Tal Derei and Sang-Jun Park

CSE 411: Project 2
Syncrobench: Memory Reclamation

Part 1: Code Review

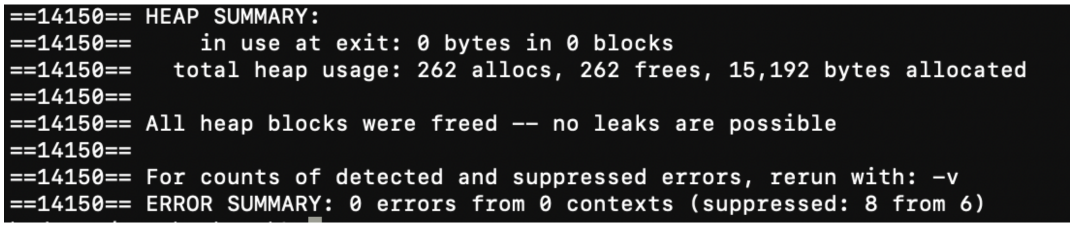
Executing the codebase inside the 'lazy-list' folder generates a 'MUTEX-lazy-list' binary, but the underlying lock-based algorithm used during runtime is obscure. There's an optional '-x' command-line flag that changes the default algorithm from [1] lock-coupling to [2] lazy, but the codebase doesn't support this flag, responding with the error message "The parameter s is not valid for this benchmark". The helper method indicates that the runtime is using lock-coupling as the default algorithm, but it was difficult to verify whether it was actually defaulting to lock-coupling or lazy. Inspecting the codebase more closely, it uses API function calls to lazy.c rather than coupling.c, even though it says it uses lock-coupling as the default algorithm in the documentation. In 'test.c', function calls are made to 'set_add_l(), set_remove_l(), and set_contains_l()' inside of 'inset.c', where they either call functions inside of 'coupling.c' or 'lazy.c' depending on the value of the TRANSACTIONAL field. This field is a typedef for 'd->unit_tx' inside of 'test.c', where 'unit_tx" represents the default DEFAULT_LOCKTYPE, which is set to 2. Therefore, although the instructions indicate that the default lock-type is 'lock-coupling', the codebase reveals that it's using lazy execution under the hood. The only way to change the default lock-based algorithm to lock-coupling is by changing the DEFAULT_LOCKTYPE to 1 inside of linkedlist-lock.h, which would trigger the functions calls inside of inset.c to be directed to coupling.c rather than lazy.c.

In regards to the overall readability of the codebase and documentation, the lack of comments make it difficult to follow the control flow of the program. As a result, some of the most important lines of code are hidden and not documented well, for example: 'pthread_create(&threads[i], &attr, test, (void *)(&data[i])) != 0)' where threads are created and passed the 'test()' function to run asynchronously in parallel. In regards to code refactoring, it seems like concurrent data structures would be more suited for a C++ workflow, as C++ supports class inheritance and polymorphism constructs. Additionally, C++ includes built-in support for threads, mutual exclusion, condition variables, and futures as part of the standard library. This would help significantly reduce the codebase and make it more compact and efficient as compared with C.

Part 2: Unsafe Memory Reclamation

[1] Implement Unsafe Reclamation
[2] Compare Performance (Original Implementation vs. Unsafe Reclamation)
[3] Artificial Scenario

