Scalable Database Logging for Multicores

Hyungsoo Jung*
Hanyang University
Seoul, South Korea
hyungsoo.jung@hanyang.ac.kr

Hyuck Han
Dongduk Women's University
Seoul, South Korea
hhyuck96@dongduk.ac.kr

Sooyong Kang
Hanyang University
Seoul, South Korea
sykang@hanyang.ac.kr

ABSTRACT

Modern databases, guaranteeing atomicity and durability, store transaction logs in a volatile, central log buffer and then flush the log buffer to non-volatile storage by the write-ahead logging principle. Buffering logs in central log store has recently faced a severe multicore scalability problem, and log flushing has been challenged by synchronous I/O delay. We have designed and implemented a fast and scalable logging method, ELEDA, that can migrate a surge of transaction logs from volatile memory to stable storage without risking durable transaction atomicity. Our efficient implementation of ELEDA is enabled by a highly concurrent data structure, GRASSHOPPER, that eliminates a multicore scalability problem of centralized logging and enhances system utilization in the presence of synchronous I/O delay. We implemented ELEDA and plugged it to WiredTiger and Shore-MT by replacing their log managers. Our evaluation showed that ELEDA-based transaction systems improve performance up to 71×, thus showing the applicability of ELEDA.

PVLDB Reference Format:
Hyungsoo Jung, Hyuck Han, and Sooyong Kang. Scalable Database Logging for Multicores. *PVLDB*, 11(2): 135 - 148, 2017.
DOI: https://doi.org/10.14778/3149193.3149195

1. INTRODUCTION

A logging and recovery subsystem has been an indispensable part of databases, since System R [4] was designed in the mid-1970s. To ensure atomicity and durability, the write-ahead logging (WAL) protocol [21, 22] was defined and later codified in ARIES [39], a de facto transaction recovery method widely deployed in relational database systems. Databases relying on ARIES use centralized logging that in general consists of two operations; (1) building an in-memory image of a log file by storing transaction logs into a volatile, central log buffer and (2) appending the image to the log file by flushing the log buffer to non-volatile storage (following the WAL protocol).

The widespread, decades-long use of centralized logging poses a significant challenge in the landscape of multicore hardware. The key challenge is to find a fast, scalable logging method that can solve the classic problem of multiple producers (i.e., transactions) and a single consumer (i.e., log flusher), with WAL being strictly enforced. Two technical obstacles are known at present:

Multicore scalability for multiple producers. As multicore platforms allow more transactions to run in parallel, storing concurrently generated logs into a central log buffer faces the serialization bottleneck [8] that hinders the entire database from being scalable. Prior studies [31, 32] have addressed this problem with noticeable improvements, but scalability bottlenecks still exist due to the non-scalable lock used in proposed solutions.

Synchronous I/O delay by a single consumer. Flushing a log buffer to stable storage by the WAL protocol has long been challenged by synchronous I/O delay. In this regard, *latency-hiding techniques*, originally proposed by DeWitt et al. [15] and Nightingale et al. [42], have successfully been used to increase system-wide throughput in the presence of synchronous I/O delay. Furthermore, emerging non-volatile memory (NVM) technologies are worth considering for low access latency. We believe that combining these can be a promising solution.

Important to notice is the intrinsic difficulty of the challenge that whichever of two salient issues remains unaddressed will be a culprit responsible for performance bottlenecks. Although a large body of work has focused on this challenge, what still remains unresolved yet is the multicore scalability that may render several databases vulnerable to performance problems on hardware with dozens of cores, which appears in the cloud [1]. This constitutes the primary motivation for the present work.

As a general solution to the performance bottlenecks of centralized logging, we present ELEDA (**E**xpress **L**ogging **E**nsuring **D**urable **A**tomicity) that can smoothly shepherd a surge of transaction logs from volatile memory to stable storage, without risking atomicity and durability. The centerpiece of ELEDA is a highly *concurrent* data structure, called GRASSHOPPER, which is designed to resolve the multicore scalability issues arising in centralized logging. To enhance system-wide utilization in the presence of synchronous I/O delay, we follow the claim of DeWitt et al. [15] and apply proven implementation techniques to ELEDA. We tightly integrate all these into a three-stage logging pipeline of ELEDA that can be *applicable* to any transaction systems suffering performance bottlenecks of centralized logging.

We implemented a prototype of the ELEDA design and plugged it to WiredTiger [41] and Shore-MT [17] that use *consolidation array techniques* [31,32] but still exhibit the performance bottlenecks under high loads on multicores. We evaluate the performance of ELEDA on a high-end computing platform equipped with 36 physical cores, 488 GiB memory and two enterprise NVMe SSDs. With the key-value workload, ELEDA-based systems improve transaction throughput by a wide margin, thus some achieving higher than

*Contact author and principal investigator

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.
Proceedings of the VLDB Endowment, Vol. 11, No. 2
Copyright 2017 VLDB Endowment 2150-8097/17/10... $ 10.00.
DOI: https://doi.org/10.14778/3149193.3149195

~ 3.9 million Txn/s with guaranteed atomicity and durability. With the online transaction processing (OLTP) workload, ELEDA-based systems also show significant improvements.

This paper makes the following contributions:

– We confirm that the performance bottlenecks in centralized logging still exist and reframe the key challenge to be the design of concurrent data structures (§2.2).
– We design ELEDA that is a fast, scalable logging method for high performance transaction systems with guaranteed atomicity and durability (§3 and §4).
– We build ELEDA and plug it to WiredTiger and Shore-MT (§5). We then evaluate ELEDA-based systems, demonstrating that our design avoids the performance bottleneck and improves performance compared to the baseline systems.

2. BACKGROUND AND RELATED WORK

In this section, we point to a few essential concepts needed for understanding our contribution.

2.1 Database Logging and Recovery

Transactional recovery is of importance to databases that are expected to guarantee ACID (**A**tomicity, **C**onsistency, **I**solation and **D**urability) properties. Authoritative coverage on the field of transaction management and its subfield of transactional recovery can be found in the textbooks by Bernstein et al. [6] and Weikum and Vossen [53]. The internal design of database systems is described by Hellerstein et al. [24].

Database logging in this context is used to capture every modification made to data by a transaction. A log sequence number (LSN) is associated with each log to establish the happen-before relation [36] between logs pertaining to the same database page, or between logs spanning multiple data pages. LSN-stamped logs are stored in a volatile, central log buffer and later synchronously written to non-volatile storage by the WAL protocol. In a nutshell, given all log records that are totally-ordered, ARIES [39], upon failures, replays all redo logs in LSN order to reconstruct the state of databases as of the crash, then it applies undo logs in reverse LSN order to rollback the effects of uncommitted transactions. Hence a globally unique LSN in ARIES (or its variants) is pivotal to imposing proper ordering on transaction logs.

2.2 Multicore Scalability

2.2.1 Sequentiality of logging

A fundamental restriction imposed on database logs is the *sequentiality of logging*, strictly requiring that the sequence of LSN and the log flushing order be the same. In other words, the LSN order should also match the log order in an in-memory image of a log file. Any violation would definitely risk the correctness of database recovery. Note that the sequentiality condition held for logs in a volatile buffer must be retained prior to the volatile log buffer being sequentially flushed to non-volatile storage. To honor the sequentiality of logging, the widespread use of a global lock began early days in many database engines, such that (1) a transaction *acquires* a lock, (2) allocates an LSN for its log and embeds the LSN in the log, (3) copies the log to the central log buffer, and (4) *releases* the lock. The use of such a global lock undoubtedly limits the scalability of database logging on multicore systems.

2.2.2 Related work

To improve the scalability of centralized logging, Johnson et al. [31, 32] proposed the consolidation array technique that is robustly implemented in two publicly available database engines (i.e., Shore-MT [17] and WiredTiger [41]). But we observe that there is still a chance to improve the multicore scalability of centralized logging in these systems, as the author of prior work also recently confirmed the same issue and proposed distributed logging on NVM instead in [52]. Distributed logging and recovery [49, 52, 54], owing to performance benefits, have been studied as such, but design complexity added to recovery and log space partitioning, which may restrict transactions to access multiple database tables belonging to different log partitions, remains to be addressed for database vendors to change the status quo. For instance, Silo [51] avoids centralized logging bottleneck by letting each worker thread copy transaction-local redo logs to per-thread log buffer after validation (following *optimistic concurrency control (OCC)*). Since Silo stores redo logs only for committed transactions, epoch-based decentralized logging ensures correctness and works well for OCC-based transaction engines. Despite this attractive feature, it is non-trivial to directly apply Silo's design to ARIES (or its variants)-based databases adopting the *steal* and *no-force* policies that require undo/redo logs.

Table 1: Control options for transaction durability.

Databases or storage engines	**Durability/Slow (no data loss)**	**Non-Durability/Fast (data loss)**
Oracle [44]	force wait	no-wait
MySQL [43]	flush log at commit	delayed flush
SQL Server [38]	full durability	delayed durability
MongoDB [40]	commit-level durability	checkpoint durability
PostgreSQL [50]	synchronous commit	asynchronous commit
Shore-MT [17]	flush log at commit	lazy flush

2.2.3 Challenge

The key challenge in eliminating the non-scalable global lock for scalable logging on multicores is to allow transactions to concurrently buffer their logs with the sequentiality being guaranteed. As prior work attempted, concurrent log buffering itself is not technically challenging. The essence of the challenge is to guarantee the sequentiality for those concurrently buffered logs, without using locks. Since the assigned LSN tells a log where to be copied in the log buffer, the sequentiality for the log being buffered cannot be ensured until its preceding logs are completely buffered. Here we define an LSN already assigned to a log not being buffered yet as an *LSN hole*, and also define the maximum LSN of which no LSN holes are behind as a *sequentially buffered LSN* (SBL). Since LSN holes undoubtedly stall the sequential log flushing, advancing SBL with LSN holes being chased is critical to the entire logging system. Hence the core challenge we address is to design concurrent data structures that perform required operations while transactions write logs to the central log buffer *concurrently*.

