CStore: A Lock-free SIMD Skip List

Ziyi Liu (ziyil), Quan Quan (qquan)

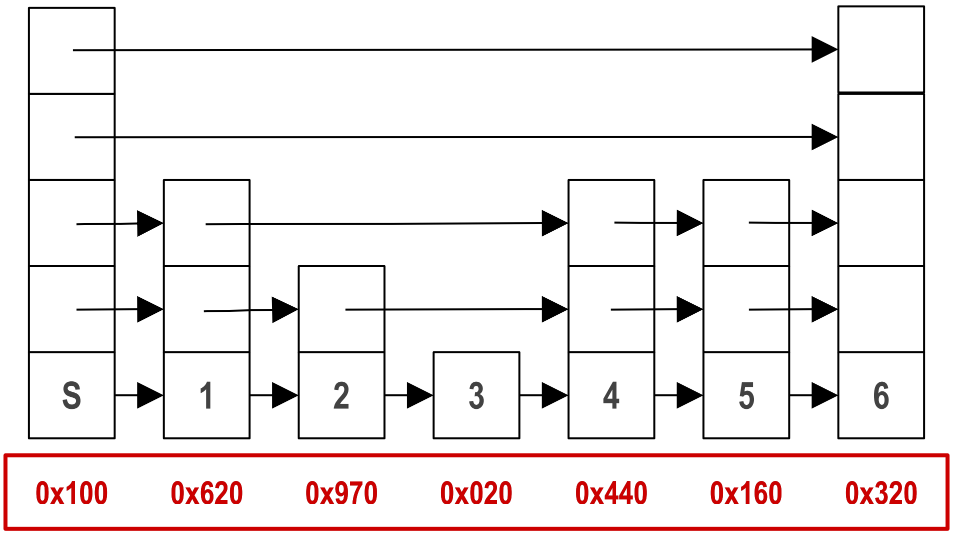
SUMMARY

In this project we implemented a lock-free SIMD skiplist which supports insert, delete and find. The project deliverable will be the library for this skiplist, and performance comparison with std::map and other lock-free non-SIMD skiplist on GHC machines.

BACKGROUND

According to Wikipedia, skiplist is a data structure that allows fast search within an ordered sequence of elements. On average it costs O(logN) to do searches.

However, since skiplist is an array of linked lists, and linked list has poor data locality (because it has to load memory again when going to a new node, as the example in the following graph shows), so it is actually slower than balance tree implementations like red-black tree.



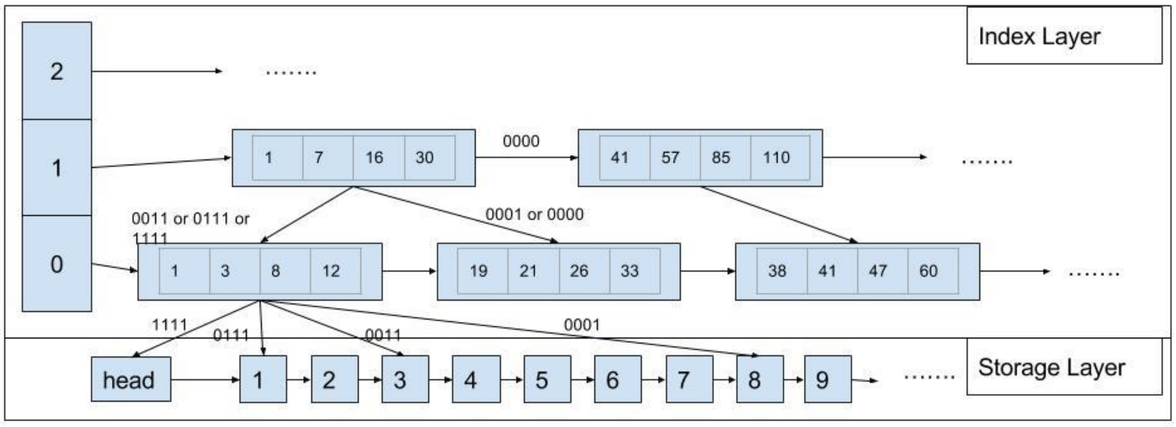
Graph 1. Skip List has poor data locality

One solution we apply in CStore is we group adjacent indexes together to improve data locality, and this also introduces the chance of applying SIMD to do comparisons and reduce the number of comparisons we have to do to search indexes horizontally.

