```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "./pcm/pcm-memory.cpp"
//#include "./pcm/pcm-numa.cpp"
#include "./papi_util.cpp"

#include "microbench.h"

#include <cstring>
#include <cctype>
#include <atomic>

#define COUNT_READ_MISS

thread_local long skiplist_steps = 0;
std::atomic<long> skiplist_total_steps;

//#define USE_TBB

#ifdef USE_TBB
#include "tbb/tbb.h"
#endif

// Enable this if you need pre-allocation utilization
//#define BWTREE_CONSOLIDATE_AFTER_INSERT

#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
#ifdef USE_TBB
  #warning "Could not use TBB and BwTree consolidate together"
  #endif
#endif

#ifdef BWTREE_COLLECT_STATISTICS
#ifdef USE_TBB
  #warning "Could not use TBB and BwTree statistics together"
  #endif
```

```
#endif

// Whether insert interleaves
//#define INTERLEAVED_INSERT

// Whether read operatoin miss will be counted
//#define COUNT_READ_MISS

typedef uint64_t keytype;
typedef std::less<uint64_t> keycomp;

static const uint64_t key_type=0;
static const uint64_t value_type=1; // 0 = random pointers, 1 = pointers to keys

extern bool hyperthreading;

// This is the flag for whather to measure memory bandwidth
static bool memory_bandwidth;
// Whether to measure NUMA Throughput
static bool numa;
// Whether we only perform insert
static bool insert_only;

bool del;

// We could set an upper bound of the number of loaded keys
static int64_t max_init_key = -1;

static std::vector<int> _listeners;

#include "util.h"

int setup_server(u_int16_t port)
{
   int _socket;

   struct sockaddr_in server;
   int iSetOption = 1;

   _socket = socket(AF_INET, SOCK_STREAM, 0);
   setsockopt(_socket, SOL_SOCKET, SO_REUSEADDR, (char*)&iSetOption,
sizeof(iSetOption));
   if (_socket < 0) {
```

```
        perror("Cannot create a socket"); exit(1);
    }
    bzero((char *) &server, sizeof(server));
    u_int16_t server_port = port;
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(port);
    // The bind function associates a local address with a socket.
    if (bind(_socket, (struct sockaddr *) &server,
            sizeof(server)) < 0) {
        perror("ERROR on binding.");
    }
    // The listen function places a socket in a state in which it is listening for an incoming
connection.
    listen(_socket,5);

    return _socket;
}


int setup_listener(int _socket)
{
    struct sockaddr_in client_addr;
    socklen_t clilen;
    clilen = sizeof(client_addr);
    // The accept function permits an incoming connection attempt on a socket.
    int new_socket = accept(_socket, (struct sockaddr *) &client_addr, &clilen);
    if (new_socket < 0)
    {
        perror("ERROR on accept\n");
    }
    return new_socket;
}

void handle_info(char *buffer, int _listener)
{
    long recv = read(_listener, buffer, 512);
    if (recv < 0) {
        perror("ERROR reading from socket\n");
    } else {
        recv = write(_listener, "OK\0", 3);
        if (recv < 0) {
            perror("ERROR writing to socket\n");
```

```
        }
    }
}

void handle_socket_response(long res)
{
    if (res < 0){
        printf("An error occurred while sending or receiving data from the client. Aborting ...\n");
        exit(-1);
    }
};

//============================================================
// EXEC
//============================================================
inline void exec(int wl, int index_type, int num_thread)
{

    Index<keytype, keycomp> *idx = getInstance<keytype, keycomp>(index_type, key_type);

#ifdef USE_TBB
    tbb::task_scheduler_init init{num_thread};

  std::atomic<int> next_thread_id;
  next_thread_id.store(0);

  auto func = [idx, &init_keys, &values, &next_thread_id](const tbb::blocked_range<size_t>& r) {
    size_t start_index = r.begin();
    size_t end_index = r.end();

    threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);

    int thread_id = next_thread_id.fetch_add(1);
    idx->AssignGCID(thread_id);

    int gc_counter = 0;
    for(size_t i = start_index;i < end_index;i++) {
      idx->insert(init_keys[i], values[i], ti);
      gc_counter++;