2.3 Synchronous I/O Delay

The strict enforcement of the WAL protocol incurs synchronous I/O delay, which is detrimental to high performance reliable transaction processing. The synchronous I/O delay therefore has placed substantial pressure on database designers for pursuing alternative options that are intended to trade transaction durability for better performance. As shown in Table 1, well-known database engines provide such options that control balance between transaction durability and performance.

2.3.1 NVM is close at hand

The advent of fast, non-volatile memory technologies has changed the age-old belief in our mind that non-volatile storage is slow. This has driven a substantial body of proposals that have attempted to reduce or to avoid the synchronous I/O delay in databases, either

by rearchitecting subcomponents of databases [2, 13, 52], or by the novel use of NVM [9, 18, 20, 23, 34]. Closer to our work is the proposal by Fang et al. [18] that combines logging in NVM with hole detection being executed during database recovery. Although the hole detection uses linear scanning of log space in NVM, the proposal retains efficiency during normal operations mainly because (i) logs are quickly hardened to fast NVM with some LSN holes being left unresolved and (ii) the linear scanning is performed only during the recovery for detecting any holes left due to the system crash. Write-behind logging (WBL) [3] is recent work designed for hybrid storage with DRAM and NVM. With WBL, database management systems write updates to NVM-resident databases prior to a log being flushed. Updates are made visible after a log is hardened. WBL can ensure good performance only if databases reside on fast, byte-addressable NVM that is again an essential part of the scheme. All in all, despite performance gain proved by these techniques, non-scalable locks in software, if left unaddressed, would raise undesirable performance issues on high-end computing platforms with dozens of cores, like what prior works [7, 10–12, 16, 29, 30, 33, 37, 48] have addressed for improving multicore scalability of components of operating systems and databases.

2.3.2 *Latency-hiding techniques*

Radically different from NVM-based approaches are some proposals that are intended to increase system-wide utilization visible to users in the presence of synchronous I/O delay. The database and operating systems communities call these *latency-hiding techniques*, and they are widely adopted in both research and commercial fields. The key idea for improving system-wide utilization is switching waiting threads/processes with runnable ones. The elaboration required here is to control when to release the computation results to external users.

Early lock release. Three decades ago, DeWitt et al. [15] proposed the *early lock release* (ELR) in the database community, stating that database locks can be released before its commit record is hardened to the disk, as long as the transaction does not return results to the client. This means, other transactions may see the modifications made by the pre-committed transaction, and these transactions are not allowed to return results to the clients. While materializing ELR, Johnson et al. [31] proposed the *flush pipelining* technique that allows the thread to detach the pre-committed transaction and to attach another transactions to reduce scheduling bottleneck. Other examples of adopting ELR are Silo [51] and SiloR [54] that have proposed high-performance in-memory database engines, based on optimistic concurrency control and distributed logging.

External synchrony. In the systems community, Nightingale et al. [42] proposed the external synchrony that may share the same design goals of increasing system-wide utilization in the presence of synchronous file I/O. The external synchrony is a model of local file I/O to provide the reliability and simplicity of synchronous I/O, while providing the performance of asynchronous I/O. The essence of the external synchrony is, a process/thread is not allowed to deliver any output to an external user until the file system transaction on which the pending output depends commits, meaning the pending output is released only after the corresponding file system transaction writes all contents to the non-volatile storage.

3. OVERVIEW

In this section, we provide an overview of ELEDA. We first describe the overall architecture and then explain how we approach technical challenges for achieving the goals of making centralized logging fast and scalable, with guaranteed atomicity and durability.

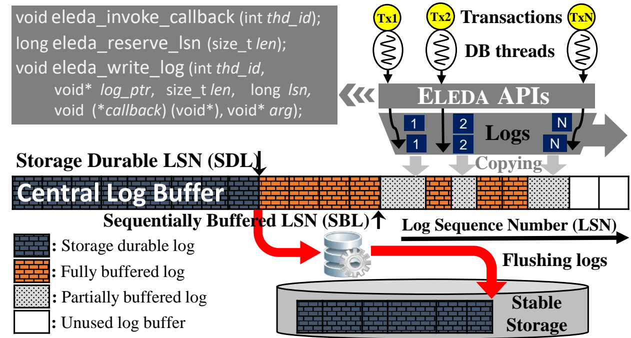


Figure 1: ELEDA logging architecture.

3.1 Overall Architecture

The logging architecture of ELEDA is based on a three-stage logging pipeline, and it consists of three types of threads: database threads, an ELEDA worker thread and an ELEDA flusher thread. Figure 1 depicts the overall architecture. The central log buffer in ELEDA is one contiguous memory logically divided into chunks by I/O unit size. Database threads do the memory copy to store their logs to the log buffer.

In the first stage of the logging pipeline, database threads, on behalf of its transaction, reserve log space (i.e., LSNs) and copy logs to the reserved buffer via ELEDA APIs, all done *concurrently*. LSN allocation is done by using the atomic `fetch-add` instruction[1], and concurrent space allocation and copying inevitably creates so-called LSN holes (i.e., *partially buffered logs* in Figure 1). That being said, database threads can make rapid progress because the ELEDA worker takes the responsibility for tracking such LSN holes and advancing SBL. Among transactions is a committing one that must provide a callback function to ELEDA and needs wait until its log and all preceding logs become durable on stable storage.

In the second stage, the ELEDA worker tracks the LSN holes and advances SBL efficiently using carefully designed concurrent data structures (i.e., GRASSHOPPER) (§4.2). In the third stage, the ELEDA flusher keeps flushing all sequentially buffered logs up to SBL to non-volatile storage. Here we define the most recently flushed LSN as a *storage durable LSN* (SDL), which is a durability indicator for database logs in stable storage (§4.3). And database threads wake up waiting transactions whose commit LSNs became durable (i.e., commit LSN $\leqslant$ SDL). This is done by invoking callback functions associated with waiting transactions, and it makes use of the *latency-hiding technique* (§3.3). As shown in Figure 1, ELEDA provides APIs for this purpose. Although ELEDA is oblivious to host engine's logging and recovery policies, it permits the host engine to pre-allocate in-memory log space at their disposal (e.g., reserving log buffer space for logs generated upon rollback). But such pre-allocated buffer space works as a large LSN hole that may hinder log flushing.

3.2 Scalable Logging

The widespread use of a *non-scalable* global lock to protect log buffering is now liable for the *serialization bottleneck* [8] on multicores. The naive elimination of a global shared lock for scalable logging however poses a challenge that requires sequentiality of logging to be preserved. The essence in addressing this challenge is to design shared data structures allowing concurrent accesses, and to devise an algorithm for fast tracking of LSN holes.

[1]This will surely hit a hardware-based synchronization bottleneck once latch contention is eliminated, and research on new designs for surpassing the limitation is ongoing in our community [47].

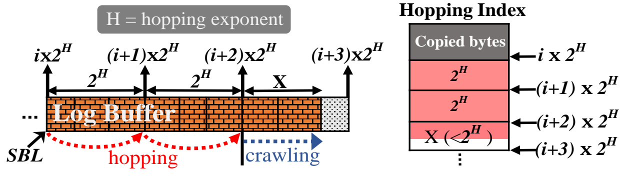


Figure 2: Grasshopper algorithm for advancing SBL.

To address the challenge, we eliminate non-scalable locks by designing concurrent data structures intended to enable multiple transactions to store their logs to the log buffer concurrently while tracking LSN holes fast and thus advancing SBL accordingly. We call this GRASSHOPPER. GRASSHOPPER advances SBL in two ways - *hopping* and *crawling* - depending on how LSN holes are traced. To put it in a nutshell, a grasshopper on grass hops using its hind legs when it feels threatened, and crawls otherwise. Our GRASSHOPPER works likewise, except that the threat we are concerned here is the gap between the current SBL and the most recently allocated LSN. The wider the gap is, the longer it takes to trace LSN holes and to flush sequentially buffered logs.

Figure 2 depicts how SBL is advanced in two different ways: hopping is used when the gap is too wide while crawling is activated otherwise. The question is how to figure out when to commence or to stop hopping. We address this by maintaining a table, called a *hopping index* (or *H-index* in short), whose entry bookkeeps the cumulative byte count of buffered logs for the LSN range of $[i \cdot 2^H, (i+1) \cdot 2^H)$, where i and 2^H are the index to the table and a hopping distance, respectively. As shown in Figure 2, entries in the hopping index are used to provide more accurate information to GRASSHOPPER. If an entry equals to 2^H, we can assert that all logs in the corresponding LSN range are buffered without LSN holes. Otherwise ($< 2^H$), one or more holes exist in the range.

3.2.1 SBL-hopping

When the current SBL - a precise indicator of the tip of the sequentially buffered logs - is far behind the most recently allocated LSN because transactions generate logs too fast, we advance SBL by *hopping* the hopping distance, provided that logs in a given LSN range are all sequentially buffered. SBL-hopping continues until GRASSHOPPER detects any LSN holes, by checking the byte count of the hopping index entry (i.e., byte count $< 2^H$). Hence SBL-hopping can be viewed as fast-path SBL advancement which works well when logs are surging fast. But the hopping is unable to precisely spot LSN holes since the hopping index does never record each individual LSN value. We cope with this issue by chasing LSN holes accurately (i.e., SBL-crawling).

3.2.2 SBL-crawling

The method of advancing SBL switches from SBL-hopping to SBL-crawling when we detect the presence of LSN holes. Once we know the presence of LSN holes, GRASSHOPPER advances SBL by chasing LSN holes thoroughly (i.e., *SBL-crawling*). SBL-crawling, despite of its relatively slow nature of advancing SBL, can handle a wide range of operating conditions efficiently.

