The Bw-Tree: A B-tree for New Hardware Platforms

Justin J. Levandoski [1], David B. Lomet [2], Sudipta Sengupta [3]

Microsoft Research
Redmond, WA 98052, USA
[1]justin.levandoski@microsoft.com, [2]lomet@microsoft.com, [3]sudipta@microsoft.com

***Abstract*— The emergence of new hardware and platforms has led to reconsideration of how data management systems are designed. However, certain basic functions such as key indexed access to records remain essential. While we exploit the common architectural layering of prior systems, we make radically new design decisions about each layer. Our new form of B-tree, called the Bw-tree achieves its very high performance via a latch-free approach that effectively exploits the processor caches of modern multi-core chips. Our storage manager uses a unique form of log structuring that blurs the distinction between a page and a record store and works well with flash storage. This paper describes the architecture and algorithms for the Bw-tree, focusing on the main memory aspects. The paper includes results of our experiments that demonstrate that this fresh approach produces outstanding performance.**

I. Introduction

A. Atomic Record Stores

There has been much recent discussion of No-SQL systems, which are essentially atomic record stores (ARSs) [1]. While some of these systems are intended as stand-alone products, an atomic record store can also be a component of a more complete transactional system, given appropriate control operations [2], [3]. Indeed, one can regard a database system as including an atomic record store as part of its kernel.

An ARS supports the reading and writing of individual records, each identified by a key. Further, a tree-based ARS supports high performance key-sequential access to designated subranges of the keys. It is this combination of random and key-sequential access that has made B-trees the indexing method of choice within database systems.

However, an ARS is more than an access method. It includes the management of stable storage and the requirement that its updates be recoverable should the system crash. It is the performance of the ARS of this more inclusive form that is the foundation for the performance of any system in which the ARS is embedded, including full function database systems.

This paper introduces a new ARS that provides very high performance. We base our ARS on a new form of B-tree that we call the Bw-tree. The techniques that we introduce make the Bw-tree and its associated storage manager particularly appropriate for the new hardware environment that has emerged over the last several years.

Our focus in this paper is on the main memory aspects of the Bw-tree. We describe the details of our latch-free technique, i.e., how we can do updates and structure modification operations without setting latches. Our approach also carefully avoids cache line invalidations, hence leading to substantially better caching performance as well. We describe how we use our log structured storage manager at a high level, but leave the specifics to another paper.

B. The New Environment

Database systems have mostly exploited the same storage and CPU infrastructure since the 1970s. That infrastructure used disks for persistent storage. Disk latency is now analogous to a round trip to Pluto [4]. It used processors whose uni-processor performance increased with Moore's Law, thus limiting the need for high levels of concurrent execution on a single machine. Processors are no longer providing ever higher uni-core performance. Succinctly, "this changes things".

1) Design for Multi-core: We live in a high peak performance multi-core world. Uni-core speed will at best increase modestly, thus we need to get better at exploiting a large number of cores by addressing at least two important aspects:

1) Multi-core cpus mandate high concurrency. But, as the level of concurrency increases, latches are more likely to block, limiting scalability [5].
2) Good multi-core processor performance depends on high CPU cache hit ratios. Updating memory in place results in cache invalidations, so how and when updates are done needs great care.

Addressing the first issue, the Bw-tree is latch-free, ensuring a thread never yields or even re-directs its activity in the face of conflicts. Addressing the second issue, the Bw-tree performs "delta" updates that avoid updating a page in place, hence preserving previously cached lines of pages.

2) Design for Modern Storage Devices: Disk latency is a major problem. But even more crippling is their low I/O ops per second. Flash storage offers higher I/O ops per second at lower cost. This is key to reducing costs for OLTP systems. Indeed Amazon's DynamoDB includes an explicit ability to exploit flash [6]. Thus the Bw-tree targets flash storage.

Flash has some performance idiosyncracies, however. While flash has fast random and sequential reads, it needs an erase cycle prior to write, making random writes slower than sequential writes [7]. While flash SSDs typically have a mapping layer (the FTL) to hide this discrepancy from users, a noticeable slowdown still exists. As of 2011, even high-end FusionIO drives exhibit a 3x faster sequential write performance than

random writes [8]. The Bw-tree performs log structuring itself at its storage layer. This approach avoids dependence on the FTL and ensures that our write performance is as high as possible for both high-end and low-end flash devices and hence, not the system bottleneck.

C. Our Contributions

We now describe our contributions, which are:

1) The Bw-tree is organized around a mapping table that virtualizes both the location and the size of pages. This virtualization is essential for both our main memory latch-free approach and our log structured storage.
2) We update Bw-tree nodes by prepending update deltas to the prior page state. Because our delta updating preserves the prior page state, it improves processor cache performance. Having the new node state at a new storage location permits us to use the atomic compare and swap instructions to update state. Hence, the Bw-tree is latch-free in the classic sense of allowing concurrent access to pages by multiple threads.
3) We have devised page splitting and merging structure modification operations (SMOs) for the Bw-tree. SMOs are realized via multiple atomic operations, each of which leaves the Bw-tree well-formed. Further, threads observing an in-progress SMO do not block, but rather take steps to complete the SMO.
4) Our log structured store (LSS), while nominally a page store, uses storage very efficiently by mostly posting page change deltas (one or a few records). Pages are eventually made contiguous via consolidating delta updates, and also during a flash "cleaning" process. LSS will be described fully in a another paper.
5) We have designed and implemented an ARS based on the Bw-tree and LSS. We have measured its performance using real and synthetic workloads, and report on its very high performance, greatly out-performing both BerkeleyDB, an existing system designed for magnetic disks, and latch-free skip lists in main memory.

In drawing broader lessons from this work, we believe that latch free techniques and state changes that avoid update-in-place are the keys to high performance on modern processors. Further, we believe that log structuring is the way to provide high storage performance, not only with flash, but also with disks. We think these "design paradigms" are applicable more widely to realize high performance data management systems.

D. Paper Outline

We present an overview of Bw-tree architecture in Section 2. In Sections 3 through 5, we describe the system we built. We start at the top layer with in-memory page organization in Section 3, followed by Bw-tree organization and structure modifications in Section 4. Section 5 details how the cache is managed. In Section 6, we describe our experiments and the performance results of them. Section 7 describes related work and how we differ significantly in our approach. We conclude with a short discussion in Section 8.

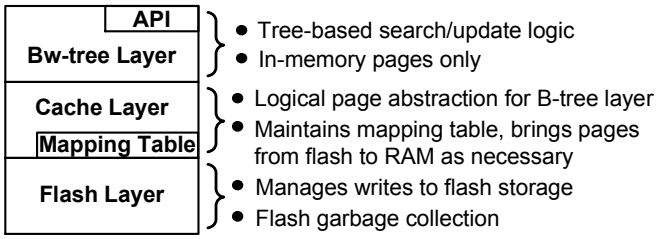


Fig. 1. The architecture of our Bw-tree atomic record store.

II. Bw-tree Architecture

The Bw-tree atomic record store (ARS) is a classic B+-tree [9] in many respects. It provides logarithmic access to keyed records from a one-dimensional key range, while providing linear time access to sub-ranges. Our ARS has a classic architecture as depicted in Figure 1. The access method layer, our *Bw-tree Layer*, is at the top. It interacts with the middle *Cache Layer*. The cache manager is built on top of the *Storage Layer*, which implements our log-structured store (LSS). The LSS currently exploits flash storage, but it could manage with either flash or disk.

