Supplementary Material for SIDLE

S1 SIDLE Framework

The SIDLE framework requires developers to adapt their tree-structure indexes in three steps: (1) adding SIDLE metadata to the original tree nodes; (2) calling SIDLE's frontend helper functions when allocating new nodes and accessing leaf nodes; and (3) implementing the interfaces related to the tree-specific logic for the background module. We present the details of this framework in the following sections. Besides, we use a B+ tree as an example to show how to integrate SIDLE with existing tree-structure indexes.

```
enum class mem_type : uint8_t {
    remote = 0, local, unknown
};
// internode metadata
struct sidle_metadata {
  mem_type type : 2;
  uint8_t depth : 6;
};
// leaf node metadata
struct sidle_leaf_metadata {
  uint16_t access_time;
  sidle_metadata metadata;
};
```

Listing 1. SIDLE's metadata.

```
// integrate sidle metadata into internal node
struct bplus_internode {
  int type;
  int parent_key_idx;
  struct bplus_internode *parent;
  int children;
  int key[BPLUS_MAX_ORDER - 1];
  ...
  + sidle_metadata sidle_meta;
};
// integrate sidle metadata into leaf node
struct bplus_leaf {
  int type;
  int parent_key_idx;
  struct bplus_internode *parent;
  int entries;
  int key[BPLUS_MAX_ENTRIES];
  ...
  + sidle_leaf_metadata sidle_meta;
};
```

Listing 2. Integration of SIDLE metadata into B+ tree nodes.

S1.1 SIDLE Metadata

Listing 1 shows the structure of SIDLE's metadata. Listing 2 gives an example of how to add SIDLE's metadata to the B+ tree nodes. Note that the field name of SIDLE's metadata should be sidle_meta, which is used by the frontend helper functions and the background module to access the metadata.

```
using I = internal_node;
using L = leaf_node;
using N = node;

// helper functions that implement frontend ops
// for layer-aware allocation
N* sidle_alloc(size_t size, I* parent, mem_type tgt_type);
void sidle_free(N *node, size_t size);
// for leaf-centric access tracking
void record_access(L* leaf);
```

Listing 3. Frontend helper functions of SIDLE.

```
// usage of sidle_alloc
- bplus_leaf* leaf_new()
+ bplus_leaf* leaf_new(bplus_internode* parent,
mem_type target_type)
{
  - bplus_leaf *node = malloc(sizeof(*node));
  + bplus_leaf *node = sidle_alloc<bplus_leaf>(
sizeof(*node), parent, target_type);
  ...
  return node;
}
// example of using the new allocation function
int insert(bplus_tree *tree, key_t key, int data)
{
  ...
  - bplus_leaf *new_leaf = leaf_new();
  // unknown means no specific memory type and uses the
     default layer-aware allocation
  + bplus_leaf *new_leaf = leaf_new(leaf->parent,
     mem_type::unknown);
  ...
}
// usage of record_access
int bplus_tree_search(key_t key)
{
  ...
  while (node != NULL) {
    if (is_leaf(node)) {
      bplus_leaf *ln = (bplus_leaf *)node;
      + record_access<bplus_leaf>(ln);
  ...
}
```

Listing 4. Using frontend helper functions in the B+ tree.

S1.2 SIDLE Frontend Helper Functions

As shown in Listing 3, for SIDLE's frontend module, SIDLE provides three template-based helper functions for developers to call. sidle_alloc implements layer-aware allocation using information from the parent node. Moreover, sidle_alloc also supports developers to specify the target memory type (i.e., fast memory or slow memory) for the new node, which is essential for the background module to migrate nodes (Listing 7: line 9). sidle_free is responsible for freeing the

```
using I = internal_node;
using L = leaf_node;
using N = node;
using T = tree;

// interfaces need to be implemented for the background
    module.
struct tree_op {
  using func = std::function;
  // traverse all leaf nodes
  func<void(T*, func<void(L*)>)> leaf_traverse;
  // migrate a leaf node to target memory type
  func<I*(L*, mem_type)> migrate_leaf;
  // migrate an internal node to target memory type
  func<I*(I*, mem_type)> migrate_internal;
  // traverse an internal node's all children
  func<void(I*, func<bool<N*>)> internal_traverse;
};

```

Listing 5. SIDLE's interfaces for background module.

memory allocated by sidle_alloc. record_access is used for leaf-centric access tracking.

