Operation-Aware Hybrid Locking for Modern In-Memory Indexes

Vishal Gupta
EPFL
vishal.gupta@epfl.ch

Martin Sanchez Lopez
EPFL
martin.sanchezlopez@epfl.ch

Victor Laforet
Inria
victor.laforet@inria.fr

Jean-Pierre Lozi
Inria
jean-pierre.lozi@inria.fr

Sanidhya Kashyap
EPFL
sanidhya.kashyap@epfl.ch

ABSTRACT

Achieving scalable performance in modern in-memory indexes is primarily limited by synchronization. Traditional synchronization approaches apply a single "one-size-fits-all" strategy, ignoring the diverse characteristics of different index operations. For instance, pessimistic lock coupling forces high atomic overhead on all tree traversals, even simple lookup operations. Meanwhile, optimistic queue-based locking, while efficient for lookups, suffers from performance collapse due to shared data movement during high-contention updates.

This paper introduces Opal, a hybrid *operation-aware* lock design for modern in-memory indexes. Opal dynamically selects among three locking mechanisms within a single lock instance based on operation type: (i) optimistic version-based locking for read-only lookups; (ii) lightweight function-pointer-based batching for updates that eliminates shared data movement; and (iii) traditional MCS-based locking for structural modification operations (SMOs), such as node splits and merges, that naturally distributes contention across multiple index nodes. We evaluate Opal on widely-used index structures: a B+ Tree and an Adaptive Radix Tree (ART). Compared to state-of-the-art optimistic locking, Opal improves throughput by up to 2.43× and reduces latency by 80%.

PVLDB Reference Format:
Vishal Gupta, Martin Sanchez Lopez, Victor Laforet, Jean-Pierre Lozi, and Sanidhya Kashyap. Operation-Aware Hybrid Locking for Modern In-Memory Indexes. PVLDB, 19(8): 1804-1817, 2026.
doi:10.14778/3811243.3811253

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rs3lab/opal.

1 INTRODUCTION

Modern OLTP databases rely on concurrent in-memory indexes to deliver high performance on multi-core systems with tens to hundreds of CPU cores [23, 26, 28, 37, 46, 48]. To enable safe concurrent access, these memory-optimized indexes employ synchronization mechanisms that must balance multiple objectives: delivering high throughput across varied workloads, scaling gracefully under contention, maintaining fairness across competing threads, and handling oversubscription scenarios [20, 21] where applications spawn more threads than available hardware contexts.

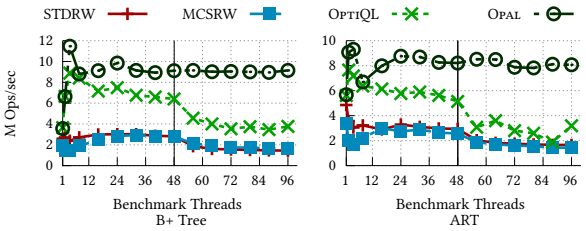

Figure 1: Impact of lock scalability using B+ Tree and ART indexing structures. We compare two pessimistic locks: STDRW and MCSRW, and an optimistic queue-based lock OptiQL [48] with our proposed lock design: Opal.

These synchronization mechanisms for memory-optimized indexes[1] significantly impact performance. For instance, traditional pessimistic lock coupling [4] guarantees safety but suffers from high overhead. Lock-free designs [28, 53] promise wait-free progress but often underperform simpler locking schemes [16, 53], while introducing significant implementation complexity [36, 52]. As a result, recent memory-optimized indexes rely on optimistic lock designs [24, 48], which combine implementation simplicity with strong performance characteristics.

Despite their different philosophies, existing synchronization mechanisms share a critical weakness: they treat all index operations identically. Each approach selects a single synchronization mechanism and applies it uniformly, regardless of whether the operation is a simple lookup, a leaf update, or complex structural change to the index. This *one-size-fits-all* design creates inherent performance tradeoffs. Consider pessimistic lock coupling: to maintain correct reader counts, every tree traversal—even read-only lookups—must execute atomic increment and decrement operations at each tree level. This uniform treatment imposes significant overhead on the most common operation. On the other hand, optimistic locks like OptiQL [48] improve upon this by replacing atomic operations during traversal with lightweight version checks, dramatically accelerating reads. However, for write operations, OptiQL inherits the MCS lock's [38] queue-based design. Lock ownership transfer between threads requires moving shared data cache lines between cores. Under write-heavy contention, particularly when updates concentrate on hot leaf nodes, this data movement becomes a bottleneck that extends sequential execution time. By Amdahl's

[1] They are often called "latches" as opposed to "locks" protecting database content from conflicting transactions. We follow the synchronization literature in using "locks".

Proceedings of the VLDB Endowment, Vol. 19, No. 8 ISSN 2150-8097.
doi:10.14778/3811243.3811253

law [1], this sequential bottleneck fundamentally limits parallel scalability for the index, as shown in Figure 1.

Recent batching-based locks [18, 30, 45] address the data movement bottleneck by executing multiple critical sections sequentially on a single core, thereby maximizing cache locality. These locks encapsulate each critical section as a function pointer, which the lock holder executes on behalf of waiting threads [30, 45], eliminating the need to transfer lock ownership across cores. However, like other synchronization mechanisms, batching-based locks apply their combining strategy uniformly to all operations. Thus, these locks also miss out on opportunities to tailor the synchronization approach to operation-specific characteristics.

Our key insight is that *index operations are not homogeneous*. Different operation types exhibit distinct access patterns, contention characteristics, and synchronization requirements. Lookup operations, which dominate many OLTP workloads [8, 49], traverse the tree by acquiring read locks at each level. These operations benefit most from lightweight, non-atomic synchronization mechanisms such as version checks. Update operations, in contrast, acquire read locks during traversal of non-leaf nodes, but require a write lock on the target leaf node to modify its contents. Because updates do not alter the index structure, they create localized contention hotspots at specific leaf nodes. When multiple threads concurrently target the same leaf, batching these operations on a single core minimizes wasted cycles from repeated cache line movement between cores. Finally, structural modification operations (SMOs) (*e.g.*, node splits and merges) acquire read locks during traversal and write locks on all nodes requiring modification. Unlike updates, SMOs distribute contention across multiple index nodes, making traditional queue-based lock handoffs efficient. Batching SMOs would introduce unnecessary complexity without performance benefit. No single synchronization mechanism can optimize simultaneously for all three operation types.

We present Opal, an operation-aware lock design tailored for modern in-memory indexes. Opal uses the appropriate mechanism for each operation type. For lookups, Opal uses optimistic version checks paired with opportunistic reads: readers perform version checks before and after traversal, proceeding without atomic operations when no writer[2] holds an exclusive lock. This approach matches the performance of existing optimistic locks. For updates that modify leaf node contents without structural changes, Opal relies on a batching-based lock design that uses function pointers. The lock holder executes waiters' requests sequentially, which keeps the leaf node's cache line resident on the local core and eliminates repeated cache line movement. For SMOs that require acquiring multiple locks across the index nodes, Opal employs the traditional MCS-style queue-based locking protocol. Because SMOs naturally distribute lock acquisitions across multiple nodes, contention remains distributed. The queue-based approach provides fair ordering and consistent performance without the complexity of attempting to batch structurally interdependent operations.

Opal also addresses oversubscription—a growing concern in modern database deployments [20, 21]. Traditional approaches experience severe performance degradation when thread count exceeds core count. This becomes critical for highly-contended update operations in memory-optimized indexes. When the OS preempts a thread holding a lock, all threads queued behind it wait idly until the preempted thread is rescheduled, which ends up extending the application's sequential execution phase. In contrast, Opal's batching-based design mitigates this performance collapse: the lock holder can execute multiple batched requests before being preempted by the OS scheduler, amortizing the cost of preemption across many operations.

To show the general applicability of our design, we evaluate Opal using two different index structures: a B+ Tree and a trie (ART [25]). Our evaluation shows that Opal boosts throughput and reduces latency for these index structures by 2.43× and 80%, respectively, compared to the state-of-the-art optimistic lock design, OptiQL. Moreover, Opal improves throughput under 4× oversubscription by up to 10000× compared to OptiQL.

In summary, this paper makes the following contributions:

- **Batching-based lock design for indexes.** We introduce a batching-based lock design that provides better throughput and lower latency across different scenarios.
- **Operation-aware hybrid lock design.** We propose operation-aware hybrid locking for modern in-memory indexes. Lookups require optimistic version checks, updates require function-pointer-based batching, and SMOs require traditional MCS-style locking.
- **Practical lock design.** We demonstrate Opal's resilience to performance collapse under oversubscription and validate its general applicability across different indexes.

2 BACKGROUND

We first describe various lock mechanisms, then explain how they are applied to enable concurrent access in index structures.

2.1 Pessimistic Lock Coupling

Index structures may use pessimistic locks, such as reader-writer locks, on each node to enable fine-grained concurrency [23]. Usually, the reader part is implemented as a centralized atomic counter [28] that tracks the number of threads currently accessing the node in shared mode. The writer part is a TAS (test-and-set) [23] or a queue-based MCS lock [38]. For correct synchronization, indexes use lock coupling with pessimistic locks, which requires acquiring the child node in shared mode before releasing the read lock on the parent node. This acquire/release of a read lock translates to atomic increments and decrements of a centralized counter, causing frequent cache line bouncing of these counter variables. This results in performance collapse as the number of threads increases.