      if(gc_counter % 4096 == 0) {
        ti->rcu_quiesce();
      }
    }
```

```
    ti->rcu_quiesce();
    idx->UnregisterThread(thread_id);

    return;
  };

  idx->UpdateThreadLocal(num_thread);
  tbb::parallel_for(tbb::blocked_range<size_t>(0, count), func);
  idx->UpdateThreadLocal(1);
#else
    auto func = [idx, num_thread, index_type](uint64_t thread_id, bool) {

        threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);

        int _listener = _listeners[thread_id];

        char buffer[512];
        std::string k_s, v_s;

        handle_info(buffer, _listener);
        int share_size = atoi(buffer);
        //printf("My share size: %d\n", share_size);

        int gc_counter = 0;
//#ifdef INTERLEAVED_INSERT
//        for(size_t i = thread_id;i < total_num_key;i += num_thread) {
//#else
        for(int i = 0;i < share_size;i++) {
//#endif
            handle_socket_response(read(_listener, buffer, 512));

            istringstream iss(buffer);
            iss >> k_s >> v_s;

            keytype key = std::stoull(k_s, 0, 10);
            uint64_t value = std::stoull(v_s, 0, 10);

#ifdef BWTREE_USE_DELTA_UPDATE
            //printf("Want to insert key %li and val %li\n", (keytype) key, value);
            bool res = idx->insert((keytype)key, value, ti);
#else
            bool res = idx->insert_bwtree_fast((keytype)key, value);
```

```
#endif
            if (res)
                handle_socket_response(write(_listener, (to_string(BWTREE_STORED) +
"\0").c_str(), 2));
            else
                handle_socket_response(write(_listener, (to_string(BWTREE_NOTSTORED) +
"\0").c_str(), 2));

            gc_counter++;
            if(gc_counter % 4096 == 0) {
                ti->rcu_quiesce();
            }
            if(gc_counter % 500000 == 0){
                printf("Thread handled %.2f%% of its insert-operations, %d of %d.\n",
((float(gc_counter)*100.0)/float(share_size)), gc_counter, share_size);
            }
        }

        ti->rcu_quiesce();
        return;
    };
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    /*
    if(memory_bandwidth == true) {
        PCM_memory::StartMemoryMonitor();
    }
    if(numa == true) {
        PCM_NUMA::StartNUMAMonitor();
    }*/
    //auto start = std::chrono::high_resolution_clock::now();

    StartThreads(idx, num_thread, func, false);

    //auto end = std::chrono::high_resolution_clock::now();
    //printf("\nExecuting input file 1 took %f ms.\n\n", std::chrono::duration<double, std::milli>(end
- start).count());
    /*
    if(memory_bandwidth == true) {
        PCM_memory::EndMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::EndNUMAMonitor();
```

```
   }*/
#endif

   // Only execute consolidation if BwTree delta chain is used
#ifdef BWTREE_CONSOLIDATE_AFTER_INSERT
   fprintf(stderr, "Starting consolidating delta chain on each level\n");
  idx->AfterLoadCallback();
#endif

   // If the workload only executes load phase then we return here
   if(insert_only) {
      delete idx;
      return;
   }
   ////////////////////////////////////////////////////////////////////////////////////////////////////////////
   ////DELETE PART START////////////////////////////////////////////////////////////////////////////////////////
   ////////////////////////////////////////////////////////////////////////////////////////////////////////////

   if (del){
      auto del_func = [num_thread, idx](uint64_t thread_id, bool) {

         int _listener = _listeners[thread_id];

         char buffer[512];
         std::string o_s, k_s, v_s;

         handle_info(buffer, _listener);
         int share_size = atoi(buffer);
         //printf("My second share size: %d\n", share_size);

         threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);

         int counter = 0;
         for(int i = 0;i < share_size;i++) {

            handle_socket_response(read(_listener, buffer, 512));

            istringstream iss(buffer);
            iss >> o_s >> k_s >> v_s;

            int op = std::stoi(o_s);
            keytype key = std::stoull(k_s, 0, 10);
