An Update-Aware Storage System for Low-Locality Update-Intensive Workloads

Dilip N Simha
Computer Science Department, Stony Brook University
dnsimha@cs.sunysb.edu

Maohua Lu
IBM Almaden Research
lum@us.ibm.com

Tzi-cker Chiueh
Computer Science Department, Stony Brook University & Industrial Technology Research Institute
tcc@itri.org.tw

Abstract

Traditional storage systems provide a simple read/write interface, which is inadequate for *low-locality update-intensive* workloads because it limits the disk scheduling flexibility and results in inefficient use of buffer memory and raw disk bandwidth. This paper describes an *update-aware* disk access interface that allows applications to explicitly specify disk update requests and associate with such requests call-back functions that will be invoked when the requested disk blocks are brought into memory. Because call-back functions offer a *continuation* mechanism after retrieval of requested blocks, storage systems supporting this interface are given more flexibility in scheduling pending disk update requests. In particular, this interface enables a simple but effective technique called *Batching mOdifications with Sequential Commit* (BOSC), which greatly improves the sustained throughput of a storage system under low-locality update-intensive workloads. In addition, together with a space-efficient low-latency disk logging technique, BOSC is able to deliver the same durability guarantee as synchronous disk updates. Empirical measurements show that the random update throughput of a BOSC-based B^+ tree is more than an order of magnitude higher than that of the same B^+ tree implementation on a traditional storage system.

Categories and Subject Descriptors D.0 [*Software*]: GENERAL; E.2 [*Data*]: DATA STORAGE REPRESENTATIONS

General Terms Design, Performance

Keywords B+ Trees, Storage, Hard disks, Buffered writes, Fast logging, Update interface, low-locality, update-intensive, BOSC

1. Introduction

A disk update request retrieves a disk block, modifies it, and writes the resulting block back to disk. A low-locality update-intensive disk access workload is one that is dominated by disk update requests with a large working set and poor locality. Such workloads commonly result from index updates in data de-duplication, user-generated content management and on-line transaction processing applications, and have posed a major performance challenge for storage stack designers. Traditional storage systems do not perform well under these workloads and thus require higher storage stack layers to carefully schedule incoming disk requests to mitigate the performance penalty associated with such workloads. Even storage systems using flash memory-based disks perform poorly in the face of these workloads, sometimes faring ever worse when compared with magnetic disks. A large body of previous research efforts on disk buffering [4, 12], caching [16, 24, 28], and scheduling [32, 50] have attempted to solve this problem, but they are generally ineffective because low access locality leads to excessive random disk I/Os.

A well-known technique to boost the performance of workloads dominated by low-locality disk writes is to structure the storage system as a log and convert each disk write into an append to the log. However, such techniques carry hidden performance overheads in the form of metadata look-up and garbage collection. For workloads dominated by low-locality disk reads, no generally effective performance-boosting measure is available. Because a disk update is a disk read followed by a disk write, on the surface it appears that there is not much one can do to boost the performance of workloads dominated by low-locality disk updates. However, this paper describes an *update-aware* disk access interface and a novel storage system exploiting this new interface that together demonstrate that it is possible to apply the same logging technique to low-locality disk update workloads and produce one to two orders of magnitude in performance improvement as compared with conventional storage systems.

Traditional storage systems support a disk access interface for higher-layer systems software, such as a file system or a DBMS, to read or write data stored on disks. The granularity of disk reads and writes ranges from disk blocks [26, 33] to more sophisticated constructs such as objects [17, 29]. Regardless of access granularity, these simple read/write interfaces are not adequate for low-locality update-intensive workloads for two reasons. First, storage systems are typically designed to minimize the response time for disk read accesses and thus tend to service them as synchronously as possible. However, for a disk update, it's OK to service the update's leading disk read access asynchronously, like its following disk write access, because a disk update is considered completed only after its following disk write is completed. Unfortunately, the standard read/write disk access interface does not allow the storage system to distinguish between stand-alone disk reads and disk reads associated with disk updates. Second and more importantly, an ideal way to optimize the performance of a set of modifications to a disk block is to aggregate them and apply them to the disk block when it is brought into memory. This is very similar to the strategy employed in InnoDB [27] which is a standard storage engine used in most of the MySQL applications. But when multiple application processes issue update requests concurrently, the simple read/write interface prevents such aggregation, because the storage system does not know how to apply individual updates and has to rely on application processes to perform these updates.

To remove the above two problems, we propose a new disk access interface that allows applications of a storage system (a) to explicitly declare a disk access request as an `update` request to a disk block, in addition to the standard `read` or `write` request, and (b) to associate with an update request a callback function that performs the actual update on its target disk block. With disk update requests explicitly labeled, a storage system can now aggregate them, including the implicit reads contained within, in the same way as it does with disk write requests. With access to application-specific disk block update functions, a storage system can directly apply proper updates to each disk block retrieved on behalf of the processes issuing the disk update requests, thus gaining much more flexibility in disk access scheduling.

The update-aware disk access interface enables a new storage system architecture called *BOSC* (Batching mOdifications with Sequential Commit), which sits between storage applications, e.g., a DBMS process or file system, and hardware storage devices, and is specifically optimized for low-locality update-intensive workloads. In BOSC, incoming disk update requests targeted at a disk block are queued in the in-memory queue associated with the disk block; in the background, BOSC *sequentially* scans the disk(s) to bring in disk blocks whose associated queue is not empty. When a disk block is fetched into memory, BOSC applies all of its pending updates to it in one shot.

BOSC treats each disk update request as an atomic operation, and optimizes them as if they are disk write requests. The resulting throughput improvement under low-locality update-intensive workloads is quite impressive, between one to two orders of magnitude. In addition, combined with a space-efficient low-latency logging technique, BOSC is able to achieve this throughput improvement while delivering the same durability guarantee and latency as if each disk update request is serviced synchronously.

2. Related Work

A common approach to improving the performance of small disk writes is to use NVRAM to buffer writes, which provides two benefits: scheduling disk writes more flexibly and combining multiple writes with the same target. However, NVRAM is expensive, and for workloads with poor locality, high update rate, and large working set such as TPC-C [15], a small amount of NVRAM can only mask the delay for a finite number of disk writes, because eventually the sustained write performance is bottlenecked by the speed at which writes are propagated to disks. Write-only disk cache [40] mitigates the performance problem due to buffer flushing by injecting disk writes between consecutive disk reads. However, a single buffer page is still required to hold the result of each disk read and the read operations can still exhibit poor performance if the input workload has poor data locality. In contrast, BOSC's low-latency logging technique can accommodate a much larger number of disk writes, its use of sequential disk I/O to commit pending updates greatly improves the sustained disk update throughput, and it does not rely on NVRAM to ensure data durability.

There has been a long line of research on efficient file system metadata update techniques that ensure metadata consistency with minimal performance overhead. HyLog [49] further reduces the performance overhead associated with LFS's cleaning [45] by treating hot and cold pages separately. The soft update technique [18, 34] avoids synchronous metadata writes by exploiting dependencies among metadata updates and makes it possible to aggregate updates as much as possible to improve the disk I/O efficiency. One problem with soft updates is that it is metadata-specific and thus needs to be tailored to each type of file system. Also, the above metadata update techniques focused mainly on the latency but not the throughput of metadata updates.

Efficient file system metadata update techniques that ensure metadata consistency with minimal performance overhead have received significant attention in the last two decades. *WAL (Write-Ahead Logging)* [14, 22] and shadow paging [9, 10, 45, 47, 49] group relate metadata updates and commit them atomically to ensure metadata consistency. Performance benefits of *WAL* mainly come from sequential disk writes and group commit.