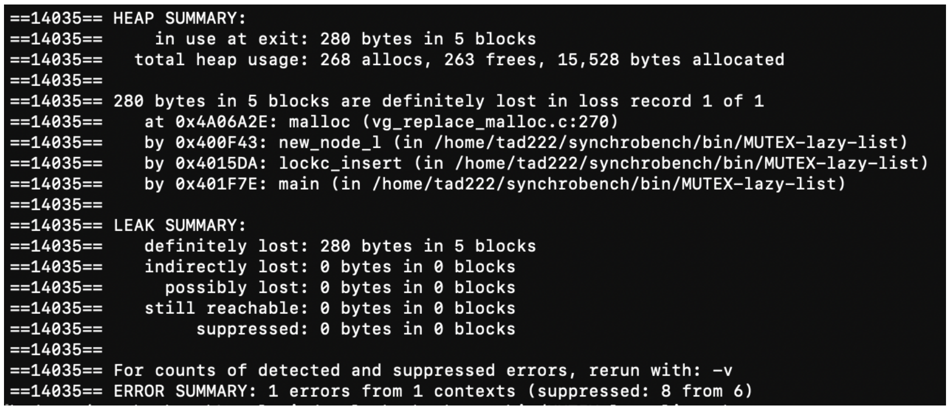
[1] Baseline tests indicate that memory reclamation is NOT implemented for the lazy-list data structure, as expected. Executing the command 'valgrind --leak-check=yes bin/MUTEX-lazy-list -t 1 -u 0' with an update ratio set to '0' yielded expected results: no-memory leaks were identified since none of the nodes were marked as logically deleted.



```
==14150== HEAP SUMMARY:
==14150==     in use at exit: 0 bytes in 0 blocks
==14150==   total heap usage: 262 allocs, 262 frees, 15,192 bytes allocated
==14150==
==14150== All heap blocks were freed -- no leaks are possible
==14150==
==14150== For counts of detected and suppressed errors, rerun with: -v
==14150== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 8 from 6)
```

Figure 1: Valgrind for Lazy-List with Update Ratio = 0% for single thread

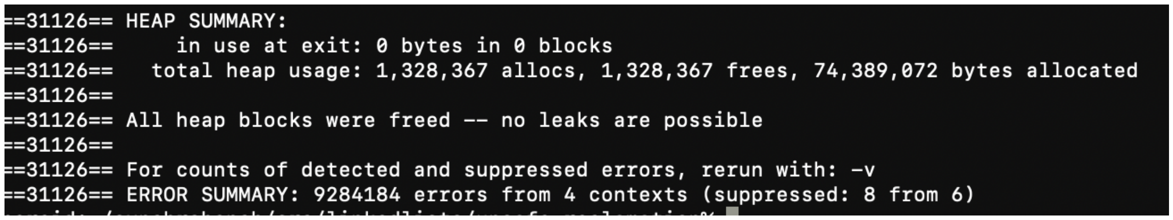
When increasing the update ratio to 20%, valgrind identified memory-leaks (as shown below) since nodes marked as logically deleted weren't being reclaimed by either the program or the lazy-list library. As the update ratio increased, the memory leaks in the heap increased proportionally.



```
==14035== HEAP SUMMARY:
==14035==     in use at exit: 280 bytes in 5 blocks
==14035==   total heap usage: 268 allocs, 263 frees, 15,528 bytes allocated
==14035==
==14035== 280 bytes in 5 blocks are definitely lost in loss record 1 of 1
==14035==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==14035==    by 0x400F43: new_node_l (in /home/tad222/synchrobench/bin/MUTEX-lazy-list)
==14035==    by 0x4015DA: lockc_insert (in /home/tad222/synchrobench/bin/MUTEX-lazy-list)
==14035==    by 0x401F7E: main (in /home/tad222/synchrobench/bin/MUTEX-lazy-list)
==14035==
==14035== LEAK SUMMARY:
==14035==    definitely lost: 280 bytes in 5 blocks
==14035==    indirectly lost: 0 bytes in 0 blocks
==14035==      possibly lost: 0 bytes in 0 blocks
==14035==    still reachable: 0 bytes in 0 blocks
==14035==         suppressed: 0 bytes in 0 blocks
==14035==
==14035== For counts of detected and suppressed errors, rerun with: -v
==14035== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 8 from 6)
```

Figure 2: Valgrind for Lazy-List with Update Ratio = 20% for single thread

In order to rectify these memory leaks, adding 'free(curr)' after marking the node as logically deleted and physically deleting it by redirecting the pointer freed the memory. By freeing the dangling 'curr' node, valgrind outputs that the memory is freed and "no memory leaks are possible" for both single and multithreaded workloads, as shown below. The commands executed, respectively: "../../../bin/unsafe-reclamation-lazy-list -i 1024 -r 2048 -t 8 -u 50'



```
==31126== HEAP SUMMARY:
==31126==     in use at exit: 0 bytes in 0 blocks
==31126==   total heap usage: 1,328,367 allocs, 1,328,367 frees, 74,389,072 bytes allocated
==31126==
==31126== All heap blocks were freed -- no leaks are possible
==31126==
==31126== For counts of detected and suppressed errors, rerun with: -v
==31126== ERROR SUMMARY: 9284184 errors from 4 contexts (suppressed: 8 from 6)
```

Figure 3: Valgrind for Lazy-List with Update Ratio = 50% for single thread

[2] Comparing the performance of this unsafe version with the original implementation (without reclamation) with respect to transactional throughput yielded expected results: the process of freeing memory reduces transactional throughput.