2.2 Optimistic Locking

Optimistic locks [28, 48] address the read-side scalability of pessimistic locks by replacing atomic counters with a version number on each node. A thread reads this version during the acquire phase and validates it during the release phase. By avoiding the atomic operation during the tree traversal, optimistic locks can keep the node cache line containing the version number in the shared state, resulting in faster traversal time compared to pessimistic locks.

For write access, OptiQL [48] uses an MCS-based queue lock [38]. Each thread waiting for the write lock joins a wait queue and waits on its own queue node for the notification from the previous lock

[2]Reading shared data uses a shared lock, and writing requires an exclusive lock. Following synchronization literature, we use "read/write" in the lock algorithm.

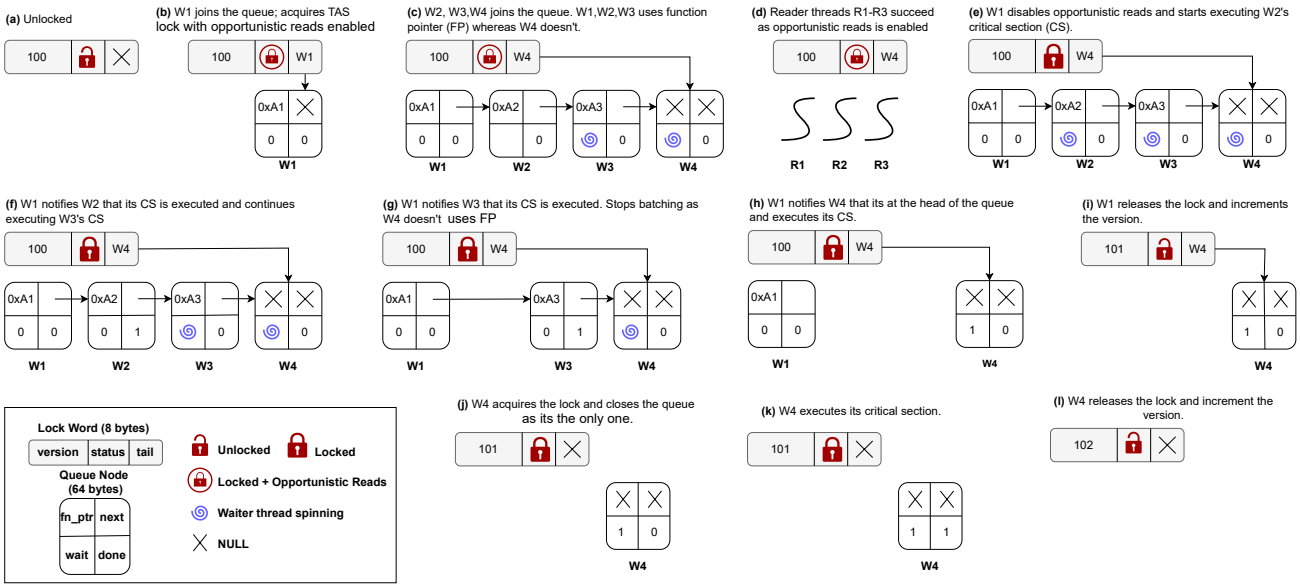

Figure 2: Lock and qnode structures of Opal. Reader threads R1-R3 access shared data. Writer threads W1-W4 need to modify shared data.

holder. Compared to a TAS lock, a queue-based waiting mechanism reduces cache line contention by having each thread spin on a flag within its own dedicated queue node, rather than on a single, shared lock variable. However, after acquiring the lock, the lock-holder thread executes the critical section and accesses the shared data. This access requires bringing the shared data cache line from the previous lock holder's L1 cache, imposing a latency penalty before starting the critical section execution. The new thread is forced to wait for an extra delay because the data it needs has to be physically transferred from the previous thread's core over to its own core. This directly impacts the performance of the lock mechanism.

2.3 Batching-Based Locks

Batching-based lock designs eliminate cache line movement by executing multiple critical sections on the same core [17, 18, 30, 31, 45]. By keeping shared data cache-resident, these designs reduce critical section latency and improve overall throughput. Existing batching approaches differ in how they transfer critical section execution to the lock holder.

Function-pointer-based batching. Multiple designs [30, 45] require applications to encapsulate critical sections as functions. Waiting threads pass function pointers to the lock holder, which executes them directly. Although this approach is lightweight, it cannot handle operations requiring overlapping lock ownership. For example, B+ Tree [23] and ART [25] insert/delete operations use hand-over-hand locking or lock coupling, acquiring the child node's lock before releasing the parent's lock. These overlapping critical sections cannot be cleanly separated into independent functions.

Transparent combining. TCLock [18] overcomes this limitation without application modification. It uses hardware registers to capture critical section state, stack switching to transfer execution to the lock holder, and explicit lock/unlock calls to demarcate critical section boundaries. While transparent combining handles arbitrary critical sections, stack switching triggers pipeline flushes that add non-trivial overhead. For simple operations like updates that do not require hand-over-hand locking, this overhead degrades index throughput compared to the function-pointer-based approach by 1.73×, thereby outweighing the benefits of batching.

Fairness and starvation. Batching improves performance but compromises short-term fairness, as the lock holder is delayed by processing requests for other threads. In existing designs, the number of critical sections is capped at a threshold [17–19], which ensures long-term fairness and prevents starvation (a timeout could also be used [11]). The threshold can be configured to tune the fairness-performance tradeoff.

2.4 Synchronization Usage in Index Structures

We now focus on how indexes use synchronization mechanisms, which reveals why a one-size-fits-all approach is suboptimal.

Lookup and updates. SQL queries like SCAN on a particular column trigger lookup operations for that column's index. Meanwhile, UPDATE queries on columns that are part of a clustered index's key can trigger in-place updates for all nonclustered indexes [14]. Both operations traverse the tree from the root to the leaf node using optimistic reads with lock coupling: acquire the child's read lock before releasing the parent's read lock. Version mismatches trigger traversal restart from the root. At the leaf level, lookups complete with only read lock acquisition, while updates acquire a write lock to modify the leaf contents.

Structural modification operations (SMOs). SQL queries like INSERT or DELETE trigger SMOs in the indexing structure to insert or delete the relevant key. These operations may trigger node splits or merges that propagate up the tree. These operations use optimistic reads during initial traversal while tracking which ancestors have available space. Upon reaching the leaf, they acquire write locks in a bottom-up order until reaching an ancestor with sufficient space, which ensures atomic completion of the structural modification operation.

```
G_UNLOCK = 0, G_LOCKED = 1 # TAS states
G_LOCKED_OPT = 3 # Locked + Enable opportunistic reads

class lock: # state for TAS lock, MCS queue tail and reader version
  state = 0, tail = None, version = 0

class queue_node_struct:
  wait = False, next = None

def read_lock(lock, &version) -> bool:
  *version = lock.version # Store the current version
  return (lock.state == G_LOCKED) # Restart if the lock is locked

def read_unlock(lock, curr_version) -> bool:
  return (lock.state != G_LOCKED && lock.version == curr_version)

def write_lock(lock):
  # Fastpath: Try to acquire the TAS lock
  if CAS(&lock.state, G_UNLOCK, G_LOCKED) == G_UNLOCK:
    return # Got the lock, going to execute the critical section

  # Slowpath: Join the queue and wait
  qnode = thread_local_storage.node
  init_qnode(qnode, wait = True, next = None) #Initialize qnode
  qnode.fn_ptr = None
  # Phase 1A: Busy waiting: Join the queue and wait until notified
  qprev = SWAP(&lock.tail, &qnode) # Atomically add node to tail
  if qprev is not None:            # Check if any waiter is present
    qprev.next = qnode             # Form the queue
    while qnode.wait is True:      # Wait for previous lock holder
      continue                     # to halt my spinning
  # Phase 1B: Head of the queue: Acquire the global TAS lock
  while True: # Wait for the state to be unlocked
    while lock.state != G_UNLOCK:
      continue
    # Acquire lock with opportunistic reads enabled
    if CAS(&lock.state, G_UNLOCK, G_LOCKED_OPT) == G_UNLOCK:
      break  # Got the TAS lock
  # Phase 1C: Notify next waiter or close the queue
  if CAS(&lock.tail, qnode, None) == qnode:
    lock.state = G_LOCKED # Only one node; Disable Opportunistic Reads
    return
  qnext = qnode.next # Someone joined the queue
  while qnext is None:
    qnext = qnode.next
  qnext.wait = False # Next waiter is now at the head of the queue
  lock.state = G_LOCKED # Disable Opportunistic Reads

def write_unlock(lock):
  lock.version += 1 # Increment version
  lock.state = G_UNLOCK # Unlock the lock
```

Listing 1: Pseudocode for read (shared) and write (exclusive) lock.

3 OPAL: OPERATION-AWARE HYBRID LOCK

Opal addresses the one-size-fits-all problem through operation-aware synchronization: matching the lock mechanism to the operation type rather than applying a uniform strategy. For lookups, Opal uses optimistic version-based validation [48]. For updates that concentrate contention at leaves, Opal employs function-pointer batching [30, 45] to eliminate cache line movement during high-contention updates. For SMOs, Opal uses a variant of the MCS-style queue lock [9, 22] for less contended exclusive accesses.

This section describes Opal's design. We begin with an overview of the lock structures and operation protocols (§3.1), then detail each protocol (§3.2), establish correctness (§3.3), and show how to apply Opal to index structures (§3.4).

