LLAMA: A Cache/Storage Subsystem for Modern Hardware

Justin Levandoski
Microsoft Research
One Microsoft Way
Redmond, WA 98052
justinle@microsoft.com

David Lomet
Microsoft Research
One Microsoft Way
Redmond, WA 98052
lomet@microsoft.com

Sudipta Sengupta
Microsoft Research
One Microsoft Way
Redmond, WA 98052
sudipta@microsoft.com

ABSTRACT

LLAMA is a subsystem designed for new hardware environments that supports an API for page-oriented access methods, providing both cache and storage management. Caching (CL) and storage (SL) layers use a common mapping table that separates a page's logical and physical location. CL supports data updates and management updates (e.g., for index re-organization) via latch-free compare-and-swap atomic state changes on its mapping table. SL uses the same mapping table to cope with page location changes produced by log structuring on every page flush. To demonstrate LLAMA's suitability, we tailored our latch-free Bw-tree implementation to use LLAMA. The Bw-tree is a B-tree style index. Layered on LLAMA, it has higher performance and scalability using real workloads compared with BerkeleyDB's B-tree, which is known for good performance.

1 INTRODUCTION

1.1 Modern Architectures

Modern computer platforms have changed sufficiently that it is timely to re-architect database systems (DBMSs) to exploit them [1, 10]. These DBMSs have not slowed down, but rather they are missing significant opportunities to perform dramatically better. Without re-architecting in a substantial way, these performance opportunities will continue to elude them.

CPU changes include multi-core processors and main memory latency that requires multiple levels of caching. Flash storage, and hard disk vendor recognition that update-in-place compromises capacity, have increased the use of log structuring. Cloud data centers increase system scale and the use of commodity hardware puts increased emphasis on high availability techniques.

Previous Deuteronomy work [15, 19] described how to provide consistency (i.e. transactions) in a cloud setting. We focus here on a Deuteronomy data component (DC) and how to maximize its performance on modern hardware. A DC manages storage and retrieval of data accessed via CRUD (*c*reate, *r*ead, *u*pdate, *d*elete) [32] atomic operations. A DC is not distributed but rather is a local mechanism that can be amalgamated into a distributed system via software layers on top of it, e.g. a Deuteronomy transactional component (TC) and/or a query engine.

Proceedings of the VLDB Endowment, Vol. 6, No. 10

We believe there are fundamental problems posed by current hardware that impact all access methods: B-trees, hashing, multi-attribute, temporal, etc. Further, these problems can be solved with general mechanisms applicable to most access methods.

1. Good processor utilization and scaling with multi-core processors via latch-free techniques.
2. Good performance with multi-level cache based memory systems via delta updating that reduces cache invalidations.
3. Write limited storage in two senses: (1) limited performance of random writes; (2) flash write limits; via log structuring.

The Bw-tree [16], an index resembling B-trees [4, 7], is an example of a DC or key-value store that exploits these techniques. Indeed, it is an instance of a paradigm for how to achieve latch-freedom and log structuring more generally. In this paper, we describe a new architecture where the latch-free and log-structure techniques of the Bw-tree are implemented in a cache/storage subsystem capable of supporting multiple access methods, in the same way that a traditional cache/storage subsystem deals with latched access to fixed size pages that are written back to disks as in-place updates.

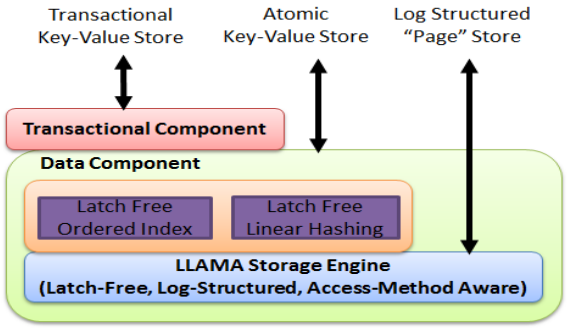

Figure 1: Architectural Layering for LLAMA

1.2 Brief Description of LLAMA

LLAMA and its capabilities, described below, are the contributions of our paper. While LLAMA's "big picture" architecture is similar to that of conventional systems, it introduces new techniques throughout that make it uniquely suitable for new platforms.

Like conventional cache/storage layers, LLAMA supports a page abstraction to support access method implementations. Further, a Deuteronomy-style transactional component can be added on top. This architectural layering is illustrated in Figure 1. A page is accessed via a "mapping table" that maps page identifiers (PIDs) to states, either in main memory cache or on secondary storage (see Figure 2). Pages are read from secondary storage into a main memory cache on demand, they can be flushed to secondary storage, and they are updated to change page state while in the cache. All page state changes (both data state and management state) are provided as atomic operations.

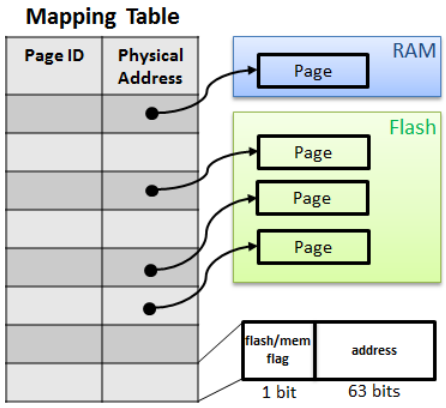

Figure 2: The mapping table in LLAMA

LLAMA, through its API, provides latch-free page updating via a compare and swap (CAS) atomic operation on the mapping table. This replaces the traditional latch that guards a page from concurrent access by blocking threads. The CAS strategy increases processor utilization and improves multi-core scaling.

In managing the cache, LLAMA can reclaim main memory by dropping only previously flushed portions of pages from memory, thus not requiring any I/O, even when swapping out "dirty" pages. This means that LLAMA can control its buffer cache memory size without input from its access method user. This is important as LLAMA is unaware of transactions and write-ahead logging.

LLAMA uses log-structuring to manage secondary storage. This provides the usual advantages of avoiding random writes, reducing the number of writes via large multi-page buffers, and wear leveling needed by flash memory. Further, LLAMA improves performance compared with conventional log structuring with partial page flushes and pages with no empty space- i.e. 100% utilization. These reduce the number of I/Os and storage consumed per page when a page is flushed, and hence reduces the write amplification usually encountered when log-structuring is used. Further all storage related operations are completely latch-free.

Finally, LLAMA supports a limited form of system transaction [19]. System transactions are not user transactions, but rather provide atomicity purely for the "private use" of the access method, e.g., for index structure modifications (SMOs) [21]. The fact that system transactions recorded separately from the transaction log can be effective is one of the key insights of the Deuteronomy approach to decomposing a database kernel.

1.3 Outline for Paper

The rest of the paper is organized as follows. We introduce the operational interface (API) that an access method implementer sees when using LLAMA and describe how it might be used in section 2. Section 3 describes the cache layer. The design of the log structured storage layer is described in Section 4. Section 5 describes our system transaction mechanism and the measures we take to provide atomicity. Section 6 discusses log structured storage recovery from system crashes. We describe our performance experiments and present their results in Section 7. Section 8 describes related work while section 9 provides a brief discussion and some conclusions about the work.

2 LLAMA INTERFACE

The design goal of LLAMA is to be as general purpose as possible. That sometimes turns into "be as low level" as possible. But that is not our intent. Rather, for LLAMA to be general purpose, it needs to know as little as possible about what an access method does in using its facilities. Thus, LLAMA operations are "primitive", targeted at cache management and the updating of pages. It has some additional facilities to support a primitive transaction mechanism that is needed for SMOs (e.g. page splits and merges).

There is nothing in the interface about LSNs, write-ahead logging or checkpoints for transaction logs. There is no idempotence test for user operations. Indeed, there is no transactional recovery in this picture. That is handled by an access method using LLAMA. We describe how the Bw-tree uses LLAMA in section 2.4. But first we describe the operations that LLAMA supports.