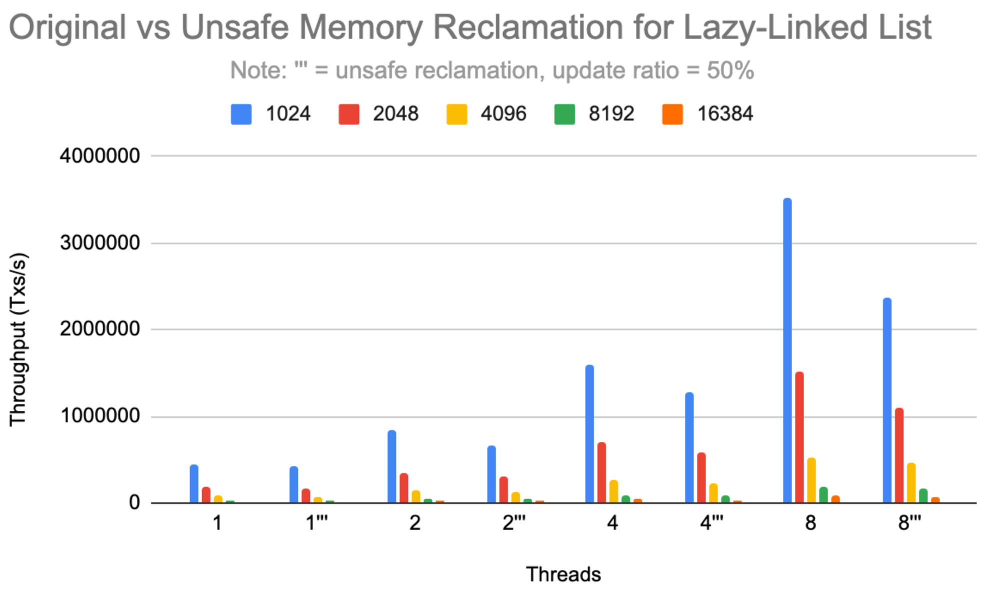
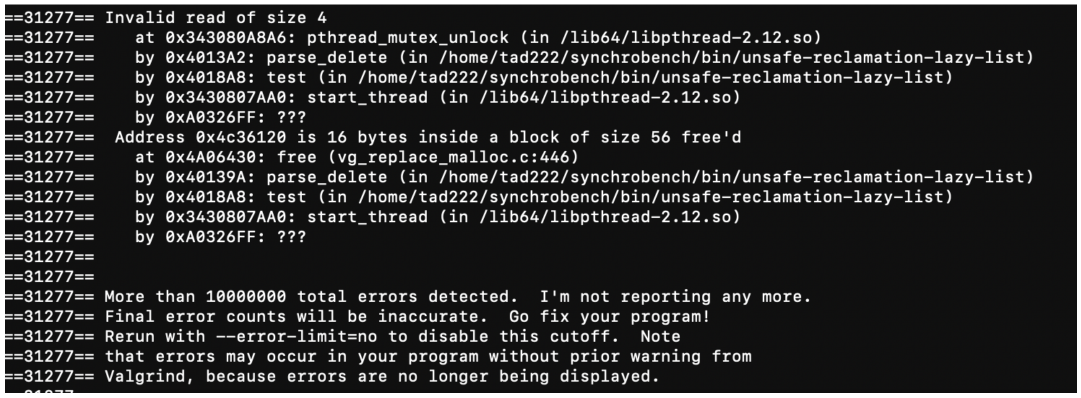


Figure 4: Transactional throughput as a Function of the Number of Threads (50% update ratio)

In the chart above, the unsafe memory reclamation is achieving slightly less throughput than the original implementation since the process of freeing memory takes time, and reduces the throughput. Comparing the results between the original implementation and the unsafe implementation for an update ratio of 0% were the same, since there's no logical/physical deletion. Therefore, an effective update ratio of 50% was enforced in order to compare the throughput for transactions executing these delete API function calls.