3.1 Opal Overview

Figure 2 provides the structure of the lock and queue node, along with an example of concurrent operations.

Opal lock structure. Opal uses a compact 8-byte lock word split into three fields: (1) lock *state*: *G_UNLOCKED* when free, *G_LOCKED* when held exclusively, or *G_LOCKED_OPT* when held exclusively with opportunistic read bit set, *i.e.*, the lock holder allows readers to proceed; (2) *tail*: the MCS tail pointing to the last waiting thread that joined the queue using an atomic swap operation; (3) *version*: modification counter for reader validation. Writers increment the version number before releasing the lock.

Opal node structure. Opal uses the node structure for exclusive accesses and consists of four fields: (1) *wait*: flag indicating whether a waiter thread is waiting for notification from the lock-holder thread. (2) *done*: indicates if the lock holder has executed the waiter's critical section. (3) *next*: points to the next node in the queue. (4) *fn_ptr*: stores the function pointer to the critical section that needs to be executed; it is `NULL` for traditional queuing.

Unlike prior works [48], Opal only maintains a single queue node per thread. This design is possible since as compared to the original MCS lock algorithm, the head of the queue in Opal spins on the lock *state* rather than on the *wait* field in its queue node. Consequently, Opal's algorithm eliminates the need to pass the queue node information in the *release* function. thereby allowing arbitrary nesting with a single queue node.

Opal workflow. Figure 2 provides a running example of Opal with four writer threads W1-W4 and three reader threads R1-R3. Writer threads execute UPDATE SQL queries that result in in-place updates of the keys, whereas reader threads execute SCAN SQL queries that result in reading the relevant keys. (a) Initially, the lock is unlocked with version 100 and a `NULL` tail pointer, indicating an empty queue. (b) Thread W1 joins by initializing its queue node with a function pointer and atomically swapping the tail. As the new head, W1 acquires the lock in *G_LOCKED_OPT* state, enabling opportunistic reads. (c) Meanwhile, threads W2, W3, and W4 enqueue themselves; W2 and W3 update their function pointer field, whereas W4 uses the traditional mechanism (no function pointer). (d) While W2,W3 and W4 spin-wait, other reader threads can successfully execute their critical sections because the opportunistic read bit is set. (e) W1, being the lock-holder thread, starts the combining/batching process. W1 first disables opportunistic reads and then executes W2's critical section via its function pointer. (f) W1 notifies W2 of its completion and (g) repeats the process for W3. (h) W1 stops batching on finding W4 not using the function pointer. W1 notifies W4 that it is the new head of the queue and then continues with its own critical section execution. (i) W1 increments the version to 101 and releases the lock. (j) W4 acquires the lock and closes the queue by setting the *tail* to `NULL` (as it is now the only node). (k) W4 executes its critical section. (l) Finally, W4 increments the version to 102 and releases the lock.

Opal invariants. Opal maintains the following invariants: (1) The lock holder (or combiner thread executing on behalf of waiters) is always at the head of the queue. (2) The thread at the head of the queue spins on the global lock state rather than its own queue node. (3) Opportunistic reads are disabled before executing any critical section. (4) The combiner thread batches waiters that are using function pointers and does not batch waiters that joined the queue using the traditional mechanism. (5) The combiner thread batches only `WAITERS_TO_BATCH` critical sections, preventing starvation.

3.2 Opal Algorithm

Following the approach of TCLock [18] and OptiQL [48], Opal augments the MCS lock with a version counter to inform readers whether another writer thread executed their critical section

between the time their read critical section started and ended. Listing 1 presents the pseudocode of Opal for acquiring the read and write lock without function pointers.

3.2.1 Optimistic Read Lock

Read-lock: Thread t_s checks whether the lock is free (line 11). The lock is free if it is in either of two states: (1) `state` is *G_UNLOCKED*—no writer thread holds the lock; or (2) `state` is *G_LOCKED_OPT*, *i.e.*, a writer thread (t_{ex}) holds the lock but has enabled opportunistic reads, indicating t_{ex} is not executing its critical section. On success, the t_s stores the current lock version (line 12). Otherwise, t_s retries.

Read-unlock: After accessing the shared data, t_s checks the following two conditions (line 15): (1) whether the lock is held by writer thread and (2) whether the version matches the version it stored in the acquire phase. If either check fails, t_s retries.

3.2.2 Write Lock with Traditional Queuing

Write-lock: Thread t_{ex} attempts to acquire the TAS lock on the fast path via compare-and-swap (Line 19). On success, t_{ex} executes its critical section. Otherwise, it enters the slowpath.

On the slowpath, t_{ex} initializes its qnode (Line 24) and sets *fn_ptr* to NULL (Line 25) to indicate traditional queuing without function pointer based combining. t_{ex} then proceeds through three phases:
Phase 1A: Busy-waiting. Thread t_{ex} enqueues itself by atomically swapping the lock's `tail` pointer with its qnode (Line 27). The swap's return value indicates whether t_{ex} is at the head of the queue (Line 28). If a predecessor exists, t_{ex} links itself as a waiter (Line 29) and spins until notified by the lock holder (Lines 30–31).
Phase 1B: Acquire global TAS lock. Thread t_{ex} spins until the lock becomes free, then it atomically sets the lock `state` to *G_LOCKED_OPT* (Lines 33– 38). Setting this state permits concurrent readers to execute opportunistically.
Phase 1C: Notify next waiter or close queue. Thread t_{ex} checks if it is the only node in the queue via atomic compare-and-swap (Lines 40–42). If successful, it disables opportunistic reads (Line 41) and executes its critical section (Line 42). Otherwise, a successor has joined, and t_{ex} waits for its address to become visible (Lines 44– 45). Finally, it notifies the next waiter that it is at the head of the queue (Line 46), disables opportunistic reads (Line 47) and continues with its critical section execution.

Write-unlock: Thread t_{ex} releases the lock by incrementing the version counter (Line 50) and resetting the lock state to *G_UNLOCKED* (Line 51).

3.2.3 Write Lock Acquisition with Function Pointer

Listing 2 presents the pseudocode of Opal's *execute_op* function that is used by the index structure's update operation.

Thread t_{fp} starts by setting up its *qnode* with the function pointer of the index structure operation to execute (Line 11) and the input struct to the function (Line 12). t_{fp} attempts to acquire the lock on the fast path using a compare-and-swap operation (line 15). If successful, t_{fp} executes its own critical section and releases the lock (Line 16). Otherwise, t_{fp} continues with its slowpath execution. The slow path starts with t_{fp} initializing its qnode (Line 20) and then continues with the following four phases:
Phase 2A: Busy-waiting. Thread t_{fp} adds itself to the waiting queue by atomically swapping the lock's tail pointer with its qnode's address (line 23). t_{fp} checks if it is at the head of the queue by

```
1  PRCSD    = 0 # Waiter's request is processed by the combiner
2  UNPRCSD  = 1 # Waiter's request is not processed until now
3  WAITERS_TO_BATCH = 16384 # Batch count
4
5  class queue_node_struct:
6    wait = False, request = UNPRCSD , next = None, skt_id = 0,
7    fn_ptr = None, tree_input_ptr = None, tree_result = 0
8
9  def execute_fn(lock, fn_ptr, tree_input_ptr) -> int:
10   qnode = thread_local_storage.node
11   qnode.fn_ptr = fn_ptr # Function pointer to tree operation
12   qnode.tree_input_ptr = tree_input_ptr # Pointer to input struct
13
14   # Fastpath: Try to acquire the TAS lock
15   if CAS(&lock.state, G_UNLOCK, G_LOCKED) == G_UNLOCK:
16     execute_cs_and_release_lock(lock, qnode)
17     return qnode.tree_result
18
19   # Slowpath: Join the queue and wait
20   init_qnode(qnode, wait = True, request = UNPRCSD , #Initialize qnode
21     next = None, skt_id = numa_id(), tree_result = 0)
22   # Phase 2A: Busy waiting: Join the queue and wait until notified
23   qprev = SWAP(&lock.tail, &qnode) # Atomically add node to tail
24   if qprev is not None:            # Waiters are already present in the queue
25     qprev.next = qnode             # Link qprev with qnode to form a queue
26     while qnode.wait is True:
27       continue                     # Wait for the combiner to halt my spinning
28     if qnode.request == PRCSD :    # Waiter request has been processed
29       return qnode.tree_result     # Return the result
30   # Phase 2B: Head of the queue: Acquire the global TAS lock
31   while True: # Wait for the state to be unlocked
32     while lock.state != G_UNLOCK:
33       continue
34     # Acquire lock with opportunistic reads enabled
35     if CAS(&lock.state, G_UNLOCK, G_LOCKED_OPT) == G_UNLOCK:
36       break  # Got the TAS lock
37   # Phase 2C: Combining-role decision: Whether to combine
38   if CAS(&lock.tail, qnode, None) == qnode:
39     disable_opportunistic_reads(lock)
40     execute_cs_and_release_lock(lock, qnode)
41     return qnode.tree_result # If only one in the queue, return
42
43   qnext = qnode.next # Someone joined the queue
44   while qnext is None:
45     qnext = qnode.next
46   # Check if there are at least two waiters and next waiter is using FP
47   if check_batching_condition(qnext):
48     qnext.wait = False # next waiter is combiner
49     disable_opportunistic_reads(lock)
50     execute_cs_and_release_lock(lock, qnode)
51     return qnode.tree_result
52   # Phase 2D: Combining: Batch requests
53   counter = 0
54   while True:  # Combiner loop
55     qcurr = qnext # Get the next waiter to batch
56     qnext = qcurr.next # Get the next node
57     counter += 1
58     disable_opportunistic_reads(lock)
59     # Execute critical section using function pointer
60     execute_cs_via_fn_ptr(qcurr)
61     # Waiter's critical section execution finished
62     enable_opportunistic_reads(lock)
63     qnode.request = PRCSD  # Mark previous waiter's request as completed
64     qnode.wait = False # Wakeup previous waiter
65     if check_batching_condition(qnext)
66       or counter >= WAITERS_TO_BATCH: # Check comb. cond.
67       break
68
69   # Combiner phase is over, now combiner runs its own CS
70   qnext.wait = False # Next waiter is now at the head of the queue
71   disable_opportunistic_reads(lock)
72   my_qnode = thread_local_storage.qnode
73   execute_cs_and_release_lock(lock, my_qnode)
74   return my_qnode.tree_result
75
76 def enable_opportunistic_reads(lock):
77   lock.state = G_LOCKED_OPT
78
79 def disable_opportunistic_reads(lock):
80   lock.state = G_LOCKED
81
82 def execute_cs_and_release_lock(lock, qnode):
83   qnode.tree_result = qnode.fn_ptr(qnode.tree_input_ptr)
84   lock.version += 1 # Increment Version
85   lock.state = G_UNLOCK # Release lock
86
87 def check_batching_condition(qnext) -> bool:
88   return qnext is None or qnext.fn_ptr is None or qnext.next is None
```

