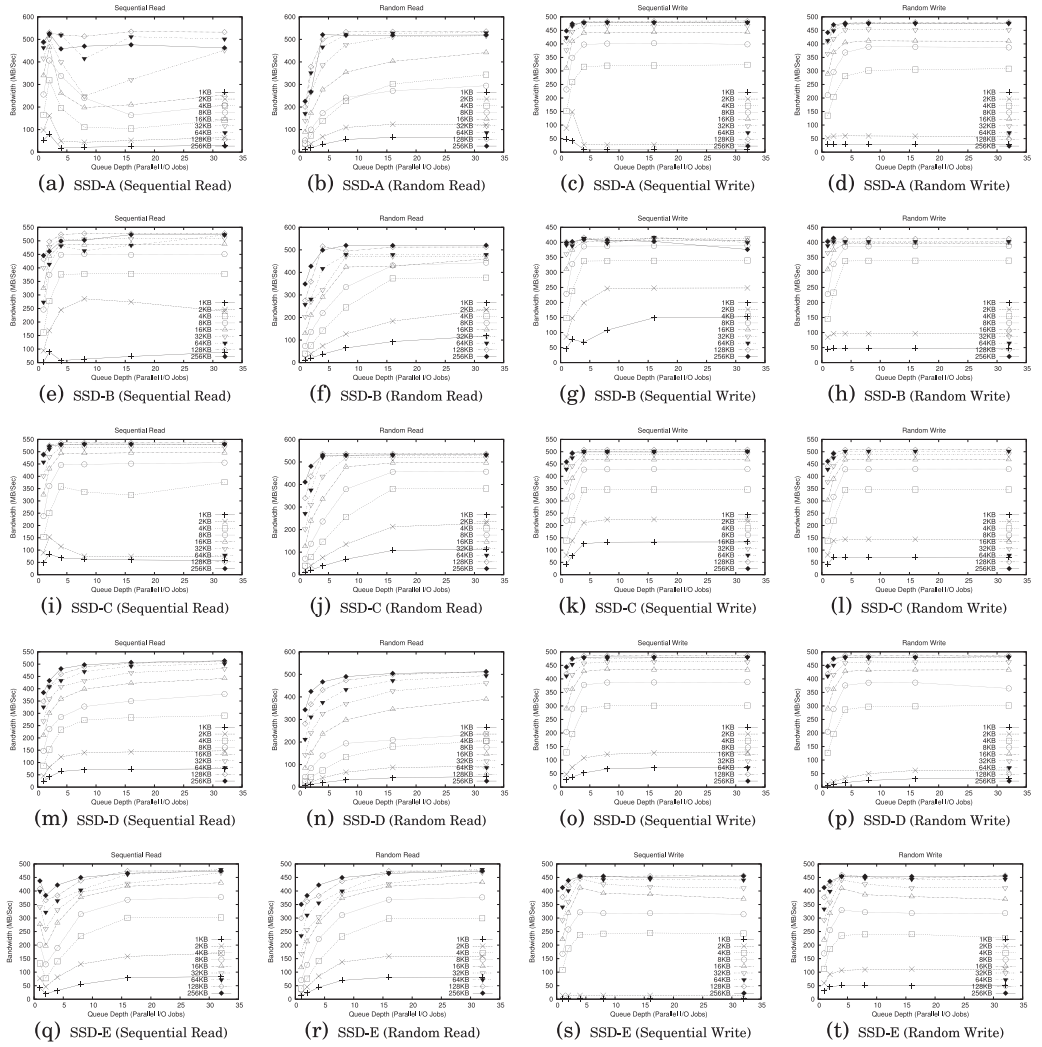
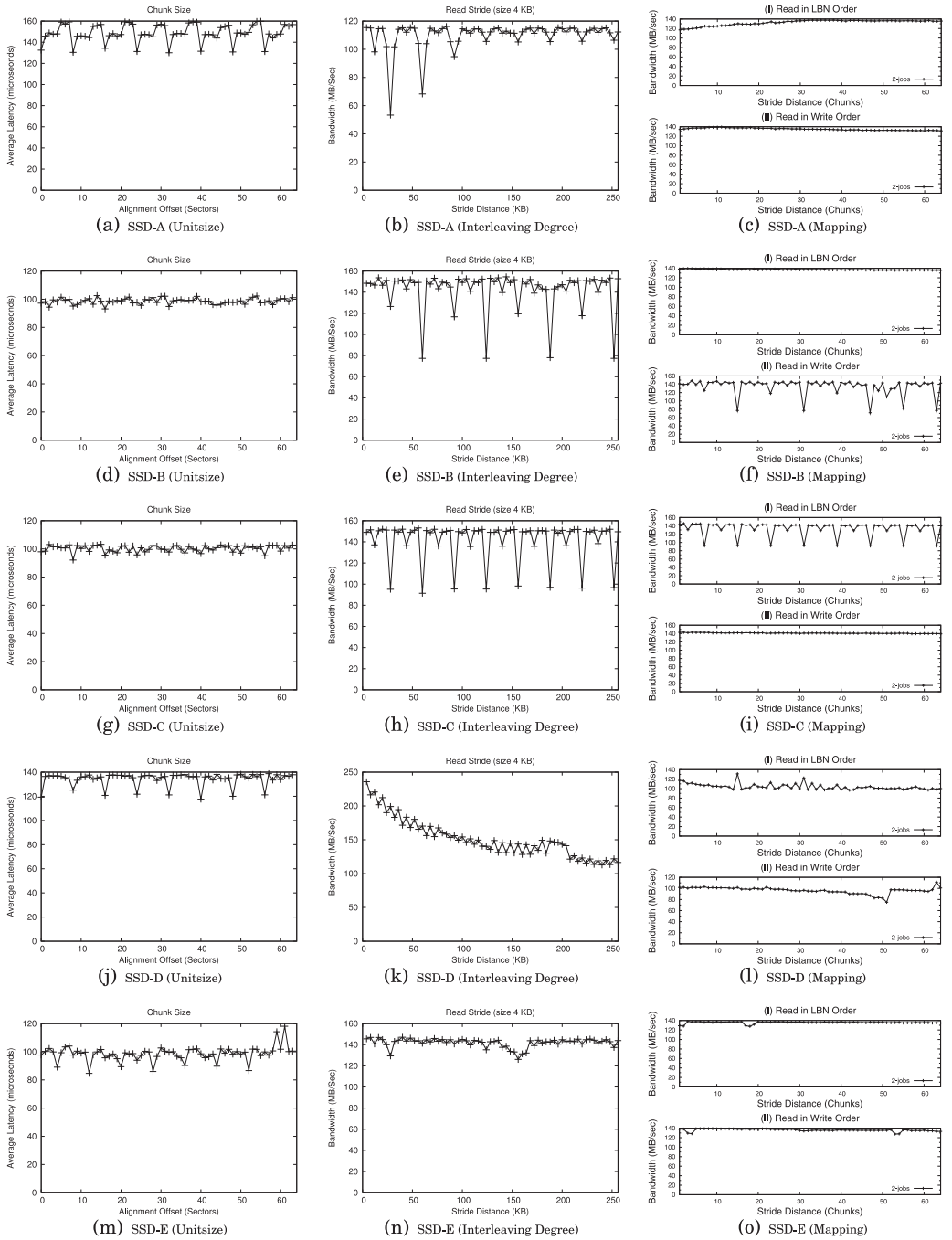


Fig. 16. Bandwidths of SSDs with increasing queue depth (the number of concurrent I/O jobs).

adopts low-cost high-density TLC planar NAND flash. Although the SSD includes a large DRAM/SLC cache on device, internal resource congestion still shows a strong interference with high I/O parallelism, and such a performance loss is evident when handling multiple concurrent I/O jobs.

9.2. Characterizing the SSD Internals

The new-generation SSDs have a complex internal structure. Some SSDs adopt new V-NAND flash chips and some have a large DRAM or SLC NAND flash cache (e.g., 512MB). Compared to early-generation SSDs, all these factors make characterizing SSD internals more difficult. In this section, we repeat the same set of experiments in Section 5. Figure 17 shows the results. In the figure, we can see that the curves do exhibit rather different patterns, as expected. We have the following observations: (1) For detecting the chunk size, some SSDs show repeatable and recognizable patterns



(a) SSD-A (Unitsize) (b) SSD-A (Interleaving Degree) (c) SSD-A (Mapping)

(d) SSD-B (Unitsize) (e) SSD-B (Interleaving Degree) (f) SSD-B (Mapping)

(g) SSD-C (Unitsize) (h) SSD-C (Interleaving Degree) (i) SSD-C (Mapping)

(j) SSD-D (Unitsize) (k) SSD-D (Interleaving Degree) (l) SSD-D (Mapping)

(m) SSD-E (Unitsize) (n) SSD-E (Interleaving Degree) (o) SSD-E (Mapping)

Fig. 17. Characterizing SSD Internals on new-generation SSDs.

but some others show no clear patterns. For example, SSD-A, SSD-D, and SSD-E show a regular pattern. SSD-A exhibits a two-step pattern for detecting the chunk size. This is likely to be caused by the different page access cost in high-density TLC NAND flash (LSB, CSB, MSB pages). According to the pattern, we still can roughly guess the logical unit size, which is very likely to be 4KB. SSD-D and SSD-E also show a clear pattern, which implies a 4KB page. SSD-B and SSD-C, however, show no strong patterns for accurately identifying the chunk size. It could be a result of the cache interference or a different implementation for handling a single chunk access. (2) For detecting the interleaving degree, SSD-A, SSD-B, and SSD-C produce regular patterns. SSD-A shows a regular but weakening pattern. According to the unit size, the number of channels is likely to be four channels. For SSD-B and SSD-C, both show a steady and repeatable pattern, based on which we believe the two SSDs have four channels as well. Interestingly, we have observed three repeatedly appearing dips on SSD-B and two dips on SSD-C, which indicates that SSD-B is more aggressively sharing the channels. SSD-D and SSD-E, which are built by the same vendor, exhibit an unclear pattern, which indicates that the internal implementation is fairly complex and interferes our detection based on an assumed architecture. (3) For the mapping policy, only SSD-B and SSD-C respond to our detecting mechanism and clearly shows two patterns: SSD-B uses a write-order based dynamic mapping, while SSD-C uses the LBN-based static mapping. The reason why the two SSDs from the same manufacture adopt two completely different mapping policy is unknown. One possible reason is that SSD-B uses a relatively lower-end TLC V-NAND flash chips, so using the dynamic write-order based mapping could better optimize the write performance. SSD-C, in contrast, uses higher-speed MLC V-NAND flash chips and therefore has a relatively lower pressure for handling writes.

In general, this set of experiments have confirmed that new-generation SSDs show fairly distinct patterns due to the complex internal designs, although some basic internal structure is still roughly characterizable. We expect that as SSD internals continue to be increasingly sophisticated, uncovering SSD internals completely by using a black-box approach would be more and more difficult. For this reason, directly exposing low-level hardware details to the upper-level components is a promising direction for future SSDs [Ouyang et al. 2014].

9.3. Impact of Garbage Collection

Garbage collection operations are the main source of internal overhead that affects observable SSD performance. While moving data internally inside an SSD, parallelizing I/Os may not achieve the same level of effectiveness as the situation without garbage collection. In this section, we illustrate such a situation by running 32 parallel 16KB requests. In order to trigger the garbage collection process, we first randomly write the device with 32 small (4KB) requests for 60s and 600 seconds, respectively, and then immediately start the workload, which generates 32 parallel 16KB requests in four different patterns. Figure 18 shows the comparison between the performance with and without garbage collection. As shown in the figures, after intensive random writes, the impact of garbage collection varies across different devices and workloads. In general, we observe less interference to reads caused by garbage collection than to writes. It is consistent with our understanding that, since most reads are synchronous and performance critical, SSD internals often optimize reads with a high priority. Also, as reads do not rely on garbage collection to refill the pool of clean blocks, garbage collection can be delayed to minimize the potential interference. In contrast, writes, especially random writes, are impacted by garbage collection greatly. With 60-second intensive writes, SSD-A, SSD-B, and SSD-C all have more than 50% bandwidth decrease for random writes. As we further increase the write volume by 10 times, we see that a

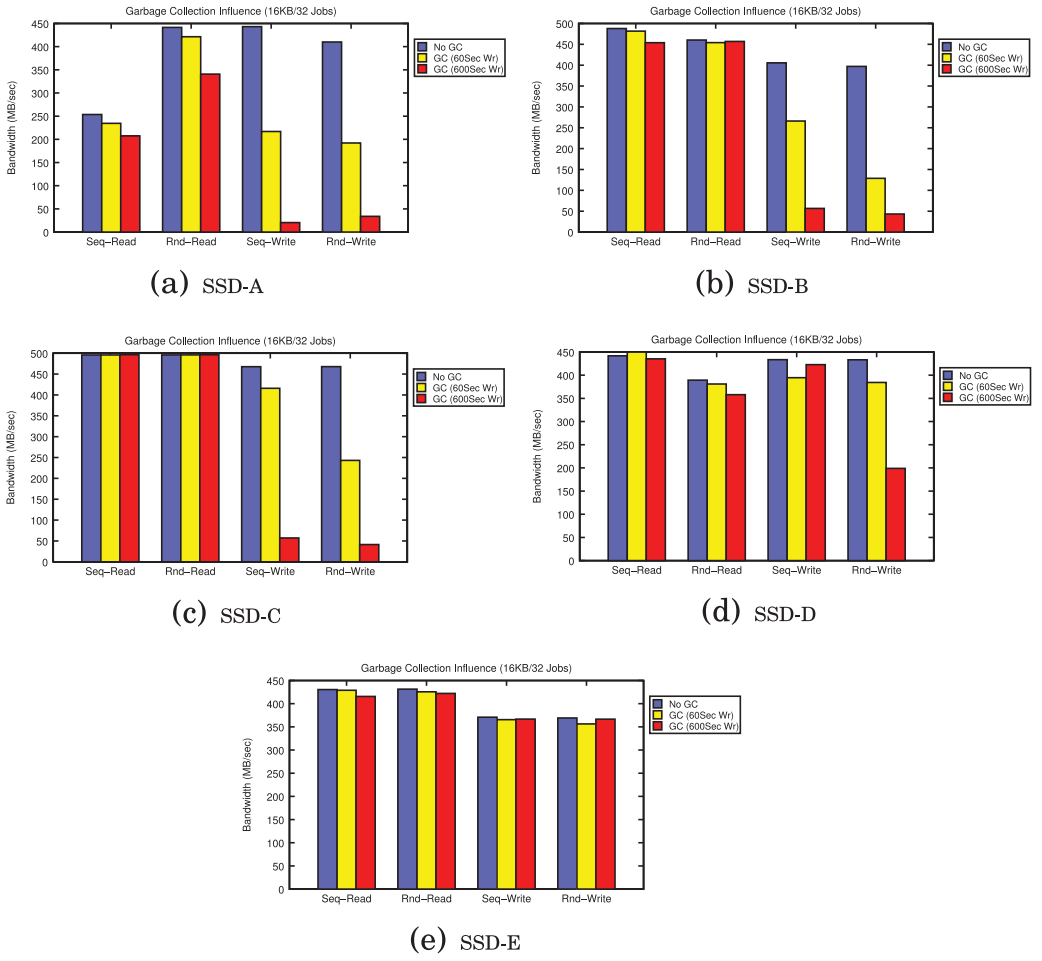


Fig. 18. Influence of garbage collection to the effectiveness of parallelizing I/Os.

significant performance decrease also appears in SSD-D, in addition to SSD-A, SSD-B, and SSD-C. SSD-E still performs well. We believe that this is due to the large over-provisioning ratio in SSD-E. In SSD-E, which is a high-end product designed for data centers, 25% of the NAND flash on SSD-E is used for over-provisioning, which absorbs heavy random write traffic. For mid-range products, such as SSD-B, SSD-C, and SSD-D, the performance impact of garbage collection after a long-time writes is evident. For this reason, some recent research, such as HIOS [Jung et al. 2014] and Harmonia [Kim et al. 2011], has worked on optimizing the efficiency of garbage collections in particular. On the other hand, we also need to note that most SSDs perform reasonably well with a moderate volume of random writes. Considering the fact that most writes are asynchronous, the end-to-end impact to applications would be further reduced by host side page cache.

9.4. Discussions

The implications of our experiments with new-generation SSDs are twofold: First, as SSD technologies advance and internal structure gets increasingly complex and

diverse, it is more and more difficult to accurately uncover the architectural details of the device. Our approach assumes a device as a black box that follows a general architecture design. Each new optimization integrated into the device will unfortunately weaken such an assumption as well as the observable performance patterns. In fact, our methodology is based on observable performance difference for handling various carefully designed workloads. With heavy optimizations, such a performance difference is diminishing or nearly unrecognizable, which is good for end users but bad news for researchers who desire to uncover the internal structures. On the other hand, we can see that many SSDs still can show certain patterns, more or less. This indicates that some general design principles are still in effect. Most importantly, the following key conclusion is still valid: In order to fully exploit the performance potential of flash SSD, we must fully utilize the internal parallelism. We believe this set of experiments with new-generation SSDs assures us the importance of internal parallelism in such new technologies.

10. IMPLICATIONS TO SYSTEM AND APPLICATION DESIGNERS

Having presented our experimental and case studies, we are in a position to present several important implications to system and application designers. Our experimental studies and analysis strongly indicate that we should consider parallelism as a *top priority* and revisit many existing optimizations designed for magnetic disks. We hope that our new findings can provide effective guidance and enhance our understanding on the properties of SSDs in the context of I/O parallelism. This section also summarizes our answers to the questions raised at the beginning of this article.

Benefits of parallelism on SSD: Parallelizing data accesses can provide substantial performance improvement, and the significance of such benefits depends on workload access patterns, resource redundancy, and flash memory mediums. In particular, small random reads benefit the most from parallelism. Large sequential reads achieve less significant but still impressive improvement. Such a property of SSDs provides many opportunities for performance optimization in application designs. For example, many critical database operations, such as index-tree search, feature intensive random reads, which is exactly a best fit in SSDs. Application designers can focus on parallelizing these operations. Meanwhile, sequential-access-dominated operations, such as hash join and sort, can be parallelized through aggressive asynchronous prefetching with proper hints from applications. In our experiments, we did not observe obvious negative effects of over-parallelization. In general, setting the concurrency level slightly over the number of channels is a reasonable choice. Finally, for practitioners who want to leverage the high parallel write performance, we recommend adopting high-end SSDs to provide more headroom for serving workloads with intensive writes. In our experiments, we also found a recent technical breakthrough in flash chips (e.g., 3D V-NAND) that makes MLC or even TLC flash SSDs perform reasonably well.

Random reads: An interesting finding is that with parallelism, random reads performance can be significantly improved to the level of large sequential reads without parallelism. Such a finding indicates that we cannot continue to assume that sequential reads are always better than random reads, which has been a long-existing common understanding for hard drives. This observation has a strong system implication. Traditionally, operating systems often make tradeoffs for the purpose of organizing large sequential reads. For example, the anticipatory I/O scheduler [Iyer and Druschel 2001] intentionally pauses issuing I/O requests for a while and anticipates organizing a larger request in the future. In SSDs, such optimization may become less effective and sometimes may be even harmful due to unnecessarily introduced delays. Issuing random reads as quickly as possible can also fully utilize the internal resources as sequential reads. On the other hand, some application designs could become more complicated.

For example, it is more difficult for the database query optimizer to choose an optimal query plan, because simply counting the number of I/Os and using static equations can no longer satisfy the requirement for accurately estimating the relative performance strengths of different join operators with parallelism. A parallelism-aware cost model is needed to handle such a new challenge.