This design is architecturally compatible with existing database kernels, while also being suitable as a standalone "data component" in a decoupled transactional system [2], [3]. However, there are significant departures from this classic picture. In this section, we provide an architectural overview of the Bw-tree ARS, describing why it is uniquely well-suited for multi-core processors and flash based stable storage.

A. Modern Hardware Sensitive

In our Bw-tree design, threads almost never block. Eliminating latches is our main technique. Instead of latches, we install state changes using the atomic compare and swap (CAS) instruction. The Bw-tree only blocks when it needs to fetch a page from stable storage (the LSS), which is rare with a large main memory cache. This persistence of thread execution helps preserve core instruction caches, and avoids thread idle time and context switch costs. Further, the Bw-tree performs node updates via "delta updates" (attaching the update to an existing page), not via update-in-place (updating the existing page memory). Avoiding update-in-place reduces CPU cache invalidation, resulting in higher cache hit ratios. Reducing cache misses increases the instructions executed per cycle.

Performance of data management systems is frequently gated by I/O access rates. We have chosen to target flash storage to ease that problem. But even with flash, when attached as an SSD, I/O access rates can limit performance. Our log structure storage layer enables writing large buffers, effectively eliminating any write bottle neck. Flash storage's high random read access rates coupled with a large main memory cache minimizes blocking on reads. Writing large multi-page buffers permits us to write variable size pages that do no contain "filler" to align to a uniform size boundary.

The rest of this section summarizes the major architectural and algorithmic innovations that make concrete the points described above.

B. The Mapping Table

Our cache layer maintains a *mapping table*, that maps logical pages to physical pages, logical pages being identified by a logical "page identifier" or PID. The mapping table translates a PID into either (1) *a flash offset*, the address of a page on stable storage, or (2) *a memory pointer*, the address of the page in memory. The mapping table is thus the central location for managing our "paginated" tree. While this indirection technique is not unique to our approach, we exploit it as the base for several innovations. We use PIDs in the Bw-tree to link the nodes of the tree. For instance, all downward "search" pointers between Bw-tree nodes are PIDs, not physical pointers.

The mapping table severs the connection between physical location and inter-node links. This enables the physical location of a Bw-tree node to change on every update and every time a page is written to stable storage, without requiring that the location change be propagated to the root of the tree (i.e., updating inter-node links). This "relocation" tolerance directly enables both delta updating of the node in main memory and log structuring of our stable storage, as described below.

Bw-tree nodes are thus logical and do not occupy fixed physical locations, either on stable storage or in main memory. Hence we are free to mold them to our needs. A "page" for a node thus suggests a policy, not a requirement, either in terms of how we represent nodes or how large they might become. We permit page size to be elastic, meaning that we can split when convenient as size constraints do not impose a splitting requirement.

C. Delta Updating

Page state changes are done by creating a delta record (describing the change) and prepending it to an existing page state. We install the (new) memory address of the delta record into the page's physical address slot in the mapping table using the atomic compare and swap (CAS) instruction[1]. If successful, the delta record address becomes the new physical address for the page. This strategy is used both for data changes (e.g., inserting a record) and management changes (e.g., a page being split or flushing a page to stable storage).

Occasionally, we consolidate pages (create a new page that applies all delta changes) to both reduce memory footprint and to improve search performance. A consolidated form of the page is also installed with a CAS, and the prior page structure is garbage collected (i.e., its memory reclaimed). A reference to the entire data structure for the page, including deltas, is placed on a pending list all of which will be reclaimed when safe. We use a form of epoch to accomplish safe garbage collection, [10].

Our delta updating simultaneously enables latch-free access in the Bw-tree and preserves processor data caches by avoiding update-in-place. The Bw-tree mapping table is the key enabler of these features via its ability to isolate the effects of node updates to that node alone.

[1]The CAS is an atomic instruction that compares a given *old* value to a *current* value at memory location L, if the values are equal the instruction writes a *new* value to L, replacing *current*.

D. Bw-tree Structure Modifications

Latches do not protect parts of our index tree during structure modifications (SMOs) such as page splits. This introduces a problem. For example, a page split introduces changes to more than one page: the original overly large page O, the new page N that will receive half $O's$ contents, and the parent index page P that points down to O, and that must subsequently point to both O and N. Thus, we cannot install a page split with a single CAS. A similar but harder problem arises when we merge nodes that have become too small.

To deal with this problem, we break an SMO into a sequence of atomic actions, each installable via a CAS. We use a B-link design [11] to make this easier. With a side link in each page, we can decompose a node split into two "half split" atomic actions. In order to make sure that no thread has to wait for an SMO to complete, a thread that sees a partial SMO will complete it before proceeding with its own operation. This ensures that no thread needs to wait for an SMO to complete.

E. Log Structured Store

Our LSS has the usual advantages of log structuring [12]. Pages are written sequentially in a large batch, greatly reducing the number of separate write I/Os required. However, because of garbage collection, log structuring normally incurs extra writes to relocate pages that persist in reclaimed storage areas of the log. Our LSS design greatly reduces this problem.

When flushing a page, the LSS need only flush the deltas that represent the changes made to the page since its previous flush. This dramatically reduces how much data is written during a flush, increasing the number of pages that fit in the flush buffer, and hence reducing the number of I/O's per page. There is a penalty on reads, however, as the discontinuous parts of pages all must be read to return a page to the main memory cache. Here is when the very high random read performance of flash really contributes to our ARS performance.

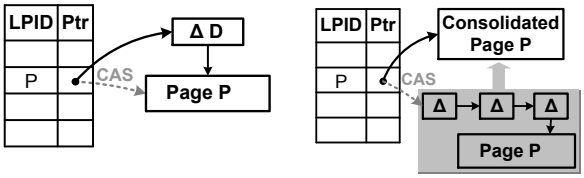
The LSS cleans prior parts of flash representing the old parts of its log storage. Delta flushing reduces pressure on the LSS cleaner by reducing the amount of storage used per page. This reduces the "write amplification" that is a characteristic of log structuring. During cleaning, LSS makes pages and their deltas contiguous for improved access performance.

是否合并delta

F. Managing Transactional Logs

As in a conventional database system, our ARS needs to ensure that updates persist across system crashes. We tag each update operation with a unique identifier that is typically the log sequence number (LSN) of the update on the transactional log (maintained elsewhere, e.g., in a transactional component). LSNs are managed so as to support recovery idempotence, i.e., ensuring that operations are executed at most once.

Like conventional systems, pages are flushed lazily while honoring the write-ahead log protocol (WAL). Unconventionally, however, we do not block page flushes to enforce WAL. Instead, because the recent updates are separate deltas from the rest of the page, we can remove "recent" updates (not yet on the stable transactional log) from pages when flushing.



(a) Update using delta record (b) Consolidating a page

Fig. 2. In-memory pages. Page updates use compare-and-swap (CAS) on the physical pointer in the mapping table.

III. In-Memory Latch Free Pages

This section describes Bw-tree in-memory pages. We begin by discussing the basic page structure and how we update pages in a latch-free manner. We then discuss occasional page consolidation used to make search more efficient. Using a tree based index enables range scans, and we describe how this is accomplished. Finally, we discuss in-memory garbage collection with our epoch safety mechanism.