Listing 2: Pseudocode of Opal to execute using the function pointer.

checking the return value of the swap operation (line 24). If another thread address exists, t_{fp} joins the queue as a waiter (line 25), and waits for notification from the combiner thread (lines 26–27). After receiving the notification, t_{fp} checks if combiner executed its critical section (line 28). If that is the case, t_{fp} returns (line 29) the value stored in the `qnode` and continues with its non-critical section execution. Otherwise, t_{fp} has reached the head of the queue and begins the subsequent phases.

Phase 2B: Global lock acquisition. Thread t_{fp} starts by spinning on the state byte of the lock until it is set to *G_UNLOCKED* (lines 31–36). t_{fp} then tries to acquire the lock by setting the state byte to *G_LOCKED_OPT* atomically (line 35). Setting this state allows the readers to continue executing their read phase. If successful, t_{fp} continues with the next phase. Otherwise, t_{fp} spins waiting for the lock to be unlocked. This phase is required as the lock can be acquired by another thread on the fast path after being released by the combiner thread.

Phase 2C: Combining-role decision. Thread t_{fp} checks whether there are at least two threads in the queue [18]. t_{fp} starts by checking if it is the only one in the queue (lines 38–41). If that is the case, t_{fp} disables opportunistic reads (line 39) and executes its critical section (Line 40). t_{fp} executes the critical section using the provided function pointer in the queue node (Line 60) and saves the result in the queue node (Line 83). Finally it releases the lock and returns the result saved in the queue node (Line 41).

Otherwise, thread t_{fp} waits for the next node to join the queue (lines 44–45). t_{fp} then checks the condition for batching (lines 47–51). This includes existence of at least two nodes in the queue and the next node using the function-pointer-based approach. If these criteria are met, t_{fp} continues with the next phase. Otherwise, t_{fp} notifies the next node that it is now at the head of the queue (line 48). t_{fp} then disables opportunistic reads (line 49) and continues with its critical section execution and releases the lock (Line 50).

Phase 4: Combining. Thread t_{fp} iterates over the queue to execute each waiter's critical section (lines 54–67). Within each iteration, t_{fp} selects the next waiter to execute (line 55). t_{fp} then disables opportunistic reads (line 58) to prevent readers from continuing their execution as the current thread can now modify the shared cache line. t_{fp} executes the waiter's critical section (line 60) using the provided function pointer in the queue node and saves the result in the queue node.

Once executed, thread t_{fp} enables the opportunistic reads (line 62) to notify the readers that it is safe for them to execute. t_{fp} then notifies the current waiter that its critical section is executed (line 63), and proceeds to check the following batching conditions (lines 65–66): (1) To execute the next waiter, check if at least two threads are present in the queue. (2) Check if the next waiter wants to be combined using the function-pointer-based approach or not. (3) Check if enough waiters have been combined. If all of the above conditions are satisfied, t_{fp} executes the next iteration of the loop, otherwise, it ends the combining phase.

Thread t_{fp} then notifies the next waiter that it is at the head of the queue (line 70). t_{fp} disables opportunistic reads (line 71), and continues with its critical section execution (Line 73). Finally, it releases the lock and returns the output of the execution (Line 74).

Executing thread t_{fp}'s critical section at the end changes the execution order of the critical section as it is different than the arrival order. However, this behavior does not have any impact in terms of correctness on the transaction execution as the behavior will be equivalent to the transaction running with traditional locks when thread t_{fp} arrives at the end and joins the queue.

3.3 Proof Sketch of Correctness

Correct synchronization across different access states.

Writer-writer synchronization. Opal ensures mutual exclusion by relying on atomic operations. Specifically, the atomic swap operation to join the queue ensures that only one thread can be at the head of the queue at any given point in time (line 27 in Listing 1 and line 23 in Listing 2). After doing an atomic swap operation, only one thread will observe that the previous value is `NULL`, making it the head of the queue. Meanwhile, other threads will join the queue and wait for the notification from the lock-holder thread. The head of the queue also acquires the lock using TAS to ensure that no other thread has acquired the lock on the fast path. The combination of these two operations ensures mutual exclusion between threads acquiring write lock.

Reader-writer synchronization. Opal maintains correct synchronization between threads competing for read or write access by ensuring the correct usage of the opportunistic read bit and the version counter. Readers are only allowed to finish their execution if the lock is free, or if the lock is acquired but the opportunistic read bit is set and the version remains the same (lines 11 and 15). These three conditions ensure that either there is no thread that has acquired the write lock, or there is such a thread, but it has explicitly allowed the readers to proceed by setting the opportunistic read bit. The version counter check (line 15) ensures that a reader blocked for some time does not miss any write that happened after acquiring the read lock but before releasing it.

Opal guarantees that the thread acquiring the write lock will reset the opportunistic read bit before executing its own critical section (lines 39, 49 and 71). The combiner loop also disables the opportunistic read bit before executing a waiter's critical section (line 58) and re-enables it after execution (line 62). The version counter is also incremented with each release of the lock (line 84), ensuring correct synchronization between threads acquiring the read or write lock.

Correct mechanism used for critical section execution.

Opal ensures that the correct mechanism is used for critical section execution as requested by the waiter thread. Using the incorrect mechanism can lead to segmentation faults. Specifically, if a waiter thread t_{ex} performs an insert operation, it will use the traditional lock mechanism. If the lock-holder thread t_{fp} is doing an update operation, it will batch waiters and execute their critical section using the function pointer. Since the function pointer in the queue node of t_{ex} will be `NULL`, this will result in segmentation fault. Opal ensures correctness by checking the `fn_ptr` value in the queue node (line 65) and only continues batching if it is not `NULL`. This ensures that the lock-holder thread t_{fp} does not execute a waiter thread t_{ex} running with traditional lock mechanism. The reverse, where the lock-holder thread t_{ex} executes with the traditional lock mechanism and the next waiter thread t_{fp} executes with a function pointer, does not create any issue as the traditional lock mechanism just notifies the next waiter thread that it is at the head of the queue (Line 46).

```
1  # Default implementation
2  def btree_update(root, key, new_val):
3  restart:
4    parent_node = root
5    v = INVALID
6    if !read_lock(parent_node.lock, v):
7      goto restart
8
9    # Tree traversal
10   while !is_leaf(node):
11     child = find_child(parent_node, key)
12     nv = INVALID
13     if is_leaf(child):
14       write_lock(child.lock)
15       if !read_unlock(parent_node.lock, v):
16           write_unlock(child.lock)
17           goto restart
18       update_leaf(child, key, new_val)
19       write_unlock(child.lock)
20       return
21     else:
22       if !read_lock(child.lock, nv):
23         goto restart
24       if !read_unlock(parent_node.lock, v):
25         goto restart
26       parent_node = child
27       v = nv
28
29
30
```

```
31 # Conversion to function pointer
32 def btree_update_fn(input_ptr) -> bool:
33   if !read_unlock(input_ptr.parent_node.lock, input_ptr.v):
34     return false
35   update_leaf(input_ptr.child, input_ptr.key, input_ptr.new_val)
36   return true
37
38 def btree_update(root, key, new_val):
39 restart:
40   parent_node = root
41   v = INVALID
42   if !read_lock(parent_node.lock, v):
43     goto restart
44
45   while !is_leaf(node):
46     child = find_child(parent_node, key)
47     nv = INVALID
48     if is_leaf(child):
49       input_ptr = create_struct(parent_node, v, child, key, new_val)
50       if !execute_fn(child.lock, btree_update_fn, input_ptr):
51         goto restart
52     else:
53       if !read_lock(child.lock, nv):
54         goto restart
55       if !read_unlock(parent_node.lock, v):
56         goto restart
57       parent_node = child
58       v = nv
```

Listing 3: B+ Tree update operation converted to use function pointers.