Much work [8, 21, 25, 35, 36, 43] has been done to optimize the disk I/O performance for inserting and querying index data structures. One particularly interesting line of research in this area is the cache-oblivious data structures and algorithms [5–8]. Take a binary tree B of height H for example. This tree is abstracted into a 2-level abstract tree AB, whose root corresponds to the first $\frac{H}{2}$ levels of B, and each of whose leaf nodes corresponds to a $\frac{H}{2}$-level subtree of B. Each node in AB is then recursively abstracted in the same way until the size of each final abstract tree node is smaller than a pre-defined threshold T. This linearization strategy for tree data structures, known as the van Emde Boas scheme, substantially reduces the number of disk accesses required in the tree look-up process if T is smaller than the cache line (page) size. The performance improvement of cache-oblivious data structures mainly comes from the fact that they put portions of a tree that are likely to be accessed together during the look-up process in the same units which are transferred in the memory hierarchy. With this set-up, when a transfer unit is brought in, it is expected to service multiple accesses to the unit before it is evicted.

There have been several research efforts on the bulk update problem, which attempts to speed up index updates in the presence of a continuous stream of inputs to a database, which require real-time updates to its indexes. Arge et al. [1, 2] proposed a bulk update mechanism for dynamic R-trees, whereas Procopiuc et al. [42] described a scalable bulk update algorithm for kd-trees. The basic idea behind these schemes is to hold the inserted input records in the internal nodes as long as possible and copy them sequentially to grow the tree when the internal nodes are filled up. In [19], Graefe proposed to add an artificial leading column to logically partition a single B^+ tree to several small B^+ trees. Similarly in the *buffer tree technique* [8], incoming updates to a B^+ tree are written to the smallest B^+ tree that can fit into the main memory. Merging is implemented as a background operation to take advantage of large sequential writes. However, read query performance again is sacrificed because multiple B^+ trees have to be queried before the final result can be computed. In [20], Graefe proposed a novel technique to improve the de-fragmentation and reorganization performance of B^+ tree. A logical pointer called *fence* instead of a physical pointer to sibling B^+ tree leaf nodes was proposed to limit the performance overhead of migrating B^+ tree leaf nodes. However, this scheme optimizes the performance of insert operations but not update-in-place operations because the latter needs to fetch target leaf nodes before modifying them.

BOSC is different from these database index optimization schemes in three ways. First, BOSC is application-independent and requires only minor modifications to the database indexes built on top of it. Second, BOSC speeds up the disk access performance through request batching and sequential commit, without requiring any additional data structure copying. Third, BOSC can handle arbitrary index modifications, i.e., insert, delete and in-place update, but most bulk update schemes are optimized for streaming inserts.

3. Update-Aware Disk Access Interface

The conventional disk access interface provides
`read(target_block_addr, dest_buf_addr)`

write(target_block_addr, src_buf_addr) for applications (including file system and DBMS) to read and write disk blocks, respectively. Under this interface, existing storage systems optimize disk write accesses by delaying and/or scheduling them to maximize their throughput, and optimizes disk read accesses by servicing them as soon as possible to minimize their latency.

A disk update involves a disk read of the target disk block and then a disk write of the same block after the block is brought into memory and modified. If a storage system could treat each disk update as an atomic operation, theoretically it can delay and schedule the disk reads associated with disk updates in the same way as it does with disk writes. However, to atomically service a disk update request, the storage system must be able to perform the request's intended update operation *without involving the application process issuing the disk update request.* To allow an application running on a storage system to explicitly declare a disk access request as a disk update request and supply the necessary information for the storage system to service it atomically, we propose an *update-aware* disk access primitive specific for disk updates, modify(target_block_addr, ptr_modification, ptr_commit_function), which specifies the target disk block to be modified, a pointer to an application-specific data structure that includes all the information required by the requested modification, and a pointer to an application-specific call-back function that commits the actual modification to disk. This primitive is sufficiently general to accommodate disk update requests from such common storage applications as database index managers, including creating a new index entry, updating an existing index entry, and deleting an existing index entry.

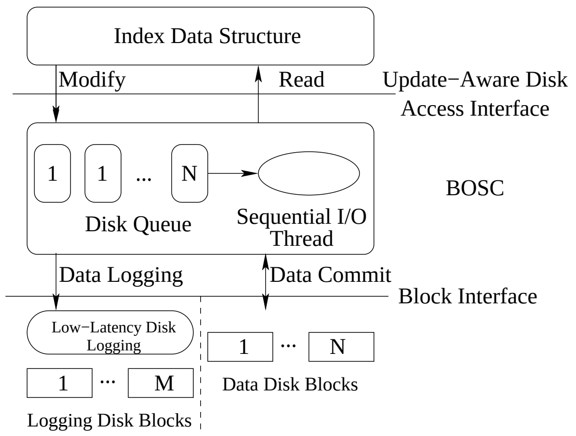


Figure 1. *BOSC associates with each disk block an in-memory update request queue. When BOSC receives a disk update request, it logs the request to disk, queues the request in the target block's associated update request queue, and performs the update operation only when the target disk block is brought into memory. BOSC brings disk blocks into memory sequentially according to their logical block addresses.*

In the proposed disk update primitive, the application-specific part of each disk update request, i.e., the internal organization of the data structure pointed to by ptr_modification and the internal logic of the function pointed by ptr_commit_function, is fully encapsulated. A storage system implementing the proposed interface carries out the requested modification of each disk update request by blindly invoking the specified function on the specified data structure, without requiring any knowledge about the data structure and function. Even the developers of BOSC's modify API do not need to know anything about BOSC's internals except following the guideline below when writing a call-back function: ”It does not contain any calls to the modify API”. In fact, such a storage system does not even need to differentiate among *create*, *update*, or *delete* operations. As a result, the update-aware disk access interface provides the underlying storage system the same flexibility of scheduling disk block create, disk block update, and disk block delete requests as disk block write requests, and thus the corresponding performance boost.

3.1 Caveats with Call-back Function

In the general case,running the call back functions in BOSC entails some security risks like a buggy or malicious program that mismanages system resources. However, these risks are reasonably low for our two target use cases.

1. When BOSC is used as a user-level library that allows a user-level application (e.g. DBMS) to directly manage a disk or disk partition, the background thread that invokes the call-back function runs at the user level and thus can at most compromise the user-level application itself.
2. When BOSC runs as a kernel module, only trusted software components such as file systems are allowed to use the BOSC interface, and the call-back functions they registered with BOSC are all kernel code. So, practically speaking, the fact that the background thread invokes these call-back functions in the kernel context does not really introduce any additional safety risks.

If an application calling an update function needs to receive the function's return code in order to proceed, it is NOT appropriate to implement such an update function on top of BOSC, because this use pattern is similar to a read function.In our experiences, there are many applications like dedupe and CDP that do not need the called update function to return results.

All the inputs that a call-back function in BOSC needs to run are either logged and put in the per-block queue or available on the corresponding fetched disk block. At recovery time, the per-block queues at the time of crash are reconstructed, and therefore, the call-back function for a given fetched disk block must be able to run after recovery. Here the assumption is that the registered function pointer remains valid across a system crash, which is the case because function pointers are virtual addresses and we assume application binaries are not modified and call-backed functions are all statically linked.

4. Batching Modifications with Sequential Commit

Figure 1 shows how BOSC leverages the update-aware disk access interface to aggregate disk update requests and reduce the overall physical disk I/O overhead. It applies a space-efficient low-latency disk logging technique to log each incoming disk update request, queues it in an in-memory per-block request queue, commits updates to disk asynchronously using mostly sequential disk I/O, and recovers from failures efficiently. The following subsections describe its design in more detail.

4.1 Low-Latency Disk Array Logging

Logging a disk update request to disk synchronously and then performing whatever operations triggered by the update asynchronously is a well-known technique. BOSC adopts this technique to service disk updates asynchronously and achieve high sustained disk update throughput, while delivering the same durability guarantee as synchronous disk updates.