Listing 4 illustrates how to use these helper functions in a B+ tree. When allocating new nodes, developers should replace the original allocation function with sidle_alloc. Additionally, developers should call record_access when accessing leaf nodes during tree operations (e.g., search).

S1.3 SIDLE Background Module Interfaces

SIDLE's background module requires some tree-specific functions to interact with the tree, for example, migration trigger and cooler need to scan all leaf nodes. Listing 5 shows these interfaces in tree_op, which developers need to implement for their tree-structure indexes to work with SIDLE's background module. The functionality of leaf_traverse is to traverse all leaf nodes and call the callback function for each leaf node. Thus, the migration trigger and cooler can utilize this callback to process the leaves. migrate_leaf and migrate_internal should implement the logic for migrating leaf/internal nodes to the target memory type, and return their parent node for further migration if needed. internal_traverse is used to traverse all children of an internal node and call the callback function for each child. The demotion executor uses this callback to check whether all children of an internal node are in slow memory.

Listing 6 demonstrates how to implement the migrate_leaf interface for a B+ tree[1] (i.e., bplus_migrate_leaf) and then pass these tree-specific functions to the background workers. It is worth noting that bplus_migrate_leaf's implementation largely draws on the logic of node-splitting in the B+ tree, such as updating the node's pointers. It is worth noting that the implementation of interfaces should follow the concurrency control mechanism used by the tree. For example, if the B+ tree uses optimistic concurrency control, the migration

[1] bplus_migrate_leaf is a demo that simplifies the concurrency control.

```
// migrate the leaf node to the target memory type
bplus_internode* bplus_migrate_leaf(bplus_leaf *leaf,
     mem_type target_type) {
  // Create a replica in the target memory tier
  bplus_leaf* new_leaf = allocate_leaf(target);
  clone_data(new_leaf, old_leaf);
  // Update the parent's pointer to the new node
  bplus_internode* p = old_leaf->get_parent();
  p->replace_child(old_leaf, new_leaf);
  // Update the leaf linked list
  update_sibling_links(old_leaf, new_leaf);
  // Mark the old node for safe reclamation, e.g., via RCU
  retire_node(old_leaf);
  // return the parent for further migration
  return p;
}

// register the tree-specific functions
tree_op bplus_tree_op = {
  .migrate_leaf = bplus_migrate_leaf,
  ...
};

// initialize background workers w/ the tree-specific ops
void init_background_workers() {
  // migration trigger is activated every wakeup_interval
  auto migration_trigger = new migration_trigger_t(
     wakeup_interval, tree, bplus_tree_op);
  auto promotion_executor = new promotion_executor_t(tree,
     bplus_tree_op);
  ...
}

// use of the migration interface in promotion executor
void promotion_executor_t::promote(bplus_leaf* l) {
  bplus_internode* n = bplus_tree_op.migrate_leaf(l,
     mem_type::local);
  while (n != nullptr && need_promote(n)) {
    n = bplus_tree_op.migrate_internal(n, mem_type::local);
  }
}
```

Listing 6. Demo of SIDLE's interface implementation in B+ tree.

should prevent concurrent reads and writes from accessing the migrating node by leveraging the tree's inherent node-level locking and version number.