Correct synchronization with multiple lock usage. Multiple locks can be acquired either for different lookup/update operations or in a nested order for structural modification operations. Since Opal's algorithm works exclusively on single lock instance and does not rely on any global state, operations that require shared or exclusive access to different locks work correctly, provided the correct lock instance is passed to the *acquire* and *release* functions. Moreover, Opal allows SMOs to have arbitrarily deep nesting of locks while maintaining only a single queue node per thread. This is possible since unlike MCS algorithm (§3.1), Opal does not pass the queue node to the *release* function, allowing to reuse the queue node for nested locks.

3.4 Using Function Pointer Execution for Index Operations

B+ Tree Listing 3 presents the B+ Tree update function and the changes required to convert it into a function-pointer-based approach. The default implementation consists of a tree traversal using lock coupling until it reaches the leaf node (Lines 10– 27). Lock coupling involves acquiring the read lock on the child node before releasing the read lock on the parent node (Lines 22– 25). After reaching the leaf node, the thread acquires a write lock on the leaf node (Line 14) and releases the read lock on the parent node (Line 15). The thread updates the key with the new value (Line 18) and releases the write lock (Line 19).

For B+ Tree update function, the critical section consists of releasing the read lock on the parent (Line 33) and updating the key with the new value (Line 35). The critical section is moved into a separate function, *btree_update_fn*, and the pointer to this function along with input arguments, *input_ptr* is passed to the *execute_fn* function in Listing 2. The combiner thread will execute this function and store the result within the qnode (Line 83 in Listing 2). The waiter thread will return the stored result in qnode (Line 29 in Listing 2). The traversal is restarted if the result is *false* indicating that the parent node has changed. Similar steps are followed to modify the ART update function to use function pointer.

3.5 Opal under Oversubscription

Batching-based critical section execution provides better performance under oversubscription compared to traditional lock designs. Traditional lock designs like OptiQL transfer lock ownership to the next waiter after executing a single critical section. Under oversubscription, where the number of threads exceeds the number of available cores, the new lock holder may be blocked immediately after acquiring the lock. This extends the duration of sequential execution, degrading overall application performance.

With Opal, a single combiner thread batches multiple critical sections before transferring the lock ownership to another thread. Since the execution of a waiter's critical section is independent of whether the waiter thread gets scheduled, the combiner thread can batch critical sections for the whole time slice. This results in faster execution of the sequential code resulting in better performance.

Ideally, the combiner thread should run until it hits its batching condition but in reality, with oversubscription, the CPU scheduler schedules other threads on the same CPU as the combiner thread. This results in smaller percentage of CPU time allocated for the combiner thread as the oversubscription increases. A system design where other threads on the same CPU as the combiner thread are migrated to a different CPU can solve this issue as it allows the combiner thread access to the full time slice. We leave this design as future work.

4 IMPLEMENTATION

Lock structure. Opal implements an 8-byte (64-bit) lock where the status is 2 bits, the tail is 10 bits, and the version is 52 bits. This allows Opal to support up to 1024 threads. A higher thread count can be supported by reducing the version number.

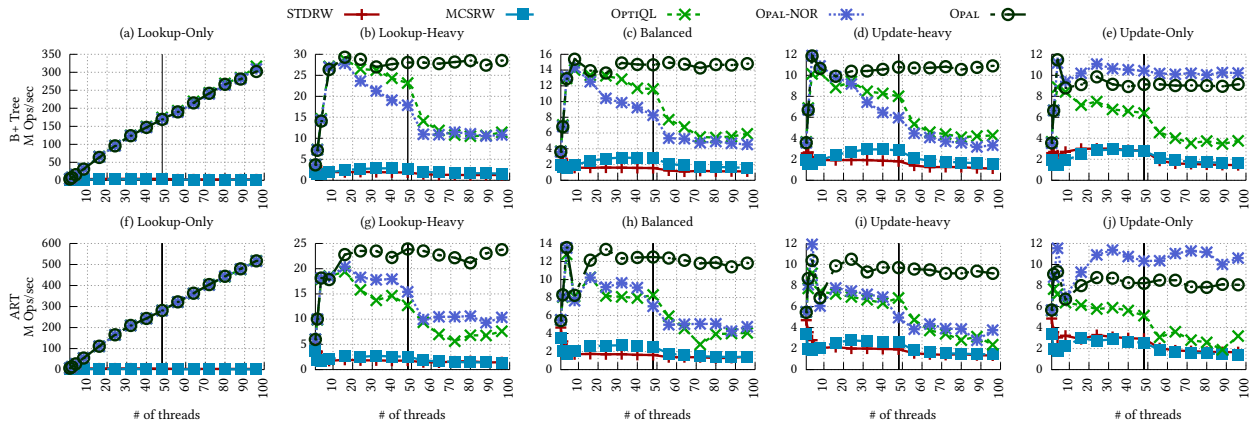

Figure 3: Throughput of B+ Tree (top) and ART (bottom) under a skewed workload (self-similar distribution with a skew factor of 0.1). The workload uses a mix of lookup and update operations.

Queue node. Opal supports arbitrary nesting in the SMO operation by requiring only a single queue node per thread. OptiQL, on the other hand, requires four queue nodes per thread. Opal does not require passing the queue node to the unlock function because the next waiter at the head of the queue spins on the lock word instead of its own queue node. This allows the release of the lock to be a simple write operation on the lock word without requiring the queue node. Similar to OptiQL, we use a mapping between the thread ID and queue node ID stored in the tail pointer and access other queue nodes using a global queue node array.

NUMA. Opal implements a NUMA-aware policy for batching where the queue nodes within the same socket are batched before passing the lock ownership to some other waiter in another socket. The NUMA-aware policy minimizes the cache line bouncing among NUMA nodes. Similar to CNA [11] and TCLock [18], Opal adopts a dynamic queue-splitting approach where a thread-local waiting queue is maintained for all the waiter threads from different sockets. At the end of the combining phase, the thread-local waiting queue is merged into the primary queue and the lock is passed to the head of this merged queue.

5 EVALUATION

We evaluate Opal by answering the following questions:

- **Q1.** How does Opal perform with different index structures under lookup-update (§5.1) and insert (§5.2) operations?
- **Q2.** How does Opal perform in oversubscribed scenarios (§5.4)?
- **Q3.** How is Opal affected by batch size and skewness (§5.5)?

Evaluation setup. We perform experiments on a dual-socket server equipped with two 48-core Intel Sapphire Rapids CPU with hyper-threading disabled. We extend OptiQL's PiBench [27] setup with our lock design to stress test the B+ Tree and ART. Following prior work [48], we use *jemalloc* [15] to reduce memory allocation overhead and evaluate with 256-byte nodes, 8-byte key/value pairs and 100 million records. When the number of threads is lower than the total CPU count, we use `taskset` to restrict them to a specific subset of CPUs selected from the minimum number of NUMA nodes necessary to provide exactly one CPU per thread. This reduces variance and makes results easier to interpret, as threads use a predictable number of NUMA nodes. For oversubscription, the whole system is available for spawned threads. We test up to 4x oversubscription.

Table 1: Experiment Setup

Experiment	YCSB	Lookups	Updates	Inserts
Lookup-only	C	100%	-	-
Lookup-heavy	B	95%	5%	-
Lookup-Update-balanced	A	50%	50%	-
Update-heavy	-	20%	80%	-
Update-only	F	-	100%	-
Insert-only	LOAD	-	-	100%
Insert-Lookup-heavy	D	95%	-	5%
Insert-Lookup-balanced	-	50%	-	50%
Insert-Update-heavy	-	-	80%	20%
Insert-Update-balanced	-	-	50%	50%

Evaluated locks. We test the following locks:

- STDRW (Pessimistic): The reader-writer lock (`pthread_rwlock_t`) in pthread (`std::shared_mutex`).
- MCSRW (Pessimistic): The reader-writer MCS lock [39].
- OptiQL (Optimistic): Optimistic queue-based lock [48].
- Opal (Optimistic): Opal proposed in §3 with opportunistic reads available.
- Opal-NOR (Optimistic): Opal proposed in §3 with opportunistic reads disabled.

Workloads. Table 1 shows the workload we use for our index experiments, which are similar to prior work [8, 48]. To test under contention, we run the B+ Tree and ART workload under a self-similar distribution with a skew factor of 0.1 (*i.e.*, 90% of the accesses target 10% of the keys). Similar to prior work [48], we make the key space dense to further stress locks (*i.e.*, the first 256 keys accept 16% of the total accesses).