Another case is we have to handle the multi-threading scenario. There has been some lock-free solutions for skiplist, but since we are changing the structure of skiplist, they cannot apply in our case. We make use of the lock-free linked list solution and come up with some adaptations to suit our design and maintain the lock-free attribute.

APPROACH

Overall structure design:



Graph 2. Overall structure of CStore

Above is a representation of the architecture of our skiplist. We divide the skiplist into index layer and storage layer. In the index layer, we group multiple indexes of a node together in one index node, so we can make use of cache locality and apply SIMD comparisons. The index layer will not change until some upper limit of the number of inserts and deletes are triggered, in which case we will build a new index layer to replace the old one. **Our assumption is the requests are uniformly distributed on the key space, so making the index layer unchanged for some time will not hurt performance much.** In the storage layer, we only have an ordered singly-linked lock-free skiplist.

Index layer design:

Each index node holds VECTOR_SIZE number of indexes. On receiving a query, each index node will launch a gang of instances, each instance comparing the key to an index and report the result in the corresponding slot in the output array. Then we use the pre-built routing-table to find where to go next. VECTOR_SIZE is a configurable parameter, on GHC machine we can use avx instruction set which can compile 8-width vector, so a VECTOR_SIZE greater or equal to 8 is preferable. But a large vector may lead to excessive computation, since in skip lists most of the time we suppose the target key is in the middle of the indexes. After doing test with various data size, a VECTOR_SIZE of 8 proves to have the best performance.

Node Count	vector_size=8	vector_size=16	vector_size=24
5,000,000	0.311	0.310	0.325
10,000,000	0.342	0.343	0.358
25,000,000	0.37	0.379	0.395
50,000,000	0.396	0.402	0.43
100,000,000	0.427	0.445	0.473

Table 1. Query time with different vector sizes
(thread number = 16, total requests = 32M, read-only test)

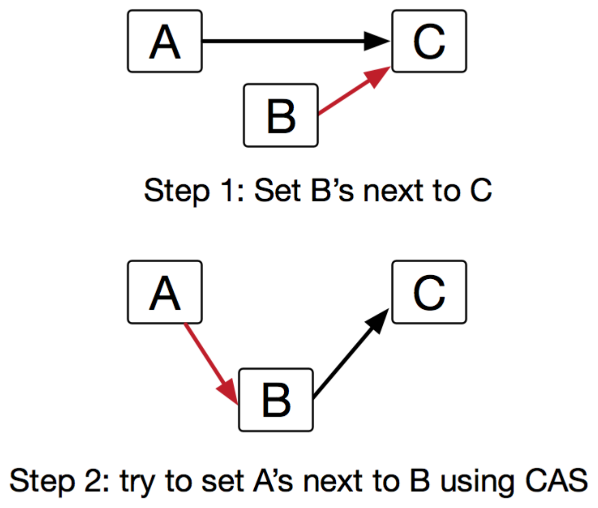
We choose to not change the index layer on inserts and deletes and only rebuild it at some interval because if we do change the index layer, we will not only have to do index movement frequently which takes time, and we will also have to come up with a lock-free/fine-locking performant index layer. This leads us to give up this plan and turn to our current solution.There is no parallelism in building the index layer, because anyway we have to go through the entire storage layer and the storage layer is a linked list which is hard to do data parallelization. We have a configurable parameter REBUILD_THRESHOLD to denote the maximum number of inserts and deletes before a rebuild of index layer. This parameter should be decided based on the number of nodes stored. In our testing, it is definitely not suggested to do frequent rebuilding, because going through the whole list to rebuild consumes a lot of resources. But scarcely rebuilding the index layer may also lead to slow lookups, so it is a tradeoff and this parameter needs to be carefully tuned. In our testing, based on a test setting of 500k intial nodes, 32 million requests with 60% inserts, 20% reads and 20% deletes, and 5 billion key space, 1/100, 1/5 and 2 times of the number of nodes all prove to be a good threshold.