2.1 Page Data Operations

An access method changes state in response to user operations. A user may want to create (C), read (R), update (U), or delete (D) a record (CRUD operations [31]). LLAMA does not directly support these operations. Rather, the access method needs to implement them as updates to the states of LLAMA pages.

There are also structure changes that are part of access method operation. For example, a Bw-tree page split involves posting a split delta to the original page *O* so that searchers know that a new page now contains data for a sub range of the keys in *O*. These too are handled as updates to a LLAMA page *O*.

LLAMA supports two forms of update, a delta update, and a replacement update. An access method can choose to exploit these forms of updates as it sees fit. For example, the Bw-tree will make a series of delta updates and at some point decide to "consolidate" and optimize the page by applying the delta updates to a base page. It then uses a replacement update to create the new base page.

1. **Update-D(PID, in-ptr, out-ptr, data):** A delta update prepends a delta describing a change to the prior state of the page. For the Bw-tree, the "data" parameter to Update-D includes at least <lsn, key, data> where the lsn enables idempotence. The "in-ptr" points to the prior state of the page. The "out-ptr" points to the new state of the page.
2. **Update-R(PID, in-ptr, out-ptr, data):** A replacement update results in an entirely new state for the page. The prior state, preserved when using an Update-D, is replaced by the "data" parameter. Thus, the "data" parameter must contain the entire state of the page with deltas "folded in".
3. **Read(PID, out-ptr):** Read returns, via "out-ptr" the address in main memory for the page. If the page is not in main memory, then the mapping table entry contains a secondary storage address. In that case, the page will be read into main memory and the mapping table updated with the new address.

2.2 Page Management Operations

In addition to supporting data operations, LLAMA needs to provide operations to manage the existence, location, and persistence of pages. To adjust to the amount of data stored, the access method will add or subtract pages from its managed collections. To provide state persistence, an access method will from time to time flush pages to secondary storage. To manage this persistence effectively, pages need to be annotated appropriately, e.g. with log sequence numbers.

1. **Flush(PID, in-ptr, out-ptr, annotation):** Flush copies the page state into the log structured store (LSS) I/O buffer. Flush is similar to Update-D in its impact on main memory, i.e. it prepends a delta (with an annotation) to the prior state. This delta is tagged as a "flush". LLAMA stores the LSS secondary storage address of the page and the caller "annotation" in the flush delta. Flush does not guarantee that the I/O buffer is stable when it returns.
2. **Mk-Stable(LSS address):** Mk-Stable ensures that pages flushed to the LSS buffer, up to the LSS address argument, are stable on secondary storage. When Mk-Stable returns, the LSS address provided and all lower LSS addresses are guaranteed to be stable on secondary storage.
3. **Hi-Stable(out-LSS address):** Hi-Stable returns the highest LSS address that is currently stable on secondary storage.
4. **Allocate(out-PID):** Allocate returns the PID of a new page allocated in the mapping table. All such pages need to be remembered persistently, so Allocate is always part of a system transaction (see 2.3 below), which automatically flushes its included operations.
5. **Free(PID):** Free makes a mapping table entry identified by the PID available for reuse. In main memory, the PID is placed on the pending free list for PIDs for the current epoch. We discuss epochs in section 3.4. Again, because active pages need to be remembered, Free is always part of a system transaction.

2.3 System Transaction Operations

LLAMA system transactions are used to provide relative durability and atomicity (all or nothing) for structure modifications (SMOs [20]). We use our LSS and its page oriented records as our "log records". All operations within a transaction are automatically flushed to an in-memory LSS I/O buffer in addition to changing page state in the cache. Each LSS entry includes the state of a page as our LSS is strictly a "page" store.

In main memory, all such operations within a transaction are held in isolation until transaction commit (details are described in Sections 5.2 and 5.3.) At commit, all page changes in the transaction are flushed atomically to the LSS buffer. On abort, all changes are discarded.

System transactions are initiated and terminated in a conventional way via LLAMA supported operations.

1. **TBegin(out-TID):** A transaction identified by a TID is initiated. This involves entering it into an active transaction table (ATT) maintained by the LLAMA CL.
2. **TCommit(TID):** The transaction is removed from the active transaction table and the transaction is committed. Page state changes in the transaction are installed in the mapping table and flushed to the LSS buffer.
3. **TAbort(TID):** The transaction is removed from the active transaction table and changed pages are reset to transaction begin in the cache and no changes are flushed.

In addition to Allocate and Free, **Update-D** operations are permitted within a transaction to change page states. **Update-R** is not used as it complicates transaction undo (see section 5.3).

Transactional operations all have input parameters: TID and annotation. TID is added to the deltas in the cache, and an annotation is added to each page updated in the transaction, as if it were being flushed. When installed in the flush buffer and committed, all updated pages in the cache will have flush deltas prepended describing their location, as if they were flushed independently of a transaction.

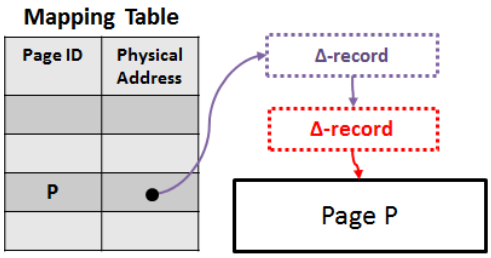

Figure 3: Installing delta updates in the mapping table.

2.4 Using the Interface

The Bw-tree [16] provides a key-value store that enables transactions to be supported. It manages LSNs, enforces the WAL protocol, and responds to checkpointing requests as required of a Deuteronomy data component (DC) [15, 19]. Here, we address how it does that when using LLAMA.

What is data to the Update-D and Update-R LLAMA operations can include keys, LSNs, and the data part of a key value store. The Bw-tree can thus, via these operations, implement a key value store, provide idempotence via LSNs, do incremental updates via Update-D, do its page consolidations via Update-R, and access pages for read or write using the LLAMA Read or Flush operation.

An access method can store LSNs in the data it provides to LLAMA via update operations. Further, the Flush operation annotation stored in a flush delta can provide additional information to describe page contents. These permit the Bw-tree to enforce the write-ahead logging. A Stabilize operation after flushing a page makes updates stable for transaction log checkpointing.

Allocate and Free permit the Bw-tree to grow and shrink its tree. BTrans and TCommit/TAbort enable the atomicity required for structure modifications operations (SMOs). Update operations are not limited to "user level" data. For example, the Bw-tree uses Update-D to post its "merge" and "split" deltas when implementing SMOs. We provide more detail about this when discussing system transactions in Section 5.

3 CACHE LAYER

3.1 Data Operations

All page updating is accomplished by installing a new page state pointer in the mapping table using a CAS, whether a delta or a replacement update (see Figure 3). A replacement update must include both the desired new state and the location of the prior state of the page in LSS. A new update delta will point to the prior state of the page, which already includes this LSS location.

This latch-free approach avoids the delays introduced by latching, but it has a penalty of its own, as do all "optimistic" concurrency control methods, i.e., the CAS can fail and the update then must be re-attempted. It is up to the LLAMA user to retry its operation as appropriate. LLAMA merely indicates when a failure occurs.

While no operation blocks when the data is in cache, reading a page from secondary storage has to wait for the page to appear in the cache. The mapping table will always point to the LSS page, even for cached pages (see above), enabling pages to be moved between cache and LSS for effective cache management.

3.2 Flushing Pages

When a page is flushed, LLAMA ensures that what is represented in the cache matches what is in LSS. Thus, the flush delta includes both PID and LSS offset, and LLAMA includes that delta in the LSS buffer and in the cache by prepending it to the page.

3.2.1 Non-contiguous Pages

Because LLAMA supports delta updating, it is possible that page state will consist of several non-contiguous pieces. Combine this with flushing activity and an in-cache page may have part of its state in LSS (having been flushed earlier) while recent updates are only present in the cache. When this occurs, it provides an opportunity to reduce the storage cost of the next flush.