Because the end-to-end throughput of BOSC is bounded by its synchronous disk logging performance, BOSC extends a low-latency disk logging technique called Trail [13] to be low-latency, space-efficient, and able to work on commodity disk arrays [31]. The key idea in this low-latency disk logging technique is to write an incoming disk block to where the disk head happens to be, which requires real-time knowledge of the disk head position of each log disk. To do so, BOSC statically extracts the physical disk geometry information from every log disk, and constantly keeps track of each log disk's disk head position at run time. Upon receiving a disk update request, BOSC prepares its log record, which includes the following information:

- A copy of the data structure pointed to by `ptr_modification`,
- A global sequence number for the current disk update request,
- A *back pointer* to the disk location of the log record that is temporally immediate before this record,
- A *global frontier*, which corresponds to the global sequence number of the *youngest* disk update request before which all disk update requests have been committed to disk, and
- A *local frontier*, which corresponds to the global sequence number of the *youngest* disk update request before which all disk update requests to the same target disk block of the current disk update request have been committed to disk.

To service physical disk write requests, BOSC maintains a separate disk request queue for each disk in the log disk array. At any point in time, one of the log disks serves as the *active* disk. In the beginning, BOSC randomly chooses one of the log disks as the active disk. Once a log disk becomes the active disk, it remains as the active disk until the waiting time of the oldest pending request in its disk request queue exceeds a threshold, T_{wait}. Whenever a new disk write request arrives, BOSC inserts the request to the active disk's queue as long as the waiting time of its oldest pending request is smaller than T_{wait} and there is enough free space in the current track to accommodate the new request; otherwise BOSC dispatches the request batch currently in the active disk's queue, chooses another log disk if available as the active disk and inserts the new request to the new active disk's request queue.

To select a new active disk for an incoming disk write request, BOSC considers the degree of possible batching and the write latency. When estimating a write request's write time on a log disk, BOSC takes into account the current position of the log disk's head and the possibility of batching the new request with others already in the disk's queue. For those log disks that are currently idle, BOSC only needs to consider the delay due to batching. A key design decision in BOSC is to dispatch a new write request to a log disk that allows as many disk write requests to be batched into one physical disk write operation as possible, rather than to one with the earliest write time for that request. However, BOSC uses T_{wait} to limit the size of batching and to ensure that the experienced latency of each incoming disk write request is always bounded.

Because of disk request batching, multiple log records could be merged into a physical disk write request when they are written to disk. Also, the actual disk location of each log record is only known at the last moment, i.e., right before they are written to disk, and BOSC keeps track of this information accurately to chain related log records together through their back pointers.

To track the log disks' disk head position, BOSC statically extracts the physical disk geometry information from every log disk, and constantly keeps track of each log disk's disk head position at run time. More concretely, after a physical disk write is completed, BOSC records the *LBA (Logical Block Address)* of its last sector, LBA_0, and its completion timestamp T_0. Assuming the disk head stays in the same track, when the next write arrives at T_1, BOSC estimates the disk head's current position $CurrentLBA$ using the following formula:

$$CurrentLBA = SPT \cdot \frac{(T_1 - T_0)\ mod\ RoTime}{RoTime} + LBA_0 \qquad (1)$$

where SPT is the number of sectors in the current track, $RoTime$ is the disk's full rotation time. The final predicted position, $DestinationLBA$, is $CurrentLBA + Lookahead$, where $Lookahead$ is an empirical value chosen to account for such delays as the controller delay and avoid a full rotation delay due to tracking errors.

4.2 Sequential Commit of Aggregated Updates

In addition to *log disks*, BOSC maintains a set of *data disks* to store application data and a set of in-memory disk update request queues, one for each data disk block. Upon receiving a disk update request, BOSC first checks if the target disk block is memory-resident; if it is, BOSC performs the update against the block immediately, otherwise BOSC appends the update request to the per-block disk update request queue associated with the request's target disk block and logs the update request to the log disk array; finally, BOSC returns control to the caller. Because of BOSC's low-latency disk logging, the perceived delay of each disk update request is relatively small, typically smaller than 1 msec. In the background a separate thread of BOSC constantly reads the data disks sequentially to fetch into memory those disk blocks whose pending request queue is non-empty. This thread goes from the beginning to the end of the data disks, and repeats the cycle. This is referred to as the *sequential commit cycle*. Every time the background BOSC thread brings in a disk block, it applies all the pending updates to the disk block and writes the block back to the disk. To further minimize the disk access overhead in this read-modify-write loop of commit processing, instead of processing one disk block at a time, BOSC physically reads and writes a continuous run of disk blocks, and commits pending updates on a run by run basis.

A disk block run corresponds to a contiguous sequence of disk blocks the percentage of which having a non-empty pending request queue is above a threshold (currently set to 0.5) and whose length is no greater than another threshold (currently set to 32 blocks). The first and last disk block in a run must have a non-empty pending request queue, and runs are disjoint. As one run is being fetched from a data disk, the background thread applies pending updates to another run that has previously been brought into memory, and pushes a third run, whose pending updates have already been applied, to another data disk. By pipelining the processing of runs, BOSC is able to eliminate unnecessary disk seek delays and reduce the number of disk rotations required to commit pending updates to a sequence of disk blocks to 2.

4.3 Recovery Processing

After a system crash, BOSC scans the log disks to discover uncommitted disk update requests, reconstructs the in-memory per-disk-block update request queues that exist immediately before the crash, and resumes its normal processing. This is similar to the idea employed in Aries [37] with some optimizations as described below. Note that BOSC chooses *not* to commit all uncommitted disk update requests to disk at recovery time. Instead, it merely aims to reconstruct the in-memory per-block update request queues and relies on BOSC's normal sequential commit mechanism to write them to disk. More concretely, BOSC's recovery procedure consists of the following steps:

1. Searching the log for the log record with the largest global sequence number,

2. Determining the *replay window* in the log that contains log records required for the reconstruction of per-disk-block update request queues, and
3. Parsing the log records in the replay window to reconstruct the per-block request queues.

To speed up Step (1), BOSC performs a binary search (rather than a sequential scan) of the tracks of the log disk array to track down the youngest log record, which is the last one to be inserted before the crash and thus corresponds to the *end* of the replay window. The binary search works because the disk logging proceeds one track by one track in a FIFO fashion.

Finding the *beginning* of the replay window is trivial because it actually corresponds to the *global frontier* field of the youngest log record, because all update requests associated with log records before the global frontier by definition have already been committed to disk. In Step (3), the log records in the replay window are traversed backwards from the end to the beginning. Given a log record, BOSC first assigns the value in its *local frontier* field to its target block's *local frontier* if the request queue associated with its target block is empty; then BOSC inserts the update request in the log record into its target block's request queue in the global sequence number order if the log record's global sequence number is larger than the target block's *local frontier*. By exploiting the global and local frontier information in log records, BOSC can avoid inserting a significant percentage of the log records in the replay window into per-block request queues.

4.4 Extensions

A straightforward way for a BOSC application like a database index manager to query if a record of a certain qualification exists in a disk block is to explicitly read in the block and scan it for records with the target qualification. However, if the desired record already exists in the disk block's pending update request queue, this approach may bring the target block into memory unnecessarily. To eliminate this unnecessary disk I/O, BOSC provides a query API that allows an application to query a specific disk block: `query(target_block_addr, ptr_query, ptr_query_function)`, where `target_block_addr` is the target disk block's ID, `ptr_query` is a pointer to a data structure containing the query's parameters, and `ptr_query_function` is a pointer to an application-specific call-back function that BOSC invokes to search the target disk block's memory-resident update requests queue and/or the target disk block itself if BOSC needs to fetch it into memory. When a disk block is read into memory because it is a target of a read request, BOSC applies all the block's pending updates to it before returning the block to the application issuing the read request. This API allows BOSC to double each in-memory per-block request queue as a cache for the associated disk block.