[3] With respect to building an artificial scenario (using delays and deterministic operations for example) that clarifies how this unsafe implementation can raise run-time errors, the original Syncrobench codebase is already an example of the prime scenario! The issue is that this reclamation technique is UNSAFE since you can't free the memory directly after deleting it, since another thread may have an active reference to that memory location. Valgrind is therefore detecting invalid reads from certain memory locations, yet still freeing the memory since the reclamation technique is "correct".



```
==31277== Invalid read of size 4
==31277==    at 0x343080A8A6: pthread_mutex_unlock (in /lib64/libpthread-2.12.so)
==31277==    by 0x4013A2: parse_delete (in /home/tad222/synchrobench/bin/unsafe-reclamation-lazy-list)
==31277==    by 0x4018A8: test (in /home/tad222/synchrobench/bin/unsafe-reclamation-lazy-list)
==31277==    by 0x3430807AA0: start_thread (in /lib64/libpthread-2.12.so)
==31277==    by 0xA0326FF: ???
==31277==  Address 0x4c36120 is 16 bytes inside a block of size 56 free'd
==31277==    at 0x4A06430: free (vg_replace_malloc.c:446)
==31277==    by 0x40139A: parse_delete (in /home/tad222/synchrobench/bin/unsafe-reclamation-lazy-list)
==31277==    by 0x4018A8: test (in /home/tad222/synchrobench/bin/unsafe-reclamation-lazy-list)
==31277==    by 0x3430807AA0: start_thread (in /lib64/libpthread-2.12.so)
==31277==    by 0xA0326FF: ???
==31277==
==31277==
==31277== More than 10000000 total errors detected.  I'm not reporting any more.
==31277== Final error counts will be inaccurate.  Go fix your program!
==31277== Rerun with --error-limit=no to disable this cutoff.  Note
==31277== that errors may occur in your program without prior warning from
==31277== Valgrind, because errors are no longer being displayed.
```

Figure 5: Valgrind Error for Unsafe Reclamation

Part 3: Memory Reclamation (Smart Pointers and Epoch-Based Reclamation)

[1] Smart-Pointers
[2] Epoch-Based Reclamation (EBR)
[3] Comparing Smart-Pointers and EBR

[1] Implementing smart pointers presented a number of challenges:

1. Converting pointers to smart pointers involves converting every 'node_l_t' pointer in lazy.c and linkedlist-lock.h to shared_ptr (e.g. std::shared_ptr<node_l_t> newnode). This is a smart pointer to a node_l_t struct (i.e. a shared pointer to a node in the set).

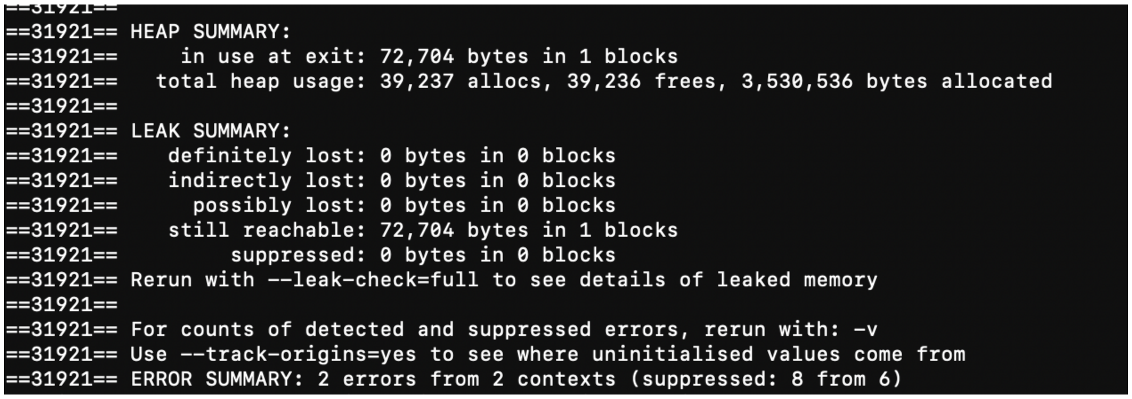
2. The next step is to implement all the assignment operations (e.g. curr = get_marked_ref(pred->next)) atomically by using load and store operations. For load/store, when you load values into pred and curr (for example pred = set->head in the original code), that would be something like pred = std::atomic_load(&set->head). And you use an ::atomic_store where you mark the node as logically and physically deleted.