A. Elastic Virtual Pages

The information stored on a Bw-tree page is similar to that of a typical B+-tree. Internal index nodes contain (separator key,pointer) pairs sorted by key that direct searches down the tree. Data (leaf) nodes contain (key, record) pairs. In addition, pages also contain (1) a *low key* representing the smallest key value that can be stored on the page (and in the subtree below), and (2) a *high key* representing the largest key value that can be stored on the page, and (3) a *side link pointer* that points to the node's immediate right sibling on the same level in the tree, as in a B-link tree [11].

Two distinct features make the Bw-tree page design unique. First, Bw-tree pages are *logical*, meaning they do not occupy fixed physical locations or have fixed sizes. (1) We identify a page using its PID index into the mapping table. Accessors to the page use the mapping table to translate the PID into its current physical address. All links between Bw-tree nodes ("search" pointers, side links) are PIDs. (2) Pages are elastic, meaning there is no hard limit on how large a page may grow. Pages grow by having "delta records" prepended to them. A delta record represents a single record modification (e.g., insert, update), or system management operation (e.g., page split).

Updates. We never update a Bw-tree page in place (i.e., modify its memory contents). Rather, we create a *delta record* describing the update and prepend it to the existing page. Delta records allow us to incrementally update page state in a latch-free manner. We first create a new delta record D that (physically) points to the page's current address P. We obtain P from the page's entry in the mapping table. The memory address of the delta record will serve as the new memory address (the new state) for the page.

To install the new page state in the mapping table (making it "live" in the index), we use the atomic compare and swap (CAS) instruction (described in Section II) to replace the current address P with the address of D. The CAS compares P to the current address in the mapping table. If the current address equals P, the CAS successfully installs D, otherwise it fails (we discuss CAS failures later). Since all pointers between Bw-tree nodes are via PIDs, the CAS on the mapping table entry is the *only* physical pointer change necessary to install a page update. Furthermore, this latch-free technique is the *only* way to update a page in the Bw-tree, and is uniform across all operations that modify a page. In the rest of this paper, we refer to using the CAS to update a page as "installing" an operation.

Figure 2(a) depicts the update process showing a delta record D prepended to page P. The dashed line from the mapping table to P represents P's old address, while the solid line to the delta record represents P's new physical address. Since updates are atomic, only one updater can "win" if there are competing threads trying install a delta against the same "old" state in the mapping table. A thread must retry its update if it fails[2].

After several updates, a "delta chain" forms on a page as depicted in the lower right of Figure 2(b) (showing a chain of three deltas). Each new successful updates forms the new "root" of the delta chain. This means the physical structure of a Bw-tree page consists of a delta chain prepended to a *base page* (i.e., a consolidate B-tree node). For clarity, we refer to a base page as the structure to which the delta chain is prepended, and refer to a page as the a base page along with its (possibly empty) delta chain.

Leaf-level update operations. At the leaf page level, updates (deltas) are one of three types: (1) *insert*, representing a new record inserted on the page; (2) *modify*, representing a modification to an existing record in the page; (3) *delete*, representing the removal of an existing record in the page. All update deltas contain an LSN provided by the client issuing the update. We use this LSN for transactional recovery involving a transaction log manager with its need to enforce the write-ahead-log (WAL) protocol (we discuss WAL and LSNs further in Section V-A). Insert and update deltas contain a record representing the new payload, while delete deltas only contain the key of the record to be removed. In the rest of this section, we discuss only record updates to leaf pages, postponing discussion of other updates (e.g., index pages, splits, flushes) until later in the paper.

Page search. Leaf page search involves traversing the delta chain (if present). The search stops at the first occurrence of the search key in the chain. If the delta containing the key represents an insert or update, the search succeeds and returns the record. If the delta represents a delete, the search fails. If the delta chain does not contain the key, the search performs a binary search on the base page in typical B-tree fashion. We discuss index page search with delta chains in Section IV.

B. Page Consolidation

Search performance eventually degrades if delta chains grow too long. To combat this, we occasionally perform *page con-*

[2]The retry protocol will depend on the specific update operation. We discuss specific retry protocols where appropriate.

solidation that creates a new "re-organized" base page containing all the entries from the original base page as modified by the updates from the delta chain. We trigger consolidation if an accessor thread, during a page search, notices a delta chain length has exceeded a system threshold. The thread performs consolidation after attempting its update (or read) operation.

When consolidating, the thread first creates a new base page (a new block of memory). It then populates the base page with a sorted vector containing the most recent version of a record from either the delta chain or old base page (deleted records are discarded). The thread then installs the new address of the consolidated page in the mapping table with a CAS. If it succeeds, the thread requests garbage collection (memory reclamation) of the old page state. Figure 2(b) provides an example depicting the consolidation of page P that incorporates deltas into a new "Consolidated Page P". If this CAS fails, the thread abandons the operation by deallocating the new page. The thread does not retry, as a subsequent thread will eventually perform a successful consolidation.

C. Range Scans

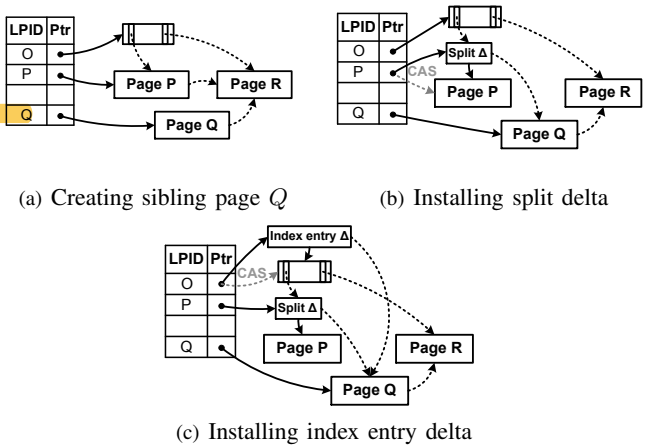
A range scan is specified by a key range (*low key*, *high key*). Either of the boundary keys can be omitted, meaning that one end of the range is open-ended. A scan will also specify either an ascending or descending key order for delivering the records. Our description here assumes both boundary keys are provided and the ordering is ascending. The other scan options are simple variants.

A scan maintains a cursor providing a key indicating how far the search has progressed. For a new scan, the remembered key is $lowkey$. When a data page containing data in the range is first accessed, we construct a vector of records containing all records on the page to be processed as part of the scan. In the absence of changes to the page during the scan, this allows us to efficiently provide the "next-record" functionality. This is the common case and needs to be executed with high performance.

We treat each "next-record" operation as an atomic unit. The entire scan is *not* atomic. Transactional locking will prevent modifications to records we have seen (assuming serializable transactions) but we do not know the form of concurrency control used for records we have not yet delivered. So before delivering a record from our vector, we check whether an update has affected the yet unreturned subrange in our record vector. If such an update has occurred, we reconstruct the record vector accordingly.

D. Garbage Collection

A latch-free environment does not permit exclusive access to shared data structures (e.g., Bw-tree pages), meaning one or more readers can be active in a page state even as it is being updated. We do not want to deallocate memory still accessed by another thread. For example, during consolidation, a thread "swaps out" the old state of a page (i.e., delta chain plus base page) for a new consolidated state and requests that the old state be garbage collected. However, we must take care not to deallocate the old page state while another thread still accesses it. Similar concerns arise when a page is removed from the Bw-tree. That is, other threads may still be able to access the now removed page. We must protect these threads from accessing reclaimed and potentially "repurposed" objects by preventing reclamation until such access is no longer possible. This is done by a thread executing within an "epoch".