BOSC treats every disk update request it receives from an application as an independent I/O transaction, and is able to guarantee their durability across system failures by synchronous logging and recovery. When a system recovers from a crash, BOSC's recovery manager first restores the side effects of all the disk update requests that BOSC considers already committed, and then invokes the application's recovery logic. However, BOSC's I/O transaction is not equivalent to an application-level transaction. If a disk update request U is contained in an application-level transaction that the application's recovery manager thinks should be aborted, it should explicitly issue a compensating update request to undo U. Further, when a transaction calling a BOSC update function is committed after the called BOSC update function is ACKed, it is OK for the transaction to release all the locks and consider everything is done, even though the actual updates underlying the BOSC update function happen much later. The reason is that BOSC's recovery mechanism guarantees that all ACKed updates eventually happen.

The current BOSC design is mainly for directly attached disks. To generalize the BOSC idea to a network storage server requires leveraging the security mechanisms developed in the Active Disk [44] project.

5. BOSC-Based B^+ Tree

We have successfully ported B^+ tree index implementation from TPIE [3, 48] to the BOSC storage system prototype. TPIE is a software environment written in C++ that is designed specifically to minimize the disk I/O cost in the face of very large data sets.

The BOSC-based B^+ tree assumes all internal tree nodes and a small subset of leaf nodes are memory-resident. To service a modification query that inserts, deletes, or updates an index record, the BOSC-based B^+ tree first traverses the internal nodes to identify the leaf node containing the target index record, then constructs a disk update request record, and finally calls BOSC's disk update API using the target leaf node's disk block address, the associated update request record and the corresponding commit function as input arguments. Upon receiving such a disk update request, BOSC logs the request to the log disks first, commits the update to the target leaf node immediately if it is currently cached in memory, and queues the update request record in the corresponding in-memory request queue associated with the target leaf node otherwise.

To ensure atomicity, the BOSC-based B^+ tree acquires a lock on a leaf node before modifying it, and releases the lock after BOSC logs the associated disk update request and queues it in the associated request queue. It is safe to release the lock associated with the target leaf node of a modification query before physically committing the requested modification to disk, because BOSC *guarantees* the effects of a modification query's associated disk update request be visible to all subsequent queries that access the same leaf node, even in the presence of power failures.

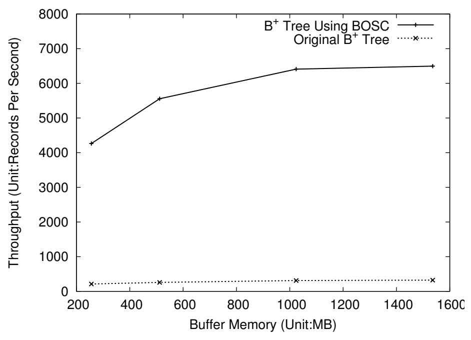
An implicit assumption underlying the design of BOSC is that each disk update request modifies only its target disk block. However, this assumption does not always hold for B^+ tree, because a modification to a tree node, e.g., an insertion of a new index record, may trigger a restructuring of the tree and thus modifications to other tree nodes. If a disk update request that triggers additional disk updates is not processed immediately at the time when it is queued but deferred until the time when it is committed to disk, a disk block's in-memory request queue may grow unbounded, because the triggered restructuring may be recursive. This makes the update commit processing time of a disk block less predictable, and increases the response time of read query requests because servicing read query requests requires scanning of per-block update request queues.

To mitigate the performance overhead due to disk update requests that trigger additional disk updates, the BOSC-based B^+ tree maintains a count for the *effective* number of index records in each leaf node, including the pending delete and insert operations, and *proactively* triggers the split of a leaf or internal node when the effective number of records in a tree node exceeds a threshold, say 70%. If the leaf node to be split does not have any index records on disk, all the node's index records are in the associated update request queue and the BOSC-based B^+ tree performs the split without incurring any disk accesses. If the leaf node to be split has some index records on disk, the BOSC-based B^+ tree defers the split operation until the time when these records are brought into memory by the background BOSC thread.

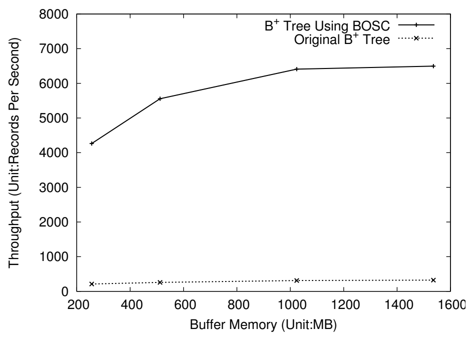
6. Performance Evaluation

6.1 Evaluation Methodology

We have built a complete Linux-based BOSC prototype. This prototype supports the update-aware disk access interface as well as



(a): BOSC vs. Vanilla for Random insert workload



(b): BOSC vs. Vanilla for Random update workload

Figure 2. *Comparison between the record insertion throughput of a BOSC-based B^+ tree implementation and a vanilla B^+ tree implementation based on the conventional disk read/write interface under the random insert workload (a) and random update workload (b) when the total amount of buffer memory is varied from 256 MB to 1.5 GB. The leaf block size is 64 KB, the record size is 64 B, and the initial index size is 16 GB.*

sequential commit of aggregated disk updates. On top of this BOSC prototype, we built a BOSC-based B^+ tree implementation, which is derived from TPIE and took about 2 man weeks.

To evaluate the efficiency of the BOSC-based B^+ tree, we used the following four synthetic workloads: (1) *sequential insert* workload, (2) *random insert* workload, (3) *clustered insert* workload, and (4) *random update* workload. In the sequential insert workload, records with sequentially increasing key values between 0 and 2^{60} are inserted into an initialized index. In the random insert workload, records with randomly generated key values, which fall between 0 and the pre-defined index size, are inserted into an initialized index. The clustered insert workload consists of a group of fixed-sized (32 by default) clusters of record insertions. Within each cluster, records with sequentially increasing key values are inserted. However, the key of the start record for each cluster is randomly generated. The random update workload updates random existing records that are inserted by the random insert workload.

The evaluation testbed for the BOSC prototype is a Dell PowerEdge 600SC machine with an Intel 2.4GHz CPU, 512KB L2 cache, 4GB memory, a 400MHz front-side bus, two Gigabit Ethernet interfaces and five 7200-RPM IBM Deskstar DTLA-307030 disks, four of which store the B^+ tree index records and one of which is dedicated to low-latency logging. We have done a separate study on the ratio of data disks to log disks, and concluded that "4 data disks and 1 logging disk" is the most performant configuration for a 5-disk storage system. Due to space constraints, the details of that study are omitted.

To be realistic, the initial B^+ tree must contain a substantial number of index records, on the order of tens of gigabytes of data. If we were to measure the throughput of the BOSC-based B^+ tree implementation against an initially empty B+ tree, then the measurement results for initial inserts/updates would be biased as they don't include such critical components as lock acquisition and tree traversal. However, it takes several hours to generate a properly initialized multi-gigabyte B^+ tree, and we need dozens of initialized B^+ trees, each with a different node or record size, in the entire evaluation study. So a fast B^+ tree initialization method is needed. The major bottleneck in the B^+ tree initialization process is the disk I/Os required to put leaf node data on disk. Because the actual contents of the leaf nodes are immaterial to our evaluation study, we could completely skip these disk I/Os in the initialization process and focus only on the creation of internal tree nodes. Therefore, when a B^+ tree is initialized this way, only its internal nodes are properly set up and its leaf nodes are only allocated on disk but not actually initialized. During the experiment run, whenever a leaf node is brought into memory for the first time, its content is filled with proper values at that point. The values filled are calculated on the fly, because the structure of the initialized B^+ tree and the key values in it are pre-determined. This B^+ tree initialization method proves invaluable to our evaluation study, because it saves us hundreds of hours, e.g., the time to initialize a 64-Gbyte B^+ tree is reduced from 36 hours to under 50 seconds.