Random writes: Contrary to our common understanding, parallelized random writes can achieve high performance, sometimes even better than reads. We also find that writes are no longer highly sensitive to patterns (random or sequential) as commonly believed. The uncovered mapping policy explains this surprising result from the architectural level: An SSD may internally handle random writes and sequential writes in the same way, regardless of access patterns. The implications are twofold. On one hand, we can consider how to leverage the high write performance through parallelizing writes. Some related problems would emerge, however, for example, how can we commit synchronous transaction logging in a parallel manner while maintaining a strong consistency [Chen 2009]? On the other hand, it means that optimizing random writes specifically for SSDs may not continue to be as rewarding as in early generations of SSDs, and trading off read performance for writes would become a less attractive option. But we should still note that in extreme conditions, such as day-long writes [Stoica et al. 2009] or under serious fragmentation [Chen et al. 2009], random writes are still a research issue.

Interference between reads and writes: In both microbenchmarks and macrobenchmarks, we find that parallel reads and writes on SSDs interfere strongly with each other and can cause unpredictable performance variance. In OS kernels, I/O schedulers should pay attention to this emerging performance issue and avoid mingling reads and writes as much as possible. A related research issue is concerned with how to maintain a high throughput while avoiding such interference. At the application level, we should also be careful of the way of generating and scheduling reads and writes. An example is the hybrid-hash joins in database systems, which have clear phases with read-intensive and write-intensive accesses. When scheduling multiple hybrid-hash joins, we can proactively avoid scheduling operations with different patterns. A rule of thumb is to schedule random reads together and separate random reads and writes whenever possible.

Physical data layout: The physical data layout in SSDs is dynamically determined by the order in which logical blocks are written. An ill-mapped data layout caused by such a dynamic write-order-based mapping can significantly impair the effectiveness of parallelism and readahead on device. In server systems, handling multiple write streams is common. Writes from the same stream can fall in a subset of domains, which would be problematic. We can adopt a simple random selection for scheduling writes to reduce the probability of such a worst case. SSD manufacturers can also consider adding a randomizer in the controller logic to avoid this problem. On the other hand, this mapping policy also provides us a powerful tool to *manipulate* data layout to our needs. For example, we can intentionally isolate a set of data in one domain to cap the usable I/O bandwidth and prevent malicious applications from congesting the I/O bus.

Revisiting application designs: Internal parallelism in SSDs brings many new issues that have not been considered before. Many applications, especially data-processing applications, are often heavily tailored to the properties of hard drives and are designed with many implicit assumptions. Unfortunately, these disk-based optimizations can become suboptimal on SSDs, whose internal physical structure completely differs. This calls for our attention to revisiting carefully the application designs to make them fit well in SSD-based storage. On the other hand, the internal parallelism in SSDs also enables many new opportunities. For example, traditionally DBMS designers often assume an HDD-based storage and favor large request sizes. SSDs can

extend the scope of using small blocks in DBMS and bring many desirable benefits, such as improved buffer pool usage, which is highly beneficial in practice.

External and internal parallelism: As SSDs are organized into an RAID storage, external parallelism interacts with internal parallelism. Our studies show that various workloads have different sensitivities to such an interaction. To effectively exploit parallelism opportunities at both levels, the key issue is to generate large, parallel I/Os to ensure each level have sufficient workload to be fully active. In our experiments, we also have confirmed that the actual effect of increasing external parallelism is application dependent, which needs our careful study on workload pattern.

Architectural complexity in new-generation SSDs: As flash memory and SSD technologies continue to advance, the increasingly diverse flash chip technologies (e.g., TLC, V-NAND) and complex FTL design make uncovering the SSD internals more and more challenging. Our studies on a set of recent products show that completely relying on black-box based techniques to uncover SSD internals will be difficult to accurately characterize all the architectural details. This suggests that SSD vendors should consider to expose certain low-level information directly to the users, which could enable a huge amount of optimization opportunities. On the other hand, our experiments also show that fully exploiting internal parallelism is still the most important goal.

In essence, SSDs represent a fundamental change of storage architecture, and I/O parallelism is *the key* to unlocking the huge performance potential of such an emerging technology. Most importantly, with I/O parallelism a low-end personal computer with an SSD is able to deliver as high an I/O performance as an expensive high-end system with a large disk array. This means that parallelizing I/O operations should be carefully considered in not only high-end systems but also in commodity systems. Such a paradigm shift would greatly challenge the optimizations and assumptions made throughout the existing system and application designs, which demands extensive research efforts in the future.

11. OTHER RELATED WORK

Flash memory-based storage technology is an active research area. In recent years, flash devices have received strong interest and been extensively studied [Chen 2009; Tsirogiannis et al. 2009; Lee et al. 2008; Canim et al. 2009; Do and Patel 2009; Agrawal et al. 2009; Lee and Moon 2007; Li et al. 2009; Nath and Gibbons 2008; Koltsidas and Viglas 2008; Graefe 2007; Lee et al. 2009b; Bouganim et al. 2009; Stoica et al. 2009; Josephson et al. 2010; Soundararajan et al. 2010; Boboila and Desnoyers 2010]. Open-source projects, such as OpenNVM [2015] and OpenSSD [2015], are also under active development. In this section we present the research work that is most related to internal parallelism of SSDs.

A detailed description of the hardware internals of flash memory-based SSD has been presented in prior work [Agrawal et al. 2008; Dirik and Jacob 2009]. Another early work has compared the performance of SSDs and HDDs [Polte et al. 2008]. These studies on architectural designs of SSDs regard parallelism as an important consideration to provide high bandwidth and improve high-cost operations. In our prior work [Chen et al. 2009], we have studied SSD performance by using non-parallel workloads, and the main purpose is to reveal the internal behavior of SSDs. This article focuses on parallel workloads. Bouganim et al. have presented a benchmark tool called uFLIP to assess the flash device performance and found that increasing concurrency cannot improve performance of early generations of SSDs [Bouganim et al. 2009]. Lee et al. have studied the advances of SSDs and reported that with NCQ support, the recent generation of SSDs can achieve high throughput [Lee et al. 2009b]. Hu et al. have studied the multilevel internal parallelism in SSDs and pointed out that different levels of parallelism have distinct impact to the effect of optimization [Hu

et al. 2013]. Jung and Kandemir have revisited SSD design [Jung and Kandemir 2013] and found that internal parallelism resources have been substantially underutilized [Jung and Kandemir 2012]. This article and its early version [Chen et al. 2011a] show that internal parallelism is a critical element to improve I/O performance and must be carefully considered in system and application designs.

Another set of prior studies has applied optimization by leveraging internal parallelism. Jung et al. have proposed a request scheduler, called PAQ, in order to avoid resource contention for reaching full parallelism [Jung et al. 2012]. Sprinkler [Jung and Kandemir 2014] further improves internal parallelism by scheduling I/O requests based on internal resource layout. Roh et al. introduce B+-tree optimization methods by utilizing internal parallelism and also confirm our findings [Roh et al. 2012]. Wang et al. present an LSM-tree-based key-value store on open-channel SSD by exploiting the abundant internal parallelism resources in SSDs [Wang et al. 2014]. Park and Shen present an I/O scheduler, called FIOS, for flash SSDs and exploiting internal parallelism is also a design consideration [Park and Shen 2012]. Ouyang et al. present a software defined flash (SDF) design to directly expose internal parallelism resources to applications for the purpose of fully utilizing available internal parallelism in flash [Ouyang et al. 2014]. Guo et al. have proposed a write buffer management scheme, called PAB, to leverage parallelism inside SSDs [Guo et al. 2013]. Vasudevan et al. also present a vector interface to exploit potential parallelism in high-throughput NVM systems.

The database community is also highly interested in flash SSDs to enhance DB performance. For example, a mechanism called *FlashLogging* [Chen 2009] adopts multiple low-cost flash drives to improve log processing, and parallelizing writes is an important consideration. Das et al. present an application-level compression scheme for flash and adopt a parallelized flush method, called MT-Flush, in the MySQL database, which optimizes the flash storage performance [Das et al. 2014]. Tsirogiannis et al. have proposed a set of techniques including fast scans and join customized for SSDs [Tsirogiannis et al. 2009]. Similarly, our case studies in PostgreSQL show that with parallelism, many DBMS components need to be revisited, and the disk-based optimization model would become problematic and even error prone on SSDs. These related works and our study show that internal parallelism is crucial to enhancing system and application performance and should be treated as a top priority in the design of systems and applications.

12. CONCLUSIONS

This article presents a comprehensive study to understand and exploit internal parallelism in SSDs. Through extensive experiments and analysis, we would like to deliver three key messages: (1) Effectively exploiting internal parallelism of SSDs indeed can significantly improve I/O performance and also largely remove performance limitations of SSDs. We need to treat I/O parallelism as a *top priority* for optimizing I/O performance. (2) We must also pay close attention to some potential side effects, such as the strong interference between reads and writes and the ill-mapped data layout problem, and minimize its impact. (3) Most importantly, we also find that many of the existing optimizations, which are specifically tailored to the properties of disk drives, can become ineffective and sometimes even harmful on SSDs. We must revisit such designs carefully. It is our hope that this work can provide insight into the highly parallelized internal architecture of SSDs and also provide guidance to the application and system designers for effectively utilizing this unique merit of SSDs.

REFERENCES

Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD/PODS Conference*. ACM, New York, NY, 967–980.

D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. 2009. Lazy-adaptive tree: An optimized index structure for flash devices. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*.

N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX'08)*.

A. Birrell, M. Isard, C. Thacker, and T. Wobber. 2005. A design for high-performance flash disks. In *2005 Microsoft Research Technical Report*.

Blktrace. 2011. Homepage. Retrieved from http://linux.die.net/man/8/blktrace.

S. Boboila and P. Desnoyers. 2010. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*.

L. Bouganim, B. Jónsson, and P. Bonnet. 2009. uFLIP: Understanding flash IO patterns. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems (CIDR'09)*.

M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. 2009. An object placement advisor for DB2 using solid state storage. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*.

F. Chen, D. A. Koufaty, and X. Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance'09)*. ACM, New York, NY.

Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011a. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.

F. Chen, T. Luo, and X. Zhang. 2011b. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*.

Feng Chen, Michael P. Mesnier, and Scott Hahn. 2014. A protected block device for persistent memory. In *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST'14)*.

S. Chen. 2009. FlashLogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of the 2009 ACM SIGMOD Conference (SIGMOD'09)*. ACM, New York, NY.

Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindström. 2014. NVM compression - hybrid flash-aware application level compression. In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*.

T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2014. Deconstructing storage arrays. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*.

C. Dirik and B. Jacob. 2009. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device, architecture, and system organization. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*.

J. Do and J. M. Patel. 2009. Join processing for flash SSDs: Remembering past lessons. In *Proceedings of the 5th International Workshop on Data Management on New Hardware (DaMon'09)*.

Janene Ellefson. 2013. NVM express: Unlock your solid state drives potential. In *2013 Flash Memory Summit*.

Filebench. 2015. Homepage. Retrieved from http://sourceforge.net/projects/filebench/.

E. Gal and S. Toledo. 2005. Algorithms and data structures for flash memories. In *ACM Comput. Surv.* 37, 2 (2005), 138–163.

Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.

G. Graefe. 2007. The five-minute rule 20 years later, and how flash memory changes the rules. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware (DaMon'07)*.

Xufeng Guo, Jianfeng Tan, and Yuping Wang. 2013. PAB: Parallelism-aware buffer management scheme for nand-based SSDs. In *Proceedings of IEEE 21st International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'13)*.

Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. 2013. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. In *IEEE Trans. Comput*. 62, 6 (2013), 1141–1155.

S. Iyer and P. Druschel. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th Symposium on Operating System Principles (SOSP'01)*.

S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. 2005. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'05)*.

W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. 2010. DFS: A file system for virtualized flash storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'10)*.

Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. 2014. HIOS: A host interface I/O scheduler for solid state disks. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA'14)*.

Myoungsoo Jung and Mahmut Kandemir. 2012. An evaluation of different page allocation strategies on high-speed SSDs. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'12)*.

Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting widely held SSD expectations and rethinking system-level implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'13)*. ACM, New York, NY.

Myoungsoo Jung and Mahmut T. Kandemir. 2014. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *Proceedings of 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*.

Myoungsoo Jung, Ellis H. Wilson, and Mahmut Kandemir. 2012. Physically addressed queuing (PAQ): Improving parallelism in solid state disks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*.

T. Kgil, D. Roberts, and T. Mudge. 2008. Improving NAND flash based disk caches. In *Proceedings of the 35th International Conference on Computer Architecture (ISCA'08)*.

Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. 2011. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST'11)*.

I. Koltsidas and S. Viglas. 2008. Flashing up the storage layer. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*.

R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. 2009a. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09)*.

S. Lee and B. Moon. 2007. Design of flash-based DBMS: An in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD Conference (SIGMOD'07)*.

S. Lee, B. Moon, and C. Park. 2009b. Advances in flash memory SSD technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD Conference (SIGMOD'09)*. ACM, New York, NY.

S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. 2008. A case for flash memory SSD in enterprise database applications. In *Proceedings of 2008 ACM SIGMOD Conference (SIGMOD'08)*. ACM, New York, NY,

Y. Li, B. He, Q. Luo, and K. Yi. 2009. Tree indexing on flash disks. In *Proceedings of the 25th International Conference on Data Engineering (ICDE'09)*.

M. Mesnier. 2011. Intel Open Storage Toolkit. http://www.source forge.org/projects/intel-iscsi. (2011).

M. Mesnier, M. Wachs, R. Sambasivan, A. Zheng, and G. Ganger. 2007. Modeling the relative fitness of storage. In *Proceedings of 2007 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*. ACM, New York, NY.

Micron. 2007. Micron 8, 16, 32, 64Gb SLC NAND Flash Memory Data Sheet. Retrieved from http://www.micron.com/.

S. Nath and P. B. Gibbons. 2008. Online maintenance of very large random samples on flash storage. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB'08)*.

NVM Express. 2015. Homepage. Retrieved from http://www.nvmexpress.org.

P. O'Neil, B. O'Neil, and X. Chen. 2009. Star Schema Benchmark. Retrieved from http://www.cs.umb.edu/~poneil/StarSchemaB.pdf.

OpenNVM. 2015. Homepage. Retrieved from http://opennvm.github.io.

OpenSSD. 2015. Homepage. Retrieved from http://www.openssd-project.org/.

Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi. 2006. A high performance controller for NAND flash-based solid state disk (NSSD). In *Proceedings of the 21st IEEE Non-Volatile Semiconductor Memory Workshop (NVSMW'06)*.

S. Park and K. Shen. 2012. FIOS: A fair, efficient flash I/O scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

D. Patterson, G. Gibson, and R. Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*. ACM, New York, NY.

PC Perspective. 2009. OCZ Apex Series 250GB Solid State Drive Review. Retrieved from http://www.pcper.com/article.php?aid=661.

Pgbench. 2011. Homepage. Retrieved from http://developer.postgresql.org/pgdocs/postgres/pgbench.html.

M. Polte, J. Simsa, and G. Gibson. 2008. Comparing performance of solid state devices and mechanical disks. In *Proceedings of the 3rd Petascale Data Storage Workshop*.

T. Pritchett and M. Thottethodi. 2010. SieveStore: A highly-selective, ensemble-level disk cache for cost-performance. In *Proceedings of International Symposium on Computer Architecture (ISCA'10)*.

Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. 2012. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. In *Proceedings of VLDB Endowment (VLDB'12)*.

M. Rosenblum and J. K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst*. 10, 1 (1992), 26–52.

Samsung. 2007. Datasheet (K9LBG08U0M). Retrieved from http://www.samsung.com.

SATA. 2011. Serial ATA Revision 2.6. Retrieved from http://www.sata-io.org.

M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. 2008. Fast scans and joins using flash drives. In *Proceedings of the 4th International Workshop on Data Management on New Hardware (DaMon'08)*.

S. Shah and B. D. Noble. 2007. A study of e-mail patterns. In *Software Practice and Experience*, Vol. 37(14). 1515–1538.

G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. 2010. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*.

R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. 2009. Evaluating and repairing write performance on flash devices. In *Proceedings of the 5th International Workshop on Data Management on New Hardware (DaMon'09)*.

Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. 2007. One size fits all? Part 2: Benchmarking studies. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems (CIDR'07)*.

D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. 2009. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD Conference (SIGMOD'09)*. ACM, New York, NY.

Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*. Amsterdam, The Netherlands.

Guanying Wu and Xubin He. 2014. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

Received December 2014; revised June 2015; accepted August 2015

CLOCK-Pro: An Effective Improvement of the CLOCK Replacement

Song Jiang

Performance and Architecture Laboratory (PAL)
Los Alamos National Laboratory, CCS Division
Los Alamos, NM 87545, USA
sjiang@lanl.gov

Feng Chen and Xiaodong Zhang

Computer Science Department
College of William and Mary
Williamsburg, VA 23187, USA
{fchen, zhang}@cs.wm.edu

Abstract

With the ever-growing performance gap between memory systems and disks, and rapidly improving CPU performance, virtual memory (VM) management becomes increasingly important for overall system performance. However, one of its critical components, the page replacement policy, is still dominated by CLOCK, a replacement policy developed almost 40 years ago. While pure LRU has an unaffordable cost in VM, CLOCK simulates the LRU replacement algorithm with a low cost acceptable in VM management. Over the last three decades, the inability of LRU as well as CLOCK to handle weak locality accesses has become increasingly serious, and an effective fix becomes increasingly desirable.

Inspired by our I/O buffer cache replacement algorithm, LIRS [13], we propose an improved CLOCK replacement policy, called CLOCK-Pro. By additionally keeping track of a limited number of replaced pages, CLOCK-Pro works in a similar fashion as CLOCK with a VM-affordable cost. Furthermore, it brings all the much-needed performance advantages from LIRS into CLOCK. Measurements from an implementation of CLOCK-Pro in Linux Kernel 2.4.21 show that the execution times of some commonly used programs can be reduced by up to 47%.

1 Introduction

1.1 Motivation

Memory management has been actively studied for decades. On one hand, to use installed memory effectively, much work has been done on memory allocation, recycling, and memory management in various programming languages. Many solutions and significant improvements have been seen in both theory and practice. On the other hand, aiming at reducing the cost of paging between memory and disks, researchers and practitioners in both academia and industry are working hard to improve the performance of page replacement, especially to avoid the worst performance cases. A significant advance in this regard becomes increasingly demanding with the continuously growing gap between memory and disk access times, as well as rapidly improving CPU performance. Although increasing memory size can always reduce I/O pagings by giving a larger memory space to hold the working set, one cannot cache all the previously accessed data including file data in memory. Meanwhile, VM system designers should attempt to maximize the achievable performance under different application demands and system configurations. An effective replacement policy is critical in achieving the goal. Unfortunately, an approximation of LRU, the CLOCK replacement policy [5], which was developed almost 40 years ago, is still dominating nearly all the major operating systems including MVS, Unix, Linux and Windows [7] [1], even though it has apparent performance disadvantages inherited from LRU with certain commonly observed memory access behaviors.