            uint64_t value = (v_s == "" ? 0 : std::stoull(v_s, 0, 10));
```

```
            bool res;

            if (op == OP_REMOVE){
                res = idx->remove(key, value, ti);
            } else {
                bool res = false;
                printf("Unknown operation %d\n", op);
            }

            if (res)
                handle_socket_response(write(_listener, (to_string(BWTREE_DELETED) +
"\0").c_str(), 2));
            else
                handle_socket_response(write(_listener, (to_string(BWTREE_NOTDELETED) +
"\0").c_str(), 2));

            counter++;
            if(counter % 4096 == 0) {
                ti->rcu_quiesce();
            }
            if(counter % 500000 == 0){
                printf("Thread handled %.2f%% of its delete-operations, %d of %d.\n",
((float(counter)*100.0)/float(share_size)), counter, share_size);
            }
        }

        // Perform GC after all operations
        ti->rcu_quiesce();

        return;
    };
    /*
    if(memory_bandwidth == true) {
        PCM_memory::StartMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::StartNUMAMonitor();
    }
    */
    //start = std::chrono::high_resolution_clock::now();
    StartThreads(idx, num_thread, del_func, false);
    //end = std::chrono::high_resolution_clock::now();
```

```
        //printf("\nExecuting input file 2 took %f seconds.\n", (std::chrono::duration<double,
std::milli>(end - start).count())/1000);
        /*
        if(memory_bandwidth == true) {
            PCM_memory::EndMemoryMonitor();
        }

        if(numa == true) {
            PCM_NUMA::EndNUMAMonitor();
        }*/
    }

    ////////////////////////////////////////////////////////////////////////////////////////////////////////
    ////DELETE PART END////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////////
    // This is used to count how many read misses we have found
    std::atomic<size_t> read_miss_counter{}, read_hit_counter{};
    read_miss_counter.store(0UL);
    read_hit_counter.store(0UL);

    auto func2 = [num_thread, idx, &read_miss_counter, &read_hit_counter](uint64_t thread_id,
bool) {

        int _listener = _listeners[thread_id];

        char buffer[512];
        std::string o_s, k_s, v_s;

        handle_info(buffer, _listener);
        int share_size = atoi(buffer);
        //printf("My second share size: %d\n", share_size);

        std::vector<uint64_t> v;
        v.reserve(10);

        threadinfo *ti = threadinfo::make(threadinfo::TI_MAIN, -1);

        int counter = 0;
        for(int i = 0;i < share_size;i++) {

            handle_socket_response(read(_listener, buffer, 512));

            istringstream iss(buffer);
```

```
        iss >> o_s >> k_s >> v_s;

        int op;
        keytype key;
        uint64_t value;

        try {
            op = std::stoi(o_s);
            key = std::stoull(k_s, 0, 10);
            value = (v_s == "" ? 0 : std::stoull(v_s, 0, 10));
        }
        catch (std::invalid_argument) {
            op = 10;
        }

        if (op == OP_INSERT) { //INSERT
            if (idx->insert(key, value, ti))
                handle_socket_response(write(_listener, (to_string(BWTREE_STORED) +
"\0").c_str(), 2));
            else
                handle_socket_response(write(_listener, (to_string(BWTREE_NOTSTORED) +
"\0").c_str(), 2));

        } else if (op == OP_READ) { //READ
            v.clear();

#ifdef BWTREE_USE_MAPPING_TABLE
            idx->find(key, &v, ti);
#else
            idx->find_bwtree_fast(key, &v);
#endif

            // If we count read misses then increment the
            // counter here if the vetor is empty
#ifdef COUNT_READ_MISS
            if(v.size() == 0UL){
                read_miss_counter.fetch_add(1);
                handle_socket_response(write(_listener, (to_string(BWTREE_NOTREAD) +
"\0").c_str(), 2));
            }
            else{
                read_hit_counter.fetch_add(1);
```

```
                handle_socket_response(write(_listener, (to_string(BWTREE_READ) +