(a) Creating sibling page Q (b) Installing split delta

(c) Installing index entry delta

Fig. 3. Split example. Dashed arrows represent logical pointers, while solid arrows represent physical pointers.

An epoch mechanism is a way of protecting objects being deallocated from being re-used too early [10]. A thread joins an epoch when it wants to protect objects it is using (e.g., searching) from being reclaimed. It exits the epoch when this dependency is finished. Typically, a thread's duration in an epoch is for a single operation (e.g. insert, next-record). Threads "enrolled" in epoch E might have seen earlier versions of objects that are being deallocated in epoch E. However, a thread enrolled in epoch E cannot have seen objects deallocated in epoch E-1 because it had not yet started its dependency interval. Hence, once all threads enrolled in epoch E have completed and exited the epoch ("drained"), it is safe to recycle the objects deallocated in epoch E. We use epochs to protect both storage and deallocated PIDs. Until the epoch has drained such objects cannot be recycled.

IV. Bw-tree Structure Modifications

All Bw-tree structure modification operations (SMOs) are performed in a latch-free manner. To our knowledge, this has never been done before, and is crucial for our design. We first describe node splits, followed by node merges. We then discuss how to ensure an SMO has completed prior to performing actions that depend on the SMO. This is essential both to avoid confusion in main memory and to prevent possible corrupt trees should the system crash at an inopportune time.

A. Node Split

Splits are triggered by an accessor thread that notices a page size has grown beyond a system threshold. After attempting its operation, the thread performs the split.

The Bw-tree employs the B-link atomic split installation technique that works in two phases [11]. We first atomically

install the split at the child (e.g., leaf) level. This is called a half split. We then atomically update the parent node with the new index term containing a new separator key and a pointer to the newly created split page. This process may continue recursively up the tree as necessary. The B-link structure allows us to separate the split into two atomic actions, since the side link provides a valid search tree after installing the split at the child level.

Child Split. To split a node P, the B-tree layer requests (from the cache layer) allocation for a new node Q in the mapping table (Q is the new right sibling of P). We then find the appropriate separator key K_P from P that provides a balanced split and proceed to create a new consolidated base state for Q containing the records from P with keys greater than K_P. Page Q also contains a side link to the former right sibling of P (call this page R). We next install the physical address of Q in the mapping table. This installation is done without a CAS, since Q is visible to only the split thread. Figure 3(a) depicts this scenario, where a new sibling page Q contains half the records of P, and (logically) points to page R (the right sibling of P). At this point, the original (unsplit) state of P is still present in the mapping table, and Q is invisible to the rest of the index.

We atomically install the split by prepending a *split delta record* to P. The split delta contains two pieces of information: (1) the separator key K_P used to invalidate all records within P greater than K_P, since Q now contains these records and (2) a logical side pointer to the new sibling Q. This installation completes the first "half split". Figure 3(b) depicts such a scenario after prepending a split delta to page P pointing to its new sibling page Q. At this point, the index is valid, even without the presence of an index term for Q in the parent node O. All searches for a key contained within Q will first go to P. Upon encountering the split delta on P, the search will traverse the side link to Q when the search key is greater than separator key K_P. Meanwhile, all searches for keys less than the K_P remain at P.

Parent Update. In order to direct searches directly to Q, we prepend an *index term delta record* to the parent of P and Q to complete the second half split. This index delta contains (1) K_P, the separator key between P and Q, (2) a logical pointer to Q, and (3) K_Q, the separator key for Q (formerly the separator directing searches to P). We remember our path down the tree (i.e. the PIDs of nodes on the path) and hence can immediately identify the parent. Most of the time, the remembered parent on the path will be the correct one and we do the posting immediately. Occasionally the parent may have been merged into another node. But our epoch mechanism guarantees that we will see the appropriate deleted state that will tell us this has happened. That is, we are guaranteed that the parent PID will not be a dangling reference. When we detect a deleted state, we go up the tree to the grandparent node, etc., and do a re-traversal down the tree to find the parent that is "still alive".

Having K_P and K_Q present in the boundary key delta is an optimization to improve search speed. Since searches must now traverse a delta chain on the index nodes, finding a boundary key delta in the chain such that a search key v is greater than K_P and less than or equal to K_Q allows the search to end instantly and follow the logical pointer down to Q. Otherwise, the search continues into the base page, which is searched with a simple binary search to find the correct pointer to follow. Figure 3(c) depicts our running split example after prepending the index entry delta to parent page O, where the dashed line represents the logical pointer to page Q.

Consolidations. Posting deltas decreases latency when installing splits, relative to creating and installing completely new base pages. Decreasing latency decreases the chance of "failed splits", i.e., the case that other updates sneak in before we try to install the split (and fail). However, we eventually consolidate split pages at a later point in time. For pages with split deltas, consolidation involves creating a new base page containing records with keys less than the separator key in the split delta. For pages with index entry deltas, we create a new consolidated base page containing the new separator keys and pointers.

B. Node Merge

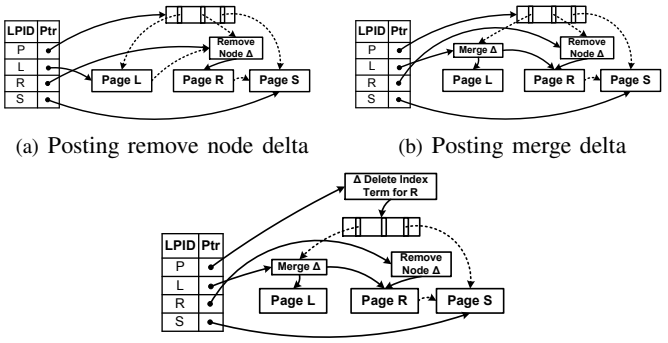
Like splits, node merges are triggered when a thread encounters a node that is below some threshold size. We peform the node merge in a latch-free manner, which is illustrated in Figure 4. Merges are decidedly more complicated than splits, and we need more atomic actions to accomplish them.

Marking for Delete. The node R to be merged (to be removed) is updated with a *remove node delta*, as depicted in Figure 4(a). This stops all further use of node R. A thread encountering a *remove node delta* in R needs to read or update the contents of R previously contained in R by going to the left sibling, into which data from R will be merged.

Merging Children. The left sibling L of R is updated with a *node merge delta* that physically points (via a memory address) to the contents of R. Figure 4(b) illustrates what L and R look like during a node merge. Note that the the *node merge delta* indicates that the contents of R are to be included in L. Further, it points directly to this state, which is now *logically* considered to be part of L. This storage for R's state is now transferred to L (except for the *remove node delta* itself). It will only be reclaimed when L is consolidated. This turns what had previously been a linear delta chain representing a page state into a tree.

When we search L (now responsible for containing both its original key space and the key space that had been R's) the search becomes a tree search which directs the accessing thread to either L's original page or to the page that it subsumed from R as a result of the merge. To enable this, the *node merge delta* includes the separator key that enables the search to proceed to the correct node.