We believe that there are two reasons responsible for the lack of significant improvements on VM page replacements. First, there is a very stringent cost requirement on the policy demanded by the VM management, which requires that the cost be associated with the number of page faults or a moderate constant. As we know, a page fault incurs a penalty worth of hundreds of thousands of CPU cycles. This allows a replacement policy to do its job without intrusively interfering with application executions. However, a policy with its cost proportional to the number of memory references would be prohibitively expensive, such as doing some bookkeeping on every memory access. This can cause the user program to generate a trap into the operating system on every memory instruction, and the CPU would consume much more cycles on page replacement than on user programs, even when there are no paging requests. From the cost perspective, even LRU, a well-recognized low-cost and simple replacement algorithm, is unaffordable, because it has to maintain the LRU ordering of pages for each page access. The second reason is that most proposed replacement algorithms attempting to improve LRU

[1] This generally covers many CLOCK variants, including Mach-style active/inactive list, FIFO list facilitated with hardware reference bits. These CLOCK variants share similar performance problems plaguing LRU.

performance turn out to be too complicated to produce their approximations with their costs meeting the requirements of VM. This is because the weak cases for LRU mostly come from its minimal use of history access information, which motivates other researchers to take a different approach by adding more bookkeeping and access statistic analysis work to make their algorithms more intelligent in dealing with some access patterns unfriendly to LRU.

1.2 The Contributions of this Paper

The objective of our work is to provide a VM page replacement algorithm to take the place of CLOCK, which meets both the performance demand from application users and the low overhead requirement from system designers.

Inspired by the I/O buffer cache replacement algorithm, LIRS [13], we design an improved CLOCK replacement, called CLOCK-Pro. LIRS, originally invented to serve I/O buffer cache, has a cost unacceptable to VM management, even though it holds apparent performance advantages relative to LRU. We integrate the principle of LIRS and the way in which CLOCK works into CLOCK-Pro. By proposing CLOCK-Pro, we make several contributions: (1) CLOCK-Pro works in a similar fashion as CLOCK and its cost is easily affordable in VM management. (2) CLOCK-Pro brings all the much-needed performance advantages from LIRS into CLOCK. (3) Without any pre-determined parameters, CLOCK-Pro adapts to the changing access patterns to serve a broad spectrum of workloads. (4) Through extensive simulations on real-life I/O and VM traces, we have shown the significant page fault reductions of CLOCK-Pro over CLOCK as well as other representative VM replacement algorithms. (5) Measurement results from an implementation of CLOCK-Pro in a Linux kernel show that the execution times of some commonly used programs can be reduced by up to 47%.

2 Background

2.1 Limitations of LRU/CLOCK

LRU is designed on an assumption that a page would be reaccessed soon after it was accessed. It manages a data structure conventionally called LRU stack, in which the Most Recently Used (MRU) page is at the stack top and the Least Recently Used (LRU) page is at the stack bottom. The ordering of other in-between pages in the stack strictly follows their last access times. To maintain the stack, the LRU algorithm has to move an accessed page from its current position in the stack (if it is in the stack) to the stack top. The LRU page at the stack bottom is the one to be replaced if there is a page fault and no free spaces are available. In CLOCK, the memory spaces holding the pages can be regarded as a circular buffer and the replacement algorithm cycles through the pages in the circular buffer, like the hand of a clock. Each page is associated with a bit, called reference bit, which is set by hardware whenever the page is accessed. When it is necessary to replace a page to service a page fault, the page pointed to by the hand is checked. If its reference bit is unset, the page is replaced. Otherwise, the algorithm resets its reference bit and keeps moving the hand to the next page. Research and experience have shown that CLOCK is a close approximation of LRU, and its performance characteristics are very similar to those of LRU. So all the performance disadvantages discussed below about LRU are also applied to CLOCK.

The LRU assumption is valid for a significant portion of workloads, and LRU works well for these workloads, which are called LRU-friendly workloads. The distance of a page in the LRU stack from the stack top to its current position is called **recency**, which is the number of other distinct pages accessed after the last reference to the page. Assuming an unlimitedly long LRU stack, the distance of a page in the stack away from the top when it is accessed is called its **reuse distance**, which is equivalent to the number of other distinct pages accessed between its last access and its current access. LRU-friendly workloads have two distinct characteristics: (1) There are much more references with small reuse distances than those with large reuse distances; (2) Most references have reuse distances smaller than the available memory size in terms of the number of pages. The locality exhibited in this type of workloads is regarded as strong, which ensures a high hit ratio and a steady increase of hit ratio with the increase of memory size.

However, there are indeed cases in which this assumption does not hold, where LRU performance could be unacceptably degraded. One example access pattern is memory scan, which consists of a sequence of one-time page accesses. These pages actually have infinitely large reuse distance and cause no hits. More seriously, in LRU, the scan could flush all the previously active pages out of memory.

As an example, in Linux the memory management for process-mapped program memory and file I/O buffer cache is unified, so the memory can be flexibly allocated between them according to their respective needs. The allocation balancing between program memory and buffer cache poses a big problem because of the unification. This problem is discussed in [22]. We know that there are a large amount of data in file systems, and the total number of accesses to the file cache could also be very large. However, the access frequency to each individual page of file data is usually low. In a burst of file accesses, most of the memory could serve as a file cache. Meanwhile, the process pages are evicted to make space for the actually infrequently accessed file pages, even though they are frequently accessed. An example scenario on this is that right after one extracts a large tarball, he/she could sense that the computer becomes slower because the previous active working set is replaced and has to be faulted in. To address this problem in a simple way, current Linux versions have to in-

troduce some "magic parameters" to enforce the buffer cache allocation to be in the range of 1% to 15% of memory size by default [22]. However, this approach does not fundamentally solve the problem, because the major reason causing the allocation unbalancing between process memory and buffer cache is the ineffectiveness of the replacement policy in dealing with infrequently accessed pages in buffer caches.

Another representative access pattern defeating LRU is loop, where a set of pages are accessed cyclically. Loop and loop-like access patterns dominate the memory access behaviors of many programs, particularly in scientific computation applications. If the pages involved in a loop cannot completely fit in the memory, there are repeated page faults and no hits at all. The most cited example for the loop problem is that even if one has a memory of 100 pages to hold 101 pages of data, the hit ratio would be ZERO for a looping over this data set [9, 24]!

2.2 LIRS and its Performance Advantages

A recent breakthrough in replacement algorithm designs, called LIRS (Low Inter-reference Recency Set) replacement [13], removes all the aforementioned LRU performance limitations while still maintaining a low cost close to LRU. It can not only fix the scan and loop problems, but also can accurately differentiate the pages based on their locality strengths quantified by reuse distance.

A key and unique approach in handling history access information in LIRS is that it uses reuse distance rather than recency in LRU for its replacement decision. In LIRS, a page with a large reuse distance will be replaced even if it has a small recency. For instance, when a one-time-used page is recently accessed in a memory scan, LIRS will replace it quickly because its reuse distance is infinite, even though its recency is very small. In contrast, LRU lacks the insights of LIRS: all accessed pages are indiscriminately cached until either of two cases happens to them: (1) they are re-accessed when they are in the stack, and (2) they are replaced at the bottom of the stack. LRU does not take account of which of the two cases has a higher probability. For infrequently accessed pages, which are highly possible to be replaced at the stack bottom without being re-accessed in the stack, holding them in memory (as well as in stack) certainly results in a waste of the memory resources. This explains the LRU misbehavior with the access patterns of weak locality.

3 Related Work

There have been a large number of new replacement algorithms proposed over the decades, especially in the last fifteen years. Almost all of them are proposed to target the performance problems of LRU. In general, there are three approaches taken in these algorithms. (1) Requiring applications to explicitly provide future access hints, such as application-controlled file caching [3], and application-informed prefetching and caching [20]; (2) Explicitly detecting the access patterns failing LRU and adaptively switching to other effective replacements, such as SEQ [9], EELRU [24], and UBM [14]; (3) Tracing and utilizing deeper history access information such as FBR [21], LRFU [15], LRU-2 [18], 2Q [12], MQ [29], LIRS [13], and ARC [16]. More elaborate description and analysis on the algorithms can be found in [13]. The algorithms taking the first two approaches usually place too many constraints on the applications to be applicable in the VM management of a general-purpose OS. For example, SEQ is designed to work in VM management, and it only does its job when there is a page fault. However, its performance depends on an effective detection of long sequential address reference patterns, where LRU behaves poorly. Thus, SEQ loses generality because of the mechanism it uses. For instance, it is hard for SEQ to detect loop accesses over linked lists. Among the algorithms taking the third approach, FBR, LRU-2, LRFU and MQ are expensive compared with LRU. The performance of 2Q has been shown to be very sensitive to its parameters and could be much worse than LRU [13]. LIRS uses reuse distance, which has been used to characterize and to improve data access locality in programs (see e.g. [6]). LIRS and ARC are the two most promising candidate algorithms that have a potential leading to low-cost replacement policies applicable in VM, because they use data structure and operations similar to LRU and their cost is close to LRU.

ARC maintains two variably-sized lists holding history access information of referenced pages. Their combined size is two times of the number of pages in the memory. So ARC not only records the information of cached pages, but also keeps track of the same number of replaced pages. The first list contains pages that have been touched only once recently (*cold pages*) and the second list contains pages that have been touched at least twice recently (*hot pages*). The cache spaces allocated to the pages in these two lists are adaptively changed, depending on in which list the recent misses happen. More cache spaces will serve cold pages if there are more misses in the first list. Similarly, more cache spaces will serve hot pages if there are more misses in the second list. However, though ARC allocates memory to hot/cold pages adaptively according to the ratio of cold/hot page accesses and excludes tunable parameters, the locality of pages in the two lists, which are supposed to hold cold and hot pages respectively, can not directly and consistently be compared. So the hot pages in the second list could have a weaker locality in terms of reuse distance than the cold pages in the first list. For example, a page that is regularly accessed with a reuse distance a little bit more than the memory size can have no hits at all in ARC, while a page in the second list can stay in memory without any accesses, since it has been accepted into the list. This does not happen in LIRS, because any pages supposed to be hot or cold are placed in the same list and compared in a consistent fash-

ion. There is one pre-determined parameter in the LIRS algorithm on the amount of memory allocation for cold pages. In CLOCK-Pro, the parameter is removed and the allocation becomes fully adaptive to the current access patterns.

Compared with the research on the general replacement algorithms targeting LRU, the work specific to the VM replacements and targeting CLOCK is much less and is inadequate. While Second Chance (SC) [28], being the simplest variant of CLOCK algorithm, utilizes only one reference bit to indicate recency, other CLOCK variants introduce a finer distinction between page access history. In a generalized CLOCK version called GCLOCK [25, 17], a counter is associated with each page rather than a single bit. Its counter will be incremented if a page is hit. The cycling clock hand sweeps over the pages decrementing their counters until a page whose counter is zero is found for replacement. In Linux and FreeBSD, a similar mechanism called page aging is used. The counter is called *age* in Linux or *act_count* in FreeBSD. When scanning through memory for pages to replace, the page age is increased by a constant if its reference bit is set. Otherwise its age is decreased by a constant. One problem for this kind of design is that they cannot consistently improve LRU performance. The parameters for setting the maximum value of counters or adjusting ages are mostly empirically decided. Another problem is that they consume too many CPU cycles and adjust to changes of access patterns slowly, as evidenced in Linux kernel 2.0. Recently, an approximation version of ARC, called CAR [2], has been proposed, which has a cost close to CLOCK. Their simulation tests on the I/O traces indicate that CAR has a performance similar to ARC. The results of our experiments on I/O and VM traces show that CLOCK-Pro has a better performance than CAR.

In the design of VM replacements it is difficult to obtain much improvement in LRU due to its stringent cost constraint, yet this problem remains a demanding challenge in the OS development.

4 Description of CLOCK-Pro

4.1 Main Idea

CLOCK-Pro takes the same principle as that of LIRS – it uses the reuse distance (called IRR in LIRS) rather than recency in its replacement decision. When a page is accessed, the reuse distance is the period of time in terms of the number of other distinct pages accessed since its last access. Although there is a reuse distance between any two consecutive references to a page, only the most current distance is relevant in the replacement decision. We use the reuse distance of a page at the time of its access to categorize it either as a cold page if it has a large reuse distance, or as a hot page if it has a small reuse distance. Then we mark its status as being cold or hot. We place all the accessed pages, either hot or cold, into one single list [2] in the order of their accesses [3]. In the list, the pages with small recencies are at the list head, and the pages with large recencies are at the list tail.

To give the cold pages a chance to compete with the hot pages and to ensure their cold/hot statuses accurately reflect their current access behavior, we grant a cold page a test period once it is accepted into the list. Then, if it is re-accessed during its test period, the cold page turns into a hot page. If the cold page passes the test period without a re-access, it will leave the list. Note that the cold page in its test period can be replaced out of memory, however, its page metadata remains in the list for the test purpose until the end of the test period or being re-accessed. When it is necessary to generate a free space, we replace a resident cold page.

The key question here is how to set the time of the test period. When a cold page is in the list and there is still at least one hot page after it (i.e., with a larger recency), it should turn into a hot page if it is accessed, because it has a new reuse distance smaller than the hot page(s) after it. Accordingly, the hot page with the largest recency should turn into a cold page. So the test period should be set as the largest recency of the hot pages. If we make sure that the hot page with the largest recency is always at the list tail, and all the cold pages that pass this hot page terminate their test periods, then the test period of a cold page is equal to the time before it passes the tail of the list. So all the non-resident cold pages can be removed from the list right after they reach the tail of the list. In practice, we could shorten the test period and limit the number of cold pages in the test period to reduce space cost. By implementing this testing mechanism, we make sure that "cold/hot" are defined based on relativity and by constant comparison in one clock, not on a fixed threshold that are used to separate the pages into two lists. This makes CLOCK-Pro distinctive from prior work including 2Q and CAR, which attempt to use a constant threshold to distinguish the two types of pages, and to treat them differently in their respective lists (2Q has two queues, and CAR has two clocks), which unfortunately causes these algorithms to share some of LRU's performance weakness.

4.2 Data Structure

Let us first assume that the memory allocations for the hot and cold pages, m_h and m_c, respectively, are fixed, where $m_h + m_c$ is the total memory size m ($m = m_h + m_c$). The number of the hot pages is also m_h, so all the hot pages are always cached. If a hot page is going to be replaced, it must first change into a cold page. Apart from the hot pages, all the other accessed pages are categorized as cold pages. Among the cold pages, m_c pages are cached, another at most m non-resident

[2] Actually it is the metadata of a page that is placed in the list.

[3] Actually we can only maintain an approximate access order, because we cannot update the list with a hit access in a VM replacement algorithm, thus losing the exact access orderings between page faults.

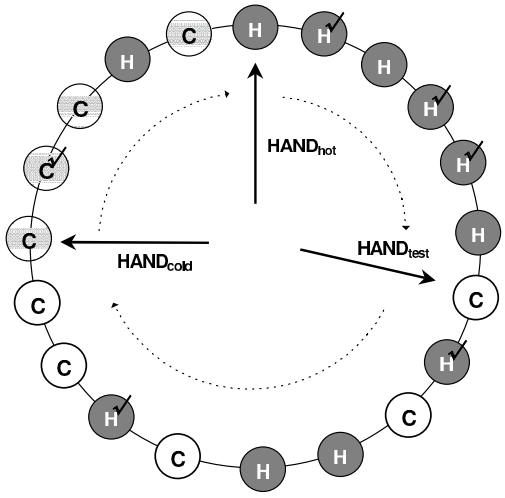


Figure 1: There are three types of pages in CLOCK-Pro, hot pages marked with "H", cold pages marked with "C" (shadowed circles for resident cold pages, non-shadowed circles for non-resident cold pages). Around the clock, there are three hands: **HAND**$_{hot}$ pointing to the list tail (i.e. the last hot page) and used to search for a hot page to turn into a cold page, **HAND**$_{cold}$ pointing to the last resident cold page and used to search for a cold page to replace, and **HAND**$_{test}$ pointing to the last cold page in the test period, terminating test periods of cold pages, and removing non-resident cold pages passing the test period out of the list. The "$\surd$" marks represent the reference bits of 1.

cold pages only have their history access information cached. So totally there are at most $2m$ metadata entries for keeping track of page access history in the list. As in CLOCK, all the page entries are organized as a circular linked list, shown in Figure 1. For each page, there is a cold/hot status associated with it. For each cold page, there is a flag indicating if the page is in the test period.

In CLOCK-Pro, there are three hands. The **HAND**$_{hot}$ points to the hot page with the largest recency. The position of this hand actually serves as a threshold of being a hot page. Any hot pages swept by the hand turn into cold ones. For the convenience of the presentation, we call the page pointed to by **HAND**$_{hot}$ as the tail of the list, and the page immediately after the tail page in the clockwise direction as the head of the list. **HAND**$_{cold}$ points to the last resident cold page (i.e., the furthest one to the list head). Because we always select this cold page for replacement, this is the position where we start to look for a victim page, equivalent to the hand in CLOCK. **HAND**$_{test}$ points to the last cold page in the test period. This hand is used to terminate the test period of cold pages. The non-resident cold pages swept over by this hand will leave the circular list. All the hands move in the clockwise direction.

4.3 Operations on Searching Victim Pages

Just as in CLOCK, there are no operations in CLOCK-Pro for page hits, only the reference bits of the accessed pages are set by hardware. Before we see how a victim page is generated, let us examine how the three hands move around the clock, because the victim page is searched by coordinating the movements of the hands.

HAND$_{cold}$ is used to search for a resident cold page for replacement. If the reference bit of the cold page currently pointed to by **HAND**$_{cold}$ is unset, we replace the cold page for a free space. The replaced cold page will remain in the list as a non-resident cold page until it runs out of its test period, if it is in its test period. If not, we move it out of the clock. However, if its bit is set and it is in its test period, we turn the cold page into a hot page, and ask **HAND**$_{hot}$ for its actions, because an access during the test period indicates a competitively small reuse distance. If its bit is set but it is not in its test period, there are no status change as well as **HAND**$_{hot}$ actions. In both of the cases, its reference bit is reset, and we move it to the list head. The hand will keep moving until it encounters a cold page eligible for replacement, and stops at the next resident cold page.

As mentioned above, what triggers the movement of **HAND**$_{hot}$ is that a cold page is found to have been accessed in its test period and thus turns into a hot page, which maybe accordingly turns the hot page with the largest recency into a cold page. If the reference bit of the hot page pointed to by **HAND**$_{hot}$ is unset, we can simply change its status and then move the hand forward. However, if the bit is set, which indicates the page has been re-accessed, we spare this page, reset its reference bit and keep it as a hot page. This is because the actual access time of the hot page could be earlier than the cold page. Then we move the hand forward and do the same on the hot pages with their bits set until the hand encounters a hot page with a reference bit of zero. Then the hot page turns into a cold page. Note that moving **HAND**$_{hot}$ forward is equivalent to leaving the page it moves by at the list head. Whenever the hand encounters a cold page, it will terminate the page's test period. The hand will also remove the cold page from the clock if it is non-resident (the most probable case). It actually does the work on the cold page on behalf of hand **HAND**$_{test}$. Finally the hand stops at a hot page.

We keep track of the number of non-resident cold pages. Once the number exceeds m, the memory size in the number of pages, we terminate the test period of the cold page pointed to by **HAND**$_{test}$. We also remove it from the clock if it is a non-resident page. Because the cold page has used up its test period without a re-access and has no chance to turn into a hot page with its next access. **HAND**$_{test}$ then moves forward and stops at the next cold page.

Now let us summarize how these hands coordinate their operations on the clock to resolve a page fault. When there is a page fault, the faulted page must be a cold page. We first run **HAND**$_{cold}$ for a free space. If the faulted cold page is not in the list, its reuse distance is highly likely to be larger than the recency of hot pages [4]. So the page is still categorized as a

[4] We cannot guarantee that it is a larger one because there are no opera-

cold page and is placed at the list head. The page also initiates its test period. If the number of cold pages is larger than the threshold ($m_c + m$), we run **HAND**$_{test}$. If the cold page is in the list [5], the faulted page turns into a hot page and is placed at the head of the list. We run **HAND**$_{hot}$ to turn a hot page with a large recency into a cold page.