3. The next challenge is understanding that we needed to replace the "bit stealing" mechanism with an atomic boolean "mark" field in the Node struct. Therefore we implemented a custom "marking" function for marking nodes as logically deleted, and removed get_marked_ref() and get_unmarked_ref() from our implementation.

4. Another challenge was understanding why you need to mark curr->next (i.e. curr->next = get_marked_ref(curr->next)) in order to mark 'curr' as logically deleted in the original code? Our smart pointer implementation includes logic to mark 'curr' as logically deleted directly, rather than using this bit-stealing logic.

5. The last challenge was freeing ALL the memory. Initially, our smart pointers were only freeing the logically deleted nodes, rather than every node in the entire set. Once we solved that problem, our smart pointer implementation was freeing every node in the set, except one block. The issue was that we're using malloc() to allocate the set on the heap, but then using a single smart pointer reference to the set (i.e. std::shared_ptr<node_l_t> head). We added 'set->head.reset()' to test.c in order to release the owner of the pointer to the set, and then calling 'free(set)' to free the set itself.

6. Finally we implemented a memory-safe, thread-safe implementation of lazy-linked using smart pointers. Valgrind failed to detect memory leaks in our implementation, running the command "../../../bin/smart-pointer-lazy-list -t 8 -A" (-A enforced update ratio to be 20%). Stackoverflow indicates that one block in 'still-reachable' state does not constitute a memory leak:



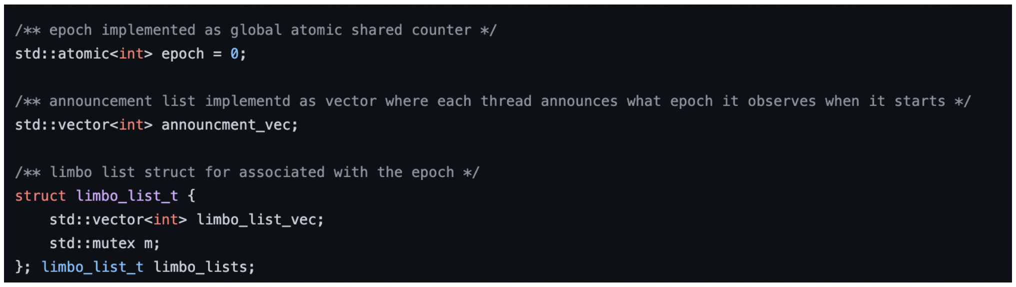
```
==31921== HEAP SUMMARY:
==31921==     in use at exit: 72,704 bytes in 1 blocks
==31921==   total heap usage: 39,237 allocs, 39,236 frees, 3,530,536 bytes allocated
==31921==
==31921== LEAK SUMMARY:
==31921==    definitely lost: 0 bytes in 0 blocks
==31921==    indirectly lost: 0 bytes in 0 blocks
==31921==      possibly lost: 0 bytes in 0 blocks
==31921==    still reachable: 72,704 bytes in 1 blocks
==31921==         suppressed: 0 bytes in 0 blocks
==31921== Rerun with --leak-check=full to see details of leaked memory
==31921==
==31921== For counts of detected and suppressed errors, rerun with: -v
==31921== Use --track-origins=yes to see where uninitialised values come from
==31921== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 8 from 6)
```