Parent Update. The parent node P of R is now updated by deleting its index term associated with R, shown in Figure 4(c). This is done by posting an *index term delete delta* that includes not only that R is being deleted, but also that L will now include data from the key space formerly contained



(a) Posting remove node delta

(b) Posting merge delta

(c) Posting index term delete delta

Fig. 4. Merge example. Dashed arrows represent logical pointers, while solid arrows represent physical pointers.

by R. The new range for L is explicitly included with a low key equal to L's prior low key and a high key equal to R's prior high key. As with node splits, this permits us to recognize when a search needs to be directed to the newly changed part of the tree. Further, it enables any search that drops through all deltas to the base page to find the right index term by a simple binary search.

Once the *index term delete delta* is posted, all paths to R are blocked. At this point we initiate the process of reclaiming Rs PID. This involves posting the PID to the pending delete list of PIDs for the currently active epoch. R's PID will not be recycled until all other threads that might have seen an earlier state of R have exited the epoch. This is the same epoch mechanism we use to protect page memory from premature recycling (Section III-D).

C. Serializing Structure Modifications and Updates

Our Bw-tree implementation assumes that conflicting data update operations are prevented by concurrency contol that is elsewhere in the system. This could be in the lock manager of an integrated database system, a transactional component of a decoupled transactional system (e.g., Deuteronomy [2]), or finally, an arbitrary interleaving of concurrent updates as enabled as by an atomic record store.

However, "inside" the Bw-tree, we need to correctly serialize data updates with SMOs and SMOs with other SMOs. That is, we must be able to construct a serial schedule for everything that occurs in the Bw-tree, where data updates and SMOs are treated as the units of atomicity. This is nontrivial.

We want to treat an SMO as atomic (think of them as system transactions), and we are doing this without using latches that could conceal the fact that there are multiple steps involved in an SMO. One way to think about this is that if a thread stumbles upon an incomplete SMO, it is like seeing uncommitted state. Being latch-free, the Bw-tree cannot prevent this from happening. Our response is to require that such a thread must complete and commit the SMO it encounters before it can either (1) post its update or (2) continue with its own SMO. For page splits, that means that when an updater or another SMO would traverse a side pointer to reach the correct page, it must complete the split SMO by posting the new *index term delta* to the parent. Only then can it continue on to its own activity. That forces the incomplete SMO to be "committed" and to serialize before the interrupted action the thread had started upon.

The same principle applies regardless of whether the SMO is a split or a node merge. When deleting a node R, we access its left sibling L to post the merge delta. If we find that L is being deleted, we are seeing an in progress and incomplete "earlier" system transaction. We need the delete of R to serialize after the delete of L in this case. Hence the thread deleting R needs to first complete the delete of L. Only then can this thread complete the delete of R. All combinations of SMO are serialized in the same manner. This can lead to the processing of a "stack" of SMOs, but given the rarity of this situation, it should not occur often, and is reasonably straightforward to implement recursively.

V. Cache Management

The cache layer is responsible for reading, flushing, and swapping pages between memory and flash. It maintains the mapping table and provides the abstraction of logical pages to the Bw-tree layer. When the Bw-tree layer requests a page reference using a PID, the cache layer returns the address in the mapping table if it is a memory pointer. Otherwise, if the address is a flash offset (the page is not in memory), it reads the page from the LSS to memory, installs the memory address in the mapping table (using a CAS), and returns the new memory pointer. All updates to pages, including those involving page management operations like split and page flush involve CAS operations on the mapping table at the location indexed by the PID.

Pages in main memory are occasionally flushed) to stable storage for a number of reasons. For instance, the cache layer will flush updates to enable the checkpointing of a transactional log when the Bw-tree is part of a transactional system. Page flushes also precede a *page swapout*, installing a flash offset in the mapping table and reclaiming page memory in order to reduce memory usage. With multiple threads flushing pages to flash, multiple page flushes that need a correct ordering must be done very carefully. We describe one such scenario for node splits in this section.

To keep track of which version of the page is on stable storage (the LSS) and where it is, we use a *flush delta record*, which is installed at the mapping table entry for the page using a CAS. Flush delta records also record which changes to a page have been flushed so that subsequent flushes send *only* incremental page changes to stable storage. When a page flush succeeds, the flush delta contains the new flash offset and fields describing the state of the page that was flushed.

The rest section focuses on how the cache layer interacts with the LSS layer by preparing and performing page flushes (LSS details covered in future work). We start by describing how flushes to LSS are coordinated with the demands of a separate transactional mechanism. We then describe the mechanics of how flushes are executed.

A. Write Ahead Log Protocol and LSNs

The Bw-tree is an ARS that can be included in a transactional system. When included, it needs to manage the transactional aspects imposed on it. The Deuteronomy [2] architecture makes those aspects overt as part of its protocol between transactional (TC) and data component (DC), so we use Deuteronomy terminology to describe this functionality. Similar considerations apply in other transactional settings.

LSNs. Record insert and update deltas in a the Bw-tree page are tagged with the Log Sequence Number (LSN) of their operations. The highest LSN among updates flushed is recorded in the flush delta describing the flush. LSNs are generated by the higher-level TC and are used by its transactional log.

Transaction Log Coordination. Whenever the TC appends (flushes) to its stable transactional log, it updates the End of Stable Log (ESL) LSN value. ESL is an LSN such that all lower valued LSNs are definitely in the stable log. Periodically, it sends an updated ESL value to the DC. It is necessary for enforcing causality via the write-ahead log protocol (WAL) that the DC not make durable any operation with an LSN greater than the latest ESL. This ensures that the DC is "running behind" the TC in terms of operations made durable. To enforce this rule, records on a page that have LSNs larger than the ESL are *not* included in a page when flushed to the LSS.

Page flushes in the DC are required by the TC when it advances its Redo-Scan-Start-Point (RSSP). When the TC wishes to advance the RSSP, it proposes an RSSPit to the DC. The intent is for this to permit the TC to truncate (drop) the portion of the transactional log earlier than the RSSP. The TC will then wait for an acknowledgement from the DC indicating that the DC has made every update with an LSN $<$ RSSP stable. Because these operations results are stable, the TC no longer has to send these operations to the DC during redo recovery. For the DC to comply, it needs to flush the records on every page that have LSNs $<$ RSSP before it acknowledges to the TC. Such flushes are never blocked as the cache manager can exclude every record with an LSN $>$ ESL. Non-blocking is *not possible* with a conventional update-in-place approach using a single page-wide LSN.

To enable this non-blocking behavior, we restrict page consolidation (Section III-B) to include only update deltas that have LSNs less than or equal to the ESL. Because of this, we can always exclude these deltas when we flush the page.

Bw-tree Structure Modifications. We wrap a system transaction [3] around the pages we log as part of Bw-tree SMOs. This solves the problem of concurrent SMOs that can result from our latch-free approach (e.g., two threads trying to split the same page). To keep what is in the LSS consistent with main memory, we do not commit the SMO system transaction with its updated pages until we know that its thread has "won" the race to install an SMO delta record at the appropriate page. Thus, we allow concurrent SMOs, but ensure that at most one of them can commit. The begin and end system transaction records are among the very few objects flushed to the LSS that are not instances of pages.

B. Flushing Pages to the LSS

The LSS provides a large buffer into which the cache manager posts pages and system transactions describing Bw-tree structure modifications. We give a brief overview of this functionality. Details will be described in another paper.