Thus, LLAMA can flush such a page by writing a delta that contains only the changes since the prior flush. Multiple update deltas in the cache can all be made contiguous for flushing by writing a contiguous form of the deltas (called a C-delta), with a pointer to the remainder of the page in LSS. Thus the entire page is accessible in LSS, but in possibly several pieces.

The Flush operation sees a cached page state that may have several parts that have been flushed over time in this way, resulting in a cached page in which the separate pieces and their LSS addresses are represented. At any time, Flush can bring these pieces together in LSS storage by writing everything contiguously (and redundantly). One might be willing to leave the pieces separate when LSS uses flash storage, while wanting contiguity when LSS uses disk storage, due to the differing read access costs.

3.2.2 A Problem

When we flush a page, we must know, prior to the flush, exactly what state of the page we are flushing. This is trivial with latches: one simply latches the page, does the flush. But in a latch free approach, we have no way to prevent updates to a page being flushed. This poses a problem when we are trying to enforce the write-ahead log protocol or when the flush occurs as part of a structure modification. We want inappropriate flushes to fail when they perform their CAS. Thus, we use the pointer to the page state we wish to flush in the CAS, which will then only capture that particular state and will fail if the state has been updated before the flush completes. But this raises another problem.

We found it surprisingly hard to provide the kind of strong invariant that we think is needed when doing cache management and flushing pages to LSS. What we want is:

(*) *A page that is flushed successfully to LSS is immediately seen in the cache as having been flushed and the flushed state of the page must indeed be in the LSS I/O buffer ahead of the flushes of all later states. A page whose flush has failed should not be seen as flushed in the cache and it should be clear when looking at LSS that the flush did not succeed.*

Consider two alternative approaches.

1. We ensure that the flush succeeds by first performing the CAS. Once the CAS succeeds, we post the page to the LSS. If we do that, a race condition undermines correct LSS recovery. We can subsequently flush a page that depends upon the earlier flush, where this "later" flush succeeds in writing to LSS before a system crash, while our "earlier" flush is too slow to complete and does not appear in the stable LSS. We have just compromised a form of causality.
2. We capture the page state that we wish to flush and write it to the LSS buffer. Then we attempt the CAS, which fails. We now have a page written to LSS with no way to distinguish whether the flush succeeded or failed should the system crash. Indeed, we can have multiple such pages written to LSS at various times. It is even possible that we have written a later state of the page that is earlier in the LSS than our failed CAS. It began later but got its buffer slot before the earlier flush.

3.2.3 A Solution

This dilemma briefly confounded us, but there is a way out. The CAS needs to be done early enough that we can know whether we will successfully flush or not—prior to copying the state of the page to the log buffer. Thus, our flush procedure is as follows.

1. Identify the state of the page that we intend to flush.
2. Seize space in the LSS buffer into which to write the state.
3. Perform the CAS to determine whether the flush will succeed. We need the LSS offset in the flush delta when we do this. But step 2 has provided us with that.
4. If step 3 succeeds, write the state to be saved into the LSS. While we are writing into the LSS, LLAMA prevents the buffer from being written to LSS secondary storage.
5. If step 3 fails, write "Failed Flush" into the reserved space in the buffer. This consumes storage but removes any ambiguity as to which flushes have succeeded or failed.

The result of this is that the LSS, during recovery, never sees pages that are the result of CASs that have failed. This preserves also the property that any page that appears later in the LSS (in terms of its position in the "log") will be a later state of the page than all earlier instances of the page in the LSS log.

3.3 Swapping Out Pages

We want LLAMA to manage the cache and effectively swap out data so as to meet its memory constraints. LLAMA knows about delta updates, replacement updates, and flushes and can recognize each of these. But LLAMA must know nothing about the contents of the pages if it is to be general purpose. Importantly, this means that the LLAMA knows nothing about whether the access method layer is supporting transactions by maintaining LSNs in the pages. So the problem becomes: how does LLAMA provide cache space management (including evicting pages) when it cannot see LSNs and enforce the write-ahead log protocol?

The important observation is that any data that has already been flushed can be dropped from the cache. Systems in which pages are updated in place are prevented from swapping out (dropping from the cache) any recently updated and dirty page. But because of delta updates, LLAMA can determine which **parts of pages** have already been flushed. Each such part is described with a flush delta. And those flushed parts can be "swapped out" of the cache.

We cannot, in "swapping out" parts of pages, simply deallocate the storage and reuse it. Doing that leaves dangling references to the swapped out parts. We need a delta describing what parts of a page have been swapped out.

For a fully swapped out page, we replace its main memory address in the mapping table with an LSS pointer from the page's most recent flush delta. For partially swapped out pages, we use a CAS to insert a "partial swap" delta record. This delta record indicates that the page has been partially swapped out (so none of the page can be accessed normally) and points to the flush delta that designates where in the LSS to find the missing part of the page.

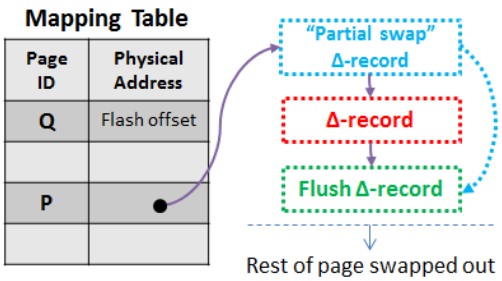

Figure 4: Swapped out page Q and a partially swapped P

Once the "partial swap" delta has been installed with a CAS, the memory for the part of the page being dropped can be freed using our epoch mechanism in Section 3.4. Partial page swap out and the partial swap delta are illustrated in Figure 4.

The approach described here has a number of advantages.

1. LLAMA's cache layer can reclaim memory without knowing anything about the content of pages.
2. Dropping flushed pages and flushed parts of pages requires no I/O operation.
3. Bringing a partially flushed page back into main memory involves fewer LSS reads than would be the case for a fully flushed page with multiple parts in LSS.

Several cache management strategies can be used to manage cache storage, e.g. LRU, LRU(k), Clock, etc. [9, 24]. These frequently require some additional bookkeeping, but pose no large difficulty.

3.4 Epochs for Resource Reuse

With our latch-free approach, operations can be examining both pages and page states even after they have been designated as "garbage". There are no "latches" that prevent either of:

1. An Update-R operation replaces the entire page state, de-allocating prior state while another operation is reading it.
2. A De-allocate operation "frees" a page in the mapping table while another operation is examining it.

We cannot allow storage or PIDs to be reused until there is no possibility that another operation is accessing them. Thus we distinguish between a freed and a re-usable resource. A freed resource has been designated as garbage by an operation. A re-usable resource has been freed and can be guaranteed not to be accessible by any other operation. Epochs are a way of protecting de-allocated objects from being re-used too early [12].

Every operation enrolls in the current epoch E prior to accessing PIDs or page states, and exits E once such access is over. An operation always posts freed resources on the list of the *current* epoch, which may be E (the epoch it joined), or a larger epoch if the current epoch has advanced. No resource on E's list is reused until all operations enrolled in E have exited.

Epochs are numbered and from time to time, a new epoch E+1 becomes the current epoch. New operations continue to enroll in the current epoch, now E+1. The epoch mechanism invariant is:

- *No operation in epoch E+1 or later epochs can have seen and be using resources freed in epoch E.*

Thus, once all operations have exited from E, no active operation can access resources freed in E. Two epochs and their garbage collection lists are illustrated in Figure 5.