3.3 Latency-hiding Design

To address the synchronous I/O delay, we use latency-hiding techniques in ELEDA for pursuing better system-wide utilization. To this end, we leverage the proven techniques used in the prior studies [32, 42, 51, 54]. This is done by first letting *synchronous waiting* be switched over other useful transaction processing work. Once I/O is completed, the computation results held by this I/O can be delivered to an external user asynchronously through the

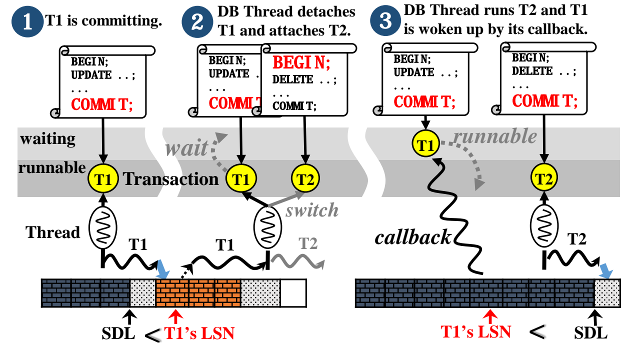


Figure 3: Latency-hiding techniques in ELEDA.

callback. Since database engines we tested (or assumed) are all based on multi-threaded architecture, we use a limited number of database threads (i.e., $\sim$ the number of available cores) as worker threads, and each of them binds itself to an active transaction among multiple candidates. We therefore assume that there is one-to-one relationship between a transaction object and an external user, though a database connection object may have such relationship with an external user instead in other engines (e.g., MariaDB).

Figure 3 shows how we implement proven techniques in ELEDA. Suppose there is a committing transaction T1 who just writes a commit log to the log buffer, but the SBL is not advanced to its LSN due to the LSN holes. Then, it is unsafe for T1 to return results to the client because its commit log is not durable yet. At this point, the database thread detaches T1 (pre-committed transaction) and put it to the *waiting* state, and then the database thread attaches a runnable transaction T2, which has useful operations to execute. When all LSN holes preceding T1's commit LSN are completely filled (i.e., SBL passes T1's commit LSN), then the log flusher advances SDL to SBL by hardening logs in between. This makes T1's commit log durable (i.e., T1's commit LSN $\leqslant$ SDL), and this is the moment ELEDA can put T1 to a running state, thereby letting T1 return its results to the external user. A callback registered for T1 is invoked to deliver an asynchronous notification to an external user.

4. ELEDA DESIGN

In this section, we describe designs and the approach we take to address the challenge in §3.2, especially a highly concurrent data structure that resolves the *serialization bottleneck* caused by the non-scalable global lock used in centralized logging of several database systems. In particular, we focus on explaining how we design shared data structures and make concurrent operations acting on the data structures correct.

4.1 Data Structures

ELEDA uses concurrent data structures (i.e., a hopping index table and GRASSHOPPER) that are intended to chase LSN holes and to advance SBL, the tip of sequentially buffered logs, without using non-scalable locks. GRASSHOPPER is designed based on two basic data structures: an augmented shared FIFO queue (i.e., a two-level list) and a binary min-heap.

Grasshopper list. An augmented shared FIFO queue is called a *grasshopper list* that manages LSNs that have been buffered but not confirmed to be durable yet. We call these LSNs *pending LSNs*. A grasshopper list is created for each database thread, and it is shared between the owner database thread and the ELEDA worker thread with different access privileges. Since a grasshopper list is basically a FIFO queue, enqueuing is solely done by the owner database thread and dequeuing is performed by the ELEDA worker thread.

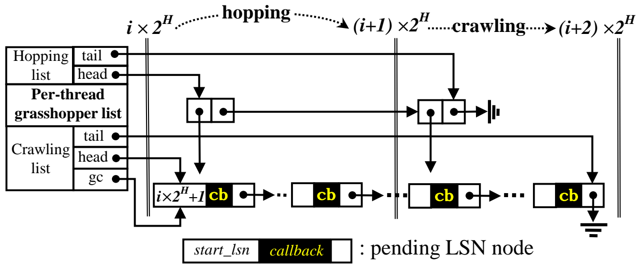


Figure 4: Per-thread grasshopper list.

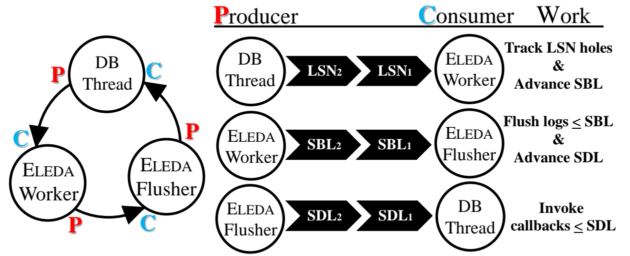


Figure 5: The producer/consumer structures in ELEDA.

A per-thread grasshopper list internally consists of two lists; a *crawling list* (or c-list) and a *hopping list* (or h-list). Figure 4 shows a per-thread grasshopper list with some pointer fields being shared by a database thread and the ELEDA worker thread. The figure depicts how we organize the crawling list with the augmented hopping list. Note that none of shared variables are protected by a lock.

The crawling list is a list of all pending LSN nodes, and a pending LSN node is a triple of <*start_lsn*, *end_lsn*, *callback*>, where the callback is valid when the LSN is for a commit log that requires a committing transaction to wait until it becomes durable. The crawling list has three pointer fields: *head*, *tail* and *gc*. Since a grasshopper list is created for each database thread, pending LSNs in each grasshopper list are totally ordered in ascending order.

The hopping list is a list of pointer pairs, one pointing to the first pending LSN node in a given LSN range of $[i \cdot 2^H, (i+1) \cdot 2^H)$, and the other one pointing to the next pair. The hopping list has two pointer fields: *head* and *tail*. The hopping list augments the crawling list in that the head pointer is updated after hopping 2^H distance when GRASSHOPPER finds itself lagging too behind the current log buffer offset, and the head pointer is used to place the head pointer of the crawling list to the LSN node where the crawling should commence afterward. Any other succinct data structure meeting the requirement can be used.

LSN-heap. A binary min-heap is called an *LSN-heap* that is exclusively owned by a dedicated ELEDA worker thread, and an LSN-heap is constructed from a collection of first (and smallest) pending LSNs of all grasshopper lists. For SBL-crawling, we can remove the top element (i.e., the smallest LSN) from the current LSN-heap *if and only if* the top LSN, which is the smallest pending LSN (<SBL) in all grasshopper lists, is contiguous to the current SBL. Then we update the old SBL (i.e., advancing SBL).

4.1.1 Access rules on shared data structures

We have presented main shared data structures and variables so far. These shared data structures in ELEDA are accessed by three types of threads: database threads, ELEDA worker and ELEDA flusher. Since shared accesses allow several concurrent executions to be exposed to shared data structures, access rules must be clearly set to avoid undefined behaviors due to malicious data races.

Design principles. In this regard, we use two design principles in making access rules. The first design principle is, ELEDA's logging pipeline architecture solely consists of producer/consumer structures, which means that every pair of a reader and a writer accessing the same shared data structure is mapped to a pair of a producer and a consumer. This makes designing concurrent algorithms straightforward since concurrent algorithms for the producer/consumer problem have been discussed thoroughly in many textbooks (e.g., Herlihy and Shavit [25]).

As shown in Figure 5, there are three such relationships. The first one is, a database thread and an ELEDA worker thread accessing a grasshopper list are mapped to a producer enqueuing pending LSN nodes to the list, and a consumer dequeuing nodes from the list, respectively. The second one is, the ELEDA worker and the ELEDA flusher are also mapped to a producer advancing SBL, and a consumer advancing SDL by flushing logs up to SBL. The third one is, the ELEDA flusher and a database thread are mapped to a producer advancing SDL and a consumer invoking callbacks up to SDL and advancing the garbage-collection point.

Table 2: Allowed operations (W: Write, R: Read and Ø: none) for different threads on variables in shared data structures.

Thread Type	Shared Variables					
	head	*tail*	*gc*	*SBL*	*SDL*	*h-index*
Database thread	Ø	W	W	Ø	R	W
ELEDA worker thread	W	R	Ø	W	Ø	R
ELEDA flusher thread	Ø	Ø	Ø	R	W	Ø

The second principle is in accordance with the first one: we never allow concurrent write privileges since arbitrating concurrent writes on shared variables complicates synchronization methods, and the performance cannot usually be guaranteed. Table 2 shows access privileges that clearly represent who can access which variables with what access privileges. By the access privileges, there is only one writer who can modify any variables in shared data structures. In ELEDA, there is only one producer thread who can modify the shared variables, which can be read by the consumer thread to detect overrunning. We resort to Table 2 in explaining concurrent algorithms for tracking LSN holes in §4.2.

General invariant. A general invariant enforced on a shared, conceptual FIFO structure supporting a single producer and a single consumer is, *the consumer can never overrun the producer*. This invariant however does never mandate that the consumer should do spin-waiting until the waiting condition is cleared. Instead, in ELEDA the consumer-side thread skips consuming work and processes other useful work to retain high utilization. The consumer later revisits and processes the work whenever it is safe. The formal descriptions of algorithm-specific invariants will be fully discussed in §4.5 after we explain complete algorithms.

4.1.2 Starting point: enqueuing an LSN node

The starting point of our logging pipeline is when a transaction needs to store its logs to the central log buffer. A database thread first gets an LSN for its logs by atomically incrementing the LSN sequencer. The allocated LSN is used as an offset inside the central log buffer and tells where to copy given logs. After embedding the assigned LSN in the logs, the database thread copies the logs to the offset in the log buffer. Once the copy is done, the database thread creates a new pending LSN node for the logs and enqueues the node to its grasshopper list. At that time, if the pending LSN node is for a commit log, then we follow the protocol to mask synchronous I/O delay (see Figure 3 in §3.3) such that the database thread must detach the current transaction by changing the state to *waiting*, and then the pending LSN node is enqueued to the grasshopper list. The callback in the pending LSN node will be used to put the waiting

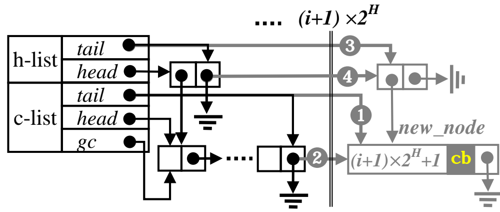


Figure 6: Enqueuing a new pending LSN node.

transaction to the runnable state once SDL passes the registered *start_lsn*, meaning all preceding LSNs are made durable. In line with this protocol, a database thread first inserts a pending LSN node into the crawling list and then adds a node to the hopping list *if and only if* the *start_lsn* of the node is the first (i.e., smallest) one in the list for a given LSN range.