Page Marshalling. The cache manager marshalls the bytes from the pointer representation of the page in main memory into a linear representation that can be written to the flush buffer. The page state is captured at the time it is intended to be flushed. This is important as later updates might violate the WAL protocol or a page split may have removed records that need to be captured in LSS. For example, the page may be split and consolidated while an earlier flush request for it is being posted to the flush buffer. If the bytes for the earlier flush are marshalled after the split has removed the upper order keys in the pre-split page, the version of the page captured in the LSS will not have these records. Should the system crash before the rest of the split itself is flushed, those records will be lost. When marshalling records on a page for flush, multiple delta records are consolidated so that they appear contiguously in the LSS.

Incremental Flushing. When flushing a page, the cache manager only marshals those delta records which have an LSN between the previously flushed largest LSN on that page and the current ESL value. The previously flushed largest LSN information is contained in the latest flush delta record on the page (as described above).

Incremental flushing of pages means that the LSS consumes much less storage for a page than is the case for full page flushing. This is very valuable for a log structured store such as our LSS for two reasons. (1) It means that a flush buffer can hold far more updates across different pages than if the entire state of every page were flushed. This increases the writing efficiency on a per page basis. (2) The log structured store cleaner (garbage collector) does not need to work as hard since storage is not being consumed as fast. This reduces the execution cost per page for the cleaner. It also reduces the "write amplification", i.e. the requirement of re-writing unchanged pages when the cleaner encounters them.

Flush Activity. The flush buffer aggregates writes to LSS up to a configurable threshold (currently set at 1MB) to reduce I/O overhead. It uses ping-pong (double) buffers and alternates between them with asynchronous I/O calls to the LSS so that the buffer for the next batch of page flushes can be prepared while the current one is in progress.

After the I/O completes for a flush buffer, the states of the respective pages are updated in the mapping table. The result of the flush is captured in the mapping table by means of a flush delta describing the flush, which is prepended to the state and installed via a CAS like any other delta. If the flush has captured all the updates to the page, the page is "clean" in that there are no uncaptured updates not stable on the LSS.

The cache manager monitors the memory used by the Bw-tree, and when it exceeds a configurable threshold, it attempts to swap out pages to the LSS. Once a page is clean, it can be evicted from the cache. The storage for the state of an evicted

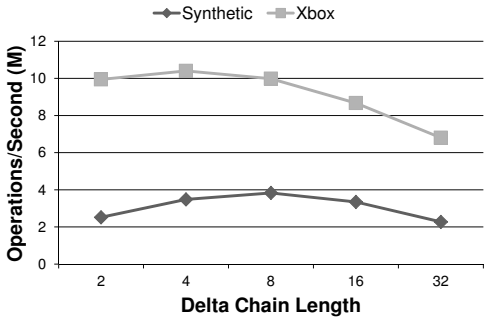


Fig. 5. Effect of Delta Chain Length

page is reclaimed via our epoch-based memory garbage collector (Section III-D).

VI. Performance Evaluation

This section provides experimental evaluation of the Bw-tree when compared to a "traditional" B-tree architecture (BerkeleyDB run in B-tree mode) as well as latch-free skiplists. Our experiments use a mix of real-world and synthetic workloads running on real system implementations. Since this paper focuses on in-memory aspects of the Bw-tree, our experiments explicitly focus on in-memory system performance. Performance evaluation on secondary storage will be the topic of future work.

A. Implementation and Setup

Bw-Tree. We implemented the Bw-Tree as a standalone atomic record store in approximately 10,000 lines of C++ code. We use the Win32 native `InterlockedCompareExchange64` to perform the CAS update installation. Our entire implementation was latch-free.

BerkeleyDB. We compare the Bw-tree to the BerkeleyDB key-value database. We chose BerkeleyDB due to its good performance as a standalone storage engine. Furthermore, it does not need to traverse a query processing layer as done in a complete database system. We use the C implementation of BerkeleyDB running in B-tree mode, which is a standalone B-tree index sitting over a buffer pool cache that manages pages, representing a typical B-tree architecture. We use BerkeleyDB in non-transactional mode (meaning better performance) that supports a single writer and multiple readers using page-level latching (the lowest latch granularity in BerkeleyDB) to maximize concurrency. With this configuration, we believe BerkeleyDB represents a fair comparison to the Bw-tree. For all experiments, the BerkeleyDB buffer pool size is large enough to accommodate the entire workload (thus does not touch secondary storage).

Skip list. We also compare the Bw-tree to a latch-free skip list implementation [13]. The skip list has become a popular alternative to B-trees for memory-optimized databases[3] since they can be implemented latch-free, exhibit fast insert performance, and maintain logarithmic search cost. Our implementation installs an element in the bottom level (a linked list) using a CAS to change the pointer of the preceding element. It decides to install an element at the next highest layer (the skip list towers or "express lanes") with a probability of $\frac{1}{2}$. The maximum height of our skip list is 32 layers.

[3]The MemSQL in-memory database uses a skip list as its ordered index [14]

	Failed Splits	Failed Consolidates	Failed Updates
Dedup	0.25%	1.19%	0.0013%
Xbox	1.27%	0.22%	0.0171%
Synthetic	8.88%	7.35%	0.0003%

TABLE I
Latch-free delta update failures

Experiment machine. Our experiment machine is an Intel Xeon W3550 (at 3.07GHz) with 24 GB of RAM. The machine contains four cores that we hyperthread to eight logical cores in all of our experiments.

Evaluation Datasets. Our experiments use three workloads, two from real-world applications and one synthetic.

1) *Xbox LIVE*. This workload contains 27 Million `get-set` operations obtained from Microsoft's Xbox LIVE Primetime online multi-player game [15]. Keys are alphanumeric strings averaging 94 bytes with payloads averaging 1200 bytes. The read-to-write ratio is approximately 7.5 to 1.
2) *Storage deduplication trace*. This workload comes from a real enterprise deduplication trace used to generate a sequence of chunk hashes for a root file directory and compute the number of deduplicated chunks and storage bytes. This trace contains 27 Million total chunks and 12 Million unique chunks, and has a read to write ration of 2.2 to 1. Keys are 20-byte SHA-1 hash values that uniquely identify a chunk, while the value payload contains a 44-byte metadata string.
3) *Synthetic*. We also use a synthetic data set that generates 8-byte integer keys with a 8-byte integer payload. The workload begins with an index of 1M entries generated using a uniform random distribution. It performs 42 million operations with a read to write ratio of 5 to 1.

Defaults. Unless mentioned otherwise, our primary performance metric is throughput measured in (Million) operations per second. We use 8 worker threads for each workload, equal to the number of logical cores on our experiment machine. The default page size for both BerkeleyDB and the Bw-tree is 8K (the skip list is a linked list and does not use page organization).

B. Bw-Tree Tuning and Properties

In this section, we evaluate two aspects of the Bw-tree: (1) the effect of delta chain length on performance and (2) the latch-free failure rate for posting delta updates.