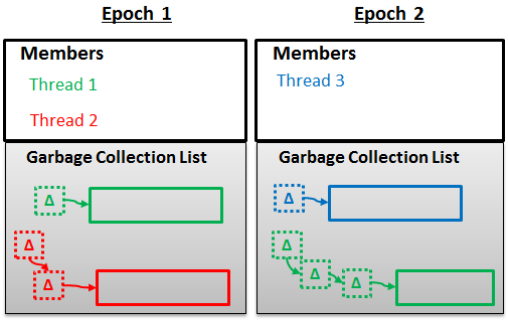

Figure 5: The epoch mechanism for garbage collection

4 STORAGE LAYER

4.1 Log Structured Storage Organization

LLAMA organizes data on secondary storage (flash in our case) in a log structured manner [27] similar to a log structured file system (LFS). Thus, each page flush relocates the position of the page on flash. This provides an additional reason for using our mapping table. Log structured storage has the substantial advantage of greatly reducing the number of writes per page and makes the writes "sequential". That is, it converts many random writes into one large multi-page write.

As discussed in Section 3.1, a logical page consists of a base page and zero or more delta records reflecting updates to the page. This allows a page to be written to flash in pieces when it is flushed. Thus, a logical page on flash corresponds to records on possibly different physical device blocks that are linked together using file offsets as pointers. Further, a physical block may contain records from multiple logical pages. This is illustrated in Figure 6. These are important differences between LLAMA and conventional LFS systems, enabling LLAMA to write a page using less storage, and hence with less write amplification.

A logical page is read from flash into memory by starting from the head of the chain on flash (whose offset is obtained from the mapping table) and following the linked records. The read process is made more efficient by consolidating multiple delta records of the same logical page into a contiguous C-delta on flash when they

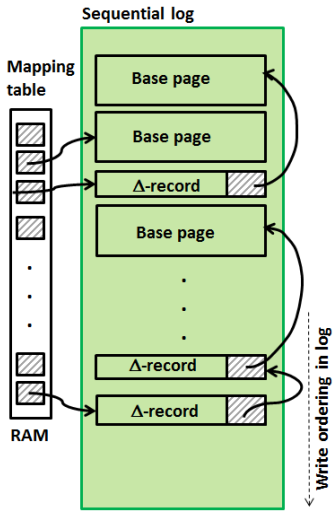

Figure 6: Log-structured storage organization on flash

are flushed together. Moreover, a logical page will get consolidated on flash when it is flushed after being consolidated in memory. These techniques improve page read performance.

4.2 Flushing

LLAMA is entirely latch-free and asynchronous. Further, we do not use dedicated threads to flush I/O buffer as this makes it harder to keep thread workload balanced. So all threads participate in managing this buffer. Most prior approaches have required latches. The best only latch while allocating space in the buffer, releasing the latch prior to data transfers, which can then proceed in parallel. We take this same basic approach but without using latches. Succeeding in this requires solving a number of problems.

4.2.1 Buffer Space Allocation

Our approach avoids the prior latch, using a CAS for atomicity, as we have done elsewhere in our system. This requires that we define the state on which the CAS executes. The constant part of buffer state consists of its address (Base) and size (Bsize). We keep track of the current high water mark of storage used in the buffer with an Offset relative to the Base. Each request for the use of the buffer begins with an effort to reserve space Size for a page flush.

To reserve space in the buffer, a thread acquires the current Offset and computes Offset+Size. If Offset+Size ≤ Bsize then the request can be stored in the buffer. The thread issues a CAS with current Offset as the comparison value, and Offset+Size as the new value. If the CAS succeeds, Offset gets the new value, the space is reserved, and the buffer writer can transfer data to the buffer.

This logic deals only with space allocation in the buffer. Writing the buffer and how to manage multiple buffers requires more in the CAS state, and we describe this next.

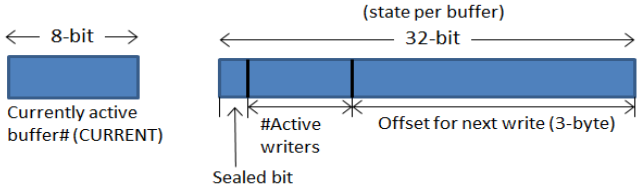

Figure 7: Flush buffer state

4.2.2 Writing the Buffer to Secondary Storage

If Offset+Size > Bsize, there is insufficient space in the buffer to hold the thread's record. At that point, the thread seals the buffer—marking it as no longer to be used and as prepared to be written to secondary storage. This is tracked with a Sealed bit in the flush buffer state. A CAS changes Sealed from F (false) to T (true). A sealed buffer can no longer be updated and a thread encountering a sealed buffer must find a different (unsealed) buffer.

A sealed buffer can no longer accept new update requests. But we cannot yet be sure that the prior writers, all of whom have succeeded in acquiring buffer space, have finished transferring their data to the buffer. An Active count indicates the number of writers transferring data to the buffer. When reserving space in the buffer, the writer's CAS includes Offset, Sealed, and Active. It acquires this structure, adds its payload size to Offset, increments Active by 1, and if ~Sealed, does a CAS to update this state and reserve space. When a writer is finished, it reacquires this state, decrements Active by one, and does a CAS to effect the change. Operations are redone as needed in case of failure.

A buffer is flushable if it is Sealed and Active = 0. The writer who causes this condition is responsible for initiating the asynchronous I/O, i.e. the thread does not wait. When the I/O is completed, the buffer's Offset and Active users are both set to zero, and the buffer is unSealed.

4.2.3 Multiple Buffers

Each of the buffers in a set has a state as indicated above. We assume that buffers are accessed and used in a round-robin style, such that as one buffer is sealed, we can simply proceed to the next buffer in the buffer "ring". We use CURRENT to indicate which of a set of buffers is currently accepting new write requests.

The thread that SEALs a currently active buffer must also update CURRENT when it SEALs the buffer. This thread then chooses the next CURRENT buffer. When a buffer I/O completes, the I/O thread unseals the buffer but does not set CURRENT as there may be another buffer serving as the current buffer. The complete flush buffer state is illustrated in Figure 7.

4.3 LSS Cleaning

LSS is a log structured store and so is conceptually "append only". To realize LSS, one must be continuously reclaiming space for the appending of new versions of pages, as any log structured file system (LFS) must. This is called "cleaning" [27].

Because versions of pages have different lifetimes, it is possible that very old parts of our "log", which we would like to reuse, will contain current page states. To reuse this "old" section of our log, we need to move the still current page states to the active tail of the log, appending them there so that the old part can be recycled for subsequent use. This side effect of cleaning increases the number of writes (called write amplification) [11].

We organize our cleaning effort very simply. We manage the log as a large "circular buffer" in which the oldest part (head of the log) is "cleaned" and added as new space at the active tail of the log where new page state is written. This is entirely conventional LFS. What is not so conventional are:

1. What we rewrite when a page is re-appended to the LSS store. Every page that is relocated is made contiguous when it is re-written. That is, as many incremental flushes as it may have had, all parts of the page are now made contiguous. This optimizes the accessibility of the page in LSS.
2. How we manage the cache so as to install the new location information. We use our usual technique, i.e., a CAS on a delta (called a relocation delta) at the mapping table entry for the page, providing the new location and describing which parts of the page have been relocated. A concurrent update or flush can cause this CAS to fail, in which case we try again.

4.4 Storage Efficiency

Storage efficiency has a positive impact on log structured storage systems. And LSS is very efficient. For any given amount of space allocated to LSS, the more efficiently it uses that space, the less cleaning it needs to do, and the fewer page moves it will need. Page moves result in additional writes to storage (write amplification).

So how is LSS storage efficient? First, there is no empty space in pages that are flushed. They are written as packed variable length strings. On average, traditional B-tree pages are only 69% utilized. Second, because we will frequently flush only deltas since the prior flush, much less space will be consumed per page flush. Finally,

swapping updated pages out of our cache will not require an additional flush as we reclaim memory only for the parts of the page previously flushed.