6.2 Overall Performance Improvement

Figure 2 shows the throughputs of a vanilla B^+ tree implementation on a conventional disk read/write interface (with 5 data disks) and a BOSC-based B^+ tree implementation under the random insert and random update workload. The throughput of the vanilla B^+ tree implementation increases only slightly with the buffer memory because the poor locality in the random insert workload does not offer much room for leaf node caching to be effective. In contrast, the throughput of the B^+ tree implementation keeps improving with the increase in buffer memory size because more pending insertion requests can be accumulated in each sequential commit cycle. This improvement saturates at 1024 MB because the given buffer memory exceeds the product of the new record insertion rate and the sequential commit cycle length. When the buffer memory size is 1024 MB, the sustained throughput of the BOSC-based B^+ tree implementation under the random insert workload reaches around 6410 requests/second, which is *20 times* higher than that of the vanilla B^+ tree implementation using the conventional disk read/write interface (311 records/second). When buffer memory is not the performance bottleneck, the throughput of the BOSC-based B^+ tree implementation is mainly bound by the physical disk I/O efficiency in the sequential commit process.

The performance of the BOSC-based B^+ tree implementation under the random update workload is almost the same as that under the random insert workload, because in both workloads the accesses to the index pages are random and consequently their performance is bottlenecked by disk I/O. Figure 2(b) shows that the throughput improvement of the BOSC-based B^+ tree implementation over the vanilla B^+ tree implementation is the same as in Figure 2(a). These two results conclusively demonstrates BOSC is as efficient for an *update-in-place* workload as for an *insert-only* workload. In contrast, most previous B^+ tree optimizations [1, 6, 21] are only applicable to insert-only workloads.

The two key performance-boosting features of BOSC are low-latency logging and asynchronous sequential commit using multiple request queues. A simpler alternative to BOSC's low-latency logging is logging by appending to the end of a file. A simpler alternative to asynchronous sequential commit is to queue all update requests in a single queue and batch-commit the head N requests in the queue according to their target disk block addresses. To evaluate the performance contribution of each of these two features, we compare the throughputs of the following four B^+ tree variants. The first variant, called *One-Queue-Append*, appends each incoming update request to the end of a log file and inserts it into a single FIFO queue. The second variant, called *One-Queue-Trail*, uses low-latency logging to log each incoming update request and inserts it into a single FIFO queue. The third variant, called *Multi-Queue-Append*, appends each incoming update request to the end of a log file and inserts it into the per-block queue associated with its target block. The fourth variant is BOSC, which uses low-latency logging to log each incoming update request and inserts it into the per-block queue associated with its target block.

To demonstrate the performance benefits of BOSC under more realistic workloads, we collected a trace of access requests to the index engine of the MySQL DBMS under the TPC-C workload [39], where the number of warehouses is set to 20, 40, 60 and 80. Each trace entry includes the type (e.g. read, update, delete and insert) and the key/data information of each request issued to the index engine. For each warehouse number, we ran the TPC-C workload for three hours to generate an index of the size 16 GB, 32 GB, 48 GB, and 64 GB, respectively, and collected the corresponding access request trace. For each index access trace collected, we replayed the first half to create an initial image of the database index, and then replayed the second half and measured the throughput of the input requests in the second half of the trace.

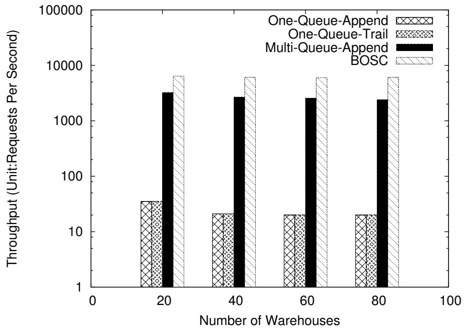


Figure 3. *Throughput comparison among the BOSC-based B^+ tree implementation, the B^+ tree implementation with multiple request queues and append-only logging, the B^+ tree implementation with one request queue and append-only logging, and the B^+ tree implementation with one request queue and low-latency logging under the four index access traces collected by running the TPC-C workload with different warehouse numbers against MySQL. The Y axis is in log scale. The leaf node size is 64 KB and the buffer memory is 1 GB.*

Figure 3 compares the throughputs of these four B^+ tree implementation variants under four different TPC-C traces. Across all warehouse parameters, as expected the BOSC-based B^+ tree implementation tops the four variants with the best throughput. For example, when the warehouse number is 80, the throughput of the BOSC-based B^+ tree is 6058 requests/second, as compared to 20 requests/second for the *One-Queue-Trail* scheme, and 2386 requests/second for the *Multi-Queue-Append* scheme. The performance gain of BOSC over the *Multi-Queue-Append* scheme comes from low-latency logging, which maximizes logging efficiency and thus the overall update throughput. The fact that the BOSC-based B^+ tree implementation is more than 2.5 times faster than the *Multi-Queue-Append* variant (the Y axis is in log scale) shows the importance of lower logging latency. The BOSC-based B^+ tree implementation is more than 300 times faster than the *One-Queue-Trail* variant, which shows the importance of sequential commit as enabled by multiple request queues is much more than low-latency logging. There is no noticeable performance difference between the *One-Queue-Trail* variant and the *One-Queue-Append* variant because both are bottlenecked by the excessive disk access overhead associated with committing pending updates to disk.

Larger warehouse number corresponds to larger database index size and lower access locality. The fact that the throughput of the BOSC-based B^+ tree implementation remains largely independent of the warehouse number suggests that BOSC enables a B^+ tree implementation to exhibit good throughput without relying on the input workload's locality characteristics. Overall, under the TPC-C workload, the BOSC-based B^+ tree implementation is 300 times faster than that of the vanilla B^+ tree implementation when there are 80 warehouses, and is 180 times faster when there are 20 warehouses.

6.3 Sensitivity Study

In this section, we evaluate the impacts of the leaf node size, the index record size, the total index size and the buffer memory size on the performance of the BOSC-based B^+ tree implementation. Each experiment run starts with a fixed-sized initial B^+ tree and continues with index record insertions/updates until the first *sequential commit cycle* is completed. At that point, we measured the total number of insertions/updates and the elapsed time.

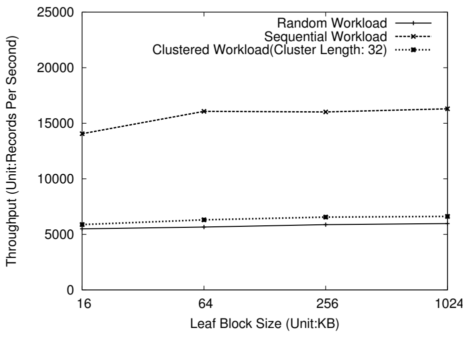


Figure 4. *Record insertion throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, and random insertion workload when the leaf node size is varied from 16 KB to 1024 KB. The X axis is log-scale. The Y axis is the number of new records inserted per second. The memory allocated for all per-block request queues is 1 GB, the record size is 64 B, and the initial index size is 64GB.*

In evaluating the impact of different parameters on the insert/update rate, there are 3 factors to consider: (1) the disk I/O efficiency, which reflects how effectively the background commit thread removes unnecessary disk access overhead, (2) the degree of batching, which determines how many requests over which each disk I/O operation's cost is amortized, and (3) the CPU overhead associated with traversing from the B^+ tree's root to the target leaf node of a given insert/update request, and queuing pending requests.