1) Delta chain length: Figure 5 depicts the performance of the Bw-tree run over the Xbox and synthetic workloads for varying delta chain length thresholds, i.e., the maximum length a delta chain grows before we trigger page consolidation. For small lengths, worker threads perform consolidation frequently. This overhead cuts into overall system performance. For the Xbox workload, search deteriorates for sequential scans larger than four deltas. While sequential scans

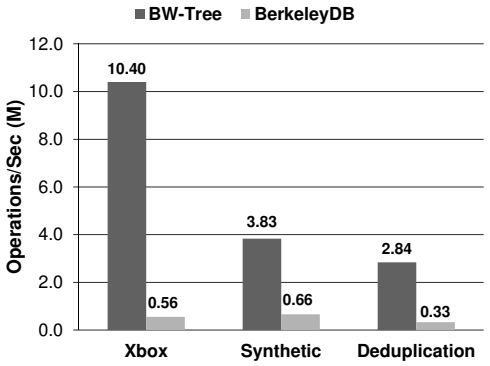


Fig. 6. Bw-tree and BerkeleyDB

over linked delta chains are good for branch prediction and prefetching in general, the Xbox workload has large 100-byte records, meaning fewer deltas will fit into the L1 cache during a scan. The synthetic workload contains small 8-byte keys, which are more amenable to prefetching and caching. Thus, delta chain lengths can grow longer (to about eight deltas) without performance consequences.

2) Update Failure in Latch-Free Environment: Given the latch-free nature of the Bw-tree, some operations will inevitably fail in the race to update page state. Table I provides the failure rate for splits, consolidates, and record updates for each workload. The record update failure rate (e.g., inserts, updates, deletes) is extremely low, below 0.02% for *all* workloads. Meanwhile, the failure rates for the split and consolidate operations are larger than the update failures at around 1.25% for both the Xbox and deduplication workloads, and 8.88% for the synthetic workload. This is expected, since splits and consolidates must compete with the faster record update operations. However, we believe these rates are still manageable. The synthetic workload failure rates represent a worst-case scenario. For the synthetic data, preparing and installing a record update delta is extremely fast since records are extremely small. In this case, a competing (and more expensive) split operation trying to install its delta on the same page has a high probability of failure.

C. Comparing Bw-tree to a Traditional B-tree Architecture

This experiment compares the in-memory performance of the Bw-tree to BerkeleyDB[4], representing a traditional B-tree architecture. Figure 6 reports the results for the Xbox, deduplication, and synthetic workloads run on both systems. For the Xbox workload, the Bw-tree exhibits a throughput of 10.4M operations/second, while BerkeleyDB has a throughput of 555K operations/second, representing a speedup of 18.7x. The throughput of both systems drops for the update-intensive deduplication workload, however the Bw-tree maintains a 8.6x speedup over BerkeleyDB. The performance gap is closer for the synthetic workload (5.8x Bw-tree speedup) due to the higher latch-free failure rates observed for the Bw-tree observed in Section VI-B.2.

[4]We use BerkeleyDB's **`memp_stat`** function to ensure it runs in memory

	Bw-tree	Skip List
Synthetic workload	3.83M ops/sec	1.02M ops/sec
Read-only workload	5.71M ops/sec	1.30M ops/sec

TABLE II
BW-TREE AND LATCH-FREE SKIP LIST

In general, we believe two main aspects lead to the superior performance of the Bw-tree: (1) *Latch-freedom*: no thread blocks on updates or reads on the Bw-tree, while BerkeleyDB uses page-level latching to block readers during updates, reducing concurrency. The Bw-tree executes with a processor utilization of about 99% while BerkeleyDB runs at about 60%. (2) *CPU cache efficiency*: since the Bw-tree uses delta records to update immutable base pages, the CPU caches of other threads are rarely invalidated on an update. Meanwhile, BerkeleyDB updates pages in place. An insert into a typical B-tree page involves including a new element in vector of key-ordered records, on average moving half the elements and invalidating multiple cache lines.

D. Comparing Bw-tree to a Latch-Free Skip List

We also compare the performance of the Bw-tree to a latch-free skip list implementation. The skip list provides key-ordered access with logarithmic search overhead, and can easily be implemented in a latch-free manner. For these reasons, it is starting to receive attention as a B-tree alternative in memory-optimized databases [14]. The first row of Table II reports the results of the synthetic workload run over both the Bw-tree and the latch-free skiplist. The Bw-tree outperforms the skip list by a factor of 3.7x. To further investigate the source of the Bw-tree performance gain, we ran both systems using a *read-only* workload that performs 30M key lookups on an index of 30M records. We report these results in the second row of Table II, showing the Bw-tree with a 4.4x performance advantage. These results suggest the Bw-tree has a clear advantage in search performance. We suspect this is due to the Bw-tree having better CPU cache efficiency than the skip list, which we explore in the next section.

E. Cache Performance

This experiment measures the CPU cache efficiency of the Bw-tree compared to the skip list. We use the Intel VTune profiler[5] to capture CPU cache hit events while running the read-only workload on each system. The workload was run single-threaded in order to collect accurate statistics. Figure 7 plots the distribution of retired memory loads over the CPU cache hierarchy for each system. As we expected, the Bw-tree exhibits very good cache efficiency compared to the skip list. Almost 90% of its memory reads come from either the L1 or L2 cache, compared to 75% for the skip list. We suspect the Bw-tree's search efficiency is the primary reason for this difference. The skip list must traverse a physical pointer before *every* comparison to traverse to the next "tower" or linked list record at a particular level. This can cause erratic CPU cache behavior, especially when branch prediction fails. Meanwhile,

[5]http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

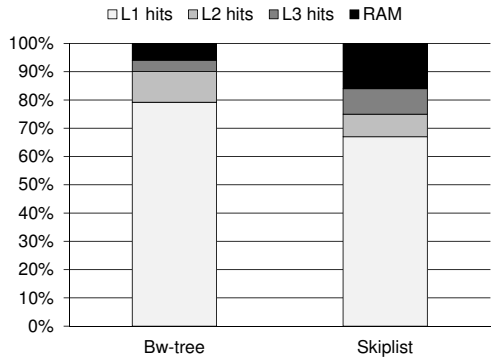


Fig. 7. Cache efficiency

Bw-tree search is cache-friendly since most of its time is spent performing binary search on a base page of keys compacted into a contiguous memory block (a few deltas may be present as well). This implies the Bw-tree will do much less pointer chasing than the skip list.

The cache friendliness of the Bw-tree becomes clearer when we consider the set of nodes accessed in each data structure up to the point of reaching the data page. *Over all possible search pathways*, this is precisely the set of internal nodes in the Bw-tree, which is 1% or less of the occupied memory. For the skip list, this is about 50% of the nodes. So, for a given cache size, more of the memory accesses involved in a search hit the cache for Bw-tree over skip list.

VII. Related Work

We have benefitted from prior work and in some cases have built upon it.

A. B-trees

Anyone working with B-trees owes a debt to Bayer and McCreight [16]. The variant that we use is the B+tree [9]. We stretch the meaning of "page-oriented" by exploiting elastic pages, in discontiguous pieces. The notion of elastic pages evolved from the hashed access method record lists used by SkimpyStash [8]. Like SkimpyStash, we consolidate the discontiguous pieces of a "page" when we do garbage collection.