5 SYSTEM TRANSACTIONS

5.1 A Limited System Transaction Capability

Access methods need to make structure modifications operations (SMOs) to permit such structures to grow and shrink. This is one of the more subtle aspects of access methods. SMOs require that there be a way to effect atomic changes of the index so that ordinary updates can execute correctly in the presence of on-going SMOs, and be atomic (all or nothing). The Bw-tree exploits LLAMA system transactions as the mechanism for its SMOs.

Durability of system transactions is realized via a log in a more or less conventional way. However, our log is not a transaction log, but our LSS "page" store. This may seem inefficient given that a transactional system usually only logs operations. But with delta updating we can log page state by logging only the delta updates since the prior page flush. Durability at commit is not required so commit does not "force" the LSS buffer. However, we guarantee that all subsequent operations that use the result of a transaction come after the transaction commit in the LSS.

Like non-transactional operations, all transaction operations are installed via a CAS on a page pointer in the mapping table. A critical aspect is to ensure that what is in the cache is represented faithfully in LSS and the reverse. Thus, all updates within a system transaction include a flush. Every system transaction update is recorded in the LSS buffer, and hence is "logged". The two representations of the information need to be equivalent. This ensures that, in case of system crash, we can faithfully reconstruct the state of the cache as of the last buffer stably captured by LSS.

This equivalence is a problem when actions involve more than one page, as SMOs do. For example, a node split SMO in a B-link tree both allocates a new page and updates its sibling page link pointer to reference the new page. SMOs in latch-based systems use latches to provide isolation so that the internal states of a multi-page SMO are not visible in the cache manager until the SMO is complete. A latch-free design means that the ability to isolate active (and hence uncommitted) transaction updates is limited.

LLAMA provides a transactional interface that permits fairly arbitrary access to pages, i.e., operations on arbitrary pages can be placed within a transaction. But "users beware". We do not protect pages updated during a transaction from access by an operation outside of the transaction. Fortunately, SMOs can be designed that do not need a fully general isolation capability. SMO transactions can be frequently captured by the "template" shown in Figure 8.

1. Allocate or free pages in the mapping table
2. Update the new pages as needed
3. Update an existing page so as to connect the new pages to the rest of the index or to remove an existing page while updating another page

Figure 8: SMO transaction template

A new node for a node split (using this template), is not visible to other threads executing operations until step 3 when it is connected to the tree and the transaction is committed. Thus, such an SMO transaction provides both atomicity and isolation.

5.2 Active System Transactions

As in conventional transactional systems, we maintain an active transaction table for our system transactions, called the ATT. The ATT has an entry per active system transaction that contains the TID for the transaction and a pointer to the immediately prior operation of the transaction, called IP, which points to the memory address of the most recent operation of the transaction.

A TBegin operation adds a new entry to the ATT, with a transaction id (TID) higher than any preceding transaction, with IP set to NULL. Execution of a transaction operation creates a "log record" for the operation pointing back to the log record for the operation identified by the IP, and IP is updated to reference the new operation. This backlinks the "log records" for operations of a transaction in the conventional way, but all "log records" are only in main memory. Further, operations within a system transaction only change cache state via mapping table updates, not LSS buffer state. All these pages are flushed on transaction commit.

When an end of transaction (commit or abort) occurs, the transaction is removed from the ATT.

5.3 System Transaction Atomicity

At commit, pages changed by a transaction need to be flushed to the LSS buffer in some atomic fashion. One might bracket these page writes with begin and end records for the transaction in the LSS. But this would require crash recovery to undo interrupted transactions. Such undo recovery would require the writing of undo information to LSS. We avoid this by doing an atomic flush at commit of all pages changed by a transaction (see 5.3.1 below).

Subsequent actions that depend on an SMO must be later in the LSS buffer than the information describing the SMO transaction. Thus, when the state of an SMO becomes visible in the cache to threads other than the thread working on the system transaction, those other threads must be able to depend upon the SMO having been committed to the LSS and already present in the LSS buffer.

It is step 3 in Figure 8 that is tricky. It must encapsulate both the updating in main memory (making the transaction state visible) and the committing of the transaction in the LSS buffer via an atomic flush. We introduce a "commit" capability for an Update-D to do this, i.e. combining an update with transaction commit.

5.3.1 Commit

LSS deals with a transactional Update-D "commit" operation by combining the update and its CAS installation with an atomic flush of all pages changed in the transaction. This flush on commit of multiple pages is done in the same style as for individual page flushes. LSS buffer space is allocated for all pages changed in the transaction. Then the CAS is executed that installs the Update-D delta prepended with a flush delta. If the CAS succeeds, the pages updated in the transaction are written to the LSS flush buffer. Only after the flush of all pages for the transaction is complete does the flush process decrement the number of writers of the flush buffer. It is the allocation of space for all pages in the transaction as a single unit with the hold until writer decrement on the LSS buffer that ensure atomicity for the transaction in the LSS store.

5.3.2 Abort

If the CAS fails, we respond as we do for other flush failures. That is, we VOID the space we had allocated so that the LSS, during recovery, does not confuse the space with anything else. In

this way, our recovery process is completely unaware of system transactions. Rather, system transactions are solely a capability of our caching layer. This means that there is no need to ensure TID uniqueness across system crashes or reboots.

Operations of an aborted system transaction need only be undone in the cache since recovery will never see incomplete transactions. Thus, we follow the back chain of log records for the transaction, and provide the undo based on the nature of the operations on the ATT list for the transaction, undoing a delta update by removing the delta, undoing an allocate with a free, and undoing a free by restoring the page to its state prior to the free. Aside from undoing a FREE, no extra information is needed beyond the information describing operation success.

5.4 Interaction with Epochs

Actions that happen within transactions are provisional. This includes the allocation and freeing of storage and mapping table page entries (PIDs). During transaction execution, PIDs are allocated or freed, and Update-D deltas are generated. The management of these resources has to be done in our epoch based way. Since an SMO is done within a single user operation request, the thread remains in its epoch for the duration of the transaction.

LLAMA reclaims resources depending on transaction commit or abort. For commit, free page PIDs are added to the PID pending free list for the current epoch. For abort, an allocated PID is freed during undo and similarly added to the PID pending free list. Finally, for an Update-D, operation, the update delta is added to the storage pending free list for the current epoch.

6 FAILURE RECOVERY

6.1 Need for Crash Recovery

When we discuss recovery here, we are *not* referring to transactional recovery. When we discuss checkpointing, we are *not* referring to checkpointing as used to manage a transactional log. Rather, recovery here refers to the need for LSS (a log structured store) to recover its mapping table of pages and their states to the time of a system crash. This recovery step is not needed for conventional update-in-place storage systems.

A way to think about crash recovery is to consider the mapping table as a "database". Updates to this database are the page states flushed to the LSS. Thus, every page flush updates the "mapping table database". Should the system crash, we replay the LSS "log" to recover the "mapping table database", using the pages flushed as redo log records to update the mapping table.

For the above strategy to work, we need to periodically checkpoint the mapping table so as to avoid having to keep LSS updates forever. Our LSS cleaning could be used for this purpose (i.e., shortening the recovery log) but leaves a recovery log (the LSS log structured store) that is too large for high speed recovery.

6.2 Checkpointing

6.2.1 Strategy for Checkpoints

We use a very simple tactic for checkpointing. LLAMA asynchronously and incrementally writes the complete mapping table during a checkpoint to one of two alternating locations. Each location, in addition to the complete mapping table, stores a recovery start position (RSP) and garbage collection offset GC as shown in Figure 9. The RSP is the end offset in the LSS store at the time we start copying the mapping table. The GC offset marks the garbage collection "frontier".

Later checkpoints have higher RSPs, as LSS offsets monotonically increase by being virtualized. After a system crash, we use the completed checkpoint with the highest RSP to initialize the state of the recovered mapping table. The RSP indicates where in the LSS "log" we begin redo recovery. To identify the last complete checkpoint, we do not write the RSP to the checkpoint until the mapping table has been fully captured. In that way, the previous high RSP (from the alternate location) will be the highest RSP until the current checkpoint is complete.