rebuild threshold	time spent
1,000	7.5
2,500	7.5
5,000	7.5
7,500	7.6
10,000	7.4
20,000	8.1
1,000,000	7.2
2,000,000	8.1
5,000,000	9.1
∞	9.3

Table 2. Query time with different rebuild threshold

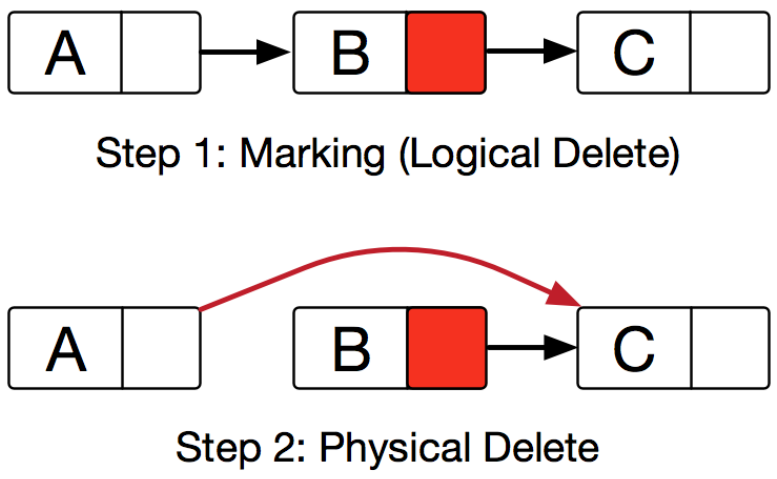
Storage layer design:

Our storage layer is an ordered linked list, so we first turn to lock-free linked list solution suggested by Harris[1]. The main idea is to utilize single-word compare-and-swap to implement lock-free insert, search, delete operations.
In insert operation implementation, we first find the right position we need to insert our node to. As shown in Figure 3, we are trying insert B between A and C. We set the node B’s next pointer to the node C. Then we try to set node A’s next pointer to node B using CAS. If CAS failed, we start from the very beginning, search the position to insert the node again.


A
C
B
Step 1: Set B’s next to C
A
C
B
Step 2: try to set A’s next to B using CAS

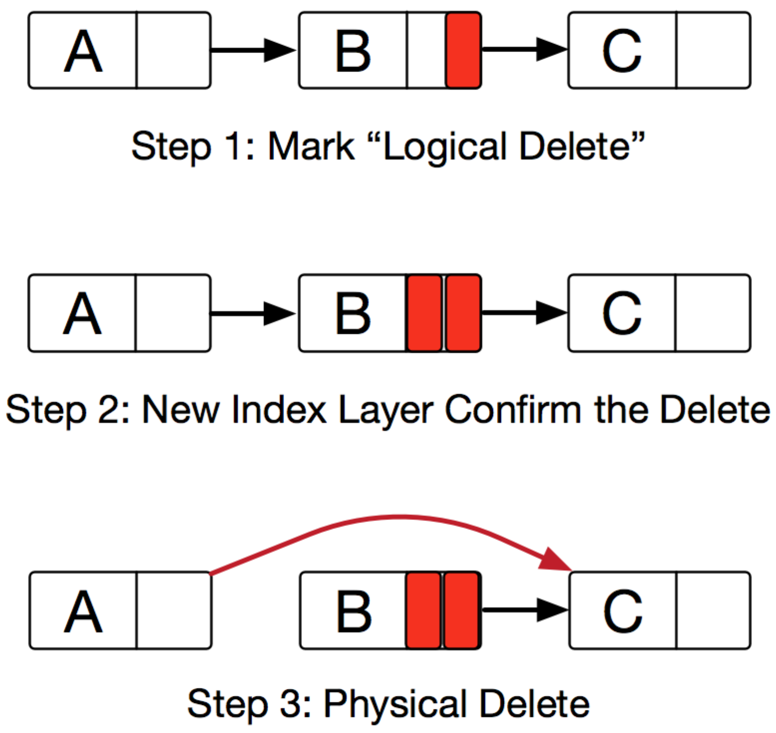
Graph 2. Lock-free Linked List Insertion Steps