The throughputs of the BOSC-based B^+ tree under the random insertion, sequential insertion and clustered insertion workload when the leaf node size varies are shown in Figure 4. The throughput performance of the BOSC-based B^+ tree index under the sequential insert workload is much higher than that under the random insert workload for two reasons. First, the average number of pending requests in each queue at the time of commit is higher under the sequential insert workload than that under the random insert workload. Second, the CPU overhead of processing insert/update requests is lower under the sequential insert workload than that under the random insert workload because of fewer L2 cache misses. For the random insert workload, it takes around 140

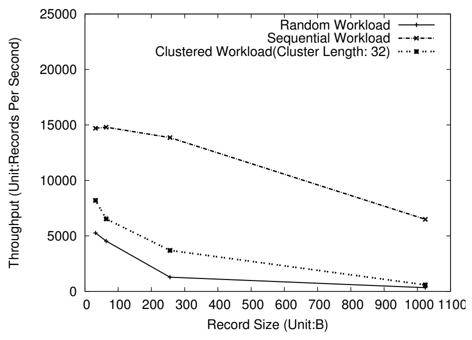


Figure 5. *Record insertion throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, and random insertion workload when the index record size is varied from 64 B to 1024 B.*

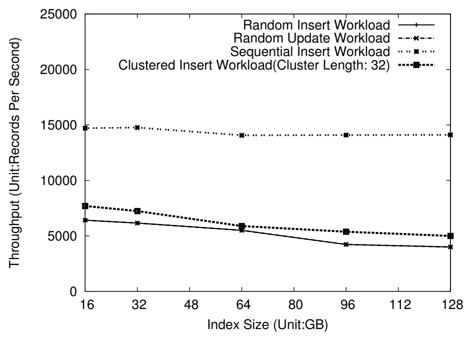


Figure 7. *Record insertion/update throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, random insertion and random update workload when the initial index size is varied from 16 GB to 128 GB.*

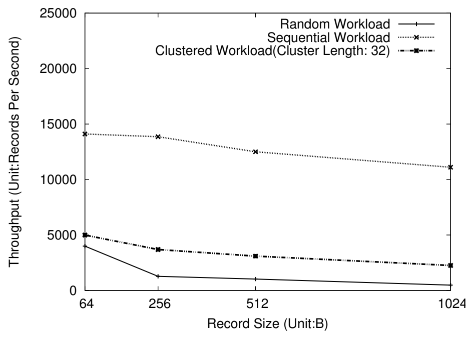


Figure 6. *Record insertion throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, and random insertion workload when the index record size increases from 64 B to 1 KB, and the leaf node size also varies proportionally so that the ratio between the two is* fixed.

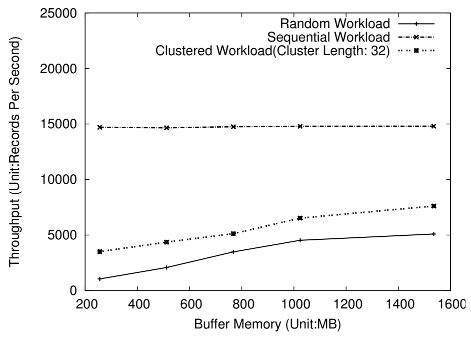


Figure 8. *Record insertion throughput of a BOSC-based B^+ tree implementation under the sequential insertion, clustered insertion, and random insertion workload when the BOSC's buffer memory is varied from 512 MB to 1536 MB.*

micro-seconds to complete an insertion request, whereas it takes only 67 micro-seconds for the sequential insert workload.

As the size of the test B^+ tree index's leaf block is increased, more index records can be packed into each leaf block, the degree of batching in terms of number of pending requests per disk block fetched is increased and so is the throughput of the BOSC-based B^+ tree index, as shown in Figure 4. This effect is more pronounced under the clustered and sequential insert workload than under the random insert workload, because there is not much batching in the random insert workload anyway.

Figure 5 shows that, as the size of the test B^+ tree's index record is increased, fewer index records can fit within each leaf block, and the degree of batching in terms of number of pending requests per disk block is decreased. The throughput degradation for the clustered insert and random insert workload is directly correlated with the decrease in the degree of batching, but that for the sequential insert workload is mainly due to additional L2 cache misses during insert request processing.

If both leaf block size and index record size are increased while keeping their ratio constant, the number of index records per leaf block remains the same, but the degree of batching in terms of number of pending requests per fixed-sized disk I/O is still decreased, e.g., the effective number of pending requests committed per 100-KB disk I/O decreases as the leaf block size is increased from 8KB to 64KB, and so is the throughput of the BOSC-based B^+ tree index, as shown in Figure 6.

As the total B^+ tree index size is increased, the average number of pending requests accumulated in each per-block queue within one sequential commit cycle becomes smaller, the degree of batching at the time of commit is thus decreased, and so is the throughput of the BOSC-based B^+ tree index, as shown in Figure 7. The throughput impact of the index size is less obvious under the sequential insert workload because the degree of batching remains largely constant regardless of the index size. Figure 7 also shows that the performance of the BOSC-based B^+ tree implementation under the random update workload is almost the same as that under the random insert workload, because in both cases accesses to the index pages are random and consequently their performance is bottlenecked by disk I/O.

As the buffer memory for per-block request queues is increased, the number of pending requests at the time of commit is increased, the degree of batching is increased, and the overall throughput under the random insert and clustered insert workload are increased, as shown in Figure 8. The performance impact of buffer memory size is minimal for the sequential insert workload because its degree of batching is already quite high and largely unaffected by the buffer memory size.

6.4 Read Query Latency

Although BOSC is designed to optimize the throughput of low-locality update-intensive workloads, it does *not* degrade the latency of read accesses to database indexes built on top it. This is unusual, because many previously proposed B^+ tree implementations opti-

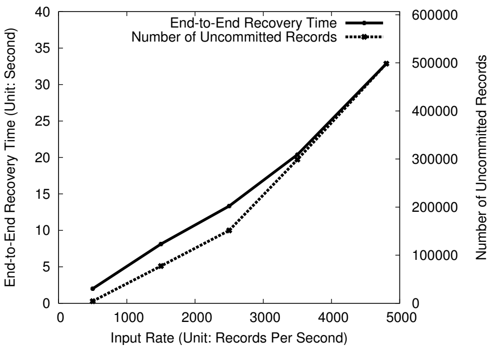


Figure 9. *The total recovery time for a 64-GB B^+ index and the number of uncommitted pending update requests in the replay window as the input update request rate is varied before the crash.*

mized for the same workload tend to trade better update throughput for longer read latency.

Index Structure	Point Query (Unit: msec)		Range Query (Unit: msec)	
	With BOSC	Without BOSC	With BOSC	Without BOSC
B^+ Tree	10.20	10.19	15.75	15.76

Table 1. *The average latency of Point and Range queries for the B^+ tree implementations with and without BOSC. The leaf block size for all index structures is 4 KB, and the buffer memory is 256 MB.*

Table 1 shows the average latency of 100 Point and Range queries against a pre-populated B^+ tree using the B^+ tree implementations with and without BOSC. For *point* queries, the key values are generated randomly from the underlying key space. For *range* queries, the starting key values are generated randomly from the key space and the maximum range size is fixed at 1,000. There is no statistically significant difference between the average read query latency of the BOSC-based B^+ tree implementation and that of the vanilla B^+ tree implementation, even though the read-path processing in BOSC requires an additional step of searching the target block's in-memory request queue. BOSC's caching has little effect on it's read performance because the locality in the input workload is relatively low, as evidenced by the relatively large average latency. This result demonstrates that the update/insert performance gain of BOSC does not come at the expense of read performance degradation, which is often the case for other B^+ tree optimizations [1, 6, 21].

6.5 Logging and Recovery Performance

BOSC relies on low-latency logging to provide the same durability guarantee as synchronous disk updates. The average latency of logging a 4-Kbyte block to an IDE disk array is under 0.5 msec, about an order of magnitude smaller than conventional disk logging implementations and the fastest ever reported in the literature. In addition, through aggressive disk request batching, BOSC is able to log more than 50000 per-insertion-request log records per second, or about 20 μs per log record. Finally, even with such high logging efficiency, BOSC is able to keep the log disks' space utilization above 70%.

There are two major steps in BOSC's recovery procedure: (1) identifying the youngest log record and (2) reconstructing the in-memory per-block request queues by analyzing the log records between the youngest log record and its associated global frontier. Because Step (1) uses a binary search through the logging disk array, it typically takes between 0.8 to 0.9 seconds to complete.