Enqueuing a new pending LSN to the grasshopper list can be separated into three constituent steps in detail:

1. A database thread appends a new pending LSN node into the *c-list* by resetting the *c-list.**tail*** to the new node.
2. **IF** the *start_lsn* of the new pending LSN node is the first one in the range of $[(i+1)\cdot 2^H, (i+2)\cdot 2^H)$, a database thread inserts a hopping node to the *h-list* by resetting the *h-list.**tail*** to the new hopping node. **ELSE** this step can be skipped.
3. Increment the cumulative byte count of the corresponding entry of the hopping index by the log size.

Figure 6 describes the steps needed to complete enqueuing a new pending LSN node to a grasshopper list, assuming that a database thread obtained the LSN of $(i+1)\cdot 2^H + 1$ for its logs.

Note that a database thread, while enqueuing a new pending LSN node, changes the *tail* fields of two lists, not changing other variables (following access rules in Table 2). We enforce that the pointer manipulation order should strictly abide by the order shown above, not to have malicious data race with the ELEDA worker thread who is advancing SBL by changing head fields of the grasshopper list concurrently.

4.2 Tracking LSN Holes

With GRASSHOPPER data structures, ELEDA performs the fast advancement of SBL that inevitably relies on how fast it can track LSN holes. In this section, we explain how we advance SBL efficiently by the following key procedures: SBL-hopping (§4.2.1) and SBL-crawling (§4.2.2). Note that none of procedures use any type of locks in performing operations. When advancing SBL, a dedicated ELEDA worker thread[2] is used to chase LSN holes, and tracking the holes involves moving *head* pointer fields of grasshopper lists. Moving head pointers has two implications in that moving *c-list.head* means that we advance SBL by SBL-crawling, while moving the *h-list.head* implies that SBL-hopping advances SBL. SBL-crawling is by far the most important workhorse under a variety of workloads. The main switch to control the mode is determined by whether or not the current SBL falls into the same entry of the hopping index with the most recently allocated LSN. We first explain how we do SBL-hopping.

4.2.1 SBL-hopping algorithm

If ELEDA finds itself lagging too behind the most recently allocated LSN, meaning that the ELEDA cannot advance SBL fast enough to process all pending LSNs being produced by database transactions, then SBL-hopping is used. To decide whether or not

[2]The performance of a single worker thread can be limited by interprocessor communication overhead in NUMA machines. Overcoming such a limitation will be left as future work.

Algorithm 1: SBL-Hopping() and SBL-Crawling()

```
Data: DBT // a group of database threads
Data: H-Index[] // hopping index
Data: H // hopping exponent
1  Procedure SBL-Hopping()
2      h-size ← H-Index.table_size
3      high ← (SBL >> H) // high-order (64-H) bits
4      index ← high modulo h-size
5      while H-Index[index].bytes == 2^H do
6          H-Index[index].bytes ← 0
7          high ← high + 1
8          index ← (index + 1) modulo h-size
9      end
10     HB ← (high << H) // hopping boundary
11     adjust heads
12     SBL ← min_{∀k∈DBT} {e_lsn | h-list_k.head.e_lsn ⩾ HB}
13     rebuild the LSN-heap
14 Procedure SBL-Crawling()
15     top ← LSN-heap.peek()
16     while top.start_lsn == SBL and H-Index[index].bytes != 2^H do
17         SBL ← top.e_lsn
18         LSN-heap.pop()
19         adjust head of top.c-list
20         LSN-heap.insert(head of top.c-list)
21         top ← LSN-heap.peek()
22     end
23     LSN-heap.insert(heads of c-lists)
```

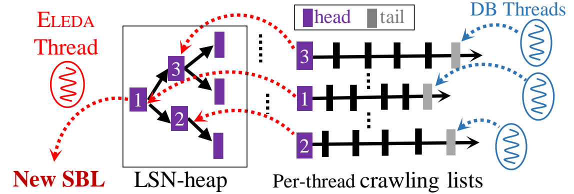


Figure 7: The overall flow of SBL-crawling.

to perform SBL-hopping, the ELEDA worker thread first checks the entry of the hopping index that current SBL belongs to. If the entry has a value of 2^H, meaning the corresponding LSN range of the log buffer is already filled up, then we guarantee that there is no LSN hole in that range. Then the thread checks the next entry of the hopping index. Checking entries of the hopping index continues until the ELEDA worker confronts with an entry whose byte count is smaller than 2^H. Then it advances the head pointers of both the h-list and c-list of all per-thread grasshopper lists, to advance SBL. In principle, we are able to advance SBL to the LSN range boundary (i.e., $i \times 2^H$) at which hopping stops. However, we have to be careful here since there can be a log placed across the LSN range boundary, which must not be partially flushed for correctness. In that case, SBL is set to the *end_lsn* of the log (see line 21 of algorithm 1).

After the ELEDA worker thread advances the head of the h-list, it deletes all pointer pairs preceding the new head in the list. It is worth noting that the SBL-Hopping algorithm modifies only the head fields of the grasshopper lists, by the access rules in Table 2. `SBL-Hopping()` of algorithm 1 describes the general algorithmic sequences of SBL-hopping.

4.2.2 SBL-crawling algorithm

For SBL-crawling, we use the LSN-heap built from the head nodes of per-thread crawling lists. Hence, the maximum number of nodes in the LSN heap is equal to the number of database threads.

As shown in Figure 7, the LSN-heap is exclusively managed by the ELEDA worker thread, and no other threads can access it. As summarized in Table 2, the head LSN nodes in crawling lists are

also modified by the ELEDA worker. These access restrictions on concerned shared data structures enable the ELEDA worker thread to manipulate the LSN-heap in a thread-safe way, especially when advancing SBL while other database threads run. Figure 7 depicts the concurrent operations; while database threads insert newly created pending LSN nodes to their own crawling list by modifying the *tail* pointer, the ELEDA worker thread can insert new head LSN nodes to the LSN-heap and adjust the head pointers of crawling lists concurrently, as long as *heads* do never overrun *tails*.

The top of the LSN-heap is guaranteed to be the smallest pending LSN among all pending ones, owing to both the total order property of each crawling list and the very nature of min-heap. The top element of the heap can be removed when its *start_lsn* is contiguous to the current SBL. As the ELEDA worker thread pops the top pending LSN from the *LSN-heap*, the *end_lsn* of the top element will be the new SBL. Once the top element is removed, the ELEDA worker adjusts the head of the c-list from which the top pending LSN came, to the next pending LSN node. Once the head of the c-list is adjusted, we insert the new head node of the c-list to the LSN-heap. When the *start_lsn* of the head of the c-list crosses the LSN range boundary (i.e., $i \cdot 2^H$), then the head of the h-list should also be adjusted to the next node, which is the first/smallest pending LSN in the next LSN range. The ELEDA worker performs SBL-crawling for advancing SBL as long as SBL falls in the same entry of the hopping index with the most recently allocated LSN. Otherwise if the hopping index entry equals 2^H, we switch to SBL-hopping.

4.3 Flushing Logs

Recall that the last stage of ELEDA logging pipeline is to flush all sequentially buffered logs to stable storage. For this purpose, ELEDA bookkeeps the most recently flushed SBL, defined as *storage durable LSN* (SDL). The SDL is stored in stable storage. The flushing operation is straightforward in that an ELEDA flusher thread keeps flushing the logs, starting from the current SDL to SBL. Since SBL is always reset to the end of a log, no logs can be partially written, leaving the remaining part in the volatile log buffer. Once the thread receives a completion notification, then it advances SDL to SBL. Once SDL is advanced, the flusher thread stores SDL to stable storage and frees the log buffer area in between SDL and SBL to be used for incoming logs. Since we use NVMe SSD devices for stable storage, the I/O unit is tailored to maximize the I/O throughput of sequential writes for given devices. This inevitably brings us a tradeoff between high bandwidth utilization and low latency. Choosing a right point on this tradeoff depends on the workload characteristics, such as an average size of a log and the maximum degree of concurrency.

4.4 Garbage-collection & Callbacks

While the ELEDA worker thread moves the heads of h- and c-lists, a database thread should move the *gc* pointer (i.e., garbage collection point) to collect garbage LSN nodes in the c-list and to invoke callback functions to inform waiting transactions of the completion. The physical removal of pending LSN nodes is done after the callback is invoked. Moving the *gc* pointer continues until the *gc* and SDL logically point to the same pending LSN. Note that *gc* pointer is exclusively accessed by the owner database thread.

4.5 Invariants for Correctness

In ELEDA, there are three types of the producer/consumer structures, all of them allow concurrent executions on concerned shared data structures. Notoriously hard to reason are these concurrent executions on shared data structures that may yield undefined behaviors. We therefore specify three important invariants that must be preserved throughout the concurrent executions. Three invariants are algorithm-specific refinements of the general invariant discussed in §4.1.1, and operations on shared variables strictly follow the access rules in Table 2. We express the specifications of invariants using Hoare logic [27, 28]: $\{P\}$ C $\{Q\}$ where P and Q are pre- and post-states satisfying relevant conditions, and C is a sequence of commands. We abbreviate SBL advancement, garbage-collection and log flushing as SA, GC and LF, respectively. Three invariants are specified as follows:

Spec1. $\underline{\{head \mapsto v \bigwedge tail \mapsto v\}\ \text{SA}\ \{head \mapsto v\}}$: If both *head* and *tail* point to the same pending LSN node (v), the ELEDA worker thread (consumer) is **not** permitted to move the *head* **while** a database thread (producer) is allowed to update the *tail*.