As shown in Listing 6 function promote, the promotion executor directly invokes the user-implemented migration interface to migrate nodes. Given that the promotion/demotion procedure involves multiple invocations of the user-implemented migration interface, concurrent write may invalidate the migration candidate node n between two invocations. To prevent this, the implementation of migrate_internal can adopt one of the following two approaches: (1) For trees using locks for concurrency control, holding the node n's write lock between two invocations, thereby ensuring that no concurrent writes can modify n. (2) For trees with lock-free techniques, migrate_internal should first check the validity of n. If it is invalid (e.g., deleted), it should retrieve the

```
node* occ_migrate_node(node* n, mem_type target) {
  lock(n);
  node* parent = n->get_parent();
  // lock parent (and siblings for linked leaves)
  lock(parent);
  // Mark the node as migrating to trigger reader retries
  n->version->set_migrating(true);
  // Allocate and copy node data
  node* new_n = sidle_alloc<node>(sizeof(n), parent, target)
  memcpy(new_n, n, sizeof(n));
  if (is_leaf(n)) // update sibling pointers
    update_sibling_pointers(n, new_n);
  // Update the parent's child pointer
  parent->update_child(n, new_n);
  new_n->version->unset_migration_and_increment();
  // Mark old node as deleted and release locks
  n->mark_deleted();
  // unlock parent and node (and siblings for linked leaves)
  unlock_all(n, parent, new_n);
  return parent;
}
```

Listing 7. OCC-style node migration (e.g., S-Masstree).

```
node_id lock_free_migrate(node_id id, mem_type target) {
 while (true) {
  // Get the address of the node (or node chain for Bw-Tree)
  node* old_addr = mapping_table->get(id);
  // Create a copy at the target memory layer
  node* new_addr = sidle_alloc<node>(sizeof(node), nullptr,
     target);
  memcpy(new_addr, old_addr, sizeof(node));
  // CAS to swap the physical address in the mapping table
  if (CAS(mapping_table->ptr(id), old_addr, new_addr)) {
    // Success: retire the old node using epoch-based GC
    epoch_retire(old_addr);
    node_id parent_id = old_addr->get_parent_id();
    return parent_id;
  }
  // Failure: node has been modified; retry
  sidle_free<node>(new_addr);
 }
}
```

Listing 8. Lock-free node migration using mapping table (e.g., Bw-Tree style).

updated candidate node and retry the check-and-migrate process in line 34–36.

S2 Migration Concurrency Strategies

Generally, the concurrency correctness of migration is guaranteed by the user's interface implementation. While the promotion/demotion procedure involves multiple invocations of the user-implemented migration interface, the correctness of concurrent foreground reads is not affected because the migration thread only checks the state of the nodes (using atomic operations) between invocations. Meanwhile, the concurrent writes between two invocations are avoided or corrected by the migration interface implementation, as described above (§S1 last paragraph).

To better help developers understand how to implement migration to ensure concurrency correctness, in this section, we present the abstracted pseudocode capturing the core logic of node migration under four typical concurrency control mechanisms.[2] This illustrated logic focuses on how each strategy guarantees the concurrency correctness of migration. The analyzed mechanisms include optimistic concurrency control (e.g., Masstree [1]), lock-free (e.g., Bw-Tree [2]), read-write locking (i.e., using read-write locks for each node), and coarse-grained locking (using one lock for the entire tree). We further provide their TLA+ verifications in https://github.com/sidle-project/sidle-supplyment/.

Optimistic concurrency control (OCC). In OCC, each node has its own lock to protect writes, and the version number to protect reads. Reads check version numbers before and after accessing the node, and retry if the version number has changed or the migrating flag is set. As shown in Listing 7, the migration process is similar to the node-splitting operation in the original tree. For concurrent writes, the lock acquisition on line 3–5 and release on line 19 ensure that the nodes involved in the migration are not modified by concurrent writes, thus ensuring the correctness of writes and migration. For concurrent reads, the migrating flag is set on line 7 and unset on line 15, together with incrementing the version number, to trigger retries when a read operation accesses the migrating nodes (i.e., n and new_n). This ensures that concurrent reads will not see intermediate states during migration, thus ensuring their correctness. Note that for trees with the linked list (e.g., B+ tree), migrating leaf nodes also need to lock and update sibling pointers. Additionally, as long as the locking order of node, parent, and siblings is consistent with other tree operations (e.g., split), deadlocks will not occur. Overall, this migration can ensure the correctness of concurrent migration and other tree operations.