The time required by Step (2) depends on the number of uncommitted pending updates, which in turn depends on the input request rate. To evaluate how the total recovery time scales with the input rate, we ran a random update workload with varying input request rates to update records in a 64-GB B^+ tree with the following configuration: 256-MB buffer memory, and 16-byte index record. In each run, we issued about 64 million update requests, shut down the B^+ tree machine, restarted it and measured its recovery time.

Figure 9 shows that the total recovery time of a BOSC-based B^+ tree implementation indeed increases with the input request rate, because higher input request rate populates the per-block request queues faster and accumulates more uncommitted pending updates in the request queues when the system is shut down. These pending updates need to be scanned and reconstructed in Step (2) of the recovery process. As expected, increase in the total recovery time is roughly linearly proportional to increase in the number of uncommitted pending updates, as shown in the right Y axis of Figure 9.

6.6 Applications of BOSC

We have built two applications on top of BOSC: an index manager for a continuous data protection (CDP) system called *Mariner* and a garbage collector for a data de-duplication engine. This subsection shows BOSC's effectiveness in improving their performance.

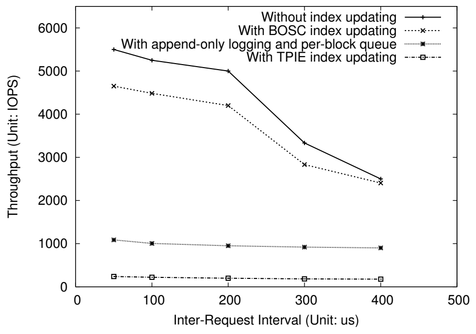


Figure 10. *The block-level logging throughputs of four* Mariner *variants: The first variant does not update any index; the second variant updates the map index using a BOSC-based B^+ tree; the third variant updates the map index using a B^+ tree that uses append-only logging and multiple per-block queues; The fourth variant updates the map index using a vanilla B^+ tree.*

6.6.1 Index Update for Continuous Data Protection

Mariner [31] is a Continuous Data Protection (CDP) system that logs every block-level disk write to a protected storage server, and creates a new version for a logical disk block whenever it is over-written. *Mariner* maintains a map between each logical block number and its physical block numbers, each corresponding to a distinct version. Upon receiving a block-level disk write request, *Mariner* logs the write request's payload to create a new version and inserts a new entry into the map.

To measure the overall disk write logging throughput of *Mariner*, we ran a kernel thread inside *Mariner* that constantly generates new 4KB block versions, with their logical block numbers uniformly and randomly distributed in $[0, 2^{41}]$. During the experiment run, there are totally 20 GB worth of new block versions logged and 935 MB ($20GB * \frac{24}{512}$) worth of indexed map entries inserted. The buffer memory for BOSC's per-block update request queuing is set to 64 MB and the leaf page cache for TPIE is also set to 64 MB.

The disk write logging performance of *Mariner* when the indexed map is not updated at all is mainly determined by the overhead of logging new block versions, and thus represents its performance upper bound. As shown in Figure 10, compared with the upper bound, the throughput degradation due to the indexed map update is up to 95% when the indexed map is implemented as a vanilla B^+ tree (i.e. TPIE), is up to 80% when the indexed map is implemented as a B^+ tree using file-level append-only logging and per-block request queuing, and is no more than 15% when the indexed map is implemented as a BOSC-based B^+ tree. This result demonstrates that the BOSC-based B^+ tree implementation successfully removes indexed map update as a performance bottleneck of *Mariner*. Across these three variants, the throughput degradation decreases with the increase in the inter-request interval, because the additional indexed map update overhead becomes less significant when the input load is less demanding. In terms of absolute performance, the disk write logging throughput of the *Mariner* version using a BOSC-based B^+ tree is more than 45 times higher than that of the *Mariner* version using a TPIE-based B^+ tree, and is more than 3 times better than that of the *Mariner* version using a BOSC-based B^+ tree without low-latency logging.

6.6.2 Garbage Collection for Data Backup

In a data backup system with data de-duplication capability, a physical block may be referenced by multiple backup snapshots. Because a backup snapshot typically has a finite retention period, the number of references to a physical block varies over time. When a physical block is no longer referenced by any backup snapshot, it should be reclaimed and reused.

When a logical block is modified, it is going to be backed up in the next backup run, and the information required to back up this logical block includes its logical block number(LBN), the physical block number (PBN) of the physical block previously mapped to this LBN (BPBN), and the PBN of the physical block currently mapped to this LBN (CPBN). The data backup system uses the logical block's payload to determine whether it is a duplicate of some existing physical block (whose PBN is DPBN) in the data backup system. Therefore, backing up a logical block triggers the updates of the GC-related metadata of up to three physical blocks. Assume a data backup system manages a 4-PB physical disk space with a 4-KB block size, then there are one billion physical blocks, and we need an array of one billion entries to record their GC-related metadata. Obviously this GC metadata array (GMA) cannot fit into the memory, and there is no reason to expect much locality in the accesses to this array.

To remove the performance bottleneck due to GMA updates, we partitioned the GMA into 128-Kbyte chunks, and queued each GMA update in the corresponding per-chunk queue after logging it. Then we used one or multiple background commit threads to commit the updates to those chunks that have enough pending updates to disk using largely sequential disk I/O. When multiple commit threads are in action, they work on disjoint but neighboring chunks simultaneously. We compared this with a vanilla GMA update implementation, which buffers GMA update requests in a queue, and uses a background thread to commit them on a first come first serve basis. The evaluation machine consists of two quad-core 3.4GHz Intel Core i-7 processor, 14GB of RAM, five 7200-RPM WD Caviar blue hard disks of 1TB each, one for the system disk, two for storing the GMA on a striped software RAID with a 64-Kbyte stripe unit size, and the remaining two for storing the deduplication fingerprints on a striped software RAID.

In Table 2, the throughput of the data deduplication engine without any GMA updates (the last column) sets an upper bound because it corresponds to a zero-cost GMA update scheme. When the BOSC-based GMA update scheme uses a single commit thread, the end-to-end throughput of the deduplication engine is decreased to 19% of the upper bound. By increasing the number of commit threads to 4 and therefore the disk I/O concurrency, it increases the end-to-end throughput to 97% of the upper bound. The number of commit threads represent a tradeoff between disk access locality and disk I/O concurrency. Empirically, the optimal number of commit threads for our experiment set-up seems to be 4. However, regardless of the number of commit threads used, the end-to-end throughput of the data deduplication engine using the vanilla GMA update scheme never exceeds 5% of the upper bound.

Commit Threads	*Dedupe + vanilla GC*	*Dedupe + BOSC-based GC*	*Dedupe without GC*
1	4441	40821	216938
2	4534	202582	216938
4	7583	209721	216938
10	6134	203363	216938

Table 2. *End-to-end throughputs(fingerprints processed/second) of a data deduplication engine with multiple garbage collector configurations.*

7. Conclusion

Conventional storage systems are seriously challenged when facing input workloads that are update-intensive and have low access locality. Such workloads are not uncommon. For example, the back-end databases in typical Internet E-commerce services are routinely bombarded with workloads with intensive update requests (e.g., due to order processing) that involve random disk accesses (e.g., due to a large number of concurrent users). As of now, no good solution can effectively handle such workloads without resorting to special caching hardware such as battery-backed DRAM. This paper describes a simple but effective solution to this problem, which consists of (1) an *update-aware* disk access interface and (2) an efficient batched processing strategy that completes pending update requests using sequential disk I/O called BOSC. We have successfully built a BOSC prototype that embodies these two ideas, and empirically demonstrated its efficiency by showing that the update request throughput of a BOSC-based B^+ tree implementations is more than an order of magnitude faster than that of a vanilla B^+ tree built on top of the conventional disk access interface. In addition, BOSC is able to deliver this performance improvement while providing the same durability guarantee as servicing update requests synchronously. In summary, the specific research contributions of this work include

- A new disk access interface that supports disk update as a first-class primitive and enables the specification of application-specific callback functions to be invoked by the underlying storage system,
- A highly efficient storage system architecture that effectively commits pending update requests in a batched fashion, and drastically improves the physical disk access efficiency by using only *sequential* disk I/O to bring in the requests' target disk blocks, and
- A complete prototype implementation of the BOSC architecture and a comprehensive evaluation of this prototype by measuring and analyzing the performance results taken on a BOSC-based B^+ tree and two applications built on BOSC.