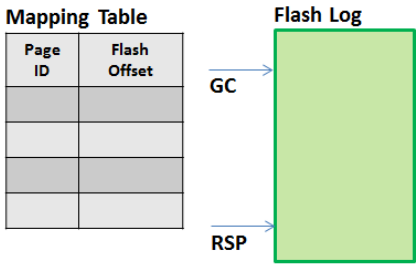

Figure 9: Checkpoint data: Mapping Table, GC, and RSP

6.2.2 "Copying" the Mapping Table

When LLAMA writes out the mapping table as part of a checkpoint, this is not a byte-for-byte copy of the mapping table as it exists in the cache for two reasons:

1. The cached form of the mapping table has main memory pointers in the mapping table entries for cached pages. Our checkpoint needs to capture the LSS addresses of the pages.
2. Mapping table entries that are not currently allocated are maintained on a free list that uses the mapping table entries as list items. Thus a free mapping table entry either has zero or the address of the immediately preceding free mapping table entry (in time order by when they were added to the free list). We cannot capture a usable free list during our asynchronous "copying" of the mapping table. Our copy of the mapping table is written asynchronously and incrementally, minimizing the impact on normal execution.

LLAMA first saves the current end offset of the LSS store as the RSP. We scan the mapping table (concurrently with ongoing operations) and identify the LSS address of the most recent flush of the page for each PID entry (stored in the most recent flush delta), and store that LSS address in our checkpoint for that mapping table entry. If the entry is free, we zero that entry in our checkpoint copy. (We reconstruct the free list at the end of redo recovery.) Finally, when we finish copying the mapping table, we save the GC as of the end of checkpoint and write both previously saved RSP and GC to the stable checkpoint area, completing the checkpoint.

6.3 Recovery

6.3.1 Redoing Operations

Recovery begins by copying the mapping table for the checkpoint with the highest RSP (i.e. the latest complete checkpoint) into cache. It then reads the log from RSP forward to the end of the LSS. Each page flush that is encountered is used to restore the page's PID in the mapping table to the flash offset for the page. When an Allocate operation is encountered, the mapping table entry for the

allocated PID is initialized to empty as required by an Allocate operation. When a Free operation is encountered, the mapping table entry is set to ZERO. The LSS cleaner will resume garbage collecting the log from the GC offset read from the checkpoint.

6.3.2 Rebuilding the Free PID List

During recovery, all free mapping table entries are set to ZERO. We scan the rebuilt mapping table. When we encounter a ZERO entry, we add it to the free list, which is managed as a stack. That is the first entry to be reused is the last one that is added to the list. In this way, we preferentially reuse the low order PIDs, which tends to keep the table size clustered and small (at least as a result of recovery). We also keep a high water mark in the mapping table indicating the highest PID used so far. When the free list is exhausted, we add PIDs from the unused part of the table, incrementing the high water mark.

7 PERFORMANCE EXPERIMENTS

This section provides an experimental evaluation of LLAMA comparing the Bw-tree [16] implemented over LLAMA with the BerkeleyDB B-tree [5], a "traditional" page-based B-tree. Our experiments use both real and synthetic workloads.

7.1 Implementation and Setup

Bw-tree and LLAMA. LLAMA is implemented in approximately 12,000 lines of C++ code. The Bw-tree on top of LLAMA is approximately 4,000 lines of code. LLAMA uses the Windows **`InterlockedCompareExchange64`** to perform the CAS, and LSS flush buffers are set to 4MB. The Bw-tree consolidates pages (e.g., installs a new page using the **`Update-R`** API) after ten or more deltas to a base page; this was found to perform well [16].

BerkeleyDB. We compare the Bw-tree on LLAMA to the BerkeleyDB key-value store, which is known for its good performance and is used as a storage layer in several well-known platforms, e.g., Project Voldemort from LinkedIn [26]. We use BerkeleyDB running in B-tree mode, a B-tree index on top of a buffer pool for its page cache. This is a traditional architecture for a B-tree. To maximize BerkeleyDB concurrency and provide a fair comparison, we run it without transactional support, which permits a single writer and multiple readers with page-level latching.

Experiment machine. Our experiment machine is an Intel Xeon W3550 (at 3.07 GHz) with 24 GB of RAM and a 160GB Fusion IO flash SSD drive. The machine has four cores that we hyperthread to eight in all of our experiments.

Data sets. We use three workloads, two from real-world applications and one synthetic. The workload characteristics are summarized in Table 1.

1. *Xbox Live*. This workload contains 27M `get-set` key operations obtained from a real-world instance of the Microsoft Xbox Live [33] backend that manages state for multi-player games. The key is dot-separated sequence of strings with a length of 94 characters and average value sizes of 1200 bytes. The ratio of `get` to `set` operations is 7.5 to 1.
2. *Storage Deduplication*. This workload is from a Microsoft deduplication trace that generates a sequence of chunk hashes for a root directory and compute the number of deduplicated chunks and storage bytes. Keys are 20 byte SHA-1 values that uniquely identify a chunk, while the value is a 44 byte metadata string. The trace contains 27M total chunks and 12M unique chunks. The workload first attempts to read a chunk, and if the chunk is not present inserts its record. The read to write ratio is 2.2 to 1.
3. *Synthetic*. This workload consists of 8 byte keys and 8 byte values. The workload begins with an index of 15M entries with keys generated with a uniform random distribution. It then performs 120M operations that are either reads or updates of existing keys. Keys for these operations come from either the hot set (lower 20% of the key range) or cold set (upper 80% of the key range), with the probability of a generating a hot key of 95%. The read to write ratio is 5 to 1.

	Op Count	Read:Write ratio	Avg Key Size	Avg Val Size
XBOX	**27 M**	**7.5:1**	**92 bytes**	**1200 bytes**
Dedup	**40 M**	**2.2:1**	**20 bytes**	**44 bytes**
Synthetic	**120M**	**5:1**	**8 bytes**	**8 bytes**

Table 1: Properties of experiment data sets

	Cleaning Overhead	LSS Read Reduction	Flush Failure Rate
XBOX	9.5%	7.0%	0.29%
Dedup	9.03%	0.29%	0.08%
Synthetic	5.75%	3.0%	0%

Table 2: LLAMA statistics

Defaults. Unless otherwise mentioned, our metric is throughput measured in operations per second. We use eight worker threads for each workload (equal to the hyperthreaded cores on our machine). The default page size for both BerkeleyDB and the Bw-tree is 8KB. The LLAMA maximum LSS file size is set to half of its observed maximum file size with cleaning turned off.

Memory Limit. For each workload, we measure the index's maximum memory usage when run completely in memory. To force each system to push data to flash, the memory limit we assign to the index for each workload is half of its observed max value.

7.2 Effect of LSS Cleaning

We ran each workload on Bw-tree/LLAMA with LSS cleaning disabled, then enabled. The 2nd column of Table 2 reports the overhead of LSS cleaning. For Xbox and Deduplication workloads, we saw a 9% overhead, while for the synthetic workload cleaning overhead was less than 5%. LSS cleaning in LLAMA entails relocating a base page on the log as well as compacting and relocating any deltas prepended to the page (Section 4.3). We believe a 9% overhead is a tolerable price to pay. Such low overhead is a necessity for log-structured storage systems to perform well.