4.4 Making CLOCK-Pro Adaptive

Until now, we have assumed that the memory allocations for the hot and cold pages are fixed. In LIRS, there is a predetermined parameter, denoted as L_{hirs}, to measure the percentage of memory that are used by cold pages. As it is shown in [13], L_{hirs} actually affects how LIRS behaves differently from LRU. When L_{hirs} approaches 100%, LIRS's replacement behavior, as well as its hit ratios, are close to those of LRU. Although the evaluation of LIRS algorithm indicates that its performance is not sensitive to L_{hirs} variations within a large range between 1% and 30%, it also shows that the hit ratios of LIRS could be moderately lower than LRU for LRU-friendly workloads (i.e. with strong locality) and increasing L_{hirs} can eliminate the performance gap.

In CLOCK-Pro, resident cold pages are actually managed in the same way as in CLOCK. **HAND**$_{cold}$ behaves the same as what the clock hand in CLOCK does: sweeping across the pages while sparing the page with a reference bit of 1 and replacing the page with a reference bit of 0. So increasing m_c, the size of the allocation for cold pages, makes CLOCK-Pro behave more like CLOCK.

Let us see the performance implication of changing memory allocation in CLOCK-Pro. To overcome the CLOCK performance disadvantages with weak access patterns such as scan and loop, a small m_c value means a quick eviction of cold pages just faulted in and the strong protection of hot pages from the interference of cold pages. However, for a strong locality access stream, almost all the accessed pages have relatively small reuse distance. But, some of the pages have to be categorized as cold pages. With a small m_c, a cold page would have to be replaced out of memory soon after its being loaded in. Due to its small reuse distance, the page is probably faulted in the memory again soon after its eviction and treated as a hot page because it is in its test period this time. This actually generates unnecessary misses for the pages with small reuse distances. Increasing m_c would allow these pages to be cached for a longer period of time and make it more possible for them to be re-accessed and to turn into hot pages without being replaced. Thus, they can save additional page faults.

For a given reuse distance of an accessed cold page, m_c decides the probability of a page being re-accessed before its being replaced from the memory. For a cold page with its reuse distance larger than its test period, retaining the page in memory with a large m_c is a waste of buffer spaces. On the other hand, for a page with a small reuse distance, retaining the page in memory for a longer period of time with a large m_c would save an additional page fault. In the adaptive CLOCK-Pro, we allow m_c to dynamically adjust to the current reuse distance distribution. If a cold page is accessed during its test period, we increment m_c by 1. If a cold page passes its test period without a re-access, we decrement m_c by 1. Note the aforementioned cold pages include resident and non-resident cold pages. Once the m_c value is changed, the clock hands of CLOCK-Pro will realize the memory allocation by temporally adjusting the moving speeds of **HAND**$_{hot}$ and **HAND**$_{cold}$.

With this adaptation, CLOCK-Pro could take both LRU advantages with strong locality and LIRS advantages with weak locality.

5 Performance Evaluation

We use both trace-driven simulations and prototype implementation to evaluate our CLOCK-Pro and to demonstrate its performance advantages. To allow us to extensively compare CLOCK-Pro with other algorithms aiming at improving LRU, including CLOCK, LIRS, CAR, and OPT, we built simulators running on the various types of representative workloads previously adopted for replacement algorithm studies. OPT is an optimal, but offline, unimplementable replacement algorithm [1]. We also implemented a CLOCK-Pro prototype in a Linux kernel to evaluate its performance as well as its overhead in a real system.

5.1 Trace-Driven Simulation Evaluation

Our simulation experiments are conducted in three steps with different kinds of workload traces. Because LIRS is originally proposed as an I/O buffer cache replacement algorithm, in the first step, we test the replacement algorithms on the I/O traces to see how well CLOCK-Pro can retain the LIRS performance merits, as well as its performance with typical I/O access patterns. In the second step, we test the algorithms on the VM traces of application program executions. Integrated VM management on file cache and program memory, as is implemented in Linux, is always desired. Because of the concern for mistreatment of file data and process pages as mentioned in Section 2.1, we test the algorithms on the aggregated VM and file I/O traces to see how these algorithms respond to the integration in the third step. We do not include the results of LRU in the presentation, because they are almost the same as those of CLOCK.

tions on hits in CLOCK-Pro and we limit the number of cold pages in the list. But our experiment results show this approximation minimally affects the performance of CLOCK-Pro.

[5] The cold page must be in its test period. Otherwise, it must have been removed from the list.

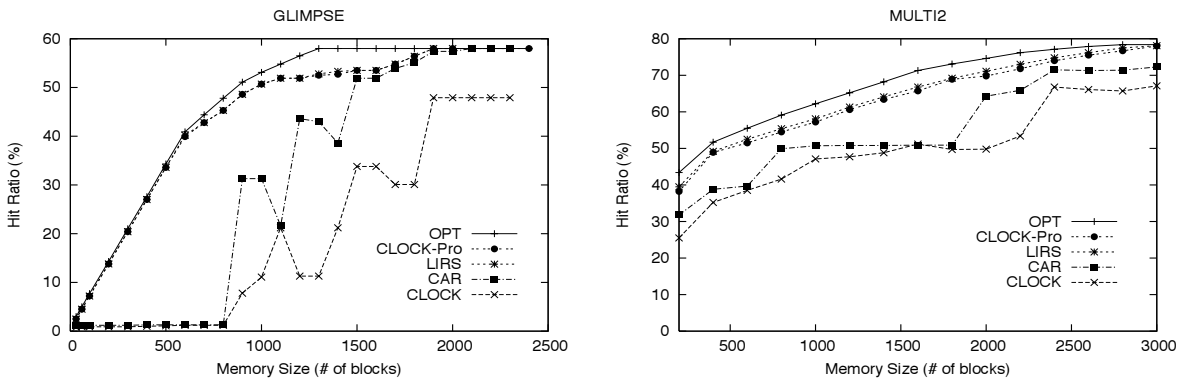


Figure 2: Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workloads $glimpse$ and $multi2$.

5.1.1 Step 1: Simulation on I/O Buffer Caches

The file I/O traces used in this section are from [13] used for the LIRS evaluation. In their performance evaluation, the traces are categorized into four groups based on their access patterns, namely, loop, probabilistic, temporally-clustered and mixed patterns. Here we select one representative trace from each of the groups for the replacement evaluation, and briefly describe them here.

1. **glimpse** is a text information retrieval utility trace. The total size of text files used as input is roughly 50 MB. The trace is a member of the loop pattern group.

2. **cpp** is a GNU C compiler pre-processor trace. The total size of C source programs used as input is roughly 11 MB. The trace is a member of the probabilistic pattern group.

3. **sprite** is from the Sprite network file system, which contains requests to a file server from client workstations for a two-day period. The trace is a member of the temporally-clustered pattern group.

4. **multi2** is obtained by executing three workloads, cs, cpp, and postgres, together. The trace is a member of the mixed pattern group.

These are small-scale traces with clear access patterns. We use them to investigate the implications of various access patterns on the algorithms. The hit ratios of $glimpse$ and $multi2$ are shown in Figure 2. To help readers clearly see the hit ratio difference for *cpp* and $sprite$, we list their hit ratios in Tables 1 and 2, respectively. For LIRS, the memory allocation to HIR pages (L_{hirs}) is set as 1% of the memory size, the same value as it is used in [13]. There are several observations we can make on the results.

First, even though CLOCK-Pro does not responsively deal with hit accesses in order to meet the cost requirement of VM management, the hit ratios of CLOCK-Pro and LIRS are very close, which shows that CLOCK-Pro effectively retains the performance advantages of LIRS. For workloads $glimpse$ and $multi2$, which contain many loop accesses, LIRS with a small L_{hirs} is most effective. The hit ratios of CLOCK-pro are a little lower than LIRS. However, for the LRU-friendly workload, $sprite$, which consists of strong locality accesses, the performance of LIRS could be lower than CLOCK (see Table 2). With its memory allocation adaptation, CLOCK-Pro improves the LIRS performance.

Figure 3 shows the percentage of the memory allocated to cold pages during the execution courses of $multi2$ and $sprite$ for a memory size of 600 pages. We can see that for $sprite$, the allocations for cold pages are much larger than 1% of the memory used in LIRS, and the allocation fluctuates over the

blocks	OPT	CLOCK-Pro	LIRS	CAR	CLOCK
20	26.4	23.9	24.2	17.6	0.6
35	46.5	41.2	42.4	26.1	4.2
50	62.8	53.1	55.0	37.5	18.6
80	79.1	71.4	72.8	70.1	60.4
100	82.5	76.2	77.6	77.0	72.6
300	86.5	85.1	85.0	85.6	83.5
500	86.5	85.9	85.9	85.8	84.7
700	86.5	86.3	86.3	86.3	85.4
900	86.5	86.4	86.4	86.4	85.7

Table 1: Hit ratios (%) of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload *cpp*.

blocks	OPT	CLOCK-Pro	LIRS	CAR	CLOCK
100	50.8	24.8	25.1	26.1	22.8
200	68.9	45.2	44.7	43.0	43.5
400	84.6	70.1	69.5	70.5	70.9
600	89.9	82.4	80.9	82.1	83.3
800	92.2	87.6	85.6	87.3	88.1
1000	93.2	89.7	87.6	89.6	90.4

Table 2: Hit ratios (%) of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload $sprite$.

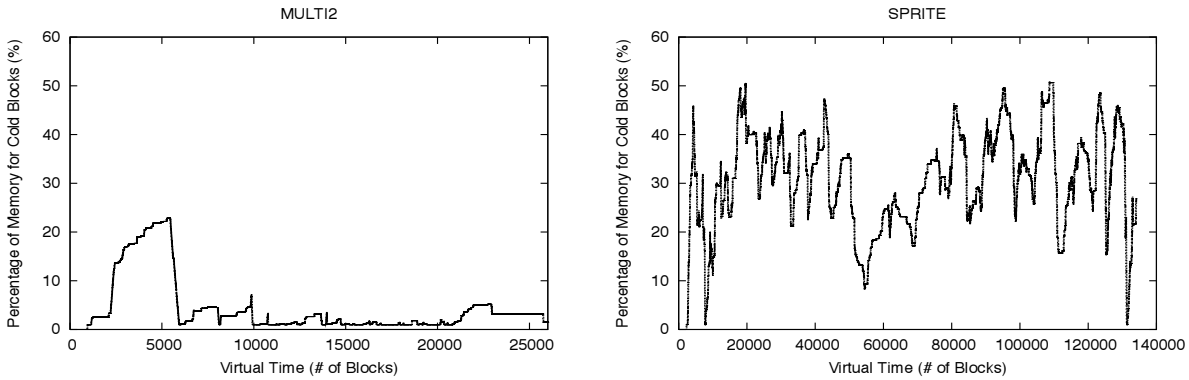


Figure 3: Adaptively changing the percentage of memory allocated to cold blocks in workloads $multi2$ and $sprite$.

time adaptively to the changing access patterns. It sounds paradoxical that we need to increase the cold page allocation when there are many hot page accesses in the strong locality workload. Actually only the real cold pages with large reuse distances should be managed in a small cold allocation for their quick replacements. The so-called "cold" pages could actually be hot pages in strong locality workloads because the number of so-called "hot" pages are limited by their allocation. So quickly replacing these pseudo-cold pages should be avoided by increasing the cold page allocation. We can see that the cold page allocations for $multi2$ are lower than $sprite$, which is consistent with the fact that $multi2$ access patterns consist of many long loop accesses of weak locality.

Second, regarding the performance difference of the algorithms, CLOCK-Pro and LIRS have much higher hit ratios than CAR and CLOCK for $glimpse$ and $multi2$, and are close to optimal. For strong locality accesses like $sprite$, there is little improvement either for CLOCK-Pro or CAR. This is why CLOCK is popular, considering its extremely simple implementation and low cost.

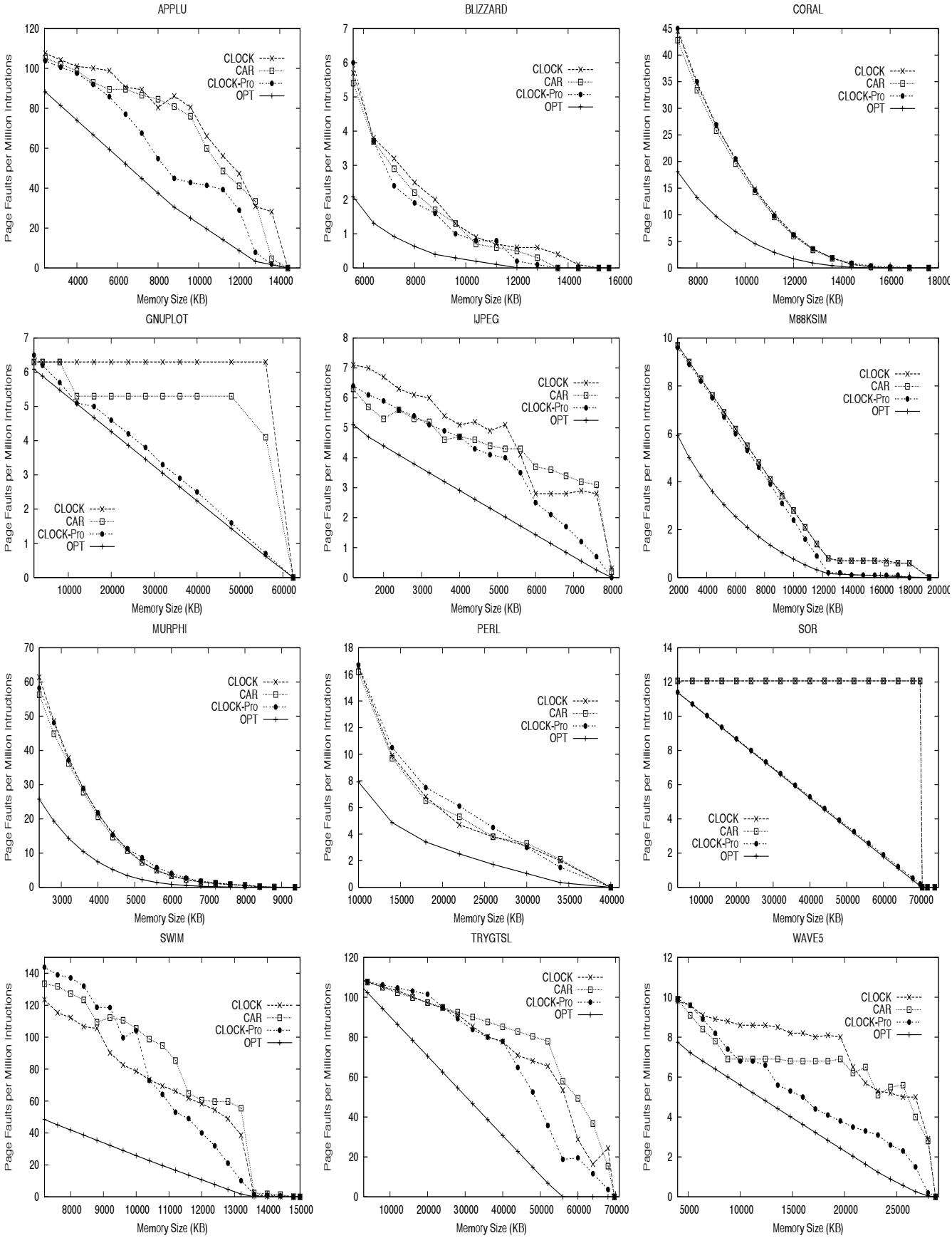
Third, even with a built-in memory allocation adaption mechanism, CAR cannot provide consistent improvements over CLOCK, especially for weak locality accesses, on which a fix is most needed for CLOCK. As we have analyzed, this is because CAR, as well as ARC, lacks a consistent locality strength comparison mechanism.

5.1.2 Step 2: Simulation on Memory for Program Executions

In this section, we use the traces of memory accesses of program executions to evaluate the performance of the algorithms. All the traces used here are also used in [10] and many of them are also used in [9, 24]. However, we do not include the performance results of SEQ and EELRU in this paper because of their generality or cost concerns for VM management. In some situations, EELRU needs to update its statistics for every single memory reference, having the same overhead problem as LRU [24]. Interested readers are referred to the respective papers for detailed performance descriptions of SEQ and EELRU. By comparing the hit ratio curves presented in those papers with the curves provided in this paper about CLOCK-Pro (these results are comparable), readers will reach the conclusion that CLOCK-Pro provides better or equally good performance compared to SEQ and EELRU. Also because of overhead concern, we do not include the LRU and LIRS performance. Actually LRU has its hit ratio curves almost the same as those of CLOCK in our experiments.

Table 3 summarizes all the program traces used in this paper. The detailed program descriptions, space-time memory access graphs, and trace collection methodology, are described in [10, 9]. These traces cover a large range of various access patterns. After observing their memory access graphs drawn from the collected traces, the authors of paper [9] categorized programs $coral$, $m88ksim$, and $murphi$ as having "no clearly visible patterns" with all accesses temporarily clustered together, categorized programs $blizzard$, $perl$, and $swim$ as having "patterns at a small scale", and categorized the rest of programs as having "clearly-exploitable, large-scale reference patterns". If we examine the program access behaviors in terms of reuse distance, the programs in the first category are the strong locality workloads. Those in the second category are moderate locality workloads. And the remaining programs in the third category are weak locality workloads. Figure 4 shows the number of page faults per million of instructions executed for each of the programs, denoted as page fault ratio, as the memory increases up to the its maximum memory demand. We exclude cold page faults which occur on their first time accesses. The algorithms considered here are CLOCK, CLOCK-Pro, CAR and OPT.

The simulation results clearly show that CLOCK-Pro significantly outperforms CLOCK for the programs with weak locality, including programs $applu$, $gunplot$, $ijpeg$, sor, $trygtsl$, and $wave5$. For $gunplot$ and sor, which have very large loop accesses, the page fault ratios of CLOCK-Pro are almost equal to those of OPT. The improvements of CAR over


APPLU
BLIZZARD
CORAL
GNUPLOT
IJPEG
M88KSIM
MURPHI
PERL
SOR
SWIM
TRYGTSL
WAVE5
CLOCK
CAR
CLOCK-Pro
OPT
Page Faults per Million Intructions
Memory Size (KB)

Program	Description	Size (in Millions of Instructions)	Maximum Memory Demand (KB)
applu	Solve 5 coupled parabolic/elliptic PDE	1,068	14,524
blizzard	Binary rewriting tool for software DSM	2,122	15,632
coral	Deductive database evaluating query	4,327	20,284
gnuplot	PostScript graph generation	4,940	62,516
ijpeg	Image conversion into IJPEG format	42,951	8,260
m88ksim	Microprocessor cycle-level simulator	10,020	19,352
murphi	Protocol verifi er	1,019	9,380
perl	Interpreted scripting language	18,980	39,344
sor	Successive over-relaxation on a matrix	5,838	70,930
swim	Shallow water simulation	438	15,016
trygtsl	Tridiagonal matrix calculation	377	69,688
wave5	plasma simulation	3,774	28,700

Table 3: A brief description of the programs used in Section 5.1.2.

