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 6: line9). sidle_free is responsible for freeing the memory allocated by sidle_alloc. record_access is used for leaf-centric access tracking.

```cpp
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.

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 data access.

S1.3 SIDLE Background Module Interfaces

SIDLE's background module requires some tree-specific functions to interact with the tree, such as migration trigger and cooler need to scan all leaf nodes. In Listing 5, tree_op contains several interfaces that developers need to implement for the 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 leaf node. migrate_leaf and migrate_internal should implement the logic for migrating leaf nodes and internal nodes to the target memory type, respectively. 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. Specifically, the implementation of interfaces needs to consider the concurrency control mechanism used by the tree. In this case, bplus_migrate_leaf utilizes the tree's inherent node-grained locking and version number for concurrency control, while Read-Copy-Update (RCU) is used for memory reclamation. Developers should keep the tree's inherent concurrency control mechanism in mind when implementing the interfaces.

```cpp
// migrate the leaf node to the target memory type
// return the parent of the leaf node, which is used for migrating the parent during structure-aware migration
bplus_internode* bplus_migrate_leaf(bplus_leaf *leaf, mem_type target_type)
{
  // check leaf's validity (skip)
  leaf->lock();
  bplus_internode* p = leaf->locked_parent();
  // check parent's validity (skip)
  bplus_leaf *new_leaf = leaf_new(p, target_type);
  // copy data from old leaf to new leaf
  // note: new leaf inherits old leaf's lock
  memcpy(new_leaf, leaf, sizeof(bplus_leaf));
  leaf->mark_migration();
  // find leaf's position in the parent node
  int pos = p->find_child(leaf);
  // change the parent-child relation pointer
  p->set_child(pos, new_leaf);
  // update the leaf's link list
  change_link_list(leaf, new_leaf);
  // delete the original node
  leaf->mark_deleted();
  leaf->deallocate_rcu();
  leaf->unlock();
  new_node->unlock();
  return p;
}

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

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

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

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