Harris’s lock-free delete solution is utilizing a special mark bit at the last digit of next field. We use two separate CAS operations to perform a two-step deletion. The first step is used to mark the next field of the deleted node. By doing so, we logically delete the node. No more unsafe operation will perform to the logically deleted node. The second step is to physically delete the node.



Graph 2. Lock-free linked list delete steps

However, this two-step deletion is not enough in our situation. We have index layer which holds the pointers to the node in storage layer. We cannot physically delete a node without the “confirm” from index layer. To address this, we introduced an extra mark bit which we called the “confirm delete” mark to indicate that index layer confirmed the deletion of this node. This will only happen when we are rebuilding a new index layer. At that time, we will traverse the nodes in storage layer.To rebuild a new index layer, we skip the logically deleted nodes and marked them as confirm deleted. A node is safe to be physically deleted after it is marked as “confirm delete”.


A
B
C
Step 1: Mark “Logical Delete”
A
B
C
Step 2: New Index Layer Confirm the Delete
A
B
C
Step 3: Physical Delete

Graph 3. Lock-free CStore storage layer delete steps

RESULTS

In order to give a comprehensive testing of CStore, we test the read-only scenarios and mix-requests scenarios separately. We will measure CStore’s performance with time spent on the queries, and compare it against std::map (red-black tree) in read-only scenario and a non-SIMD lock-free skiplist implementation we found in this github repo[2] (author: Jung-Sang Ahn jungsang.ahn@gmail.com). In both scenario, we generate requests with 16 threads launched by pthread_create() for each data structure, and each thread will send 200,000 requests and that’s 3,200,000 in total. For each test run we measure the time spent as the time of the last thread exits minus the time of the first thread starts. The accepted time is the minimum time of three consecutive tests.