CLOCK are far from being consistent and significant. In many cases, it performs worse than CLOCK. The poorest performance of CAR appears on traces $gunplot$ and sor – it cannot correct the LRU problems with loop accesses and its page fault ratios are almost as high as those of CLOCK.

For programs with strong locality accesses, including $coral$, $m88ksim$ and $murphi$, there is little room for other replacement algorithms to do a better job than CLOCK/LRU. Both CLOCK-Pro and ARC retain the LRU performance advantages for this type of programs, and CLOCK-Pro even does a little bit better than CLOCK.

For the programs with moderate locality accesses, including $blizzard$, $perl$ and $swim$, the results are mixed. Though we see the improvements of CLOCK-Pro and CAR over CLOCK in the most cases, there does exist a case in $swim$ with small memory sizes where CLOCK performs better than CLOCK-Pro and CAR. Though in most cases CLOCK-Pro performs better than CAR, for $perl$ and $swim$ with small memory sizes, CAR performs moderately better. After examining the traces, we found that the CLOCK-Pro performance variations are due to the working set shifting in the workloads. If a workload frequently shifts its working set, CLOCK-Pro has to actively adjust the composition of the hot page set to reflect current access patterns. When the memory size is small, the set of cold resident pages is small, which causes a cold/hot status exchange to be more possibly associated with an additional page fault. However, the existence of locality itself confines the extent of working set changes. Otherwise, no caching policy would fulfill its work. So we observed moderate performance degradations for CLOCK-Pro only with small memory sizes.

To summarize, we found that CLOCK-Pro can effectively remove the performance disadvantages of CLOCK in case of weak locality accesses, and CLOCK-Pro retains its performance advantages in case of strong locality accesses. It exhibits apparently more impressive performance than CAR, which was proposed with the same objectives as CLOCK-Pro.

5.1.3 Step 3: Simulation on Program Executions with Interference of File I/O

In an unified memory management system, file buffer cache and process memory are managed with a common replacement policy. As we have stated in Section 2.1, memory competition from a large number of file data accesses in a shared memory space could interfere with program execution. Because file data is far less frequently accessed than process VM, a process should be more competitive in preventing its memory from being taken away to be used as file cache buffer. However, recency-based replacement algorithms like CLOCK allow these file pages to replace process memory even if they are not frequently used, and to pollute the memory. To provide a preliminary study on the effect, we select an I/O trace (WebSearch1) from a popular search engine [26] and use its first 900 second accesses as a sample I/O accesses to co-occur with the process memory accesses in a shared memory space. This segment of I/O trace contains extremely weak locality – among the total 1.12 millions page accesses, there are 1.00 million unique pages accessed. We first scale the I/O trace onto the execution time of a program and then aggregate the I/O trace with the program VM trace in the order of their access times. We select a program with strong locality accesses, $m88ksim$, and a program with weak locality accesses, sor, for the study.

Tables 4 and 5 show the number of page faults per million of instructions (only the instructions for $m88ksim$ or sor are counted) for $m88ksim$ and sor, respectively, with various memory sizes. We are not interested in the performance of the I/O accesses. There would be few page hits even for a very large dedicated memory because there is little locality in their accesses.

From the simulation results shown in the tables, we observed that: (1) For the strong locality program, $m88ksim$, both CLOCK-Pro and CAR can effectively protect program execution from I/O access interference, while CLOCK is not able to reduce its page faults with increasingly large memory sizes. (2) For the weak locality program, sor, only CLOCK-

Memory(KB)	CLOCK-Pro	CLOCK-Pro w/IO	CAR	CAR w/IO	CLOCK	CLOCK w/IO
2000	9.6	9.94	9.7	10.1	9.7	11.23
3600	8.2	8.83	8.3	9.0	8.3	11.12
5200	6.7	7.63	6.9	7.8	6.9	11.02
6800	5.3	6.47	5.5	6.8	5.5	10.91
8400	3.9	5.22	4.1	5.8	4.1	10.81
10000	2.4	3.92	2.8	4.9	2.8	10.71
11600	0.9	2.37	1.4	4.2	1.4	10.61
13200	0.2	0.75	0.7	3.9	0.7	10.51
14800	0.1	0.52	0.7	3.6	0.7	10.41
16400	0.1	0.32	0.6	3.3	0.7	10.31
18000	0.0	0.22	0.6	3.1	0.6	10.22
19360	0.0	0.19	0.0	2.9	0.0	10.14

Table 4: The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program $m88ksim$ with and without the interference of I/O file data accesses.

Memory(KB)	CLOCK-Pro	CLOCK-Pro w/IO	CAR	CAR w/IO	CLOCK	CLOCK w/IO
4000	11.4	11.9	12.1	12.2	12.1	12.2
12000	10.0	10.7	12.1	12.2	12.1	12.2
20000	8.7	9.6	12.1	12.2	12.1	12.2
28000	7.3	8.6	12.1	12.2	12.1	12.2
36000	5.9	7.5	12.1	12.2	12.1	12.2
44000	4.6	6.5	12.1	12.2	12.1	12.2
52000	3.2	5.4	12.1	12.2	12.1	12.2
60000	1.9	4.4	12.1	12.2	12.1	12.2
68000	0.5	3.4	12.1	12.2	12.1	12.2
70600	0.0	3.0	0.0	12.2	0.0	12.2
74000	0.0	2.6	0.0	12.2	0.0	12.2

Table 5: The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program sor with and without the interference of I/O file data accesses.

Pro can protect program execution from interference, though its page faults are moderately increased compared with its dedicated execution on the same size of memory. However, CAR and CLOCK cannot reduce their faults even when the memory size exceeds the program memory demand, and the number of faults on the dedicated executions has been zero.

We did not see a devastating influence on the program executions with the co-existence of the intensive file data accesses. This is because even the weak accesses of $m88ksim$ are strong enough to stave off memory competition from file accesses with their page re-accesses, and actually there are almost no page reuses in the file accesses. However, if there are quiet periods during program active executions, such as waiting for user interactions, the program working set would be flushed by file accesses under recency-based replacement algorithms. Reuse distance based algorithms such as CLOCK-Pro will not have the problem, because file accesses have to generate small reuse distances to qualify the file data for a long-term memory stay, and to replace the program memory.

5.2 CLOCK-Pro Implementation and its Evaluation

The ultimate goal of a replacement algorithm is to reduce application execution times in a real system. In the process of translating the merits of an algorithm design to its practical performance advantages, many system elements could affect execution times, such as disk access scheduling, the gap between CPU and disk speeds, and the overhead of paging system itself. To evaluate the performance of CLOCK-Pro in a real system, we have implemented CLOCK-Pro in Linux kernel 2.4.21, which is a well documented recent version [11, 23].

5.2.1 Implementation and Evaluation Environment

We use a Gateway PC, which has its CPU of Intel P4 1.7GHz, its Western Digital WD400BB hard disk of 7200 RPM, and its memory of 256M. It is installed with RedHat 9. We are able to adjust the memory size available to the system and user programs by preventing certain portion of memory from being allocated.

In Kernel 2.4, process memory and file buffer are under an unified management. Memory pages are placed either in an

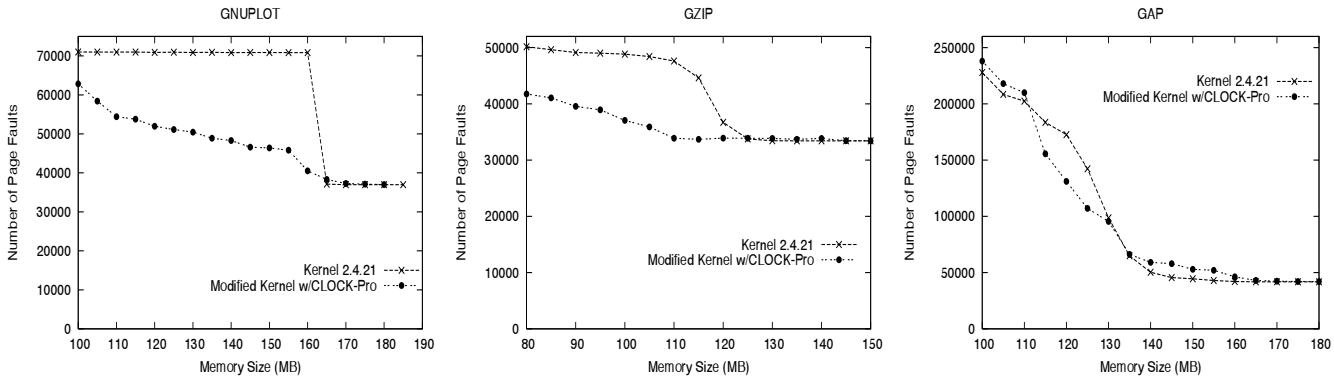


Figure 5: Measurements of the page faults of programs *gnuplot*, *gzip*, and *gap* on the original system and the system adopting CLOCK-Pro.

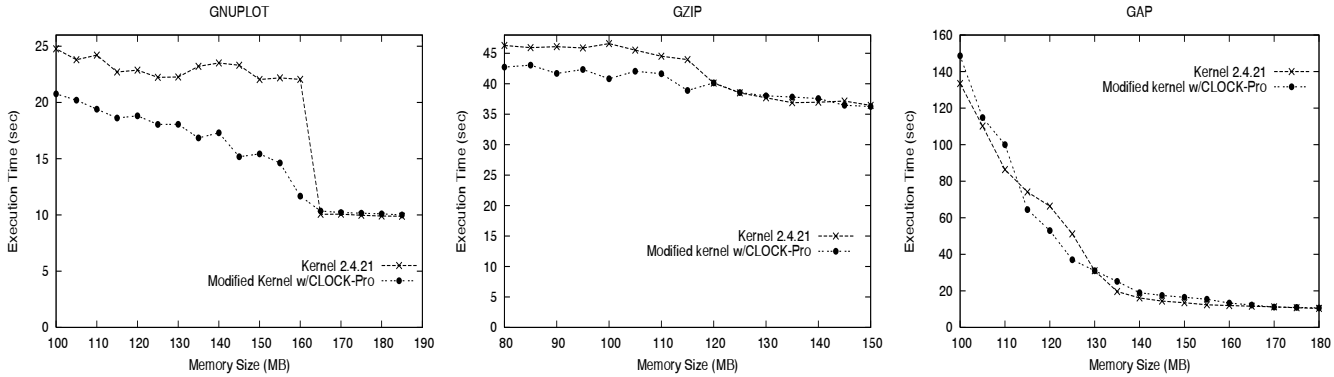


Figure 6: Measurements of the execution times of programs *gnuplot*, *gzip*, and *gap* on the original system and the system adopting CLOCK-Pro.

active list or in an inactive list. Each page is associated with a reference bit. When a page in the inactive list is detected being referenced, the page is promoted into the active list. Periodically the pages in the active list that are not recently accessed are removed to refill the inactive list. The kernel attempts to keep the ratio of the sizes of the active list and inactive list as 2:1. Each list is organized as a separate clock, where its pages are scanned for replacement or for movement between the lists. We notice that this kernel has adopted an idea similar to the 2Q replacement [12], by separating the pages into two lists to protect hot pages from being flushed by cold pages. However, a critical question still remains unanswered: how are the hot pages correctly identified from the cold pages?

This issue has been addressed in CLOCK-Pro, where we place all the pages in one single clock list, so that we can compare their hotness in a consistent way. To facilitate an efficient clock hand movement, each group of pages (with their statuses of hot, cold, and/or on test) are linked separately according to their orders in the clock list. The ratio of cold pages and hot pages is adaptively adjusted. CLOCK-Pro needs to keep track of a certain number of pages that have already been replaced from memory. We use their positions in the respective backup files to identify those pages, and maintain a hash table to efficiently retrieve their metadata when they are faulted in.

We ran SPEC CPU2000 programs and some commonly used tools to test the performance of CLOCK-Pro as well as the original system. We observed consistent performance trends while running programs with weak, moderate, or strong locality on the original and modified systems. Here we present the representative results for three programs, each from one of the locality groups. Apart from *gnuplot*, a widely used interactive plotting program with its input data file of 16 MB, which we have used in our simulation experiments, the other two are from SPEC CPU2000 benchmark suite [27], namely, *gzip* and *gap*. *gzip* is a popular data compression program, showing a moderate locality. *gap* is a program implementing a language and library designed mostly for computing in groups, showing a strong locality. Both take the inputs from their respective training data sets.

5.2.2 Experimental Measurements

Figures 5 and 6 show the number of page faults and the execution times of programs *gnuplot*, *gzip*, and *gap* on the original system and the modified system adopting CLOCK-Pro. In the simulation-based evaluation, only page faults can be obtained. Here we also show the program execution times, which include page fault penalties and system paging overhead. It is noted that we include cold page faults in the statistics, because they contribute to the execution times. We see that the variations of the execution times with memory size generally

keep the same trends as those of page fault numbers, which shows that page fault is the major factor to affect system performance.

The measurements are consistent with the simulation results on the program traces shown in Section 5.1.2. For the weak locality program *gnuplot*, CLOCK-Pro significantly improves its performance by reducing both its page fault numbers and its execution times. The largest performance improvement comes at around 160MB, the available memory size approaching the memory demand, where the time for CLOCK-Pro (11.7 sec) is reduced by 47% when compared with the time for the original system (22.1 sec). There are some fluctuations in the execution time curves. This is caused by the block layout on the disk. A page faulted in from a disk position sequential to the previous access position has a much smaller access time than that retrieved from a random position. So the penalty varies from one page fault to another. For programs *gzip* and *gap* with a moderate or strong locality, CLOCK-Pro provides a performance as good as the original system.

Currently this is only a prototype implementation of CLOCK-Pro, in which we have attempted to minimize the changes in the existing data structures and functions, and make the most of the existing infrastructure. Sometimes this means a compromise in the CLOCK-Pro performance. For example, the hardware MMU automatically sets the reference bits on the *pte* (Page Table Entry) entries of a process page table to indicate the references to the corresponding pages. In kernel 2.4, the paging system works on the active or inactive lists, whose entries are called *page descriptors*. Each *descriptor* is associated with one physical page and one or more (if the page is shared) *pte* entries in the process page tables. Each *descriptor* contains a *reference flag*, whose value is transfered from its associated *pte* when the corresponding process table is scanned. So there is an additional delay for the reference bits (flags) to be seen by the paging system. In kernel 2.4, there is no infrastructure supporting the retrieval of *pte* through the *descriptor*. So we have to accept this delay in the implementation. However, this tolerance is especially detrimental to CLOCK-Pro because it relies on a fine-grained access timing distinction to realize its advantages. We believe that further refinement and tuning of the implementation will exploit more performance potential of CLOCK-Pro.

5.2.3 The Overhead of CLOCK-Pro

Because we almost keep the paging infrastructure of the original system intact except replacing the active/inactive lists with an unified clock list and introducing a hash table, the additional overhead from CLOCK-Pro is limited to the clock list and hash table operations.

We measure the average number of entries the clock hands sweep over per page fault on the lists for the two systems. Table 6 shows a sample of the measurements. The results show that CLOCK-Pro has a number of hand movements comparable to the original system except for large memory sizes, where the original system significantly lowers its movement number while CLOCK-Pro does not. In CLOCK-Pro, for every referenced cold page seen by the moving $\mathbf{HAND}_{cold}$, there is at least one $\mathbf{HAND}_{hot}$ movement to exchange the page statuses. For a specific program with a stable locality, there are fewer cold pages with a smaller memory, as well as less possibility for a cold page to be re-referenced before $\mathbf{HAND}_{cold}$ moves to it. So $\mathbf{HAND}_{cold}$ can take a small number of movements to reach a qualified replacement page, and the number of additional $\mathbf{HAND}_{hot}$ movements per page fault is also small. When the memory size is close to the program memory demand, the original system can take less hand movements during its search on its inactive list, due to the increasing chance of finding an unreferenced page. However, $\mathbf{HAND}_{cold}$ would encounter more referenced cold pages, which causes additional $\mathbf{HAND}_{hot}$ movements. We believe that this is not a performance concern, because one page fault penalty is equivalent to the time of tens of thousands of hand movements. We also measured the bucket size of the hash table, which is only 4-5 on average. So we conclude that the additional overhead is negligible compared with the original replacement implementation.

Memory (MB)	110	140	170
Kernel 2.4.21	12.4	14.2	6.9
CLOCK-Pro	16.2	20.6	18.5

Table 6: Average number of entries the clock hands sweep over per page fault in the original kernel and CLOCK-Pro with different memory sizes for program *gnuplot*.

6 Conclusions

In this paper, we propose a new VM replacement policy, CLOCK-Pro, which is intended to take the place of CLOCK currently dominating various operating systems. We believe it is a promising replacement policy in the modern OS designs and implementations for the following reasons. (1) It has a low cost that can be easily accepted by current systems. Though it could move up to three pointers (hands) during one victim page search, the total number of the hand movements is comparable to that of CLOCK. Keeping track of the replaced pages in CLOCK-Pro doubles the size of the linked list used in CLOCK. However, considering the marginal memory consumption of the list in CLOCK, the additional cost is negligible. (2) CLOCK-pro provides a systematic solution to address the CLOCK problems. It is not just a quick and experience-based fix to CLOCK in a specific situation, but is designed based on a more accurate locality definition – reuse distance and addresses the source of the LRU problem. (3) It is fully adaptive to strong or weak access patterns without any predetermined parameters. (4) Extensive simulation experiments

and a prototype implementation show its significant and consistent performance improvements.

Acknowledgments

We are grateful to our shepherd Yuanyuan Zhou and the anonymous reviewers who helped further improve the quality of this paper. We thank our colleague Bill Bynum for reading the paper and his comments. The research is partially supported by the National Science Foundation under grants CNS-0098055, CCF-0129883, and CNS-0405909.

References

[1] L. A. Belady "A Study of Replacement Algorithms for Virtual Storage", *IBM System Journal*, 1966.

[2] S. Bansal and D. Modha, "CAR: Clock with Adaptive Replacement", *Proceedings of the 3nd USENIX Symposium on File and Storage Technologies*, March, 2004.

[3] P. Cao, E. W. Felten and K. Li, "Application-Controlled File Caching Policies", *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994.

[4] J. Choi, S. Noh, S. Min, Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme", *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999, pp. 239-252.

[5] F. J. Corbato, "A Paging Experiment with the Multics System", *MIT Project MAC Report MAC-M-384*, May, 1968.

[6] C. Ding and Y. Zhong, "Predicting Whole-Program Locality through Reuse-Distance Analysis", *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June, 2003.

[7] M. B. Friedman, "Windows NT Page Replacement Policies", *Proceedings of 25th International Computer Measurement Group Conference*, Dec, 1999, pp. 234-244.

[8] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management", *ACM Transaction on Database Systems*, Dec, 1984, pp. 560-595.

[9] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior", *Proceedings of 1997 ACM SIGMETRICS Conference*, May 1997, pp. 115-126.

[10] G. Glass, "Adaptive Page Replacement". Master's Thesis, University of Wisconsin, 1997.

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

[12] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm", *Proceedings of the 20th International Conference on VLDB*, 1994, pp. 439-450.

[13] S. Jiang and X. Zhang, "LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance", *In Proceeding of 2002 ACM SIGMETRICS*, June 2002, pp. 31-42.

[14] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim "A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References", *4th Symposium on Operating System Design & Implementation*, October 2000.

[15] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho and C. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies", *Proceeding of 1999 ACM SIGMETRICS Conference*, May 1999.

[16] N. Megiddo and D. Modha, "ARC: a Self-tuning, Low Overhead Replacement Cache", *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies*, March, 2003.

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

[18] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering", *Proceedings of the 1993 ACM SIGMOD Conference*, 1993, pp. 297-306.