Spec2. $\underline{\{gc \mapsto v \bigwedge SDL \mapsto v\}\ \text{GC}\ \{gc \mapsto v\}}$: If both *gc* and SDL logically point to the same pending LSN (v), the database thread (consumer) is **not** permitted to move the *gc* **while** the ELEDA flusher thread (producer) is allowed to update the *SDL*.

Spec3. $\underline{\{SDL \mapsto v \bigwedge SBL \mapsto v\}\ \text{LF}\ \{SDL \mapsto v\}}$: If both *SDL* and *SBL* have the same value (v), the ELEDA flusher thread (consumer) is **not** permitted to flush anymore **while** the ELEDA worker thread (producer) is allowed to update SBL.

Three invariants forbid one of concurrent threads from mutating concerned data, and these specifications indeed work as *linearization points* [26] that are used as a correctness condition for our GRASSHOPPER data structures. We strictly enforce these invariants in concerned producer/consumer structures to ensure correctness of concurrent executions on shared data structures in ELEDA.

Memory model. Modern microprocessors allow out-of-order execution to enhance performance of programs, and compilers do similar work (i.e., instruction reordering) when generating machine code. This inevitably brings memory ordering issues to multithreaded programs. To enforce memory ordering constraints on code that reads or updates shared variables in ELEDA, we explicitly use memory barriers (i.e., `mfence`) to guarantee the *read-after-write* (RAW) order for correct synchronization.

5. IMPLEMENTATIONS

We have implemented a prototype of ELEDA as a shared library and then plugged it to two storage engines, WiredTiger storage engine and the official EPFL branch of Shore-MT, with proper modifications being made. WiredTiger and Shore-MT use the consolidation array technique [31, 32] in their log manager.

WiredTiger storage engine. WiredTiger has been adopted recently as a default storage engine for MongoDB and offers multiversion concurrency control via a concurrent skip-list for maximizing update concurrency. Unlike conventional database engines, WiredTiger's transaction creates a single log for all updates it made when it commits. This limits the highest isolation level to SNAPSHOT ISOLATION [5]. Owing to the absence of database lock management, WiredTiger storage engine achieves higher performance than Shore-MT engine. ELEDA replaced WiredTiger's log manager implementing the consolidation array technique. While we implement *latency-hiding techniques*, we add thread-local *parking slots* to park pre-committing transactions. Parked transactions will later be woken up and be allowed to deliver results to the client.

Shore-MT storage engine. The EPFL branch of Shore-MT fully implemented several techniques [30, 32, 45–47] offered to improve the multicore scalability of database locking, latching, and logging. In the context of this paper, we have only enabled the logging optimizations from this codebase. We have implemented ELEDA to Shore-MT with Aether [31]. We replaced its consolidation array-based logging subsystem and modified its flush pipelining implementation for transaction switching.

Table 3: Supermicro Server 6028R-TR specifications.

Component	Specification
Processor	18-Core Intel Xeon E5-2699 v3
Processor Sockets	2 Sockets
Hardware Threads	36 (HyperThreading Disabled)
Clock Speed	2.3 GHz
L3 Cache	45 MiB (per socket)
Memory	488 GiB DDR4 2400 MHz
Storage	Samsung SM1725 NVMe SSD (3.5 TB)

6. EVALUATION

For evaluation, we use three workloads; the key-value workload (YCSB) [14], the online transaction processing workload (sysbench) [35] and a logging microbenchmark.

6.1 System Setup

All storage engines are running on a 36-core server specified in Table 3. To reduce irrelevant lock contention, we run all experiments with Hyperthreading being disabled. Two NVMe SSDs, each of which has the peak I/O speed of 3.5 GiB/s with 50 μs delay for the sequential writes of 16 KiB block, are used. ELEDA stripes log data across them with 512 KiB of stripe unit. The I/O unit for flushing logs is set to 64 KiB throughout all experiments, unless stated otherwise. For all systems, we use three variants; 1) a vanilla system on NVMe SSDs (NVMe), 2) a vanilla system on the *tmpfs* in-memory file system (tmpfs) and 3) an ELEDA-based system on NVMe SSDs (ELEDA). We used *tmpfs*, although durability is broken, to focus on the multicore scalability of log buffering in vanilla systems. Hopping distance is set to 4 MiB.

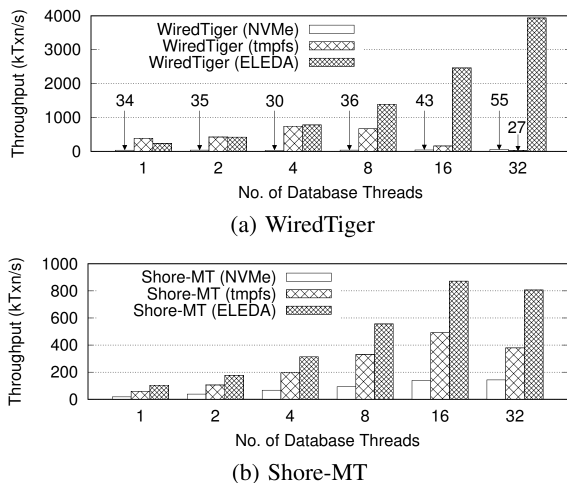
We run both vanilla and ELEDA-based Shore-MT storage systems with the default configuration, except that we provide 200 GiB memory for a buffer pool and a 10 GiB log file. The log buffer size is set to 32 MiB. For log flushing policy, we use the default setting which forces the log buffer to the log file by the WAL principle. This means that it ensures full ACID compliance. All transactions in Shore-MT experiments are committed after flushing logs. WiredTiger is configured with 60 GiB of cache size. A transaction in WiredTiger flushes its log to the stable storage when it commits (i.e., commit-level durability). To expose performance bottlenecks in database logging and preclude irrelevant overhead arising from other components, we use the isolation level `READ UNCOMMITTED` in all storage engines. By doing this, we focus our attention to the impact of non-scalable locks and the performance upper bound of ELEDA-based systems, not on discovering latent bottlenecks in other places. Note that workloads are configured to access disjoint records or key-values to make the resulting transaction history `SERIALIZABLE`.

6.2 Key-value Workload

The YCSB benchmark consists of representative transactions handled by web-based companies, as can be seen in many studies [3, 37, 51, 54]. To demonstrate that ELEDA-based systems can perform fast, scalable logging under high update workloads, we run ELEDA-based WiredTiger and Shore-MT on the YCSB benchmark with workload A(50% Reads and 50% Writes) and 1 KiB payload.

6.2.1 Throughput

Figure 8 shows the throughput results. In all systems, ELEDA-based systems improved performance with a wide margin. Noticeable is the wide performance gap (~10×) between ELEDA-based and vanilla systems with 1 database thread that is primarily owing to the latency-hiding technique allowing a database thread to execute transactions not conflicting with parked ones. The benefit of the latency-hiding technique under low load can be consistently observed throughout all experiments. In Figure 8(a), WiredTiger



(a) WiredTiger

(b) Shore-MT

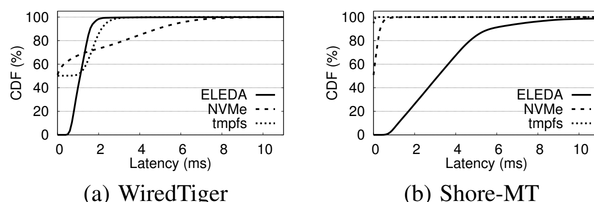
Figure 8: Throughput on the key-value workload.

(ELEDA) peaks at 3.9 million Txn/s with 32 database threads. Meanwhile, the throughput of WiredTiger (NVMe) is not commensurate with the number of database threads: the throughput almost flattens out from the beginning. Interestingly, WiredTiger (*tmpfs*) severely suffers the scalability problem after it peaked at 740 kTxn/s (with 4 database threads). The throughput improvement that ELEDA contributes with 32 threads is 71×. The I/O bandwidth consumed by WiredTiger (ELEDA) peaks at 2.2 GiB/s with 32 database threads, while WiredTiger (NVMe) peaks at 30 MiB/s with 32 database threads. Noticeable is the performance drop in WiredTiger (*tmpfs*), due to the non-scalable lock in the consolidation array technique.

Figure 8(b) shows that Shore-MT (ELEDA) improves the transaction throughput compared to the other two systems. We note that Shore-MT (ELEDA) exhibits slight performance drop-off after it peaks at 898 kTxn/s with 16 threads. In-depth looking through profiling reveals that it is due to the non-scalable algorithm for inserting a new transaction into the transaction list to get a transaction ID, which was also spotted as the main culprit for scalability problems in [33]. Although Shore-MT (ELEDA) showed saturated performance due to other cause that is beyond the scope of the present work, the performance behaviour observed up to 16 threads indicates that ELEDA can scale as more threads join if the non-scalable algorithm is resolved. Shore-MT (NVMe) and Shore-MT (*tmpfs*) peak at 143 kTxn/s (32 threads) and 492 kTxn/s (16 threads). The I/O bandwidth consumed by Shore-MT (ELEDA) peaks at ~190 MiB/s with 16 threads, while Shore-MT (NVMe) also peaks at ~35 MiB/s with 32 threads. The throughput improvement ELEDA contributed to the Shore-MT engine with 32 threads is ~6.3×.