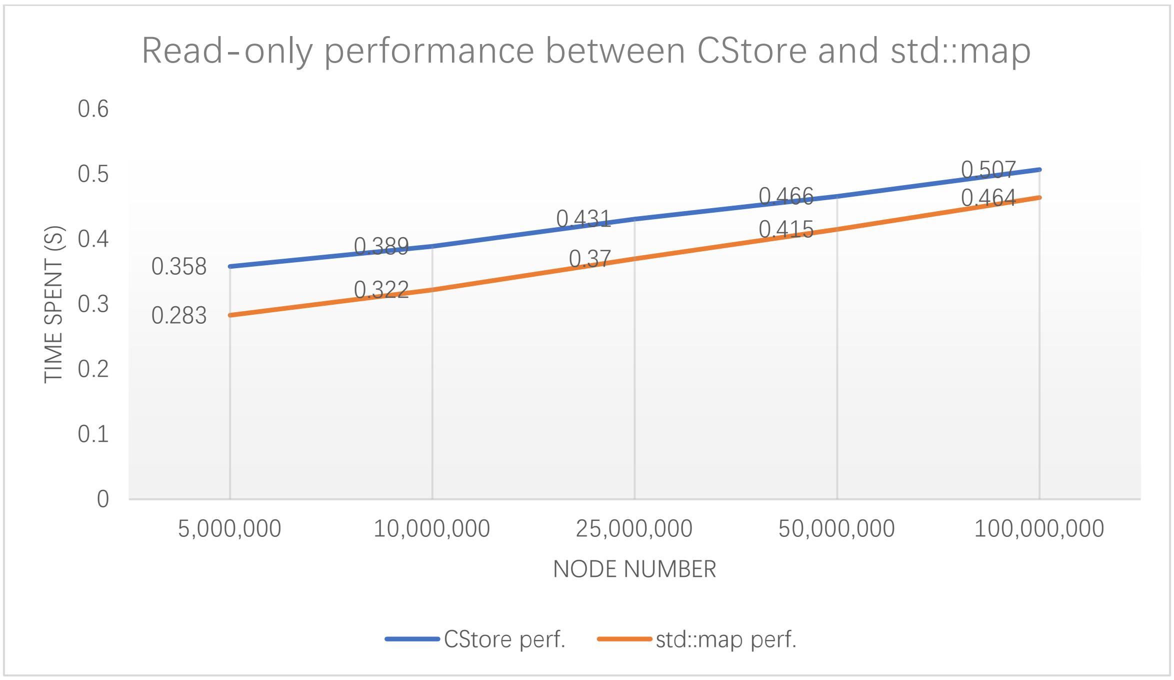
Read only test

First let’s look at the read-only scenario. As setup, we give CStore and std::map the same number of initial key-value nodes, and the initial key-value data for both data structures are the same. Each thread will send the exact same requests to both data structures.

Node Count	CStore perf.	std::map perf.
5,000,000	0.358	0.283
10,000,000	0.389	0.322
25,000,000	0.431	0.37
50,000,000	0.466	0.415

100,000,000	0.507	0.464

Table 3. Read-only performance comparison with std::map



Graph 4. Read-only performance comparison with std::map

We can see from the result that when the number of nodes is not so huge, std::map has notable better performance than CStore, however with the number of nodes growing, The performance difference shortens. This is because skip list is based on probabilistic model and with less nodes, it cannot hold O(logN) stably. Once the number of nodes are large enough, the performance would be more stable.

However, although std::map() performs well in read-only scenario, because it is very hard to implement fine-locking or lock-free for red-black tree, a read-write lock solution for std::map() is about 30~40 times slower than CStore. Since it has little meaning to make the detailed comparison, we will skip it here for simplicity.