Figure 6: Valgrind for Smart Pointer

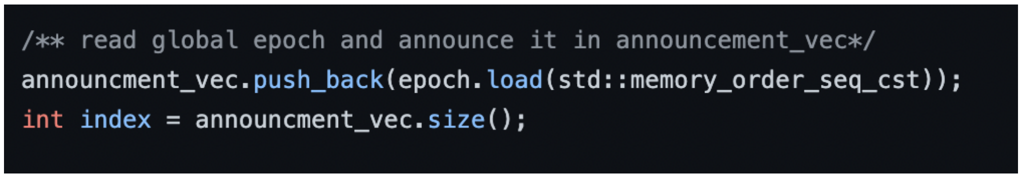
[2] The native approach to reclaiming memory involves unlinking all the nodes, and waiting for all concurrent threads to finish, and then reclaim memory. But this method is blocking...therefore EBR implements a smarter version of this process. **Epoch-Based Reclamation (EBR):** execution is divided into epochs, and each record removed from the data structure in epoch e is placed into a shared limbo bag for epoch e. Each time a process starts an operation, it reads and announces the current epoch, and checks the announcements of other processes. If all processes have announced the current epoch, then a new epoch begins, and the contents of the oldest limbo bag can be reclaimed. Implementing epoch-based reclamation involves understanding that safe reclamation happens at the END of the grace period, indicating that every thread had a quiescence state and won't reference the physically deleted nodes again, so it's safe to reclaim the memory.

At a high level, the process for safe reclamation involves: unlink(x) —> grace period —> reclaim(x). The steps are as follows for each operation by thread t: **1.** Read the global epoch and announce it in array[t] = e, and then announce it as idle with array[t] = -1 after the "transaction" (i.e. API function call). **2.** Retire the physically deleted node by adding it to the limbo list of epoch e. **3.** Reclaim all retired nodes in limbo list e-2 and atomically increment epoch e to e + 1.

In our *original* implementation, we implemented the structure above. Implemented the 'epoch' as a global atomic shared counter, an 'announcement list' implemented as a vector where each thread announces what epoch it observes, and 'limbo list' struct for the associated epoch. There were synchronization issues amongst the threads in our implementation, and we therefore decided to pivot to a new method. The screenshot below shows the data structures utilized in our original implementation:



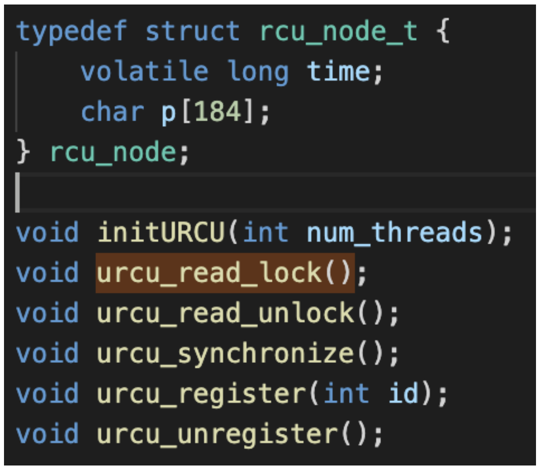
```
/** epoch implemented as global atomic shared counter */
std::atomic<int> epoch = 0;

/** announcement list implementd as vector where each thread announces what epoch it observes when it starts */
std::vector<int> announcment_vec;

/** limbo list struct for associated with the epoch */
struct limbo_list_t {
    std::vector<int> limbo_list_vec;
    std::mutex m;
}; limbo_list_t limbo_lists;
```



```
/** read global epoch and announce it in announcement_vec*/
announcment_vec.push_back(epoch.load(std::memory_order_seq_cst));
int index = announcment_vec.size();
```

Figure 7: Original EBR Implementation

In our *new* implementation, we decided to use bin/MUTEX-RCU-TREE as a reference for implementing EBR. bin/MUTEX-RCU-TREE uses EBR according to their README, but the executable has memory leaks, so we had to modify some of the code and inject it into our lazy-linked list. We reference the ‘urcu.c’ library in our implementation, with the following helper functions:

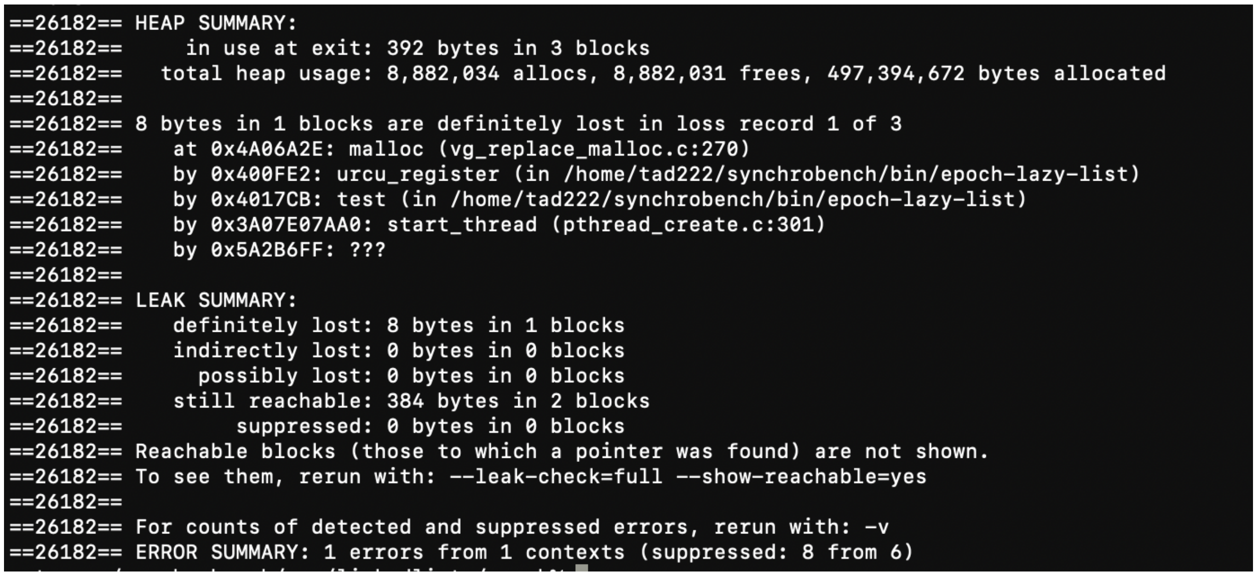


```
typedef struct rcu_node_t {
    volatile long time;
    char p[184];
} rcu_node;

void initURCU(int num_threads);
void urcu_read_lock();
void urcu_read_unlock();
void urcu_synchronize();
void urcu_register(int id);
void urcu_unregister();
```

Figure 8: URCU Library

lazy.c is not *modified* except by adding a rcu_read_lock() call inside the parse_insert(), parse_find(), and parse_delete() functions. rcu_read_lock() allows several read threads and one write thread, they read/write the same data, and data is copied for each thread. Inside of **test.c**, we use the ‘thread_data_t *d = (thread_data_t *)data’ object and declare the ID of the thread by invoking: ‘urcu_register(d->id)’. Then we initialize the threads with the initURCU() command, and at the end, deregister the thread IDs with ‘urcu_unregister()’. But most importantly, inside parse_delete() in lazy.c, we implement the ‘urcu_synchronize()’ function for synchronizing all of the threads on the same epoch. We ended up using this code since it’s on synchrobench, and therefore available for us to use in another data structure.

Running valgrind on a multi-threaded execution displayed *one* memory leak. The command executed is: ‘../../../bin/epoch-lazy-list -i 1024 -r 2048 -t 8 -u 20’.



```
==26182== HEAP SUMMARY:
==26182==     in use at exit: 392 bytes in 3 blocks
==26182==   total heap usage: 8,882,034 allocs, 8,882,031 frees, 497,394,672 bytes allocated
==26182==
==26182== 8 bytes in 1 blocks are definitely lost in loss record 1 of 3
==26182==    at 0x4A06A2E: malloc (vg_replace_malloc.c:270)
==26182==    by 0x400FE2: urcu_register (in /home/tad222/synchrobench/bin/epoch-lazy-list)
==26182==    by 0x4017CB: test (in /home/tad222/synchrobench/bin/epoch-lazy-list)
==26182==    by 0x3A07E07AA0: start_thread (pthread_create.c:301)
==26182==    by 0x5A2B6FF: ???
==26182==
==26182== LEAK SUMMARY:
==26182==    definitely lost: 8 bytes in 1 blocks
==26182==    indirectly lost: 0 bytes in 0 blocks
==26182==      possibly lost: 0 bytes in 0 blocks
==26182==    still reachable: 384 bytes in 2 blocks
==26182==         suppressed: 0 bytes in 0 blocks
==26182== Reachable blocks (those to which a pointer was found) are not shown.
==26182== To see them, rerun with: --leak-check=full --show-reachable=yes
==26182==
==26182== For counts of detected and suppressed errors, rerun with: -v
==26182== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 8 from 6)
```