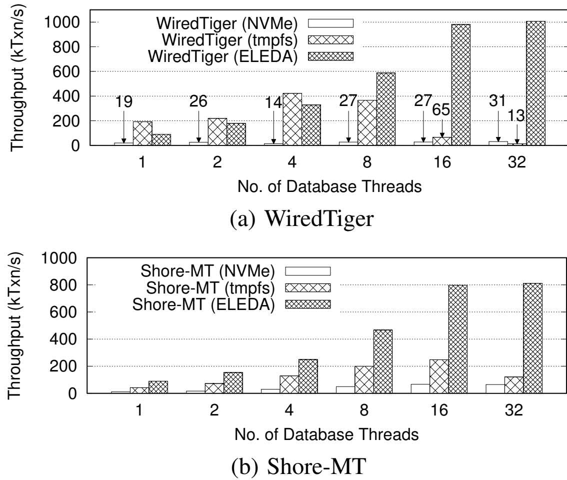
6.2.2 Commit latency

Figure 9 shows the CDF of the commit latencies for all storage engines with different logging methods. WiredTiger (ELEDA)



(a) WiredTiger (b) Shore-MT

Figure 9: CDFs for the commit latencies on the key-value workload: WiredTiger and Shore-MT with 32 threads.



(a) WiredTiger

(b) Shore-MT

Figure 10: Throughput on the OLTP workload.

has the average commit latency of 1.3 ms, whereas WiredTiger (NVMe) has 576 μs. Shore-MT (ELEDA) shows worse behaviour (the average commit latency of 6.4 ms) than Shore-MT (NVMe) (171 μs for the commit latency). In Shore-MT, a transaction generates 3 logs, and 2 out of 3 logs have a 40 bytes payload. This incurs nontrivial processing overhead in maintaining a crawling list. In terms of the average commit latency, Shore-MT (*tmpfs*) is better than ELEDA-based and vanilla systems. However, the average commit latency of WiredTiger (*tmpfs*) is 1.1 ms, which is worse than WiredTiger (NVMe) or comparable to WiredTiger (ELEDA) with 32 database threads because of heavy lock contention in logging.

6.3 Online Transaction Processing Workload

For experiments with the OLTP workload, we first use a variant of sysbench OLTP workloads and measure throughput and commit latency of ELEDA-based systems under high loads. For evaluation, we wrote sysbench-like programs using transaction APIs exported from Shore-MT and WiredTiger storage engines.

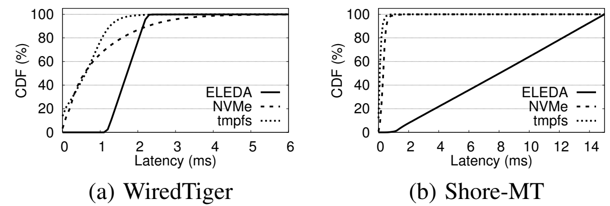
6.3.1 Throughput

Figure 10 shows the transaction throughput results. WiredTiger (ELEDA) showed similar behavior as was shown in Figure 8 in that it scales well as we increase the number of database threads. In Figure 10(a), WiredTiger (ELEDA) peaks at 1 million Txn/s with 32 threads, while WiredTiger (NVMe) and WiredTiger (*tmpfs*) peak at 30 kTxn/s (32 threads) and 421 kTxn/s (4 threads), respectively. Note that WiredTiger (*tmpfs*) again shows performance collapses as load increases due to the non-scalable lock. The I/O bandwidth consumed by WiredTiger (ELEDA) peaks at 606 MiB/s while WiredTiger (NVMe) peaks at 84 MiB/s, both with 32 threads. The throughput improvement in WiredTiger with 32 threads is $\sim$32$\times$.

Shore-MT also shows similar performance behavior likewise. The throughput of Shore-MT (ELEDA) peaks at 810 kTxn/s with 32 threads, while Shore-MT (NVMe) and Shore-MT (*tmpfs*) peak at 67 kTxn/s (16 threads) and 248 kTxn/s (16 threads), respectively. The I/O bandwidth consumed by Shore-MT (ELEDA) peaks at 315 MiB/s with 32 database threads while Shore-MT (NVMe) peaks at 269 MiB/s with 16 database threads. The throughput improvement in Shore-MT with 32 database threads is $\sim$12$\times$.

6.3.2 Commit latency

Figure 11 shows the CDF of the commit latencies with the OLTP workload. WiredTiger (ELEDA) has an average commit latency of



(a) WiredTiger (b) Shore-MT

Figure 11: CDFs for the commit latencies on the OLTP workload: WiredTiger and Shore-MT with 32 threads.

3.7 ms, whereas WiredTiger (NVMe) and WiredTiger (*tmpfs*) have 1 ms and 2.5 ms. Shore-MT (ELEDA) shows worse behaviour (the average commit latency of 10.9 ms) than Shore-MT (NVMe) and Shore-MT (*tmpfs*) (439 μs and 222 μs for the average commit latencies). Although the average commit latency of WiredTiger (ELEDA) is longer than the other two systems, it scales the performance as the count of threads increases by eliminating lock contention observed in vanilla systems.

6.3.3 Throughput with TPC-C workloads

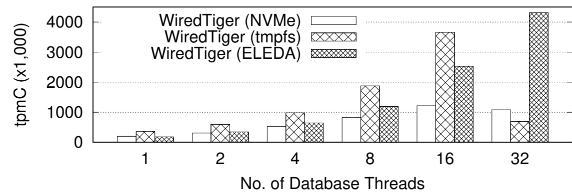


Figure 12: TPC-C benchmark result of WiredTiger.

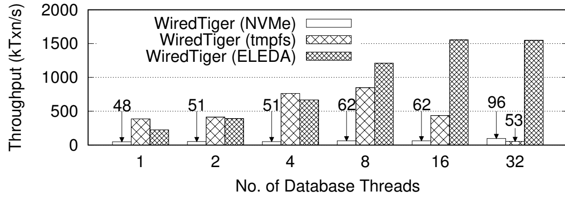
Next, we use TPC-C workloads for measuring transaction throughput and also discuss effects of long transactions using `NEW_ORDER` and `PAYMENT` transactions. For this experiment, we implemented the WiredTiger-based TPC-C application running under `SNAPSHOT ISOLATION` (SI)[3]. The number of warehouses is 32 with each database thread executing a transaction against the dedicated warehouse, avoiding unwanted contention. Figure 12 shows tpmC as we increase database threads. It shows that WiredTiger (tmpfs) scales tpmC throughput up to 16 threads, and then tpmC collapses afterward due to the contention in the log manager. WiredTiger (ELEDA) shows no performance collapse and handles long transactions without extra processing overhead.

6.4 Experiments with Different Settings

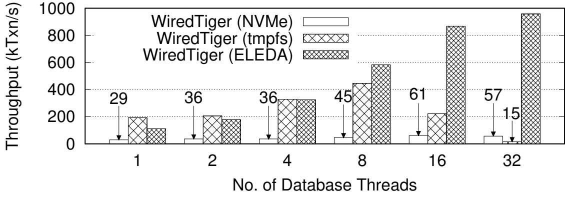
6.4.1 Evaluation under snapshot isolation

To see the performance behavior of ELEDA-based systems under higher isolation level, we conduct experiments measuring the performance of WiredTiger under `SNAPSHOT ISOLATION` (SI) that is the highest isolation level supported by WiredTiger. Figure 13 shows results with key-value and OLTP workloads. As shown in the figure, WiredTiger (NVMe) is mainly affected by synchronous I/O delay, and WiredTiger (*tmpfs*) suffers the serialization bottleneck by lock contention. WiredTiger (ELEDA) shows better performance, but the performance is suddenly limited as the count of database threads exceeds 16. In-depth profiling spotted a culprit for this issue. Code flows intended to obtain a snapshot that determines the visibility of key-value records, spend $\sim$30% of time with 32 database threads. This is engine-specific overhead, and it

[3]Fekete at al. [19] proved that TPC-C benchmarks, running under SI, ensures serializable execution.



(a) Key-value workload



(b) OLTP workload

Figure 13: Throughput of WiredTiger under SI.

is irrelevant to ELEDA. If it were removed, WiredTiger (ELEDA) would show the same peak throughput as shown in Figure 8(a). With OLTP workloads (Figure 13(b)), WiredTiger (ELEDA) shows a similar behavior as was shown in Figure 10(a). Though further investigation is needed, we believe that the overhead of WiredTiger's snapshot routines only matters under very high loads (> 2M Txn/s).

6.4.2 Key-value workload with varying size

Next, we explore the performance of WiredTiger with payload size being varied. For the evaluation, we use key-value workloads by varying the payload size from 512 bytes to 5 KiB. Since the workload is configured to use larger payload, the I/O unit is also reset to 512 KiB for ELEDA to utilize maximum device bandwidth. Figure 14 shows the results of WiredTiger (ELEDA) with the number of database threads being fixed to 32. As payload size increases, the performance of WiredTiger (ELEDA) gradually degrades mainly because the amount of log data grows. Throughput of other two systems remains the same, since they suffer either synchronous I/O delay or scalability bottleneck with 32 database threads. As shown in Figure 14, the I/O bandwidth consumed by WiredTiger (ELEDA) peaks at 6.2 GiB/s when the payload size is 5 KiB, while the write bandwidth consumed by WiredTiger (NVMe) is unchanged regardless of payload size.

6.4.3 Comparison with NVM-based logging

As discussed in §2.3.1, NVM-based logging has clear advantages over other proposals owing to the attractive characteristic of NVM itself. The proposal of Fang et al. [18] is a good example. In this section, we conduct performance comparison between the approach of Fang et al. on the emulated NVM with ELEDA on

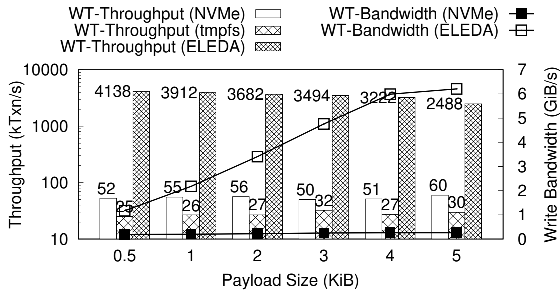
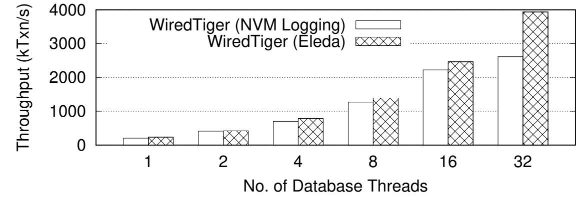
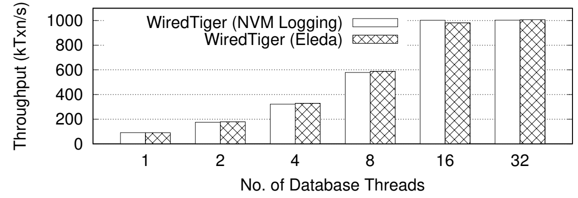


Figure 14: Performance of WiredTiger w/ varying payload size.