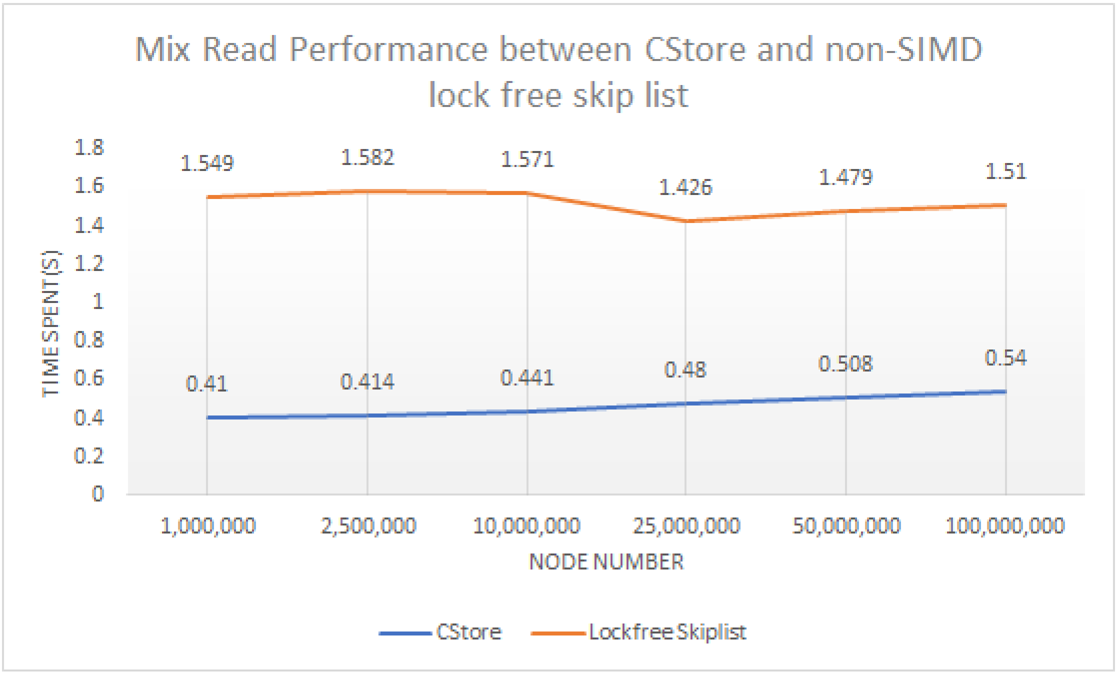
Mix test

Next we will look at CStore's performance against a non-SIMD lock free skip list. The mix requests consists of ~20% reads, ~60% inserts and ~20% deletes.

Initial Node Count	CStore	Lockfree Skiplist
1,000,000	0.41	1.549
2,500,000	0.414	1.582
10,000,000	0.441	1.571
25,000,000	0.48	1.426
50,000,000	0.508	1.479

100,000,000	0.54	1.51

Table 4. Mix request performance comparison with non-SIMD lock-free skiplist



Graph 5. Mix request performance comparison with non-SIMD lock-free skiplist

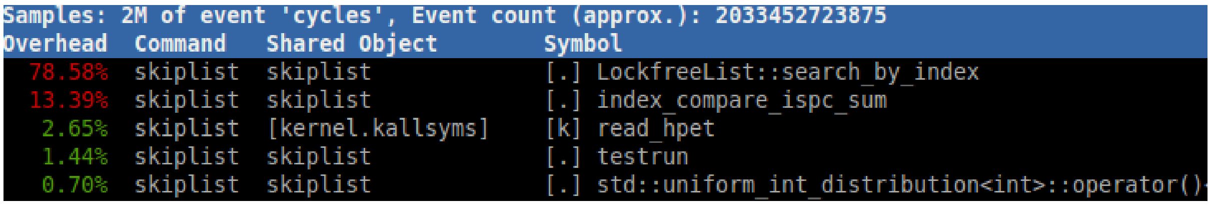
In this test our solution is about 3 times faster than the non-SIMD lock free skiplist solution. Although CStore is also a lock-free skip list solution, it has stronger data locality and requires less modification to the indexes, which makes CStore outperforms the other solution. Because of the fixed number of requests, with less nodes, the performance tends to be not consistent with that of many nodes because of high contention.

LIMITATION ANALYSIS

Because we are optmizing a linked-list based data structure, there are many techniques we learnt from this class cannot be applied here. Also, although combining indexes together improves data locality, because we are splitting the indexes for each data node into several levels of index nodes, we actually created more linked-list nodes, and introduced constant more memory access (each time going down one level, need to load the memory once more).

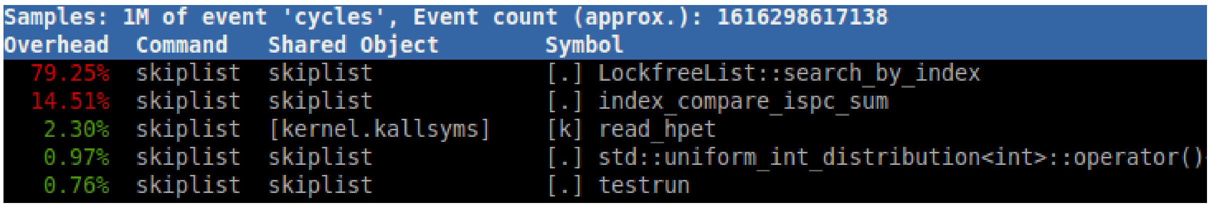
In our implementation, we have different scenarios of read-heavy, insert heavy or read-only. We use perf to profile our implementation.