5.1 Workload with Lookup-Update Operations

Update-only workload. Figure 3 (e) shows Opal's throughput benefit with an update-only workload for B+ Tree. We observe that, within a socket, Opal outperforms traditional pessimistic (STDRW and MCSRW) and optimistic lock (OptiQL) by up to 3.23×, 3.72× and 1.42×, respectively. This is because Opal reduces shared data movement compared to traditional locks. Specifically, Opal

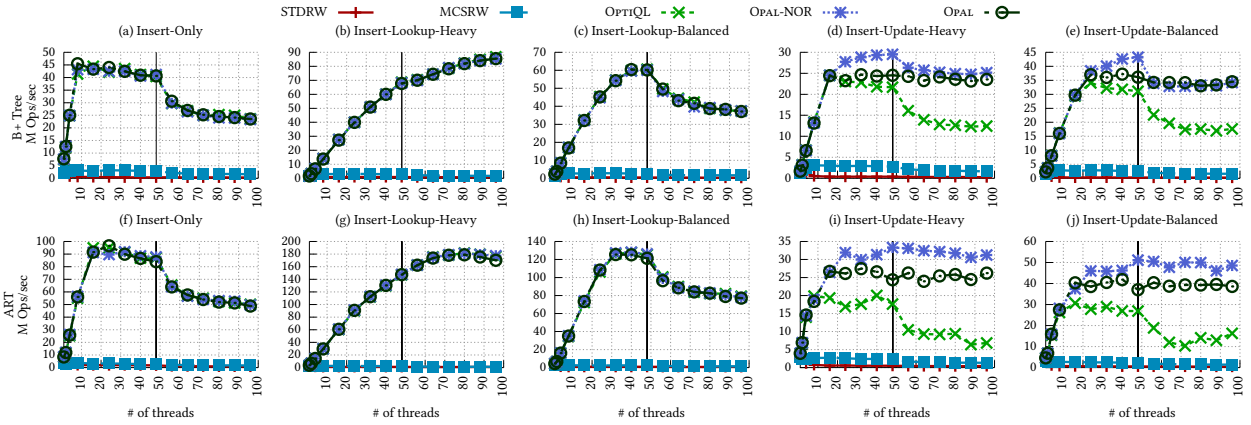

Figure 4: Throughput of different index structures with a mix of insert and lookup/update operations.

batches on average 16K critical sections before transferring the lock ownership to another thread. Batching critical sections results in more L1 cache hits for shared leaf nodes, thereby improving performance within a socket. Moreover, Opal also has minimal overhead in its combiner loop (Phase 4 in Listing 2). On average, the combiner loop is around 470 cycles long whereas the critical section is around 241 cycles long. The remaining time is spent in notifying the previous waiter about its completion, figuring out the next waiter to execute and enabling/disabling the opportunistic bit.

Across sockets, Opal outperforms STDRW, MCSRW and OptiQL by 6.34×, 6.31× and 2.43× respectively. OptiQL without any NUMA-aware policy incurs frequent cache line bouncing among NUMA nodes. Since the inter-socket cache line latency is 3× higher than the intra-socket latency on our test machine, this results in lower performance. Opal's policy of batching critical sections within a socket before passing ownership to a thread across sockets results in minimal inter-socket cache line movement, resulting in maintaining similar performance as within a socket.

Opal-NOR outperforms Opal by 1.12× in the update-only workload as it removes the setting and unsetting of opportunistic read bit in the combiner loop. Opal incurs two write accesses to the contended cache line containing the lock word when it disables and enables opportunistic read bit before executing the waiter's critical section. By removing the opportunistic read overhead, Opal-NOR reduces the time spent in the combiner loop from 470 to 413 cycles, which improves performance.

Similar to B+ Tree, Figure 3 (j) shows that Opal outperforms OptiQL within a socket and across sockets by up to 1.6× and 2.53×, respectively, due to minimizing shared data movement. Similar to B+ Tree, Opal batches on average 16K critical sections where each critical section is around 451 cycles. Moreover, Opal-NOR outperforms Opal by 1.31× by removing the overhead of enabling/disabling the opportunistic read bit in the combiner loop.

Lookup-update workload. Figure 3 (b–d and g–i) show performance with different ratios of lookup and update operations. Opal outperforms OptiQL in all index structures with different access patterns. Specifically in B+ Tree, Opal improves performance compared to OptiQL by 2.49×, 2.51×, and 1.89× in lookup-heavy, balanced and update-heavy workloads, respectively. This is because Opal executes each batched write critical section faster, even though the total number of batched critical sections decreases as the proportion of lookup operation increases and update operation decreases. Faster execution of the write critical section allows the thread to execute its next operation, reducing the queuing effect. Faster execution also results in fewer retries for readers as the write lock is held for a shorter time. Specifically, for the balanced B+ Tree workload, Opal reduces the number of reader retries compared to OptiQL by 3.08× and 7.54× within and across sockets, respectively. Opal's NUMA-aware policy helps in further reducing the critical section execution time as it reduces shared data cache line bouncing across sockets, resulting in fewer reader retries. We observe a similar trend with the ART index and other combinations of lookup and update operations.

Opal outperforms Opal-NOR by 3.26× because opportunistic reads allow readers to proceed without retrying. Although opportunistic reads add 60 cycles to the combiner loop due to enabling and disabling the opportunistic bit in the contended lock cache line, this overhead is justified as it reduces reader retries by 107×. Without opportunistic reads, a reader must retry from the root when it encounters a write-locked node with the opportunistic bit disabled. These retries block the thread from proceeding to its next operation because the combiner thread holds the lock for an extended duration while executing batched critical sections—much longer than traditional MCS-style locks. Consequently, without opportunistic reads, readers suffer from a higher retry rate.

Opal outperforms the pessimistic locks STDRW and MCSRW by 13.09× and 9.25×, respectively, because these locks perform atomic increments/decrements while traversing the tree, resulting in high contention on the root node. Since STDRW is a blocking lock, it is more severely impacted than MCSRW as the waiting thread can be put to sleep and incur a costly wakeup latency.

Lookup-only workload. Figure 3 (a and f) show performance with lookup-only workload. All optimistic locks perform similarly, as the optimistic parts are similar across different locks. The performance with the pessimistic lock is worse compared to the optimistic locks because pessimistic locks perform atomic increments/decrements while doing tree traversal. Optimistic locks provide 150× improvement compared to pessimistic lock in this case.

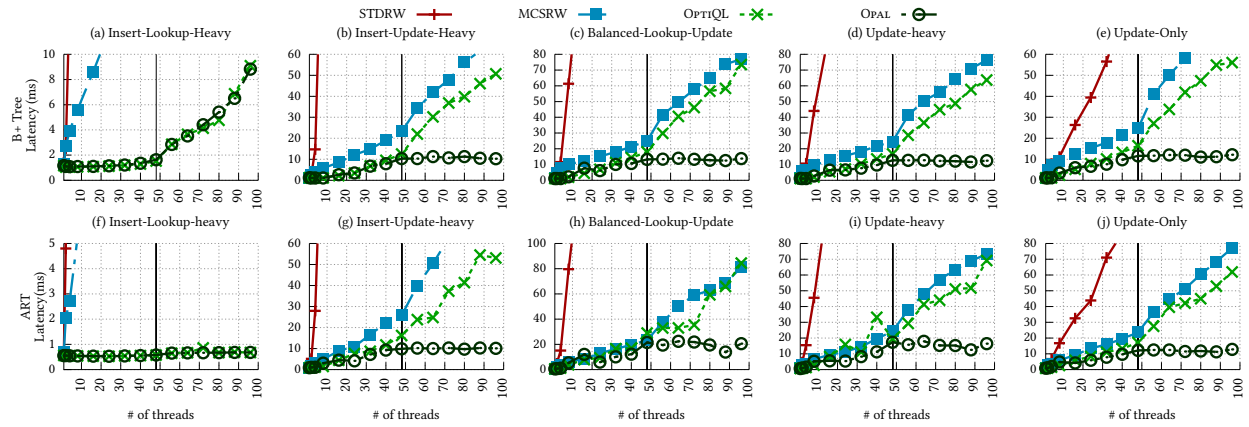

Figure 5: 99.9%ile latency of different index structure operations.

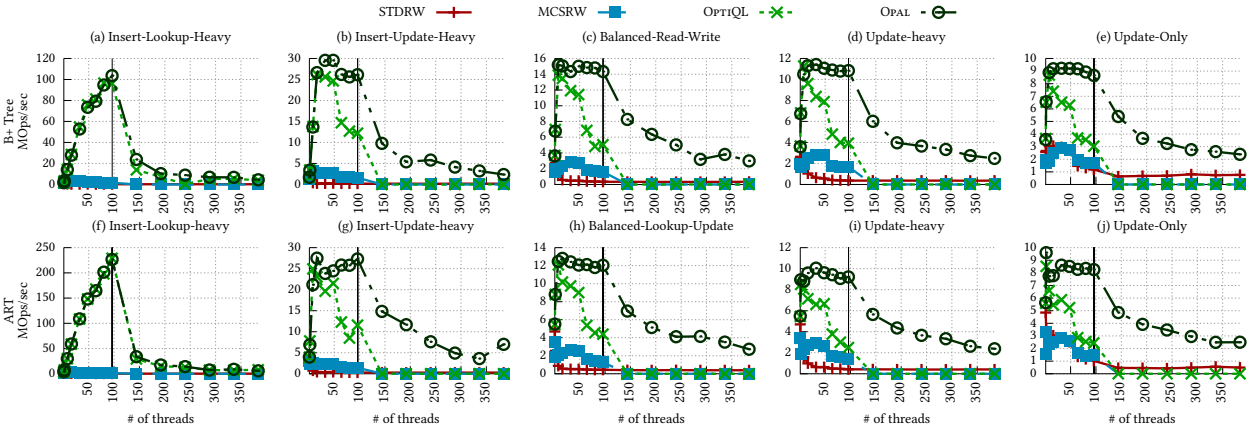

Figure 6: Throughput under oversubscription scenario where there are more threads than core. The horizontal line in each graph signifies the start of oversubscription. We test up to 4× oversubscription.

5.2 Insert Access Pattern

Figure 4 (a-c, f-h) shows throughput with insert-only and insert+read workloads. We observe that both OptiQL and Opal perform similarly in this case as both use an MCS-style lock for the insert workload and optimistic reads for the lookup workload. The pessimistic locks perform poorly compared to optimistic ones because of the usage of atomic operations during tree traversal.