(a) Key-value workload



(b) OLTP workload

Figure 15: Throughput comparison between NVM-based logging [18] and ELEDA

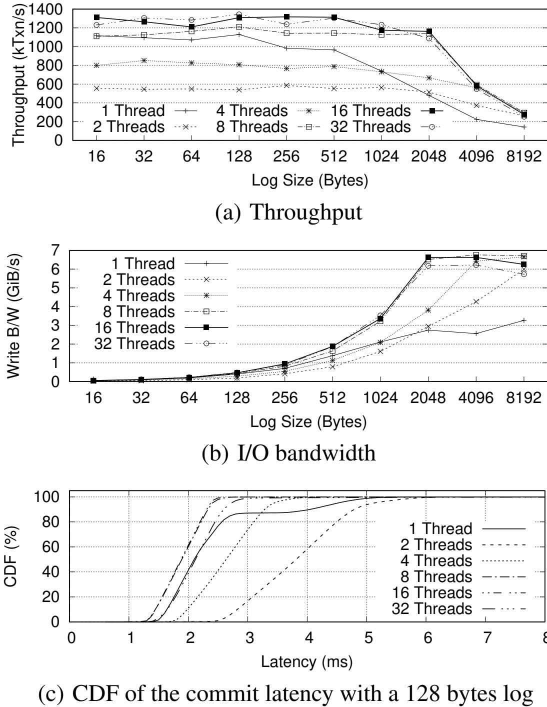
SSDs. The objective of this experiment is to shed light on ELEDA's potential application in conventional database systems that mostly run on commodity hardware without NVM, by comparing performance between *software-only* ELEDA and *hardware-assisted* NVM logging. For this experiment, we use a small segment of shared-memory and treat it as fast NVM. We also implement the approach of Fang et al. in ELEDA, by following their algorithms. We then measure the performance of WiredTiger with key-value and OLTP workloads. As shown in Figure 15, experimental results demonstrate that pure software architecture (ELEDA) on a commodity server can attain comparable (or sometimes better) performance than NVM-based logging. It is noteworthy that, to exploit NVM-based logging in conventional database systems, we need to expense not only the cost for hardware (NVM module) but also (and more importantly) the cost for software modification to make them NVM-aware.

6.5 In-depth Analysis

6.5.1 Microbenchmark

To measure various performance metrics, we build a database logging microbenchmark that creates a transaction which is designed to buffer three separate logs to ELEDA, followed by the transaction commit. For accurate measurements, transaction switching and callback mechanisms are also implemented, and callbacks are invoked after ELEDA flushes logs to the storage devices. We measure throughput, I/O bandwidth and the CDF of commit latency by varying the number of database threads and the log size. To maximize device utilization, we reset the I/O unit to 512 KiB.

Figure 16(a) shows the throughput graphs of ELEDA, as we vary the number of database threads and the log size. ELEDA's throughput increases as we add more database threads. The peak throughput (1.34 million Txn/s) is achieved with a 128 bytes log, and sustained until the log size is increased to 2 KiB. After that point, the throughput decreases as we grow the log size. Figure 16(b) shows I/O bandwidth consumed by ELEDA. Since we use two NVMe SSDs, the maximum aggregated bandwidth for sequential write is around 6.92 GiB/s. The figure clearly shows that ELEDA can utilize more bandwidth as we increase the log size and the count of database threads. When the log size reaches 4 KiB, ELEDA utilizes full device bandwidth. Figure 16(c) shows the CDF of a commit latency with a 128 bytes log. The CDF indicates that increased CPU



(a) Throughput

(b) I/O bandwidth

(c) CDF of the commit latency with a 128 bytes log

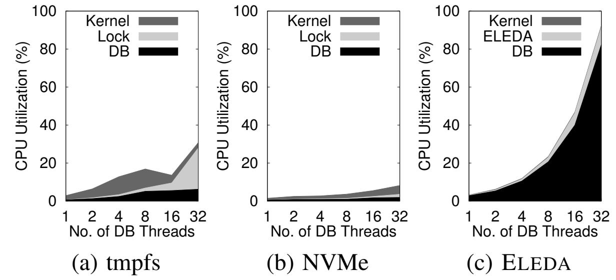
Figure 16: Microbenchmark measures throughput, I/O bandwidth and CDF of the commit latency.

utilization by more active threads widens the latency distributions. One interesting phenomenon comes to our attention; performance results with a single database thread are better than those with two and four threads. This is because Linux dispatches threads evenly to different processor sockets to maximize the use of cache memory, but this increases inter-socket memory bus traffic with cache invalidation messages in ELEDA. Giving the processor affinity to threads easily makes the graph look better, but we do not change the configuration since databases would not do that.

6.5.2 *Profiling CPU utilization*

Profiling reveals detailed information about various system activities, especially about where the time goes. Since WiredTiger shows larger performance improvement with ELEDA than Shore-MT, we choose it for profiling.

Figure 17 shows the breakdown of profiled results of WiredTiger (*tmpfs*), WiredTiger (NVMe) and WiredTiger (ELEDA), with the key-value workload. As we increase the number of database threads, vanilla engines cannot achieve commensurate CPU utilization due to too much portion of waiting/sleeping activities primarily caused by the lock contention. With 32 threads, WiredTiger (*tmpfs*) spends 20% time in spin-waiting while WiredTiger (NVMe) has negligible portion. Since the modern implementation of a mutex, upon contention, first incurs spin-waiting, followed by sleeping, WiredTiger (*tmpfs*) shows substantial portion of spin-waiting due to the removal of synchronous I/O delay. Spin-waiting on a mutex variable however bursts memory bus with cache invalidation messages, thus resulting in performance collapses in WiredTiger (*tmpfs*). This definitely prohibits the system from gaining more utilization. The same explanation applies to WiredTiger (NVMe). The difference is, WiredTiger (NVMe) with 32 database threads spends 88% of time in doing nothing, but sleeping. At this time, the synchronous I/O delay causes lock contention to reach sleeping most time. This is why we observed such low throughput numbers from vanilla



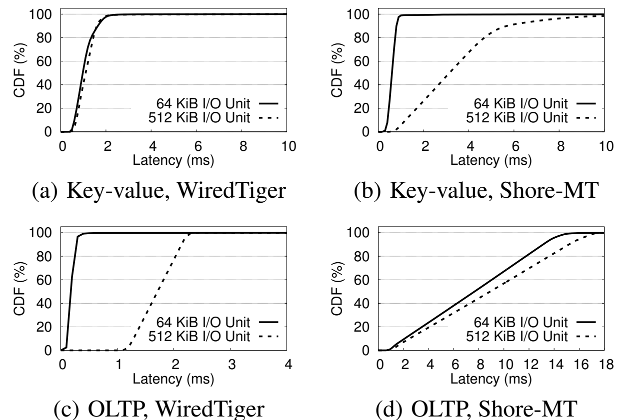
(a) tmpfs (b) NVMe (c) ELEDA

Figure 17: Profiling CPU utilization: WiredTiger on the key-value workload.

WiredTiger systems. In contrast, WiredTiger (ELEDA) show neither spin-waiting nor sleeping owing to the absence of lock contention, and CPU utilization is therefore commensurate with the count of database threads. Figure 17(c) shows time spent in ELEDA. Note that WiredTiger (ELEDA), unlike other systems, does have negligible kernel time that is usually increased due to lock contention (e.g., `futex()`).

6.5.3 *Effects of I/O unit size*

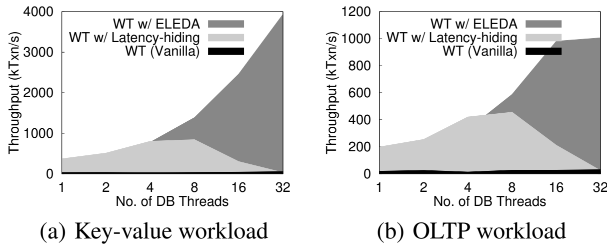
As we discussed in §4.3, the commit latency is critically influenced by the I/O unit. A larger I/O unit for higher device utilization takes longer to fill the I/O buffer, but this increases the commit latency. To analyze the effect of I/O unit on commit latency and bandwidth utilization, we use different I/O units - 64 KiB and 512 KiB - for ELEDA-based systems, all running 32 database threads with key-value and OLTP workloads. In all results shown in Figure 18, the smaller I/O unit provides much shorter commit latencies than the larger I/O unit. Hence, if there is a strict service-level agreement on response time, adjusting the I/O unit for different types of workloads is required. We then measure the max bandwidth utilization for different I/O units by using large size logs, although results are not plotted here. The max bandwidth utilization ELEDA-based systems can achieve for 64 KiB I/O unit is ∼3.2 GiB/s, while ELEDA with 512 KiB I/O unit utilizes full device bandwidth (i.e., 6.9 GiB/s). However, the commit latency for 64 KiB I/O unit is an order of magnitude shorter than that for 512 KiB I/O unit. One may think that multiple flusher threads may help increasing the utilization with a smaller I/O unit, but preserving the sequentiality of logging with multiple flusher threads would be very challenging. We leave this issue open to our community.