"\0").c_str(), 2));
            }
#endif
        } else if (op == OP_UPSERT) { //UPDATE

            if(!idx->upsert(key, value, ti)) // The negation is correct here since the bwtree code
originally returns false if we end up updating a key
                handle_socket_response(write(_listener, (to_string(BWTREE_UPDATED) +
"\0").c_str(), 2));
            else
                handle_socket_response(write(_listener, (to_string(BWTREE_NOTUPDATED) +
"\0").c_str(), 2));

        } else if (op == OP_REMOVE){
            if(idx->remove(key, value, ti))
                handle_socket_response(write(_listener, (to_string(BWTREE_DELETED) +
"\0").c_str(), 2));
            else
                handle_socket_response(write(_listener, (to_string(BWTREE_NOTDELETED) +
"\0").c_str(), 2));
        } else {
            handle_socket_response(write(_listener, (to_string(BWTREE_UNKNOWNOP) +
"\0").c_str(), 2));
            printf("Unknown operation %d\n", op);
        }

        counter++;
        if(counter % 4096 == 0) {
            ti->rcu_quiesce();
        }
        if(counter % 500000 == 0){
            printf("Thread handled %.2f%% of its extra-operations, %d of %d.\n",
((float(counter)*100.0)/float(share_size)), counter, share_size);
        }
    }

    // Perform GC after all operations
    ti->rcu_quiesce();

    return;
};
/*
```

```
    if(memory_bandwidth == true) {
        PCM_memory::StartMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::StartNUMAMonitor();
    }
    */
    //start = std::chrono::high_resolution_clock::now();
    StartThreads(idx, num_thread, func2, false);
    //end = std::chrono::high_resolution_clock::now();
    //printf("\nExecuting input file 2 took %f seconds.\n", (std::chrono::duration<double,
std::milli>(end - start).count())/1000);
    /*
    if(memory_bandwidth == true) {
        PCM_memory::EndMemoryMonitor();
    }

    if(numa == true) {
        PCM_NUMA::EndNUMAMonitor();
    }
    */
    // Print out how many reads have missed in the index (do not have a value)
#ifdef COUNT_READ_MISS
    fprintf(stderr,
          "  Read misses: %lu; Read hits: %lu\n",
          read_miss_counter.load(),
          read_hit_counter.load());
#endif

    delete idx;

    return;
}

void process(int _socket){


    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    printf("Waiting for client to connect ... \n");
    int _listener = setup_listener(_socket);

    char buffer[512];
```

```
    handle_info(buffer, _listener);
    int wl = atoi(buffer);

    handle_info(buffer, _listener);
    int index_type = atoi(buffer);

    handle_info(buffer, _listener);
    int num_thread = atoi(buffer);

    handle_info(buffer, _listener);
    insert_only = atoi(buffer);

    handle_info(buffer, _listener);
    numa = atoi(buffer);

    handle_info(buffer, _listener);
    hyperthreading = atoi(buffer);

    handle_info(buffer, _listener);
    memory_bandwidth = atoi(buffer);

    handle_info(buffer, _listener);
    int repeat_counter = atoi(buffer);

    handle_info(buffer, _listener);
    del = atoi(buffer);

    for (int i = 0; i < num_thread; i++)
        _listeners.emplace_back(setup_listener(_socket));

    printf("\nReceived: %d %d %d %d %d %d %d %d %d\n", wl, index_type, num_thread,
insert_only, numa, hyperthreading, memory_bandwidth, repeat_counter, del);

    close(_listener);
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    while (repeat_counter > 0){
        auto start = std::chrono::high_resolution_clock::now();

        exec(wl, index_type, num_thread);

        auto end = std::chrono::high_resolution_clock::now();
```

```
        //printf("\nThe whole thing took %f seconds.\n\n", (std::chrono::duration<double,
std::milli>(end - start).count())/1000);
        repeat_counter--;
        std::cout << "\n";
    }

    for (int i = 0; i < _listeners.size(); i++)
        close(_listeners[i]);

}

int main()
{
    int _socket = setup_server(11211);
    for (int i = 0; i < 250; i++){
        process(_socket);
        printf("done it %d\n", i);
    }
    close(_socket);
}
```