References

[1] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms (extended abstract). In *WADS '95: Proceedings of the 4th International Workshop on Algorithms and Data Structures*, pages 334–345, London, UK, 1995. Springer-Verlag.

[2] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002.

[3] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-Efficient Data Structures Using TPIE. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 88–100, London, UK, 2002. Springer-Verlag.

[4] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 176, Washington, DC, USA, 2004. IEEE Computer Society.

[5] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. Lopez-Ortiz. The cost of cache-oblivious searching. In *Proceedings of FOCS 2003*, pages p. 271–282, 2003.

[6] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *SIAM Journal of Computing*, volume 35(2), pages p. 341–358, 2005.

[7] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious b-trees. In *Proceedings of SPAA 2005*, pages p. 228–237, 2005.

[8] L. Biveinis, S. Šaltenis, and C. S. Jensen. Main-memory operation buffering for efficient R-tree update. In *VLDB 2007: Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 591–602. VLDB Endowment, 2007.

[9] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A History and Evaluation of System R. *Communications of the ACM*, 24(10):632–646, 1981.

[10] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A High-Performance Parallel Storage Device with Strong Recovery Guarantees. Technical Report HPL-CSP-92-9 rev 1, HewlettPackard Laboratories Report, November 1992.

[11] Y.-Y. Chen, Q. Gan, and T. Suel. I/O-Efficient Techniques for Computing PageRank. In *CIKM '02: Proceedings of the 11th International Conference on Information and Knowledge Management*, pages 549–557, New York, NY, USA, 2002. ACM Press.

[12] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical Evaluation of Multi-level Buffer Cache Collaboration for Storage Systems. *SIGMETRICS Perform. Eval. Rev.*, 33(1):145–156, 2005.

[13] T. Chiueh and L. Huang. Track-Based Disk Logging. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 429–438, Washington, DC, USA, 2002. IEEE Computer Society.

[14] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Fransisco, CA, USA, 1992. USENIX Association.

[15] T. P. P. Council. *TPC Benchmark C Standard Specification*, volume 1 and 2. Waterside Associates, Fremont, CA, 1.0.a edition, Aug, 1996.

[16] A. Dan and D. Towsley. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. *SIGMETRICS Perform. Eval. Rev.*, 18(1):143–152, 1990.

[17] A. Devulapalli, D. Dalessandro, P. Wyckoff, and N. Ali. Attribute Storage Design for Object-based Storage Devices. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 263–268, Washington, DC, USA, 2007. IEEE Computer Society.

[18] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft Updates: a Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[19] G. Graefe. Sorting and Indexing with Partitioned B-Trees. In *Conference on Innovative Data Systems Research*, 2003.

[20] G. Graefe. Write-optimized B-trees. In *VLDB 2004: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 672–683. VLDB Endowment, 2004.

[21] G. Graefe. B-tree Indexes for High Update Rates. *SIGMOD Rec.*, 35(1):39–44, 2006.

[22] R. Hagmann. Reimplementing the Cedar File System using Logging and Group Commit. In *SOSP '87: Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 155–162, New York, NY, USA, 1987. ACM Press.

[23] T. Haveliwala. Efficient Computation of PageRank. Technical Report 1999-31, Stanford University, Stanford University, Feburary 1999.

[24] L. O. X. B. He, M. J. Kosa, and S. L. Scott. A Unified Multiple-Level Cache for High Performance Storage Systems. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[25] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. WebBase: a Repository of Web Pages. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : the International Journal of Computer and Telecommunications Networking*, pages 277–293, Amsterdam, The Netherlands, 2000. North-Holland Publishing Co.

[26] G. F. Hughes and J. F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives using SATA Disk Drives. *Transactions on Storage*, 1(1):95–107, 2005.

[27] *The InnoDB Storage Engine*.

[28] N. P. Jouppi. Cache Write Policies and Performance. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 191–201, New York, NY, USA, 1993. ACM Press.

[29] V. Y. K. Kher. Decentralized Authentication Mechanisms for Object-based Storage Devices. In *SISW '03: Proceedings of the Second IEEE International Security in Storage Workshop*, page 1, Washington, DC, USA, 2003. IEEE Computer Society.

[30] M. Lifantsev and T. Chiueh. I/O-Conscious Data Preparation for Large-Scale Web Search Engines. In *VLDB '2002: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 382–393, Hongkong, China, 2002. VLDB Endowment.

[31] M. Lu, S. Lin, and T. Chiueh. Efficient Logging and Replication Techniques for Comprehensive Data Protection. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 171–184, Washington, DC, USA, 2007. IEEE Computer Society.

[32] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[33] C. Malakapalli and V. Gunturu. Evaluation of SCSI over TCP/IP and SCSI over Fibre Channel Connections. In *HOTI '01: Proceedings of the The Ninth Symposium on High Performance Interconnects (HOTI '01)*, page 87, Washington, DC, USA, 2001. IEEE Computer Society.

[34] M. K. McKusick and G. R. Ganger. Soft Updates: a Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 1999. USENIX Association.

[35] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a Distributed Full-Text Index for the Web. In *WWW '01: Proceedings of the 10th International Conference on World Wide Web*, pages 396–406, New York, NY, USA, 2001. ACM Press.

[36] S. Mitra, W. W. Hsu, and M. Winslett. Trustworthy Keyword Search for Regulatory-Compliant Records Retention. In *VLDB '2006: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1001–1012. VLDB Endowment, 2006.

[37] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. In *ACM Transactions on Database Systems, vol. 17*, pages 94–162, 1992.

[38] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the Sync. In *OSDI'2006: Proceedings of the 7th conference on USENIX*

Symposium on Operating Systems Design and Implementation, pages 1–14, Berkeley, CA, USA, 2006. USENIX Association.

[39] Open Source Development Labs (OSDL). Database Test Suite: DBT-[1,2,3,4,5]. http://osdldbt.sourceforge.net/, 2003.

[40] C. U. Orji and J. A. Solworth. Write-Only Disk Cache Experiments on Multiple Surface Disks. In *ICCI '92: Proceedings of the Fourth International Conference on Computing and Information*, pages 385–388, Washington, DC, USA, 1992. IEEE Computer Society.

[41] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford Digital Library Technologies Project, 1998.

[42] O. Procopiuc, P. Agarwal, L. Arge, and J. Vitter. Bkd-tree: A dynamic scalable kd-tree, 2002.

[43] B. Ribeiro-Neto, E. S. Moura, M. S. Neubert, and N. Ziviani. Efficient Distributed Algorithms to Build Inverted Files. In *SIGIR '99: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 105–112, New York, NY, USA, 1999. ACM Press.

[44] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks: Remote execution for network-attached storage. Technical Report CMU-CS-97-198, Parallel Data Lab, Carnegie Mellon University, December 1997.

[45] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[46] R. Stata, K. Bharat, and F. Maghoul. The Term Vector Database: Fast Access to Indexing Terms for Web Pages. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : the International Journal of Computer and Telecommunications Networking*, pages 247–255, Amsterdam, The Netherlands, 2000. North-Holland Publishing Co.

[47] M. Stonebraker. The Design of the POSTGRES Storage System. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, pages 289–300, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[48] D. E. Vengroff and J. S. Vitter. I/O-Efficient Algorithms and Environments. *ACM Computing Surveys (CSUR)*, 28(4):212, 1996.

[49] W. Wang, Y. Zhao, and R. Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 145–158, Berkeley, CA, USA, 2004. USENIX Association.

[50] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 24, Berkeley, CA, USA, 2002. USENIX Association.