[19] V. Phalke and B. Gopinath, "An Inter-Reference gap Model for Temporal Locality in Program Behavior", *Proceeding of 1995 ACM SIGMETRICS Conference*, May 1995.

[20] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, "Informed Prefetching and Caching", *Proceedings of the 15th Symposium on Operating System Principles*, 1995, pp. 1-16.

[21] J. T. Robinson and N. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement", *Proceeding of 1990 ACM SIGMETRICS Conference*, 1990.

[22] R. van Riel, "Towards an O(1) VM: Making Linux Virtual Memory Management Scale Towards Large Amounts of Physical Memory", *Proceedings of the Linux Symposium*, July 2003.

[23] R. van Riel, "Page Replacement in Linux 2.4 Memory Management", *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June, 2001.

[24] Y. Smaragdakis, S. Kaplan, and P. Wilson, "The EELRU adaptive replacement algorithm", *Performance Evaluation (Elsevier)*, Vol. 53, No. 2, July 2003.

[25] A. J. Smith, "Sequentiality and Prefetching in Database Systems", *ACM Trans. on Database Systems*, Vol. 3, No. 3, 1978, pp. 223-247.

[26] Storage Performance Council, http://www.storageperformance.org

[27] Standard Performance Evaluation Corporation, SPEC CPU2000 V1.2, http://www.spec.org/cpu2000/

[28] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems, Design and Implementation*, Prentice Hall, 1997.

[29] 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.

hStorage-DB: Heterogeneity-aware Data Management to Exploit the Full Capability of Hybrid Storage Systems

Tian Luo[1] Rubao Lee[1] Michael Mesnier[2] Feng Chen[2] Xiaodong Zhang[1]

[1]The Ohio State University
Columbus, OH
{luot, liru, zhang}@cse.ohio-state.edu

[2]Intel Labs
Hillsboro, OR
{michael.mesnier, feng.a.chen}@intel.com

ABSTRACT

As storage systems become increasingly heterogeneous and complex, it adds burdens on DBAs, causing suboptimal performance even after a lot of human efforts have been made. In addition, existing monitoring-based storage management by access pattern detections has difficulties to handle workloads that are highly dynamic and concurrent. To achieve high performance by best utilizing heterogeneous storage devices, we have designed and implemented a heterogeneity-aware software framework for DBMS storage management called hStorage-DB, where semantic information that is critical for storage I/O is identified and passed to the storage manager. According to the collected semantic information, requests are classified into different types. Each type is assigned a proper QoS policy supported by the underlying storage system, so that every request will be served with a suitable storage device. With hStorage-DB, we can well utilize semantic information that cannot be detected through data access monitoring but is particularly important for a hybrid storage system. To show the effectiveness of hStorage-DB, we have implemented a system prototype that consists of an I/O request classification enabled DBMS, and a hybrid storage system that is organized into a two-level caching hierarchy. Our performance evaluation shows that hStorage-DB can automatically make proper decisions for data allocation in different storage devices and make substantial performance improvements in a cost-efficient way.

1. INTRODUCTION

Database management systems (DBMSs) have complex interactions with storage systems. Data layouts in storage systems are established with different types of data structures, such as indexes, user tables, temporary data and others. Thus, a DBMS typically issues different types of I/O requests with different quality of service (QoS) requirements [13]. Common practice has treated storage as a black box for a long time. With the development of heterogeneous devices, such as solid-state drives (SSDs), phase-change memories (PCMs), and the traditional hard disk drives (HDDs), storage systems are inevitably becoming hybrid [5, 6, 17, 20]. The "black-box" concept of management for storage is hindering us from benefiting from the rich resources of advanced storage systems.

There are two existing approaches attempting to best utilize heterogeneous storage devices. One is to rely on database administrators (DBAs) to allocate data among different devices, based on their knowledge and experiences. The other is to rely on a management system where certain access patterns are identified by runtime monitoring data accesses at different levels of the storage hierarchy, such as in buffer caches and disks.

The DBA-based approach has the following limitations: (1) It incurs a significant and increasing amount of human efforts. DBAs, as database experts with a comprehensive understanding of various workloads, are also expected to be storage experts [3]. For example, it is a DBA's decision to use certain devices (such as SSDs) for indexes and some frequently used small tables, and less expensive devices (such as HDDs) for other data. (2) Data granularity has become too coarse to gain desired performance. As table size becomes increasingly large, different access patterns would be imposed on different parts of a table. However, all requests associated to the same table are equally treated. DBAs' efforts to divide a table into multiple partitions [18], where each partition could get a different treatment, have been in an ad-hoc manner, without a guarantee of an effective performance optimization result. (3) Data placement policies that are configured according to the common access patterns of workloads have been largely static. Changes to the policies are avoided as much as possible, because data movement in a large granularity might interrupt user applications. Therefore, such static policies are difficult to adapt to the dynamic changes of I/O demands.

Monitoring-based storage management for databases can perform well when data accesses are stable in a long term, where certain regular access patterns can be identified via data access monitoring at runtime. Examples include general-purpose replacement algorithms in production systems: LRU, LIRS [12] and ARC [15], as well as recently proposed TAC [4] and Lazy-Cleaning [7]. However, monitoring-based management may not be effective under the following three conditions. First, monitoring-based methods need a period of ramp-up time to identify certain regular access patterns. For highly dynamic workloads and commonly found data with a short lifetime, such as temporary data, the ramp-up time may be too long to make a right and timely decision. Second,

Proceedings of the VLDB Endowment, Vol. 5, No. 10

a recent study shows that data access monitoring methods would have difficulties to identify access patterns for concurrent streams on shared caching devices due to complex interferences [9]. In other words, concurrent accesses can cause unpredictable access patterns that may further reduce the accuracy of monitoring results. Third, certain information items are access-pattern irrelevant, such as content types and data lifetime [19], which are important for data placement decisions among heterogeneous storage devices. Monitoring-based approaches would not be able to identify such information. Furthermore, monitoring-based management needs additional computing and space support, which can be expensive to obtain a deep history of data accesses.

1.1 Outline of Our Solution: hStorage-DB

In order to address the limitations of DBA-based approach and particularly monitoring-based storage management, and to exploit the full capability of hybrid storage systems, we argue for a fundamentally different approach: making a direct communication channel between a DBMS and its underlying hybrid storage system. Our system framework is called heterogeneity-aware data management, or simplified as *hStorage-DB*.

We are motivated by the abundance of semantic information that is available from various DBMS components, such as the query optimizer and the execution engine, but has not been considered for database storage management. A DBMS storage manager is typically an interface to translate a DBMS data request into an I/O request. During the translation, all semantic information is stripped away, leaving only physical layout information of a request: logical block address, direction (read/write), size, and the actual data if it is a write. This in effect creates a *semantic gap* between DBMSs and storage systems.

In hStorage-DB, we bridge the semantic gap by making selected and important semantic information available to the storage manager, which can therefore classify requests into different types. With a set of predefined rules, each type is associated with a QoS policy that can be supported by the underlying storage system. At runtime, using the Differentiated Storage Services [16] protocol, the associated policy of a request is delivered to the storage system along with the request itself. Upon receiving a request, the storage system, first extracts the associated QoS policy, and then uses a proper mechanism to serve the request as required by the QoS policy.

As a case study, we have experimented with a hybrid storage system which is organized into a two-level hierarchy. Level one, consisting of SSDs, works as a cache for level two, consisting of HDDs. This storage system provides a set of caching priorities as QoS policies. Experiment results show the strong effectiveness of hStorage-DB.

Comparing with monitoring-based approaches [4, 7, 12, 15], hStorage-DB has the following unique advantages: (1) Under this framework, a storage system has the accurate information of how (and what) data will be accessed. This is especially important for highly dynamic query executions and concurrent workloads. (2) A storage system directly receives the QoS policy for each request, thus could quickly invoke the appropriate mechanism to serve. (3) Besides having the ability to pass access-pattern irrelevant semantic information, in hStroage-DB, storage management does not need special data structures required by various monitoring-based operations, thus incurs no additional computation and space overhead. (4) A DBMS can directly communicate with a storage system about the QoS policy for each request in an automatic mode, so that DBAs can be relieved from the burdens of storage complexity.

1.2 Critical Technical Issues

In order to turn our design of hStorage-DB into a reality, we must address the following technical issues.

Associating a proper QoS policy to each request: Semantic information does not directly link to proper QoS policies that can be understood by a storage system. Therefore, in order for a storage system to be able to serve a request with the correct mechanism, we need a method to accomplish the effective mapping from semantic information to QoS policies. However, a comprehensive solution must systematically consider multiple factors, including the diversity of query types, the complexity of various query plans, and the issues brought by concurrent query executions.

Implementation of hStorage-DB: Two challenges need to be addressed. (1) The QoS policy of each request eventually needs to be passed into the storage system. A DBMS usually communicates with storage through a block interface. However, current block interfaces do not allow passing anything other than the physical information of a request. (2) A hybrid storage system needs an effective mechanism to manage heterogeneous devices, so that data placement would match the unique functionalities and abilities of each device. Data also needs to be dynamically moved among devices to respond access pattern changes.

1.3 Our Contributions

This paper makes the following contributions. (1) We have identified a critical issue in the storage management of DBMSs, namely a semantic gap between the requirements of DBMS I/O requests and the supported services of heterogeneous storage services. Bridging this gap would significantly improve the performance of databases, particularly for complex OLAP queries and highly concurrent workloads, by addressing the limits of DBA-based and monitoring-based storage management approaches. We have designed hStorage-DB that restructures the storage management layer with semantic information to interface with the storage system. (2) We have implemented a system prototype of hStorage-DB that exploits the full capability of hybrid storage systems for database workloads by making informed, fine-grained and dynamic data block placement in a storage system. (3) We have evaluated the effectiveness of our prototype with a hybrid storage system, within which we use an SSD as a cache on top of hard disk drives. This storage system provides a set of caching priorities, that can be assigned to different types of requests. Performance of this system can be significantly improved by well utilizing the limited SSD capacity.

The rest of this paper is organized as follows. Section 2 outlines the architecture of hStorage-DB. Section 3 introduces QoS policies. Section 4 carries the core design of hStorage-DB by presenting a set of rules for assigning QoS policies to I/O requests. Section 5 overviews the Differentiated Storage Services protocol and its reference implementation (a hybrid storage system with caching priorities). Section 6 presents our experimental results. Section 7 discusses related work. Section 8 concludes this paper.

2. ARCHITECTURE OF hStorage-DB

Figure 1 shows the architecture of hStorage-DB. When the buffer pool manager sends a request to the storage manager, associated semantic information is also passed. We extend the storage manager with a "policy assignment table", which stores the rules to assign each request a proper QoS policy, according to its semantic information. The QoS policy is embedded into the original I/O request and delivered to the storage system through a block interface. We have implemented hStorage-DB by using the Differentiated Storage Services protocol from Intel Labs [16] to deliver a request and its associated policy to a hybrid storage system. Upon receiving a request, the storage system first extracts the policy, and invokes a mechanism to serve this request.

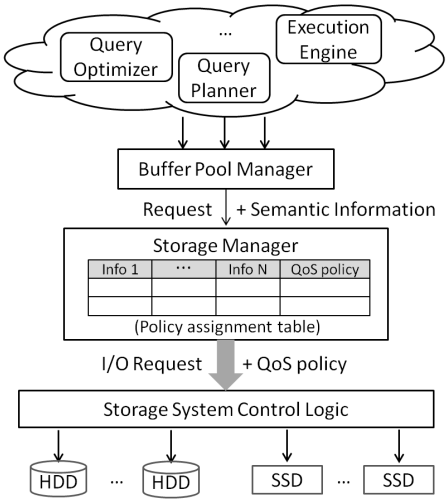


Figure 1: The architecture of hStorage-DB.

Our implementation of hStorage-DB is based on PostgreSQL 9.0.4. It mainly involves three issues: (1) We have instrumented the query optimizer and the execution engine to retrieve semantic information embedded in query plan trees and in buffer pool requests. (2) We have augmented the data structure of the buffer pool to store collected semantic information. The storage manager has also been augmented to incorporate the "policy assignment table". (3) Finally, since PostgreSQL is a multi-process DBMS, to deal with concurrency, a small region of the shared memory has been allocated for global data structures (Section 4.3) that need to be accessed by all processes.

The key to make hStorage-DB effective is associating each request with a proper QoS policy. In order to achieve our goal, we need to take the following 2 steps:

1. Understanding QoS policies and their storage implications;
2. Designing a method to determine an accurate mapping from request types to QoS policies.

In the following two sections, we will discuss the details.

3. QOS POLICIES

We will first discuss general QoS policies and then introduce the specific policies used in this paper.

3.1 Overview of QoS Policies

QoS policies provide a high-level service abstraction for a storage system. Through a set of well defined QoS policies, a storage system can effectively quantify its capabilities without exposing device-level details to users.

A QoS policy can either be performance related, such as latency or bandwidth requirements, or non-performance related, such as reliability requirements. All policies of a storage system are dependent on its hardware resources and organization of these resources. For example, if a storage system provides a reliability policy, then for an application running on such a storage system, when it issues write requests of important data, it can apply a reliability policy to these requests. Thus, when such a request is delivered, the storage system can automatically replicate received data to multiple devices.

On the one hand, QoS policies can isolate device-level complexity from applications, thus reducing the knowledge requirement on DBAs, and enabling heterogeneity-aware storage management within a DBMS. On the other hand, these policies determine the way in which a DBMS can manage requests. It is meaningless to apply a policy that cannot be understood by the storage system. Therefore, a different storage management module may be needed if a DBMS is ported to another storage system that provides a fundamentally different set of policies.

3.2 QoS Policies of a Hybrid Storage System

In this paper, we will demonstrate how to enable automatic storage management with a case study where the QoS policies are specified as a set of *caching priorities*.

The underlying system is a hybrid storage system prototype (detailed in Section 5) organized into a two-level hierarchy. The first level works as a cache for the second level. We use SSDs at the first level, and HDDs at the second level. In order to facilitate the decision making on cache management (which block should stay in cache, and what should not), its QoS policies are specified as a set of *caching priorities*, which can be defined as a 3-tuple:

$\{N, t, b\}$, where $N > 0$, $0 \leq t \leq N$, and $0\% \leq b \leq 100\%$.

Parameter N defines the total number of priorities, where a smaller number means a higher priority, i.e., a better chance to be cached.

Parameter t is a threshold for "non-caching" priorities: blocks accessed by a request of a priority $\geq t$ would have no possibility of being cached. In this paper, we set $t = N - 1$. So, there are two non-caching priorities, $N - 1$ and N. We call priority $N - 1$ *"non-caching and non-eviction"*, and call N *"non-caching and eviction"*.

There is a special priority, called *write buffer*, configured by parameter b. More details about these parameters will be discussed later.

For each incoming request, the storage system first extracts its associated QoS policy, and then adjusts the placement of all accessed blocks accordingly. For example, if a block is accessed by a request associated with a "high-priority", it will be fetched into cache if it is not already cached, depending on the relative priority of other blocks that are already in cache. Therefore in practice, the priority of a request is eventually transformed to the priority of all accessed data blocks. In the rest of paper, we will also use "priority of a block" without further explanation.

4. QOS POLICY FOR EACH REQUEST

In this section, we will present a set of rules that associate different I/O requests with appropriate QoS policies.

4.1 Request Types

A database I/O request has various semantic information. For the purpose of caching priorities, in this paper, we consider semantic information from the following categories.

Content type: We focus on three major content types: regular table, index and temporary data. Regular tables define the content of a database. They are the major consumers of database storage capacity. Indexes are used to speedup the accessing of regular tables. Temporary data, such as a hash table [8], would be generated during the execution of a query, and removed before the query is finished.

Access pattern: It refers to the behavior of an I/O request. It is determined by the query optimizer. A table may be either sequentially scanned or randomly accessed. An index is normally randomly accessed.

According to collected semantic information, we can classify requests into the following types: (1) *sequential requests*; (2) *random requests*; (3) *temporary data requests*; (4) *update requests*. The discussion of QoS policy mapping will be based on these types.

4.2 Policy Assignment in a Single Query

We will present five rules that are used to map each request type to a proper QoS policy which, in this case study, is a caching priority. For each request, the rules mainly consider two factors: 1) performance benefit if data is served from cache, and 2) data reuse possibility. These two factors determine if we should allocate cache space for a disk block, and if we decide to allocate, how long should we keep it in cache. In this subsection, we will consider priority assignment within the execution of a single query, and then discuss the issues brought by concurrent query executions in the next subsection.

4.2.1 Sequential Requests

In our storage system, the caching device is an SSD, and the lower-level uses HDDs, which can provide a comparable sequential access performance to that of SSDs. Thus, it is not beneficial to place sequentially accessed blocks in cache.

RULE 1: *All sequential requests will be assigned the "non-caching and non-eviction" priority.*

A request with the "non-caching and non-eviction" priority has two implications: (1) If the accessed data is not in cache, it will not be allocated in cache; (2) If the accessed data is already in cache, its priority, which is determined by a previous request, will not be affected by this request. In other words, requests with this priority do not affect the existing storage data layout.

4.2.2 Random Requests

Random requests may benefit from cache, but the eventual benefit is dependent on data reuse possibility. If a block is randomly accessed once but never randomly accessed again, we should not allocate cache space for it either. Our method to assign priorities for random requests is outlined in Rule 2.

RULE 2: *Random requests issued by operators at a lower-level of its query plan tree will be given a higher caching priority than those that are issued by operators at a higher-level of the query plan tree.*

This rule can be further explained with the following auxiliary descriptions.

Level in a query plan tree: For a multi-level query plan tree, we assume that the root is on the highest level; the leaf that has the longest distance from the root is on the lowest level, namely Level 0.

Related operators: This rule relates to random requests that are mostly issued by "index scan" operators. For such an operator, the requests to access a table and its corresponding index are all random.

Blocking operators: With a blocking operator, such as hash or sorting, operators at higher levels or its sibling operator cannot proceed unless it finishes. Therefore, the levels of affected operators will be recalculated as if this blocking operator is at Level 0.

Priority range: Note that there are totally N different priorities, but not all of them will be used for random requests. Instead, random requests are mapped to a consecutive priority range $[n_1, n_2]$, where $n_1 \leq n_2$. So, n_1 is the highest available priority for random requests; and n_2 is the lowest available priority.

When multiple operators access the same table: For some query plans, the same table may be randomly accessed by multiple operators. In this case, the priorities of all random requests to this table are determined by the operator at the lowest level of the query plan tree. If there is an operator that sequentially access the same table, the priority of this operator's requests is still determined by Rule 1.

Function (1) formalizes the process of calculating the priority of a random request, issued by an operator at Level i of the query plan tree. Assume l_{low} is the lowest level of all random access operators in the query plan tree, while l_{high} is the highest level. And L_{gap} represents this gap, where $L_{gap} = l_{high} - l_{low}$. Assume C_{prio} is the size of the available priority range $[n_1, n_2]$, so $C_{prio} = n_2 - n_1$.

$$p(i) = \begin{cases} n_1 & \text{if } C_{prio} = 0 \\ n_1 & \text{if } L_{gap} = 0 \\ n_1 + i - l_{low} & \text{if } C_{prio} \geq L_{gap} \\ n_1 + \lfloor C_{prio} * \frac{i - l_{low}}{L_{gap}} \rfloor & \text{if } C_{prio} < L_{gap} \end{cases} \quad (1)$$

The last branch of this function describes the case when a tree has too many levels that there are not enough priorities to assign for each level. In this case, we can assign priorities according to the relative location of operators, and operators at neighboring levels may share the same priority.