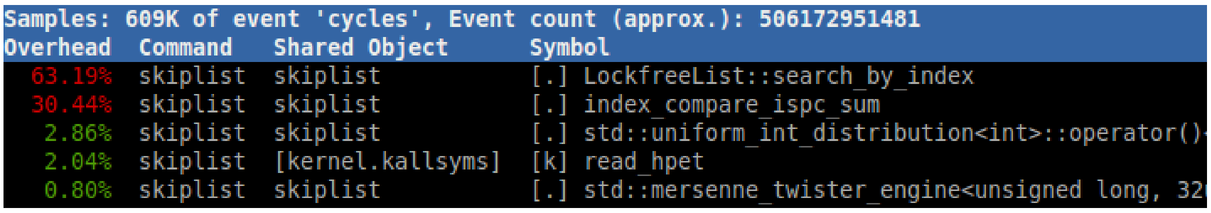
Insert 60%, read 20%, delete 20%



```
Samples: 2M of event 'cycles', Event count (approx.): 2033452723875
Overhead  Command   Shared Object      Symbol
  78.58%  skiplist  skiplist           [.] LockfreeList::search_by_index
  13.39%  skiplist  skiplist           [.] index_compare_ispc_sum
   2.65%  skiplist  [kernel.kallsyms]  [k] read_hpet
   1.44%  skiplist  skiplist           [.] testrun
   0.70%  skiplist  skiplist           [.] std::uniform_int_distribution<int>::operator()
```

Insert 20%, read 60%, delete 20%


```
Samples: 1M of event 'cycles', Event count (approx.): 1616298617138
Overhead  Command   Shared Object        Symbol
  79.25%  skiplist  skiplist             [.] LockfreeList::search_by_index
  14.51%  skiplist  skiplist             [.] index_compare_ispc_sum
   2.30%  skiplist  [kernel.kallsyms]    [k] read_hpet
   0.97%  skiplist  skiplist             [.] std::uniform_int_distribution<int>::operator()
   0.76%  skiplist  skiplist             [.] testrun
```
raed only


```
Samples: 609K of event 'cycles', Event count (approx.): 506172951481
Overhead  Command   Shared Object        Symbol
  63.19%  skiplist  skiplist             [.] LockfreeList::search_by_index
  30.44%  skiplist  skiplist             [.] index_compare_ispc_sum
   2.86%  skiplist  skiplist             [.] std::uniform_int_distribution<int>::operator()
   2.04%  skiplist  [kernel.kallsyms]    [k] read_hpet
   0.80%  skiplist  skiplist             [.] std::mersenne_twister_engine<unsigned long, 32
```
As we can see from the results, the main bottleneck is in the storage layer lock-free searching part, especially in insert heavy scenarios when conflict and backoffs are more common. This is hard to optimize since we already have an optimal lock-free solution. In read-only situation, the ispc SIMD compare takes a good amount of time. This is because each read/insert/delete operation has to go through the index layer and in each index node we have to run ispc SIMD compare to route.

Another problem is it seems SIMD instructions actually provides little speedup. The reason is if having proper max height and reasonable index height for each data node, the number of times to access index node for a lookup is small. For example, in our test above we generally have 10 million nodes. In this case a height of 23 (because 10 million is around 2^23) is sufficient for efficient lookup. Then, based on the probablistic model, we merely need to traverse through the index layer horizontally and only need to access 23 index nodes to get to the storage layer. Comparing to the memory loading time, the speedup brought by reducing 4 or 5 cycles on comparison is minor.

Citations

[1] Harris, Timothy L. "A pragmatic implementation of non-blocking linked-lists." *International Symposium on Distributed Computing*. Springer Berlin Heidelberg, 2001.
[2] https://github.com/greensky00/skiplist

LIST OF WORK BY EACH STUDENT

Equal work was performed by both project members.