Lock-free. Lock-free trees typically use atomic compare-and-swap (CAS) operations to ensure the atomicity of updates. For example, Bw-Tree [2] utilizes a *mapping table* to manage the pointers of nodes. Specifically, all pointers in the tree (e.g., parent node pointers and sibling pointers) store node logical IDs (NIDs), and the mapping table maps each NID to the physical memory address of the corresponding node. Thus, all updates to the tree (e.g., insert, delete, split) create a new version of the node and update the mapping table using CAS to point to the new version.

The migration process for lock-free trees like Bw-Tree is shown in Listing 8. Concurrent writes will trigger retries (line 2–17) due to the CAS operation on line 9, which ensures that the migration and foreground tree write operations do not interfere with each other. Thus, the correctness of writes is guaranteed. As reads always access nodes through the

[2]The presented pseudocode focuses on the core logic of single node migration, omitting auxiliary operations for ensuring concurrency safety between invocations.

```
node* rw_lock_migrate(node* n, mem_type target) {
  // get parent write lock & check parent-child relationship
  p->write_lock_and_validate_child(n);
  if (is_leaf(n) && n->linked_left_sibling())
    n->left->write_lock();
  // Acquire the node write lock
  n->write_lock();
  if (is_leaf(n) && n->linked_right_sibling())
    n->right->write_lock();
  // Allocate new node at target memory type
  node* new_n = sidle_alloc<node>(sizeof(node), p, target);
  memcpy(new_n, n, sizeof(node));
  if (is_leaf(n)) // update linked-list pointers
    update_sibling_pointers(n, new_n);
  // Update parent-child relationship in the tree
  p->update_child(n, new_n);
  // Mark old node as deleted and release locks
  n->mark_deleted();
  unlock_all();
  return p;
}
```

Listing 9. Node migration with RW-Lock (e.g., B+ tree style).

mapping table and the CAS operation ensures that mapping table updates are atomic (line 9), concurrent reads will not see intermediate states during migration, thus ensuring the correctness of reads. Moreover, since all pointers in the tree are NIDs, there is no need to update the parent and sibling pointers during migration.

Read-Write (RW) locking. For trees using RW locks, the RW lock ensures that multiple readers can access the tree concurrently, while writers have exclusive access. The migration process in this case is shown in Listing 9. For concurrent writes and reads, the write lock acquisition on line 3–9 and release on line 19 ensure that the nodes involved in the migration are not modified by concurrent writes and cannot be read during migration. This ensures that both writes and reads are correct. Note that the lock acquisition must follow the lock crabbing approach [3], where the parent node's lock is acquired before the child node's lock, to prevent deadlocks and data inconsistency. Moreover, for migrating leaf nodes with sibling pointers, the lock acquisition order of sibling nodes must be consistent with other tree operations (e.g., split) (here we assume a left-to-right order is used in tree operations, line 4–9) to prevent deadlocks.

Coarse-grained locking. In coarse-grained locking, a single lock protects the entire tree. As shown in Listing 10, the migration process simply acquires the tree lock (line 3) to prevent concurrent operations from interfering with the migration. Since all tree operations need to acquire the same lock, this migration can ensure the correctness of concurrent migration and other tree operations.

```
node* coarse_grained_migrate(node* n, mem_type target) {
  // Acquire tree lock to prevent concurrent data access
  tree->tree_lock();
  // Allocate new node at target memory type
  node* p = n->parent;
  node* new_n = sidle_alloc<node>(sizeof(n), p, target);
  memcpy(new_n, n, sizeof(node));
  if (is_leaf(n)) // update linked-list pointers
    update_sibling_pointers(n, new_n);
  // Update parent-child relationship in the tree
  p->update_child(n, new_n);
  delete n;
  t->tree_unlock();
  return p;
}
```

Listing 10. Migration using coarse-grained locking.

References

[1] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, Bern, Switzerland, 2012. ACM.

[2] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, page 302–313, USA, 2013. IEEE Computer Society.

[3] R. Bayer and M. Schkolnick. *Concurrency of operations on B—trees*, page 216–226. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.