Figure 4 (d-e, i-j) shows Opal's throughput benefit with insert+update workloads. Here, we observe Opal's benefit of batching on the update side compared to OptiQL. Within and across sockets, Opal outperforms OptiQL by 1.38× and 3.83×, respectively. This is because Opal batches around 15K critical sections for update operations. In a closed-loop setup, where a thread waits for the previous operation to complete before starting a new operation, batching accelerates update operations, allowing Opal to achieve higher throughput for both insert and update operations.

5.3 Tail Latency

Due to space constraints, Figure 5 shows the 99.9%ile latency for five workloads: Insert-Lookup-heavy, Insert-Update-heavy, Balanced-Lookup-Update, Update-heavy, and Update-only on B+ Tree and ART. Optimistic locks provide better latency than pessimistic locks because tree traversal requires only version checks instead of atomic operations at each level.

Opal reduces 99.9%ile latency compared to OptiQL by 29.4% within a socket and 80% across sockets. Although Opal batches multiple critical sections, it achieves lower tail latency because each critical section executes faster than in OptiQL. This speedup occurs as Opal localizes shared data within the combiner's cache. This allows for lower queuing delays in the system for the contended lock, resulting in lower tail latency. Since MCSRW is a spinning-based lock, it has a lower tail latency than the blocking-based STDRW. However, it has a 6.33× worse tail latency than Opal due to contention on atomic operations for reads and cache line bouncing for writes.

5.4 Oversubscription

Figure 6 shows the performance with up to 4× oversubscription. With oversubscription, there are multiple threads per core. The threads are not pinned and the scheduler is free to move them amongst all cores. However, we have observed that the OS scheduler balances the number of threads evenly across cores and migrates very few threads during the execution of the workload. This happens because the synchronization primitives use spinning as the waiting mechanism instead of going to sleep while waiting. From the OS scheduler perspective, it is as if the threads are doing useful work and it cannot differentiate whether the thread is waiting for a lock or executing a tree operation. Hence with 4× oversubscription, only four threads run on the same core for the whole duration of

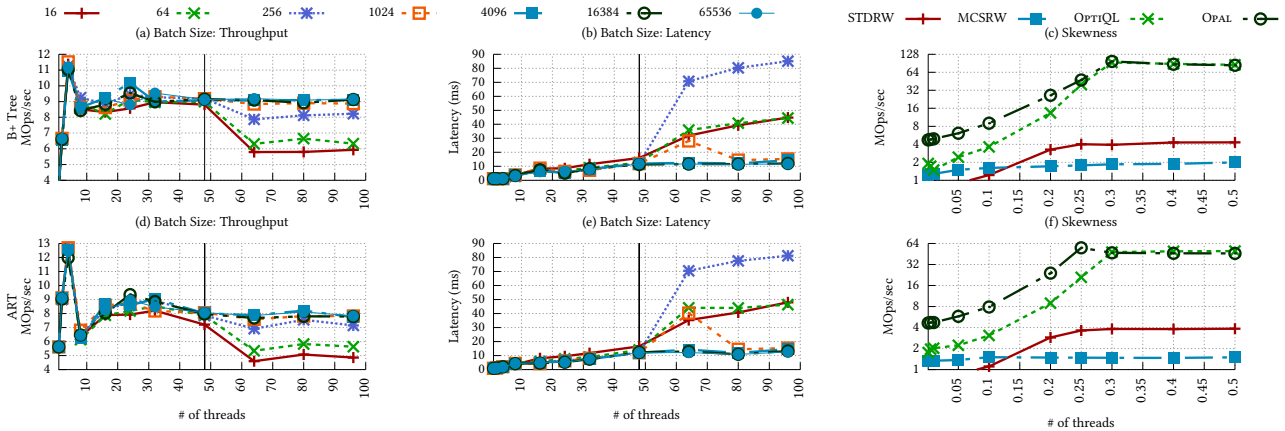

Figure 7: Opal sensitivity study.

the workload. Hence, each individual thread gets one fourth of the CPU time compared to the workload without oversubscription.

5.4.1 Lookup-Update Workload.

OptiQL Figure 6(c-e, h-j) shows that OptiQL's performance drops exponentially up to 10000× with 4× oversubscription. OptiQL's MCS-based lock design executes one critical section per core. With 4× oversubscription, multiple threads are running on the same CPU as the lock-holder thread, resulting in the lock-holder thread running one-fourth of the time and being blocked three-fourth of the time. Moreover, the next lock owner thread might also be blocked by other threads, resulting in further overhead before the next critical section can be executed. As the oversubscription increases, the performance drops exponentially as the individual lock-holder thread gets less time to run on a single CPU. Similar to OptiQL, MCSRW waits using a spin-loop, resulting in similar exponential performance degradation.

STDRW Figure 6(c-e, h-j) shows that STDRW performs similarly with and without oversubscription. Being a blocking-based design, where the waiting thread sleeps instead of spinning, the CPU is free for the lock-holder thread to execute its critical section. However, being a pessimistic lock with most of the contention happening at the root node during traversal, STDRW still performs 8.82× worse than spinning-based Opal at 4× oversubscription.

Opal Figure 6(c-e, h-j) shows that Opal performance degrades linearly as the oversubscription increases. Opal's batching-based approach enables a single lock holder to execute multiple critical sections before transferring the lock ownership to another thread. This allows the lock owner to execute critical sections as fast as possible before being descheduled by the OS. Compared to no oversubscription, the CPU time decreases linearly as the oversubscription increases. Since Opal's performance is capped by the amount of time the lock holder gets to execute other critical sections, the performance also degrades linearly instead of exponentially in OptiQL.

Opal also reduces the number of reader retries for balanced workloads as compared to OptiQL, as the critical section is executed faster. Specifically, for this workload, Opal reduces the reader retries compared to OptiQL by 2019.1× with 4× oversubscription.

5.4.2 Lookup-Update + Insert Workload. Figure 6(a-b, f-g) shows that the performance of OptiQL and MCSRW drops exponentially with Insert workloads as well. Opal performance also degrades with Insert+Update workload as Opal uses a traditional MCS-style lock design for insert operations. However, Opal usage of batching for update operations provides 1425× performance improvement compared to OptiQL with 4× oversubscription. With Insert+Lookup workload, the performance collapses for both Opal and OptiQL as it uses MCS-style locks for inserts and optimistic reads for lookups. STDRW maintains similar performance with oversubscription, but having a pessimistic lock is detrimental, as STDRW performs 12.96× worse than Opal at 4× oversubscription.

To prevent this linear performance degradation with Opal, the combiner thread can be isolated on a single CPU by moving the waiter thread to another CPU. This allows the combiner thread to run uninterrupted while it is batching waiters' critical sections. We leave exploration of this approach as future work.

5.5 Sensitivity study

Batch Size: Figure 7 (a,d) shows the performance with an Update-only workload with different batch sizes. We observe that batch size has a smaller impact within a socket compared to across sockets. Smaller batch size results in frequent movement of shared data among cores. Since the inter-socket latency is 3× higher compared to the intra-socket latency, Opal has to batch more waiters' critical sections to offset the overhead of moving shared data cache lines across sockets. Opal achieves 1.54× higher throughput with a batch size of 16K compared to a batch size of 16.

Figure 7 (b,e) shows the 99.9%ile latency of an update operation with different batch sizes. Similar to throughput results, latency follows a similar trend where the batch size has a smaller impact on latency within a socket compared to across sockets. Opal achieves 3.76× latency reduction with a batch size of 16K compared to a batch size of 16 across sockets.

Skewness Figure 7 (c,f) shows Opal performance with an Update-only workload on 96 cores with different skewness. We use a self-similar distribution where skew of 0.1 means 10% of the data is accessed 90% of the time. A skewness of 0.5 corresponds to a uniform distribution.

Opal improvement compared to OptiQL decreases as the value of skew increases. This happens because the accesses are more spread out in the tree. Opal batching decreases from 15384 at skew

of 0.1 to 1.77 at skew of 0.3. With less batching, the improvement of Opal is lower, resulting in similar performance as OptiQL.

Figure 7 (c,f) also shows that even with a uniform workload where contention is spread out evenly in the tree, pessimistic locks still provide 19× lower throughput compared to optimistic locks. This happens because most of the contention in pessimistic locks happens on the root node when the thread starts its traversal from root to the leaf node. The atomic counter to track the number of threads acquiring read lock in pessimistic locks becomes a point of contention. Meanwhile, for optimistic locks, the traversal uses version checks, and without any SMO operations, the version remains stable, resulting in faster traversal.

6 DISCUSSION

Nested locking with function pointer. Certain update operations on a key A may depend on another key B within the same index structure. In such cases, the procedure requires nested locking, where a read or write lock on key B is obtained after acquiring a write lock on key A. To prevent deadlocks when both keys share the same lock, the lock user elides the nested lock acquisition if the lock addresses of A and B match.

Opal efficiently handles scenarios involving nested read locks because each lock instance operates independently. Given that Opal's locking mechanism acts exclusively on individual lock instances during both acquisition and release, it supports arbitrary nesting when all nested locks are read locks.