Figure 9: Valgrind on EBR Implementation

[3] Comparing Smart-Pointers and EBR:

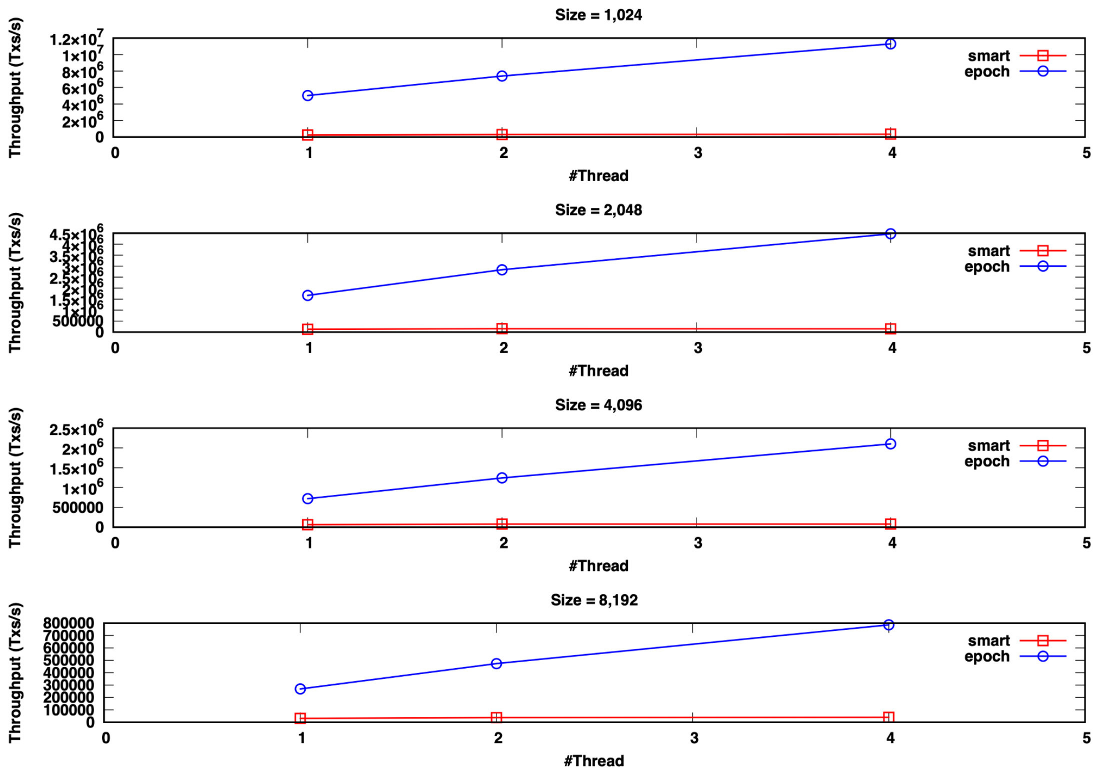


Figure 9: Comparing EBR and Smart Pointers

	1,024	**2,048**	**4,096**	**8,192**
Thread 1	1 leak (16 bytes)	1 leak (16 bytes)	1 leak (16 bytes)	1 leak (16 bytes)
Thread 2	2 leaks (32 bytes)	2 leaks (32 bytes)	2 leaks (32 bytes)	2 leaks (32 bytes)
Thread 4	4 leaks (128 bytes)	4 leaks (128 bytes)	4 leaks (128 bytes)	4 leaks (128 bytes)

Figure 10: Table for Memory Leaks of EBR Implementation

Overall, the transactional throughput for EBR was much *higher* than smart pointers since EBR does not use reference counting as compared with shared smart pointers. Our EBR implementation was interesting, in that every thread had *one* associated memory leak (i.e. 4 threads = 4 memory leaks).