Let us take the query plan tree in Figure 2 as an example. In this example, three tables are accessed: $t.a$, $t.b$ and $t.c$. We assume that the available priority range is [2,5]. Both operators that access $t.a$ are index scans. Since the lowest level of random access operators for $t.a$ is Level 0, all random requests to $t.a$ and its index would be assigned Priority 2. It also means that requests from the "index scan" operator at Level 1 are assigned the same priority: Priority 2.

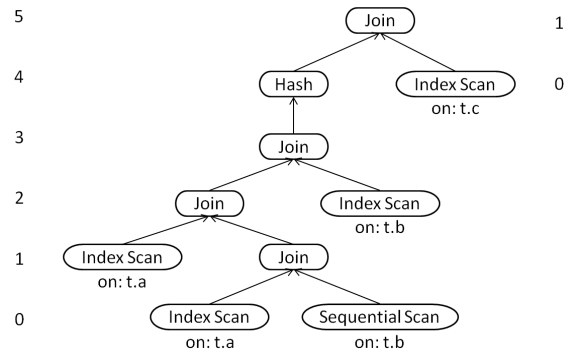


Figure 2: An example query plan tree. This tree has 6 levels. Root is on the highest level: Level 5. Due to the blocking operator "hash", the other two operators on Level 4 and 5 are re-calculated as on Level 0 and 1.

As to $t.b$, there are also two related operators. But according to Rule 1, all requests from the "sequential scan" operator (Level 0) are assigned the "non-caching and non-eviction" priority. Requests from the other operator (Level 2) that accesses $t.b$ are random. According to Function (1), requests from this operator should be assigned Priority 4.

For Table $t.c$, it is accessed by random requests from an "index scan" operator. However, due to the blocking operator "hash" on Level 4, the "index scan" operator is considered at Level 0 in priority recalculation, and thus all random requests to table $t.c$ would be assigned Priority 2.

4.2.3 Temporary Data Requests

Queries with certain operators may generate temporary data during execution. There are two phases associated with temporary data: *generation phase* and *consumption phase*. During generation phase, temporary data is created by a write stream. During consumption phase, temporary data is accessed by one or multiple read streams. In the end of consumption phase, the temporary data is deleted to free up disk space. Based on this observation, we should cache temporary data blocks once they are generated, and immediately evict them out of cache at the end of their lifetime.

RULE 3: *All read/write requests to temporary data are given the highest priority. The command to delete temporary data is assigned the "non-caching and eviction" priority.*

A request with the "non-caching and eviction" priority has two implications: (1) If the accessed data is not in cache, it will not be promoted into cache; (2) If the accessed data is already in cache, its priority will be changed to "non-caching and eviction", and can be evicted timely. Thus, requests with the "non-caching and eviction" priority only allow data to leave cache, instead of getting into cache.

Normally, if a DBMS is running with a file system, the file deletion command only results in metadata changes of the file system, without notifying the storage system about which specific blocks have become useless. This becomes a problem because temporary data may not be evicted promptly. And because of its priority, temporary data cannot be replaced by other data. Gradually, the cache will be filled with obsolete temporary data.

This issue can be addressed by the newly proposed TRIM command [1], which can inform the storage system of what LBA (logical block address) ranges have become useless due to file deletions, or other reasons. Supported file systems, such as EXT4, can automatically send TRIM commands once a file is deleted. For a legacy file system that does not support TRIM, we can use the following workaround to achieve the same effect: Before a temporary data file is deleted, we issue a series of read requests, with the "non-caching and eviction" priority, to scan the file from beginning to end. This will in effect tell the storage system that these blocks can be evicted immediately. This workaround incurs some overhead at an acceptable level, because the read requests are all sequential.

4.2.4 Update Requests

We allocate a small portion of the cache to buffer writes from applications, so that they do not access HDDs directly. With a write buffer, all written data will first be stored in the SSD cache, and flushed into the HDD asynchronously. Therefore, we apply the following rule for update requests:

RULE 4: *All update requests will be assigned the "write buffer" priority.*

There is a parameter b that determines how much cache space is allocated as a write buffer. When the occupied space of data written by update requests exceeds b, all content in the write buffer is flushed into HDD. Note that the write buffer is not a dedicated space. Rather, it is a special priority that an update request can "win" cache space over requests of any other priority. For OLAP workloads in this paper, we set b at 10%.

4.3 Concurrent Queries

When multiple queries are co-running, I/O requests accessing the same object might be assigned different priorities depending on which query they are from. To avoid such non-deterministic priority assignment, we apply the following rule for concurrent executions.

RULE 5:

1. *For sequential requests, temporary data requests and updates, the priority assignment still follows Rule 1, Rule 3 and Rule 4;*

2. *For random requests that access the same object (table or index) but for different queries, they are assigned the highest of all priorities, each of which is determined by Rule 3 and independently based on the query plan of each running query;*

To implement this rule, we store some global information for all queries to access: a hash table $H < oid, list >$, two variables gl_{low} and gl_{high}.

1. The key "*oid*" stores an object ID that is either of a table or of an index.
2. The structure "*list*" for each *oid* is a list.
3. Each element of *list* is a 2-tuple $< level, count >$. It means that among all queries, there are totally *count* operators accessing *oid*, and all of these operators are on Level *level* in their own query plan tree. If some operators on different levels of a query plan tree are also accessing *oid*, we need another element to store this information.
4. Variable gl_{low} (the global lowest level of all random operators) stores the minimum value of all l_{low} according

to each query; similarly, gl_{high} stores the maximum value of all l_{high} according to each query.

All these data structures are updated upon the start and end of each query. To calculate the priority of a random request, with concurrency in consideration, we can still use Function (1), just changing l_{low} into gl_{low}, and similarly l_{high} to gl_{high}. and thus L_{gap} would be $gl_{high} - gl_{low}$. Figure 3 describes this process.

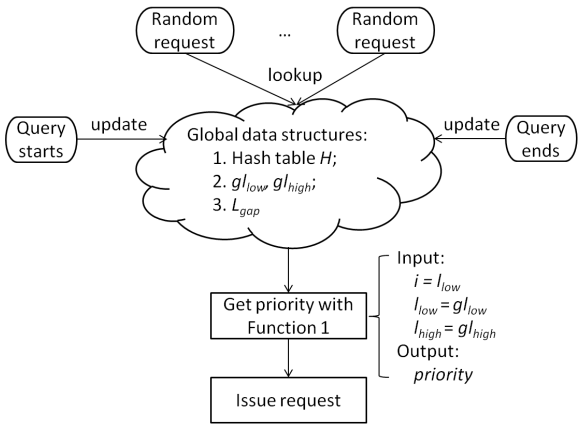


Figure 3: The process to calculate request priorities.

Table 1 summarizes all the rules hStorage-DB uses to assign caching priorities to requests.

Request type	Priority	Rule
temporary data requests	1	Rule 3
random requests	2 ... N-2	Rules 2, 5
sequential requests,	N-1	Rule 1
TRIM to temporary data	N	Rule 3
updates	write buffer	Rule 4

Table 1: Rules to assign priorities.

5. STORAGE SYSTEM PROTOTYPE

The hybrid storage system we experimented with in this paper is a pre-release version of Intel's Open Storage Toolkit [11], which is organized into a two-level hierarchy. We use SSDs on the first level, working as a caching device for HDDs on the second level.

Using the Differentiated Storage Services protocol, an I/O request may not only contain physical information, but may also carry semantic information. This protocol provides backward compatibility with current block interfaces, such as SCSI, so that a classification-enabled DBMS can still run on top of a legacy storage system, while the semantic information of each request will simply be ignored.

5.1 Cache Management

As with any cache device, the most important data placement decisions are cache admission and eviction. They decide which data should be placed in cache and which data should be replaced. Within this storage system, both decision making processes are based on priorities. Therefore, they are called *selective allocation* and *selective eviction*.

- **Selective allocation:** Regarding the non-caching threshold t, only blocks with priorities $< t$ will be considered to cache.The final decision depends on the current cache capacity and relative priority of other in-cache blocks. For an incoming block, denoted as N_{new}, whose priority is k and $k < t$, if the cache has additional space, it will be cached. Otherwise, if there exists a block, denoted as N_{old}, whose priority is k' and $k' \geq k$, which means block N_{old} has a lower priority than N_{old}. In this case, N_{new} will also be cached, but after N_{old} gets evicted (see below).

- **Selective eviction:** Eviction happens when cache needs to make room for new blocks. In order to determine which in-cache block should be evicted, the cache device first identifies blocks with Priority k, such that all other blocks in the cache have their priorities $< k$. Then among the blocks of Priority k, the "least-recently-used" one is selected to be evicted.

Cached blocks are organized into N priority groups, where N is the total number of priorities. Group $k, k \in [1, N]$, only contains blocks of Priority k. Each group is managed by the *LRU* (Least Recently Used) algorithm. Depending on the value of the "non-caching" threshold t, some low-priority groups may always be empty.

There are two types of blocks in cache: valid and invalid. Each valid block corresponds to a unique block within a level-two device, while invalid (or free) blocks do not correspond to any blocks in level-two devices. A valid block has two states: *clean* or *dirty*. A block is clean if there is an identical copy within a level-two device. Otherwise it is dirty. Based on selective allocation and selective eviction, the cache may perform one of the following six actions.

1. **Cache hit:** This action is taken when blocks accessed by an incoming request are already in cache. In this case, the caching device will directly communicate with OS. Based on the priority assigned to this request, there might be a follow-up action: "re-allocation". It will be explained later.

2. **Read allocation:** If the blocks accessed by an incoming read request are not in cache, but they are qualified for being cached, read allocation is involved. In this case, cache will first allocate enough space. Some in-cache blocks may be evicted if necessary. After that, new blocks will be read from level-two devices into cache, marked as clean, and then served to OS[1].

3. **Write allocation:** If the blocks contained in an incoming write request are not in cache, but they are qualified for being cached, write allocation is involved. After enough cache space is allocated, incoming blocks will be placed in cache, and marked as dirty. As soon as marking is done, the write request is returned. For both write allocation and read allocation, there might be data transmitted from cache to level-two devices, due to the eviction of dirty blocks.

[1]This is called synchronous read allocation, because data is placed into cache before the read request returns. Its opposite is asynchronous read allocation: blocks are served from level-two storage devices directly into OS, and placed into cache during idle time.

4. **Bypassing:** Bypassing is involved when blocks accessed by an incoming request are not in cache and are not qualified for being cached either. Since a storage-level cache is not necessarily on the critical path of data flow, for a read/write request with a low enough priority, its accessed blocks can be directly transmitted between OS and level-two devices, thus "bypassing" the cache.

5. **Re-allocation:** This action is taken when an accessed block is already in cache, but assigned a new priority. As described earlier, blocks in cache are organized into multiple priority groups. With a new priority, the block will be "removed" from its current group, and "inserted" into the corresponding new group.

6. **Eviction:** At times, a certain number of in-cache blocks (victims) should be evicted, to make room for new blocks. The selection of victims are described in the above "selection eviction". Once selected, victim blocks will be removed from their corresponding priority groups. For victim blocks that are also dirty, they will be written into level-two devices.

5.2 Metadata Management

In the storage system, blocks are managed by N priority groups and a hash table. The total size of the metadata is proportional to the cache size.

The hash table is designed to facilitate the look-up of cached blocks. Each item in the hash table can be defined as $< lbn, V >$:

- *lbn* is the logical block number. It is used as the hash key. If a logical block number is found in the hash table, the corresponding block is in cache; otherwise it is not cached.

- The value V is itself a two-tuple $< pbn, prio >$. *pbn* is a physical block address; it indicates where to find the block *lbn* in the SSD cache. Since the cache device is invisible to the OS or applications, we cannot directly use *lbn* to access a cached block. *prio* stores the associated priority of the block, and it indicates which priority group this block belongs to.

All the data structures are stored in the main memory of the storage system.

6. PERFORMANCE EVALUATION

In this section, we will first present the effectiveness of hStorage-DB on accelerating executions of single queries. Then we will discuss concurrent workloads.

6.1 Experiment Setup

Our experiment platform consists of two machines, connected with a 10 Gb Ethernet link. One machine runs the DBMS, and the other is a storage server. Both are of the same configurations: 2 Intel Xeon E5354 processors, each having four 2.33GHz cores, 8 GB of main memory, Linux 2.6.34, two Seagate Cheetah 15.7K RPM 300 GB HDDs. The storage server has an additional Intel 320 Series 300 GB SSD to be our cache device. Key specifications of this SSD are shown in Table 2. Although a more high-end SSD would certainly improve cache performance, as we will find later, such an entry-level SSD could already demonstrate strong effectiveness of hStorage-DB.

Sequential Read/Write	Random Read/Write
270 MB/s / 205 MB/s	39.5K IOPS / 23K IOPS

Table 2: Performance specification of the cache device [10].

On the storage server, one HDD runs the operating system; the other HDD and the SSD consist of the caching hierarchy as described in Section 5. The storage system is exported as a normal SCSI device to the DBMS server through iSCSI (Internet SCSI). On the DBMS server, all database data requests are sent to the storage server. except for transaction logs, which go to a dedicated local HDD.

We choose TPC-H [2] at a scale factor of 30 as our OLAP benchmark. With 9 indexes (shown in Table 3), the total dataset size is 46GB.

1	lineitem (l_partkey);
2	lineitem (l_orderkey);
3	orders (o_orderkey);
4	partsupp (ps_partkey);
5	part (p_partkey);
6	customer (c_custkey);
7	supplier (s_suppkey);
8	region (r_regionkey);
9	nation (n_nationkey);

Table 3: Indexes built for TPC-H.

6.2 Diversity of Request Types

Classification is meaningful only if a DBMS issues I/O requests of different types. In order to verify this assumption, for each query, we run it once, and count the number of I/O requests of each type, as well as the total number of disk blocks served for requests of each type.

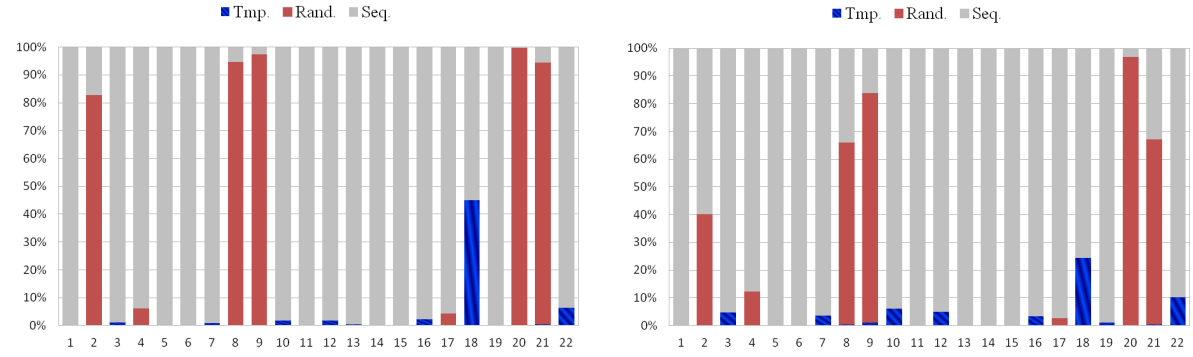
As shown in Figure 4, we can observe requests of various types: *sequential requests*, *random requests* and *temporary data requests*.

6.3 Query Performance

For each query, we run it with the following four different storage configurations. (1) HDD-only; (2) LRU; (3) hStorage-DB; (4) SSD-only. HDD-only shows the baseline case when all I/O requests are served by a hard disk drive; SSD-only shows the ideal case when all I/O requests are served by an SSD; LRU emulates a classical approach when cache is managed by the LRU (least recently used) algorithm; In hStorage-DB, the storage system is managed under the framework as proposed in this paper. When we experiment with LRU and hStorage-DB, the SSD cache size is set to be 32GB, unless otherwise specified.

6.3.1 Sequential Requests

To demonstrate the ability of hStorage-DB to avoid unnecessary overhead for allocating cache space for low-locality data, we have experimented with Queries 1, 5, 11 and 19, whose executions are dominated by sequential requests, according to Figure 4. Test results are shown in Figure 5.



(a) Percentage of each type of requests.

(b) Percentage of each type of disk blocks.

Figure 4: Diversity of IO requests in TPC-H queries. X-axis: Name of queries in TPC-H. Y-axis: Percentage of each type.

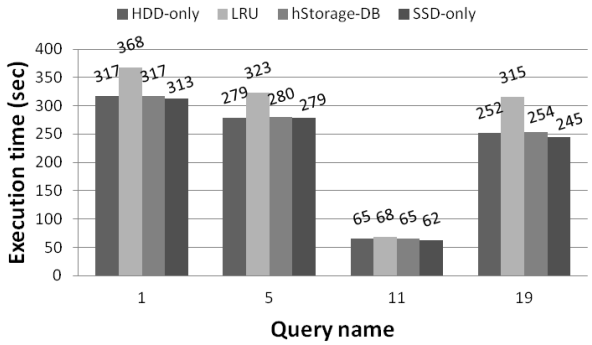


Figure 5: Execution times of queries dominated by sequential requests.

There are three observations from Figure 5: (1) The advantage of using SSD is not obvious for these queries. (2) If the cache is managed by LRU, which is not sensitive to sequential requests, the overhead can be significant. For example, compared with the baseline case, the execution time of LRU cache increased from 317 to 368 seconds for Q1, and from 252 to 315 seconds for Q19, resulting in a slowdown of 16% and 25% respectively. (3) Within the framework of hStorage-DB, sequential requests are associated with the "non-caching and non-eviction" priority, so they are not allocated in cache, and thus incurs almost no overhead.

	# of accessed blocks	# of hits	hit ratio
Q1	6,402,496	19,251	0.3%
Q5	8,149,376	17,694	0.2%
Q11	1,043,710	0	0%
Q19	6,646,328	16,798	0.3%

Table 4: Cache statistics for sequential requests with LRU.

We have listed in Table 4 the number of accessed blocks and the number of cache hits for each query, when cache is managed by LRU. From this table, we can see that although caching data requested by sequential requests can bring cache hits for some queries, the hit ratio is negligible. Even the highest cache hit ratio is 0.3%, for Q1 and Q19.

6.3.2 *Random Requests*

In order to show the effectiveness of Rule 2 (Section 4.2.2). we have experimented with Q9 and Q21. Both have a significant amount of random requests, according to Figure 4. Results are plotted in Figure 6.

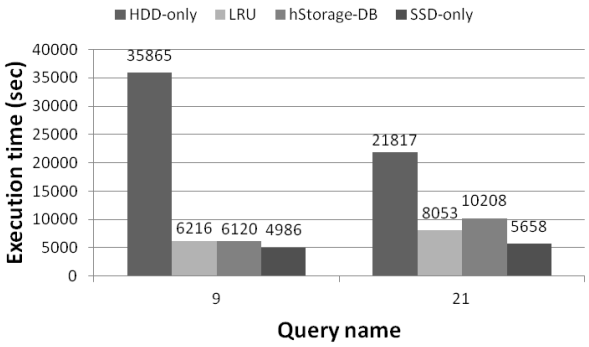


Figure 6: Execution times of queries dominated by random requests.

We have three observations from Figure 6. (1) The advantage of using SSD is obvious for the such queries. The performance of the ideal case (SSD-only) is far more superior than the baseline case (HDD-only). For Q9 and Q21, the speedup is 7.2 and 3.9 respectively. (2) Both queries have strong locality. LRU can effectively keep frequently accessed blocks in cache, through its monitoring of each block, and hStorage-DB achieves the same effect, through a different approach. (3) For Q21, hStorage-DB is slightly lower than LRU, which will be explained later.