Opal also accommodates nested write locks. However, in such configurations, the function-pointer-based approach can be applied to only one lock within the nesting hierarchy. The lock user can freely decide whether to use this approach for the outer or inner lock, while all other locking instances revert to the conventional MCS-based design. Moreover, since Opal's algorithm for write lock acquisition does not require passing a queue node between lock and unlock operations, it inherently permits arbitrary nesting of write locks, as demonstrated in SMO operations.

The function-pointer-based approach is best suited for the most contended lock in the nested hierarchy as it allows for better performance due to the batching of multiple critical sections. Identifying this lock requires measuring contention at each hierarchical level, which we leave as future work.

Additional use cases. Batching-based designs could also be applied to other contended locks in database systems. For example, in PostgreSQL, the buffer manager is a highly concurrent subsystem [29] that employs several synchronization mechanisms, some of which may benefit from batching [6]. In particular, the *buffer_content* [42] and *buffer_strategy_lock* [41] locks have been reported to become highly contended in some configurations [3]. Replacing these locks with Opal's batching mechanism is a promising direction to explore for reducing contention on exclusive accesses.

Strife [43] partitions transactions into clusters that can be executed in parallel, along with a residual set that requires exclusive execution. In such settings, Opal could be used to implement the exclusive phase (e.g., to serialize access to shared indexing structures), which may improve performance when this phase becomes a bottleneck as the thread count increases.

7 RELATED WORK

Our work is closely related to different types of synchronization mechanisms and concurrent index structure designs.

Synchronization mechanisms. Building on the original MCS lock [38], many queue-based lock designs [9, 10, 12, 35, 47] have been proposed that can reduce the lock handover cost and shared data movement. Hierarchical [7, 13, 34, 44] and NUMA-aware locks [11, 22] have been proposed to further reduce cache line bouncing in non-uniform memory access (NUMA) machines. However, these traditional lock designs do not have optimistic locking and suffer from synchronization overhead during tree traversal in index structures. Recent work [5, 51] has shown how to combine pessimistic writers with optimistic readers. However, these designs remain subject to the same performance limitations as OptiQL [48], as they still incur shared data movement.

Multiple batching-based lock designs have also been proposed that work by executing the critical section on a single core [17, 18, 30, 31, 45]. We draw inspiration from these designs for the update operation and enable correct hybrid locking by adding an MCS-style lock design for inserts and opportunistic reads for lookup operations.

Concurrent indexes. Several approaches have been proposed for concurrent indexes ranging from traditional lock coupling [4, 26, 48] to lock-free designs [2] aiming at providing performance scalability. Lock-free indexes have been shown to require more effort to implement and debug compared to lock-based approaches [53]. Persistent memory (PM) indexes [32, 33, 40, 50] have also been proposed that use optimistic locking for better scalability. Recently, Tabular [54] has shown how to build concurrent indexes using an "objects-on-DB" design. A hybrid batching-based approach can also be applied to this design to improve index structure performance.

8 CONCLUSION

Existing optimistic queue-based lock designs suffer from performance degradation due to shared data movement under high contention in modern concurrent indexes. Batching-based designs have been shown to perform better in this scenario. However, these designs suffer from performance degradation due to the usage of pessimistic reads for lookups and unnecessary overhead at low contention. In this paper, we introduce Opal, a hybrid operation-aware lock design for modern in-memory indexes. We use optimistic version-based locking for lookups, function-pointer-based batching for updates and an MCS queue-based lock design for SMOs. Our evaluation shows that Opal is able to outperform existing optimistic and pessimistic lock designs under a variety of scenarios, including thread oversubscription.

ACKNOWLEDGMENTS

We thank Musa Unal for his help in creating the figures and members of the RS3LAB for their comments on the initial draft. We also thank the anonymous reviewers for their helpful feedback. This work is supported by the Swiss National Science Foundation project grant #212884 and #10008650, Agence Nationale de la Recherche convention ANR-23-PECL-0004 (PEPR Cloud, DiVA project) and Inria's Defi OS.

REFERENCES

[1] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967 Spring Joint Computer Conference (SJCC)*. Atlantic City, NJ, USA.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[3] AWS Blog. [n. d.]. Avoid PostgreSQL LWLock:buffer_content locks in Amazon Aurora: Tips and best practices. https://aws.amazon.com/blogs/database/avoid-postgresql-lwlockbuffer_content-locks-in-amazon-aurora-tips-and-best-practices [Accessed on 31/01/2026].

[4] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of operations on B-trees. *Acta inform.* 9, 1 (1977), 1–21.

[5] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 1–8.

[6] Postgres Shared Buffer. [n. d.]. Notes About Shared Buffer Access Rules. https://github.com/postgres/postgres/blob/master/src/backend/storage/buffer/README [Accessed on 31/01/2026].

[7] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. San Francisco, CA, 12 pages.

[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[9] Jonathon Corbet. 2014. MCS locks and qspinlocks. https://lwn.net/Articles/590243/, [Accessed on 31/01/2026].

[10] Travis Craig. 1993. *Building FIFO and priority-queuing spin locks from atomic swap*. Technical Report. Technical Report TR 93-02-02, Department of Computer Science, University of Washington.

[11] Dave Dice and Alex Kogan. 2019. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 12, 15 pages.

[12] Dave Dice and Alex Kogan. 2021. Hemlock: Compact and Scalable Mutual Exclusion. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. 173–183.

[13] David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. New Orleans, LA, 247–256.

[14] Durant, David. [n. d.]. Insert, Update, and Delete Indexes: Stairway to SQL Server Indexes. https://www.sqlservercentral.com/steps/insert-update-and-delete-indexes-stairway-to-sql-server-indexes-level-13 [Accessed on 31/01/2026].

[15] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[16] Jose M Faleiro and Daniel J Abadi. 2017. Latch-free synchronization in database systems: Silver bullet or fool's gold?. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*. Chaminade, FL.

[17] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. New Orleans, LA, 257–266.

[18] Vishal Gupta, Kumar Kartikeya Dwivedi, Yugesh Kothari, Yueyang Pan, Diyu Zhou, and Sanidhya Kashyap. 2022. Ship your Critical Section, Not Your Data: Enabling Transparent Delegation with TCLocks. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA.

[19] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 355–364.

[20] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The art of latency hiding in modern database engines. In *Proceedings of the 49th International Conference on Very Large Data Bases (VLDB)*, Vol. 17. Vancouver, Canada, 577–590.

[21] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. 2010. Decoupling Contention Management from Scheduling. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Pittsburgh, PA, 117–128.

[22] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. 2019. Scalable and Practical Locking With Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Ontario, Canada.

[23] Philip L. Lehman and S. Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 650–670.

[24] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[25] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*. Brisbane, Australia, 38–49.

[26] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Article 3, 8 pages.

[27] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019), 574–587.

[28] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*. Brisbane, Australia, 302–313.

[29] Li, Dichen. [n. d.]. 30 years of PostgreSQL buffer manager locking design evolution. https://medium.com/@dichenldc/30-years-of-postgresql-buffer-manager-locking-design-evolution-e6e861d7072f [Accessed on 31/01/2026].

[30] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote core locking: Migrating Critical-Section execution to improve the performance of multithreaded applications. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 65–76.

[31] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2016. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst.* 33, 4, Article 13 (Jan. 2016), 62 pages.

[32] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: a high-performance learned index on persistent memory. *arXiv preprint arXiv:2105.00683* (2021).

[33] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302* (2020).

[34] Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing* (Dresden, Germany) *(Euro-Par'06)*. 801–810.

[35] Peter Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*. IEEE, 165–171.

[36] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. 2015. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. 8, 11 (Aug.–Sept. 2015), 1298–1309.

[37] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. 183–196.

[38] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. (Feb. 1991), 21–65.

[39] John M Mellor-Crummey and Michael L Scott. 1991. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices* 26, 7 (1991), 106–113.

[40] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. 371–386.

[41] Postgres Buffer Strategy Lock. [n. d.]. Buffer Strategy Lock. https://github.com/postgres/postgres/blob/master/src/backend/storage/buffer/freelist.c#L33 [Accessed on 31/01/2026].

[42] Postgres Content Lock. [n. d.]. Buffer Content Lock. https://github.com/postgres/postgres/blob/master/src/include/storage/buf_internals.h#L359 [Accessed on 31/01/2026].

[43] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling highly contended OLTP workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 527–542.

[44] Zoran Radovic and Erik Hagersten. 2003. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, Washington, DC, USA, 241–252.

[45] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 342–358.

[46] Mohammad Sadoghi and Spyros Blanas. 2019. *Transaction Processing on Modern Hardware*. Morgan & Claypool Publishers. doi:10.2200/S00896ED1V01Y201901DTM058

[47] Michael L. Scott and William N. Scherer. 2001. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. Salt Lake City, UT, 44–52.

[48] Ge Shi, Ziyi Yan, and Tianzheng Wang. 2023. OptiQL: Robust optimistic locking for memory-optimized indexes. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.

[49] Transaction Processing Performance Council (TPC. [n. d.]. TPC-C Standard Specification. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf [Accessed on 31/01/2026].

[50] Lukas Vogel, Alexander Van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A write-optimized persistent log-structured hash-table. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2895–2907.

[51] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment* 10, 2 (2016), 49–60.

[52] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*. Paris, France, 461–472.

[53] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. 473–488.

[54] Ziyi Yan, Mohamed Farouk Drira, Tianxun Hu, and Tianzheng Wang. 2025. Tabular: Efficiently Building Efficient Indexes. *Proceedings of the VLDB Endowment* 18, 6 (2025), 1991–2004.