(a) Key-value, WiredTiger (b) Key-value, Shore-MT

(c) OLTP, WiredTiger (d) OLTP, Shore-MT

Figure 18: Effects of the I/O unit to commit latencies.



(a) Key-value workload (b) OLTP workload

Figure 19: Performance breakdown of WiredTiger (ELEDA).

6.5.4 Performance breakdown of ELEDA

The next experiment focuses on ELEDA and measures the contribution of GRASSHOPPER and the *latency-hiding technique* on the achieved throughput. The contribution of GRASSHOPPER is further broken down by SBL-crawling and SBL-hopping. For the evaluation, we selectively enable one or both of the latency-hiding technique and GRASSHOPPER in WiredTiger (ELEDA); they are denoted as “WT w/ Latency-hiding” and “WT w/ ELEDA”, respectively. For comparison, we use vanilla WiredTiger (NVMe) as a baseline (i.e., “WT (Vanilla)”).

Figure 19 shows the performance breakdown of ELEDA with key-value and OLTP workloads. The contribution of the latency-hiding technique is dominant when the count of database threads is low because a small number of threads incur negligible lock contention overhead. As load increases, the sole use of the latency-hiding technique reveals its limitation in scaling the performance, primarily due to lock contention with multiple threads of execution. If lock contention is left unaddressed, the effect of the latency-hiding technique diminishes, and the system suffers the serialization bottleneck (see the throughput collapse of “WT w/ Latency-hiding” in Figure 19).

Once the latency-hiding technique reaches its limitation (i.e., ∼4 database threads), GRASSHOPPER plays a major role in achieving scalable performance by eliminating the scalability bottleneck. Hence GRASSHOPPER contributes to achieving commensurate performance as the count of database threads increases (i.e., the results of “WT w/ ELEDA”). Since GRASSHOPPER is designed to resolve serialization bottleneck on multicores, it is of use to the system that shows a clear indication of performance bottlenecks caused by lock contention; in other words it is of no use to the system if synchronous I/O delay remains unresolved. Overall, performance breakdown shows that ELEDA successfully resolves whichever of the synchronous I/O delay and the scalability bottleneck is a matter of pressing concern.

We further break down the impact of GRASSHOPPER on ELEDA to have a deeper understanding of interplay between SBL-crawling and SBL-hopping. For this evaluation, we use WiredTiger (ELEDA)

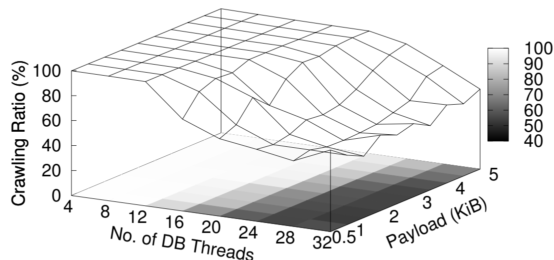


Figure 20: Performance breakdown of GRASSHOPPER.

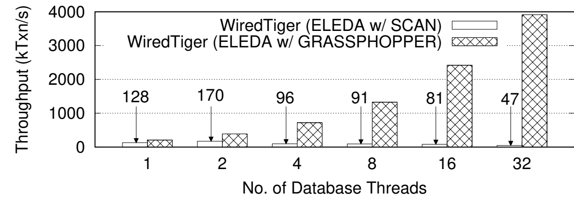


Figure 21: Throughput comparison between GRASSHOPPER and a linear scanning approach on the key-value workload.

on the key-value workload and measure the contribution of SBL-crawling on the measured throughput. The contribution of crawling is quantified by the ratio of the SBL increment conducted by SBL-crawling to the total, and it is denoted as “Crawling Ratio” in Figure 20. SBL-crawling makes a principal contribution to the performance when the thread count is low where the overhead of managing the LSN-heap can be neglected. As the count of threads increases and payload size is decreased, SBL-crawling incurs significant overhead in manipulating the LSN-heap. This causes SBL-crawling to start falling behind, and this is the moment when SBL-hopping plays a key role in advancing SBL in a timely manner. The effect of this interplay is well presented in Figure 20.

Finally, we compare GRASSHOPPER with a simple scanning approach for tracking LSN holes to see the clear benefit of GRASSHOPPER. For the evaluation, we use a bitmap with each bit representing a single byte of a log buffer. A dedicated worker simply scans a bitmap to advance SBL while database threads set bits after copying their logs, except that database threads stop before they outrun the dedicated worker thread. We run the same YCSB workload used before against the WiredTiger engine. As shown in Figure 21, the scanning approach hits its max speed quickly (i.e., ∼40 MiB/s) even with 2 database threads. We believe that designing an efficient scanning mechanism would require nontrivial effort and warrant meaningful research outcomes.

7. CONCLUSIONS

We have observed that contemporary database systems relying on ARIES (or its variants) have faced significant performance bottlenecks in centralized logging on computing platforms with dozens of processor cores and fast storage devices. With two open-source systems, we have identified non-scalable locks (or algorithms) used in the centralized logging as the main bottleneck. To address the problem, we proposed a fast, scalable logging method, ELEDA, that is based on highly concurrent data structures. ELEDA also integrated transaction switching and asynchronous callback mechanisms to address synchronous I/O delay. Our evaluation demonstrated that ELEDA is modular and applicable to any transaction systems suffering the bottleneck in centralized logging. As hardware vendors provision more cores, scalability issues in system software deserve careful consideration so as not to have undesirable performance problems. In this regard, ELEDA provides one method to address the key challenge in database logging, that is an essential prerequisite for high performance transaction systems.

8. ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea grants (2017R1A2B4006134 and 2015M3C4A7065645) funded by the Korea government. This work was also supported by the R&D program of MOTIE/KEIT (10077609). We would like to thank the anonymous PVLDB reviewers for valuable comments which helped to improve the paper.

9. REFERENCES

[1] Amazon Web Services Inc. SAP HANA on the AWS X1 Instance with 2 TiB Memory and 4-way Intel® Xeon® E7-8880 v3. https://aws.amazon.com/sap/solutions/saphana/.

[2] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[3] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, Nov. 2016.

[4] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[7] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[8] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2012.

[9] S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 73–86, New York, NY, USA, 2009. ACM.

[10] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[11] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[12] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. ACM.

[13] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[15] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[16] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. ACM.

[17] EPFL Official Shore-MT Page. Shore-MT: A scalable storage manager for the multicore era. https://sites.google.com/site/shoremt/home.

[18] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1221–1231, Washington, DC, USA, 2011. IEEE Computer Society.

[19] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[20] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. PCMLogging: Reducing transaction logging overhead with PCM. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2401–2404, New York, NY, USA, 2011. ACM.

[21] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.

[22] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the system R database manager. *ACM Comput. Surv.*, 13(2):223–242, June 1981.

[23] G. Heiser, E. Le Sueur, A. Danis, A. Budzynowski, T.-l. Salomie, and G. Alonso. Rapilog: Reducing system complexity through verification. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 323–336, New York, NY, USA, 2013. ACM.

[24] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Found. Trends databases*, 1(2):141–259, Feb. 2007.

[25] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[26] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[28] C. A. R. Hoare. Proof of a program: Find. *Commun. ACM*, 14(1):39–45, Jan. 1971.

[29] T. Horikawa. Latch-free data structures for dbms: Design,

implementation, and evaluation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 409–420, New York, NY, USA, 2013. ACM.

[30] R. Johnson, I. Pandis, and A. Ailamaki. Improving oltp scalability using speculative lock inheritance. *PVLDB*, 2(1):479–489, Aug. 2009.

[31] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A scalable approach to logging. *PVLDB*, 3(1-2):681–692, Sept. 2010.

[32] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal*, 21(2):239–263, 2011.

[33] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 73–84, New York, NY, USA, 2013. ACM.

[34] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting NVRAM in write-ahead logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 385–398, New York, NY, USA, 2016. ACM.

[35] A. Kopytov. Sysbench: A system performance benchmark. `https://github.com/akopytov/sysbench/`.

[36] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[37] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in deuteronomy. *PVLDB*, 8(13):2146–2157, Sept. 2015.

[38] Microsoft Corporation. SQL Server 2016: Control transaction durability. `https://msdn.microsoft.com/en-us/library/dn449490.aspx`.

[39] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[40] MongoDB Inc. Wiredtiger: Commit-level durability that can affect performance. `http://source.wiredtiger.com/develop/tune_durability.html#tune_durability_flush_config`.

[41] MongoDB Inc. WiredTiger storage engine. `https://docs.mongodb.org/manual/core/wiredtiger/`.

[42] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 1–14, Berkeley, CA, USA, 2006. USENIX Association.

[43] Oracle Corporation. MySQL-5.7: Control option for balancing between strict ACID compliance for commit operations, and higher performance. `http://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html`.

[44] Oracle Corporation. Oracle 12c: Nowait option when transactions commit. `http://docs.oracle.com/database/121/SQLRF/statements_4011.htm#SQLRF01110`.

[45] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2):928–939, Sept. 2010.

[46] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: Page latch-free shared-everything oltp. *PVLDB*, 4(10):610–621, July 2011.

[47] D. Porobic, E. Liarou, P. Tzn, and A. Ailamaki. Atrapos: Adaptive transaction processing on hardware islands. In *2014 IEEE 30th International Conference on Data Engineering*, pages 688–699, March 2014.

[48] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 6(2):145–156, Dec. 2012.

[49] R. Sears and E. Brewer. Segment-based recovery: Write-ahead logging revisited. *PVLDB*, 2(1):490–501, Aug. 2009.

[50] The PostgreSQL Global Development Group. Asynchronous commit that allows flexible trade-offs between performance and certainty of transaction durability. `http://www.postgresql.org/docs/9.4/static/wal-async-commit.html`.

[51] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[52] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, June 2014.

[53] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[54] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 465–477, Berkeley, CA, USA, 2014. USENIX Association.