A benefit of LSS cleaning is the creation of consolidated pages on the LSS that reduce the number of reads LLAMA performs on flash (Section 3.2). We instrumented LLAMA with counters to measure the number of flash reads performed during each run. The 3rd column of Table 2 reports the reduction in flash reads we observed with cleaning enabled. Clearly, consolidation has a positive effect on read performance. The benefit of consolidations is correlated to

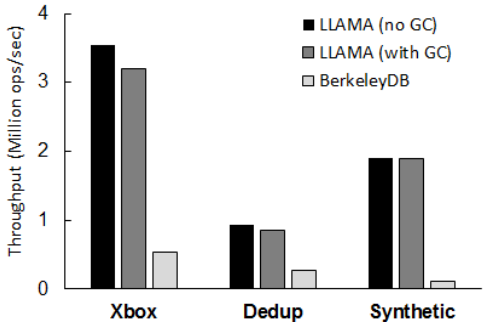

Figure 10: Throughput performance

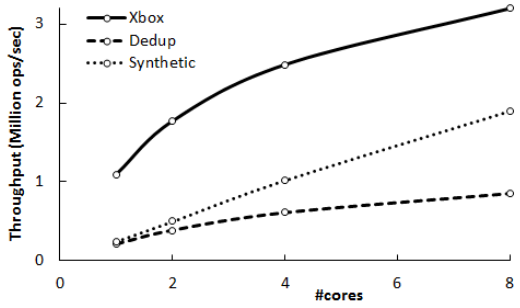

Figure 11: Throughput scaling with number of cores

the read ratio of the workload. Xbox, being a read-heavy workload, benefits the most from page consolidation (7% drop in read count). The update-heavy Deduplication workload sees a smaller benefit since it creates a relatively larger number of deltas on flash, not allowing the cleaner to catch up. The improvement for the synthetic workload falls in between the two other workloads.

7.3 Flush Failure

This section studies LLAMA's flush failure rate. Since all memory operations in LLAMA are latch-free, page flushing may fail due to a CAS failure on a flush delta after reserving buffer space (Section 3.2). A high flush failure rate results in more garbage on LSS (the unused reserved buffer space). Column 4 of Table 2 provides the flush failure rate when running all three workloads on Bw-tree/LLAMA with cleaning enabled. All rates are well below 1%, with the highest being 0.29% for the Xbox workload. Thus, flush failures have a negligible impact. Such low contention is possible since LLAMA avoids waiting for the page to be copied into the flush buffer before installing a flush delta. LLAMA simply reserves space in the buffer before installing the delta. The Xbox workload exhibits the highest contention due to larger keys and data. This causes pages to split more often, which trigger the Bw-tree to force two page flushes to correctly order the split on LSS [16]. More flush traffic increases the chances a flush will occasionally wait for a buffer to unseal. The Deduplication and synthetic workloads have relatively smaller keys and data, leading to less flush traffic.

7.4 Throughput

Figure 10 provides workload results for Bw-tree/LLAMA and BerkeleyDB. The graph plots throughput for BerkeleyDB and Bw-tree/LLAMA with and without cleaning. For fairness our discussion refers only to the numbers with cleaning.

For the XBOX workload, BerkeleyDB's throughput is 539K ops/sec, while Bw-tree/LLAMA has a throughput of 3.2M ops/sec representing a 5.9x speedup. For Deduplication, the performance of BerkeleyDB is 267K ops/sec, while Bw-tree/LLAMA has throughput of 859K ops/sec (a 3.2x speedup). The performance drop of Bw-tree/LLAMA is mostly additional flash reads. Since the Deduplication workload uses hashed keys, there is no key locality, i.e., pages are accessed at random. This increases the chance an operation on Bw-tree/LLAMA waits for a flash I/O to bring a page to memory. Finally, Bw-tree/LLAMA exhibits a 17x speedup over BerkeleyDB on the synthetic workload. This wide gap is mainly due to the latch-free behavior of LLAMA. Since most updates go to a hot set of records in this workload, only a relatively small number of high-contention pages are updated. Bw-tree/LLAMA allows concurrent page access to both readers and writers. BerkeleyDB's page-level latches block both writers and readers during an update, causing performance to suffer.

We believe three factors account for the Bw-tree/LLAMA superior performance. (1) *Latch-freedom*: the Bw-tree makes use of LLAMA's latch-free design. Latch-free page updates increase in-memory concurrency (and reduce latency) for threads updating pages. Latch-free write buffers allow page flushes (e.g., during an SMO) to proceed without blocking. Meanwhile, BerkeleyDB requires page-level latches that block readers and other writers during a page update. We believe BerkeleyDB also requires latches to reserve space in its write buffer. (2) *Delta updates*: In-memory LLAMA delta updates avoid invalidating CPU caches of other threads accessing the same page memory concurrently (except for the 8 byte mapping table being updated). Meanwhile, BerkeleyDB updates in-memory pages in place, leading to several more cache line invalidations. (3) *Log structuring*, LLAMA only writes sequentially to flash to maintain a high write bandwidth and obviating the need for a flash FTL mapping layer. LLAMA often avoids re-writing entire pages by flushing only delta updates. BerkeleyDB updates whole pages in place on secondary storage, leading to inefficient random writes on flash.

7.5 Scalability

This experiment reports the multi-core scalability of Bw-tree/LLAMA (not cross-CPU scalability), and demonstrates LLAMAs ability to increase performance as CPU power grows by adding more cores. Using our same experimental setup, we not only measured peak performance but also scaling from a single core to exploiting all four cores, and then turning on hyperthreading to exploit eight "logical" cores. Each increase in number of cores produced a performance improvement, though scaling was not linear.

All workloads demonstrate close to linear scaling when moving from one to two cores. Scalability is lower in moving to four cores, though our synthetic workload scalability is linear throughout. Though scalability is less than linear for our real workloads, it still results in substantially higher performance at four cores (about 3x for Deduplication, 2.5x for Xbox). Hyperthreading scalability is more limited, though still producing performance gains. Going to 8 hyperthreaded cores increases performance by about another third for both real workloads over using four cores without hyperthreading. Again, our synthetic workload scalability remained linear, which we suspect results from much smaller record size and working set, leading to better cache performance.

8 RELATED WORK

8.1 Database System Architecture

Database systems have always exploited caching and managed storage. These are essential: caching for performance, storage (secondary) for durability. System R [2] divided its database engine into RDS (relational data system) and RSS (research storage system). However, the database kernel (RSS) has classically been treated as a monolith, given the intertwined nature of transactional concurrency control and recovery with access methods and cache management.

Early attempts to modularize database systems [3] stopped short of decomposing the kernel. The first successful attempt at a decomposition of which we are aware was in the Deuteronomy project [15 19], which separated transactional concurrency control and recovery from the data management aspects of access methods and cache management. Other efforts [30] have created prototypes where this separation was exploited as an architectural feature.

We know of no work separating an access method layer from cache/storage management, as done in LLAMA. One reason for this is the need to enforce the write-ahead log protocol. Before flushing a page, a conventional database kernel checks the page LSN to see whether there are updates not yet stable in the transactional log. LLAMA cache management can exploit our delta updates to "swap out" a partial page. It can drop from the cache the part of the page already present on secondary storage (which does not include recent delta updates). The access method layer will be regularly flushing for transactional log checkpointing. So the cache manager will find sufficient candidate (possibly partial) pages to satisfy any buffer size constraint.

8.2 NO SQL and Key Value Systems

There are a large number of indexing subsystems that are not contained within a surrounding database system. Such indexing subsystems have been present for a long time, e.g. IBM's VSAM [7], but have become increasingly popular over the past 10 years. Indeed, Wikipedia lists around 70 such systems [32], e.g. MongoDB [22], memcachedb, etc. on its "No SQL" page [20]. Some provide persistence as well as indexing, and support a variety of data models. We are not aware of any that are latch-free or exploit log structured storage.

Most key value systems are distributed in some fashion, usually by sharding/partitioning. LLAMA has no aspects of distribution, but it could be appropriate for the support of a single shard so long as distribution is handled outside or above the storage layer. In this way, it could play the same role for these systems that it might play in database systems generally. And, like database systems, key value stores when running on modern hardware, would benefit from its latch-freedom and log structuring.