B. Latch Free

Until our work, it was not clear that B-trees could be made lock or latch-free. Skip lists [13] serve as an alternative latch-free "tree" when multiple threads must access the same page. Partitioning is another way to avoid latches so that each thread has its own partition of the key space [17], [18]. The Bw-tree avoids the overhead of partitioning by using the compare and swap (CAS) instruction for all state changes, including the "management" state changes, e.g., SMOs and flushes

C. Flash Based, Log Structured

Early work with flash storage exploited in-page logging [19], placing log records near a B-tree page instead of updating the page "in-place" which would require an erase cycle. While our delta records resemble somewhat in-page log records, they have nothing to do with transactional recovery. We view them not as log records, but as the new versions of the records. Transactional log records, if they exist at all, are elsewhere. We gain our latch-free capability from these deltas as well.

We have treated our flash SSD as a generic storage device. Hence, we have not tried to exploit parallelism within it, as done by [20]. Exploiting this parallelism might well provide even higher performance than we have achieved thus far.

Log structuring was first applied to file systems [12]. But the approach is now more widely used for both disks and as the translation layer for flash. In our system, we use solid state disks, i.e. flash based devices that are accessed via the traditional I/O interface and use a translation layer. Despite this, we implemented our own log structured store. This enabled us to pack pages together in our buffer so that no empty space exists on flash. Further, it permitted us to blur the distinction between a page store and a record store. This reduces even further the storage consumed in flushing pages. Storage savings improve LSS performance as garbage collection, which consumes cycles, amplifies writes, and wears out flash, is the largest cost in log structuring.

D. Combination Efforts

Earlier indexes have used some of the elements above, though not in exactly the same way. Hyder [21] uses a log structured store built on flash memory. In Hyder, all changes propagate to the root of the tree. The changes at the root are batched, and the paths being included are compressed. Hyder, at least in one mode, supports transactions directly, with their log structured store used as both the database and the log. We do not do that, relying instead on a transactional component when transaction support is desired.

BFTL [22] is another example of an index implemented over log structured flash. It is perhaps the closest to the Bw-tree among earlier work in how it's b-tree manages storage. It has a mapping table, called a node translation table, and writes deltas (in our terminology) to flash. It also has a process for making the deltas contiguous on flash when the number of deltas gets large. However, the BTFL b-tree does not handle multi-threading, concurrency control and cache management, topics that we view as crucial to providing high throughput performance for an atomic record store.

VIII. Discussion

A. Performance Results

Our Bw-tree implementation achieves very high performance. And it is sufficiently complete that we can measure normal operation performance reliably and consistently. We exploit the database ability to cache updates until it is convenient to post them to stable storage. This is a key to database system performance, and is explicitly enabled via control operations in the Deuteronomy architecture's interface between TC and DC. However, exploiting it, as we have done, means that one needs to be careful in comparing Bw-tree performance with atomic record stores that immediately make update operations stable.

B. Log Structured Store

We have focused this paper on the Bw-tree part of our atomic record store (ARS) or data component (DC). Thus, we have not discussed issues related to managing LSS. A latch-free environoment is very challenging for many elements of data management systems, including storage management as done by our LSS. We will describe details of how the LSS is implemented in another paper. Here we want to provide a few highlights of areas needing attention.

- We manage our flush buffer in a latch-free way. To our knowledge this has not been done before. This means that there is no thread blocking when items are posted to the buffer.
- We use system transactions to capture Bw-tree structure modification operations (SMOs). This ensures that each SMO can be considered atomic during recovery.
- We must keep main memory state consistent with the state being written to LSS, non-trivial without latches. We gave much thougth to this subtle aspect in designing the interface to the LSS.

C. In-Page Search Optimization

Our excellent performance results were achieved without tuning search performance using a cache sensitive page search technique [23], [24]. We expect further improvement in Bw-tree search performance as a result of implementing techniques like these. We have no concerns about update performance, as we would continue to use our delta record technique for that, with all of its advantages.

D. Conclusion

We have designed and implemented the Bw-tree such that it can exist as a free standing atomic record store, be a data component in a Deuteronomy style system, or be embedded in a traditional database system. We have followed the classic architecture of access method layered on cache manager layered on storage manager. But at every level, we have introduced innovations that stretch prior methods and tailor our system for the newer hardware setting of multi-core processors and flash storage.

Our innovations, e.g. elastic pages, delta updates, shared read-only state, latch-free operation, and log structured storage, eliminate thread blocking, improve processor cache effectiveness, and reduce I/O demands. These techniques should work well in other settings as well, including hashing and multi-attribute access methods.

We were acutely aware that we were implementing a component that had been successfully implemented any number of times. In such a case, when all is said and done, it is system performance that determines whether the effort bears fruit. While we were "confident" in our design choices, we were nonetheless pleasantly surprised by how good the results were. Supporting millions of operations per second on a single "vanilla" cpu offers strong confirmation for our design choices. That we out-performed very good competing approaches by such a large margin was "icing on the cake".

REFERENCES

[1] "MongoDB. **http://www.mongodb.org/**."

[2] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao, "Deuteronomy: Transaction Support for Cloud Data," in *CIDR*, 2011, pp. 123–133.

[3] D. Lomet, A. Fekete, G. Weikum, and M. Zwilling, "Unbundling Transaction Services in the Cloud," in *CIDR*, 2009, pp. 123–133.

[4] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet, "AlphaSort: A Cache-Sensitive Parallel External Sort," *VLDB Journal*, vol. 4, no. 4, pp. 603–627, 1995.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?" in *VLDB*, 1999, pp. 266–277.

[6] "Amazon DynamoDB. **http://aws.amazon.com/dynamodb/**."

[7] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09, 2009, pp. 10:1–10:9.

[8] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage," in *SIGMOD*, 2011, pp. 25–36.

[9] D. Comer, "The Ubiquitous B-Tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.

[10] H. T. Kung and P. L. Lehman, "Concurrent manipulation of binary search trees," *TODS*, vol. 5, no. 3, pp. 354–382, 1980.

[11] P. L. Lehman and S. B. Yao, "Efficient Locking for Concurrent Operations on B-Trees," *TODS*, vol. 6, no. 4, pp. 650–670, 1981.

[12] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[13] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[14] "MemSQL Indexes. **http://developers.memsql.com/docs/1b/indexes.html**."

[15] "Xbox LIVE. **http://www.xbox.com/live**."

[16] R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta Inf.*, vol. 1, no. 1, pp. 173–189, 1972.

[17] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki, "PLP: Page Latch-free Shared-everything OLTP," *PVLDB*, vol. 4, no. 10, pp. 610–621, 2011.

[18] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors," *PVLDB*, vol. 4, no. 11, pp. 795–806, 2011.

[19] S.-W. Lee and B. Moon, "Design of Flash-Based DBMS: An In-Page Logging Approach," in *SIGMOD*, 2007, pp. 55–66.

[20] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee, "B+-tree Index Optimizations by Exploiting Internal Parallelism of Flash-based Solid State Drives," *PVLDB*, vol. 5, no. 4, pp. 286–297, 2012.

[21] P. A. Bernstein, C. W. Reid, and S. Das, "Hyder - a transactional record manager for shared flash," in *CIDR*, 2011, pp. 9–20.

[22] C.-H. Wu, T.-W. Kuo, and L. P. Chang, "An Efficient B-tree Layer Implementation for Flash-Memory Storage Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, July 2007.

[23] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, "Fractal Prefetching B$^+$Trees: Optimizing Both Cache and Disk Performance," in *SIGMOD*, 2002, pp. 157–168.

[24] D. B. Lomet, "The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account," *SIGMOD Record*, vol. 30, no. 3, pp. 64–69, 2001.