To help better understand the performance numbers, Figure 7 shows the query plan tree of Q9. As we can see, there are two randomly accessed tables "supplier" and "orders". According to Rule 2, requests to "supplier" and its index are assigned Priority 2, and requests to "orders" and its index are assigned Priority 3.

Table 5 shows the cache statistics for requests of the two different priorities when executed by hStorage-DB. We can

see that random requests of Q9 are served with a high cache hit ratio. LRU has a similar cache hit ratio, so we have omitted its numbers.

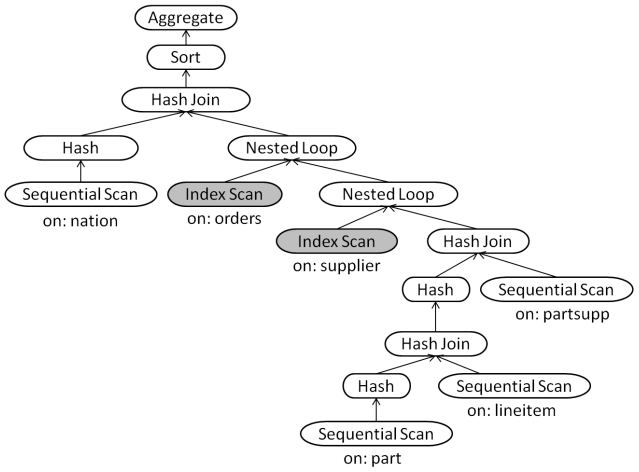


Figure 7: Query plan tree of Query 9.

	Priority 2	Priority 3
# of accessed blocks	10,556,346	30,429,858
Cache hits	9,619,456	26,981,259
Cache misses	936,890	3,448,499
hit ratio	91%	89%

Table 5: Cache statistics for random requests of Query 9 with hStorage-DB.

Similarly, Figure 8 shows the query plan tree of Q21. According to this figure, tables "orders" and "lineitem" are randomly accessed. Based on Rule 2, requests to "orders" and its index are assigned Priority 2, and requests to "lineitem" and its index are assigned Priority 3.

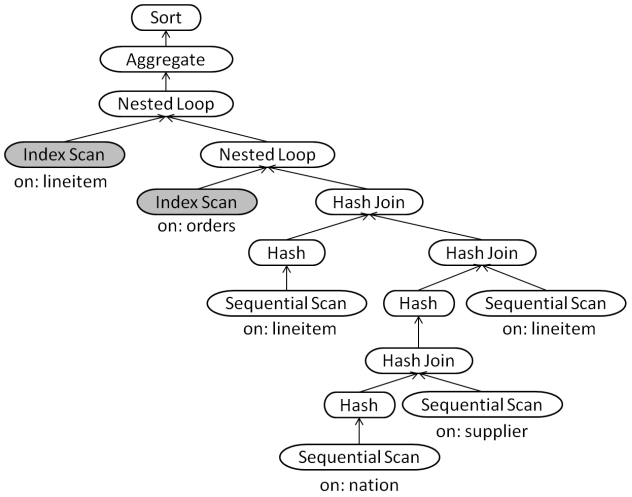


Figure 8: Query plan tree of Query 21.

hStorage-DB			
	Priority 2	Priority 3	Sequential
# of accessed blks	18,353,605	11,591,715	12,816,956
Cache hits	16,585,399	7,366,930	147,656
hit ratio	90.3%	40.1%	0.1%
LRU			
	Priority 2	Priority 3	Sequential
# of accessed blks	18,211,959	10,876,511	12,816,959
Cache hits	16,430,097	8,954,023	6,524,852
hit ratio	90.2%	82.3%	50.9%

Table 6: Cache hits/misses for Query 21.

Table 6 shows the cache statistics for requests of the two different priorities[2]. According to this table, both hStorage-DB and LRU can deliver a high cache hit ratio for Priority 2 requests, which are random requests to table "orders" and its index. But compared with hStorage-DB, LRU delivers a higher cache hit ratio for Priority 3 and sequential requests, which are all related to table "lineitem". As we can see from the query plan in Figure 8, "lineitem" is accessed by 3 operators, two "sequential scan" operators and one "index scan" operator. Therefore, they benefit from each other with LRU. In hStorage-DB, a block accessed by a sequential request would not be placed into cache, unless it is already cached. This is why, in terms of the execution time of Q21, hStorage-DB slightly underperforms LRU. However, equally treating sequential and random requests only benefits this case. We will show the disadvantages of being unable to discriminate sequential requests from other requests in Section 6.4.

6.3.3 *Temporary Data Requests*

In this section, we demonstrate the effectiveness of Rule 3 by experimenting with Q18, which generates a large number of temporary requests during its execution. Results are plotted in Figure 9.

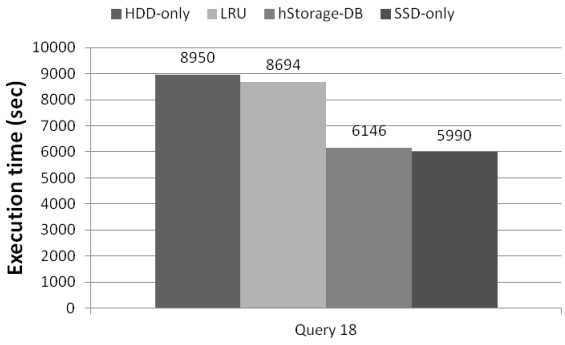


Figure 9: Execution time of Query 18.

This figure gives us the following three observations. (1) The advantage of using SSD is also obvious for this query, giving a speedup of 1.45 over the baseline case (HDD-only).

[2]In the lower half of the table, although we record statistics separately for requests of different priorities, all requests are managed through a single LRU stack.

(2) LRU also improves performance, because some temporary data blocks can be cached. But they are not kept long enough, so the speedup is not obvious. (3) hStorage-DB achieves a significant speedup because temporary data is cached until the end of its lifetime. The nature of "temporary data" can only be informed semantically.

As can be seen from the query plan (Figure 10), temporary data is generated by "hash" operators (in shaded boxes). We show the cache statistics in Table 7. Because writes of temporary data are all cache misses, we only consider temporary data reads. According to this table, LRU improves performance because some cached blocks are served from the cache, however, the cache hit ratio is only 1.8% for reads of temporary data. On the contrary, the hit ratio of temporary data reads is 100% with hStorage-DB.

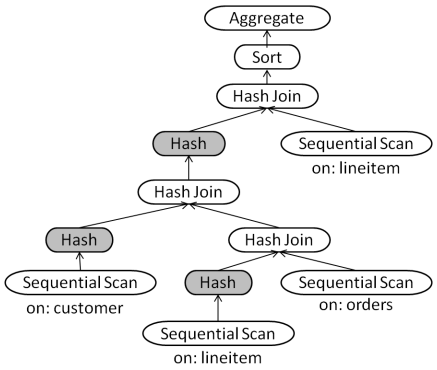


Figure 10: Query plan tree of Query 18.

hStorage-DB		
	Sequential	Temp. read
# of accessed blks	19,409,504	5,374,440
Cache hits	0	5,374,440
hit ratio	0%	100%
LRU		
	Sequential	Temp. read
# of accessed blks	19,409,358	5,374,486
Cache hits	64,552	96,741
hit ratio	0.3%	1.8%

Table 7: Cache hits of different blocks for Query 18.

6.3.4 *Running a Sequence of Queries*

We have tested the performance of hStorage-DB with a stream of "randomly" ordered queries. We use the order of power test by the TPC-H specification [2]. We omit the results from a LRU-managed SSD cache, which has already been shown much less efficient than hStorage-DB.

Results are shown in Figure 11. In this figure, "RF1" is the update query at the beginning, and "RF2" is the update query in the end. For readability, the results of short queries and those of long queries are shown separately. According to the results, hStorage-DB shows clear performance improvements for most queries. In terms of total execution time of the sequence of queries, as shown in Table 8, hStorage-DB also improves significantly over the baseline case.

This experiment with a long sequence of queries took hStorage-DB more than 10 hours to finish, which shows the stability and practical workability of the hStorage-DB solution. Different from running each query independently, the success of this experiment involves the issues of data reuse and cache space utilization during a query stream. Particularly, useless cached data left from a previous query need to be effectively evicted from cache. Experiment results have shown that in the framework of hStorage-DB, (1) temporary data can be evicted promptly, by requests with the "non-caching and eviction" priority; and (2) data randomly accessed by a previous query can be effectively evicted by random requests of the next query, if it will not used again.

HDD-only	hStorage-DB	SSD-only
86,009	39,132	23,953

Table 8: Total execution time (seconds) of the sequence.

6.4 Concurrency

We have run a throughput test by the TPC-H specification [2]. In this test, we set the scale factor at 10, and the total dataset size was 16GB. We used 2GB main memory and 4GB SSD cache. During the test, we used 3 query streams and 1 update stream. Table 9 shows the overall results.

HDD-only	LRU	hStorage-DB	SSD-only
13	28	43	114

Table 9: TPC-H throughput results.

According to this table, for throughput test, hStorage-DB has a 3.3X speedup over the baseline case, and a 1.5X speedup over the performance of LRU. We can see that these speedups are much larger than those observed in previous experiments for single queries. To explain this, we look into the execution time of different types of queries in throughput test and single query test.

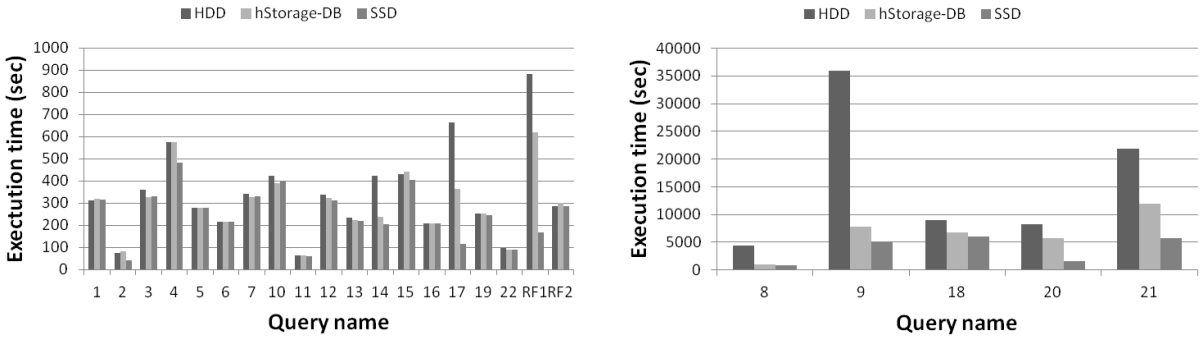
To clearly show the benefits of hStorage-DB, we closely study two queries: Q9 and Q18. We have confirmed that at a scale factor of 10, Q9 also has a significant number of random requests, while Q18 also has many temporary data requests. Figure 12a shows their performance numbers when running independently. Figure 12b shows the average execution time of the two queries in the throughput test.

For Q9, hStorage-DB can still give a performance result that is close to the ideal case (SSD only). This is because the data randomly accessed by this query has been successfully protected from other cache-polluting data during its execution. In comparison, even though the performance of LRU is close to that of hStorage-DB when running Q9 independently, in the case of concurrent workloads, it is 2.8 times lower than hStorage-DB.

Q18 is another example, which has temporary data during execution. When running Q18 independently, LRU is only 1.2 times slower than hStorage-DB, but in the case of concurrent workloads, it becomes 1.85 times slower.

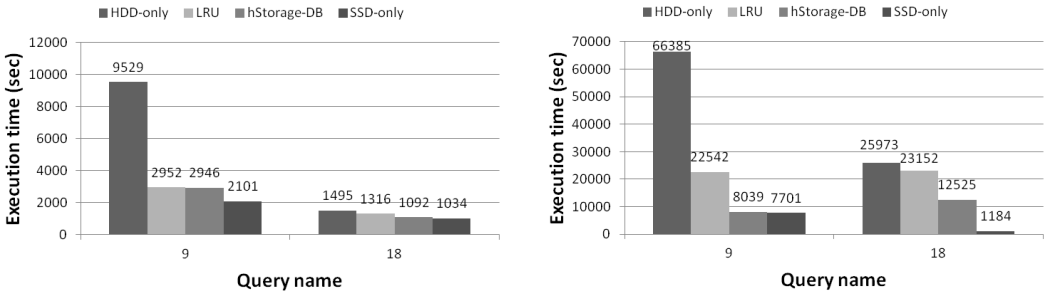
6.5 Summary of Experiment Results

Our experiments have shown the effectiveness of hStorage-DB in three unique ways: (1) For workloads whose access patterns cannot be effectively detected by monitoring-based methods, hStorage-DB can directly pass critical semantic



(a) Short queries (b) Long queries

Figure 11: Execution times of queries when packed into one query stream.



(a) Standalone execution times. (b) Average execution times in throughput test.

Figure 12: Comparison of the execution times for Q9 and Q18.

information to the storage system for effective data placement decisions (results presented in Section 6.3.1). In addition, for workloads whose access patterns can be effectively detected by monitoring-based methods, hStorage-DB could achieve comparable performance (results presented in Section 6.3.2). (2) The hStorage-DB framework can well utilize access-pattern irrelevant semantic information items, e.g., content type and lifetime of temporary data. Such information items play important roles for storage management (results presented in Section 6.3.3). (3) Within an environment of concurrent query executions, hStorage-DB shows its strong advantages over monitoring-based methods by accurately recognizing the relative importance of different data blocks, so that cache pollution could be effectively prevented and important data is cached as long as necessary (results presented in Section 6.4).

7. OTHER RELATED WORK

There have been several different approaches to managing storage data in databases, each of which has unique merits and limits. The most classical approach is to apply a replacement mechanism to keep or evict data at different levels of the storage hierarchy. Representative replacement algorithms include LIRS [12] that is used in MySQL and other data processing systems, and ARC [15] that is used in IBM storage systems and ZFS file system. The advantage of this approach is that it is independent of workloads and underlying systems. The nature of general-purpose enables this approach to be easily deployed in a large scope of data management systems. However, the two major disadvantages are (1) this approach would not take advantage of domain knowledge even it is available; and (2) it requires a period of monitoring time before determining access patterns for making replacement decisions.

Another approach is more database storage specific. There are two recent and representative papers focusing on SSD management in this category. In [4], authors propose an SSD buffer pool admission management by monitoring data block accesses to disks and distinguishing blocks into warm regions and cold regions. A temperature-based admission decision to the SSD is made based on the monitoring results: admitting the warm region data and randomly accessed data to the SSD, and making the cold region data stay in hard disks. In [7], authors propose three admission mechanisms to SSDs after the data is evicted from memory buffer pool. The three alternative designs include (1) clean write (CW) that never writes the dirty pages to the SSD; (2) dual-write (DW) that writes dirty pages to both the SSD and hard disks; and (3) lazy-cleaning (LC) that writes dirty pages to the SSD first, and lazily copies the dirty pages to hard disks. Although specific to the database domain, this approach has several limitations that are addressed by hStorage-DB.

Compared the aforesaid prior work, hStorage-DB leverages database semantic information to make effective data placement decisions in storage systems. Different from application hints, which can be interpreted by a storage system in different ways or simply ignored [14], semantic information in hStorage-DB requests a storage system to make data

placement decisions. In particular, hStorage-DB has the following unique advantages.

First, rich semantic information in a DBMS can be effectively used for data placement decisions among diverse storage devices, as have been shown in our experiments. Existing approaches have employed various block-level detection and monitoring methods but cannot directly exploit such semantic information that is already available in a DBMS.

Second, some semantic information that plays an important role in storage management cannot be detected. Take temporary data as an example. The hStorage-DB performs effective storage management by (1) caching temporary data during its lifetime, and (2) immediately evicting temporary data out of cache at the end of its lifetime. In TAC [4], temporary data writes would directly go to the HDD, instead of the SSD cache, because such data are newly generated, and are not likely to have a "dirty" version in the cache that need to be updated. In the three alternatives from [7], only DW and LC are able to cache generated temporary data. However, in the end of lifetime, temporary data cannot be immediately evicted out of cache to free up space for other useful data.

Furthermore, some semantic information, such as access patterns, may be detected, but with a considerable cost. Within hStorage-DB, such information is utilized with no extra overhead. For small queries, the execution may be finished before its access pattern could be identified. One example is related to sequential requests. In [7], special information from SQL Server is used to prevent special sequential request from being cached in SSD cache. In contrast, hStorage-DB attempts to systematically collect critical semantic information in a large scope.

8. CONCLUSION AND FUTURE WORK

We have identified two sets of problems related to DBA-based and monitoring-based storage management approaches for database systems with heterogeneous storage devices. These problems can be addressed by filling in the semantic gap between a DBMS and its storage system. We have proposed and implemented hStorage-DB to achieve this goal. Instead of relying on the storage system to detect the best way to serve each I/O request, hStorage-DB takes a top-down approach in three steps: classification for each request, a mapping from each request type to a proper QoS policy, and invoking the right mechanism that is supported by the storage system to serve the request properly. In the framework of hStorage, QoS policies provide a high level abstraction of the services supported by a storage system. This system enables block-granularity and dynamic data placement. Our experiment results have shown the strong effectiveness of hStorage-DB. We are currently extending hStorage-DB for OLTP workloads. We will also consider semantic information from database applications.

9. ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their constructive comments. The work was supported in part by the National Science Foundation under grants of CCF-0913050 and CNS-1162165.

10. REFERENCES

[1] http://en.wikipedia.org/wiki/TRIM.

[2] http://www.tpc.org/tpch.

[3] Mustafa Canim, Bishwaranjan Bhattacharjee, George A. Mihaila, Christian A. Lang, and Kenneth A. Ross. An Object Placement Advisor for DB2 Using Solid State Storage. *PVLDB*, 2(2):1318–1329, 2009.

[4] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD Bufferpool Extensions for Database Systems. *PVLDB*, 3(2):1435–1446, 2010.

[5] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *ICS*, pages 22–32, 2011.

[6] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing. In *HPCA*, pages 266–277, 2011.

[7] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, pages 1113–1124, 2011.

[8] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. Sort versus Hash Revistied. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):934–944, 1994.

[9] Avinatan Hassidim. Caching for Multicore Machines. In *Innovations in Computer Science, 2009*.

[10] Intel. Intel®SSD 320 Series. http://ark.intel.com/products/56567/Intel-SSD-320-Series-(300GB-2_5in-SATA-3Gbs-25nm-MLC).

[11] Intel. Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[12] Song Jiang and Xiaodong Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *SIGMETRICS*, pages 31–42, 2002.

[13] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *SIGMOD*, pages 1075–1086, 2008.

[14] Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li. CLIC: CLient-Informed Caching for Storage Servers. In *FAST*, pages 297–310, 2009.

[15] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST, 2003*.

[16] Michael P. Mesnier, Feng Chen, Jason B. Akers, and Tian Luo. Differentiated Storage Services. In *SOSP*, pages 57–70, 2011.

[17] Timothy Pritchett and Mithuna Thottethodi. SieveStore: A Highly-Selective, Ensemble-level Disk Cache for Cost-Performance. In *ISCA*, pages 163–174, 2010.

[18] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (3. ed.)*. McGraw-Hill, 2003.

[19] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or Death at Block-Level. In *OSDI*, pages 379–394, 2004.

[20] Qing Yang and Jin Ren. I-CASH: Intelligently Coupled Array of SSD and HDD. In *HPCA*, pages 278–289, 2011.