8.3 Latch-Free Techniques

There are several ways to avoid latches. A number of papers, e.g. [25, 29], suggest scheduling threads so that they do not conflict on the usual units of latches, i.e. pages. Thus, a thread never encounters a page being changed by another thread. There can be an interlock present, however, in the preprocessing step that does this partitioning. Further, balancing the workload among threads accessing partitioned data can be a problem.

LLAMA's latch-freedom is the form that avoids latches while allowing concurrent access by any thread to any data. This kind of latch-freedom is provided by systems using skip-lists [20]. Our experience with skip-lists [16] suggests that the Bw-tree/LLAMA approach out-performs skip-lists because of better cache locality.

Hekaton, a recently announced main memory feature for SQL Server [34], is completely latch-free. Hekaton uses no latches in its lock manager, its hash table, and its ordered index, which is a Bw-tree. Particularly notable about Hekaton is its latch-free lock manager [13], which is also lock-free by using optimistic multi-version concurrency control. Hekaton is purely a main memory system, however, while LLAMA's purpose is to bring latch-freedom and log structuring to systems that also exploit secondary storage.

8.4 Flash Storage without In-Place Updates

Flash SSDs have a "flash translation layer" (FTL) that avoids update-in-place. Even so, write costs are high and random write performance is low. Having the "application" avoid update-in-place frequently improves performance [14]. The technique there is to log updates related to a page near the original data, in a "log block". What LLAMA does has a similar effect, and coupled with large write buffering, also greatly reduces the number of write I/Os.

8.5 Log Structured Storage

Log structured storage appeared first in file systems in LFS [27]. LFS dramatically reduced the number of writes by batching page writes into a large sequential buffer, and writing it in a single large sequential write. The price for this is an indirection table tracking pages re-located when written. But for write-intensive workloads, e.g. TPC-A, the number of I/Os is cut substantially [18].

Log structured merge trees (LSM-trees) [23] use log structured techniques, exploiting large sequential writes, by periodically merging a batch of recent "B-tree" updates, maintained in cache in a main memory B-tree, into an existing densely packed secondary storage B-tree. LSM-trees have gained popularity recently [28] as a way to deal with heavy write workloads in the cloud, where inexpensive disks have very limited I/O access rates.

The Hyder system [6] uses log structuring at the record level. Hyder exploits flash memory, visible to a cluster of processors in a data sharing manner. That is, multiple processors are allowed to hold an intersecting set of pages in their caches. Hyder ensures that each processor is notified when the flash memory is written, so that each processor can perform appropriate cache invalidation. Further, it is this visibility of changes to the flash memory that is used to maintain a consistent binary tree that provides access to the data records. Every update in Hyder results in changes to this binary tree being propagated to the root of the tree.

The log-structuring of updates to a page through chaining delta records on flash evolved from the SkimpyStash [8] hashed access method record lists. Like SkimpyStash, LLAMA consolidates the dis-contiguous pieces of a "page" when it does garbage collection.

LLAMA is page oriented like LFS, with an indirection table. It is like Hyder in its ability to limit what is written to only the changed data. LLAMA avoids Hyder's propagating of changes to the root of the tree, while gaining much of Hyder's record level advantage. Further, it spreads the synchronization burden over many pages, rather than focusing it on the root of the tree, where Hyder requires a subtle merge algorithm for high concurrency.

9 DISCUSSION

We have introduced and described LLAMA, our caching and storage subsystem. LLAMA is unique in a number of ways.

1. It is an independent architectural subsystem, cleanly separated from both transactional functionality and the details of access method implementation. Its general purpose operations can be used by access methods of the "grow and post" variety [17] (tree index growth by node splitting and index terms posting at parent nodes). LLAMA operations enable access methods to easily become latch-free and exploit log structured storage.
2. Both the latch-free and the log structuring techniques are done in a new and particularly effective way to provide an efficient and high performance implementation. Further, they exploit a common mapping table. This synergism between the two previously separate techniques makes the system easier to describe, easier to implement, and more efficient.

By enabling access methods to be latch-free and log structured, LLAMA achieves about an order of magnitude performance improvement for the Bw-tree over even a high quality, well-tuned conventional B-tree. And this performance scales well as the number of cores in a multi-core cpu increases.

Part of our work going forward will be the implementation of additional access methods on top of LLAMA. We see no reason why hash based, multi-attribute, and temporal access methods should not be able to use LLAMA successfully. This, of course, remains to be demonstrated in a concrete manner, so stay tuned.

10 REFERENCES

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

[2] M. Astrahan, M. Blasgen, D. Chamberlin, et al.: System R: Relational Approach to Database Management. ACM TODS 1(2): 97-137 (1976)

[3] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, T. E. Wise: GENESIS: An Extensible Database Management System. IEEE Trans. Software Eng. 14(11): 1711-1730 (1988)

[4] R. Bayer and E. M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Inf. 1(1) pp. 173–189, 1972.

[5] BerkeleyDB. http://www.oracle.com/technology/products/berkeley-db/index.html.

[6] P. Bernstein, C. Reid, and S. Das: Hyder - a transactional record manager for shared flash. CIDR, 2011, pp. 9–20.

[7] D. Comer: The Ubiquitous B-tree. ACM Comp. Surveys 11, 2 (June 1979) 121 – 137.

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

[9] W. Effelsberg, T. Haerder: Principles of database buffer management. ACM TODS 9 (4) 560 –595, 1984

[10] S. Harizopoulos, D. Abadi, S. Madden, M. Stonebraker: OLTP through the looking glass, and what we found there. SIGMOD 2008: 981-992

[11] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, Write amplification analysis in flash-based solid state drives. iSYSTOR 2009: pp. 10:1–10:9.

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

[13] P-A Larson, S. Blanas, C. Diaconu, et al: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309 (2011)

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

[15] J. Levandoski, D. Lomet, M. Mokbel, K. Zhao, Deuteronomy: Transaction Support for Cloud Data. CIDR, 2011, pp. 123–133.

[16] J. Levandoski, D. Lomet, S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. ICDE 2013, pp. 302-313.

[17] D. B. Lomet: Grow and Post Index Trees: Roles, Techniques and Future Potential. SSD 1991: 183-206

[18] D. Lomet: The Case for Log Structuring in Database Systems. HPTS (1995)

[19] D. Lomet, A. Fekete, G. Weikum, M. Zwilling. Unbundling Transaction Services in the Cloud. CIDR, 2009: 123–133.

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

[21] C. Mohan, Frank E. Levine: ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. SIGMOD 1992: 371-380

[22] "MongoDB. http://www.mongodb.org/MongoDB. http://www.mongodb.org.

[23] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O'Neil: The Log-Structured Merge-Tree (LSM-Tree). Acta Inf. 33(4): 351-385 (1996)

[24] E. O'Neil, P. O'Neil, G. Weikum. The LRU-K page replacement algorithm for database disk buffering. SIGMOD 1993. pp 297–306.

[25] I. Pandis, P. T¨oz¨un, R. Johnson, and A. Ailamaki, "PLP: Page Latch-free Shared-everything OLTP," PVLDB 4(10) pp. 610–621, 2011.

[26] Project Voldermont. http://www.project-voldemort.com/voldemort/

[27] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM TOCS* 10(1) pp. 26–52, 1992.

[28] R. Sears and R. Ramakrishnan, bLSM: A General Purpose Log Structured Merge Tree. SIGMOD 2012: 217 – 228.

[29] J. Sewall, J. Chhugani, C. Kim, et al. "PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB* 4 (11) pp. 795–806, 2011.

[30] A.Thomson, T. Diamond, S-C Weng, K. Ren, et al.: Calvin: fast distributed transactions for partitioned database systems. SIGMOD 2012: 1-12

[31] Wikipedia: (CRUD) http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[32] Wikipedia: (NoSQL) http://en.wikipedia.org/wiki/NoSQL

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

[34] C. Diaconu, C. Freedman, P.-°A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. SIGMOD 2013: 1243–1254