Flat combined red black trees

CS5510 Multiprocessor Programming
Sergio Sainz

Problem

- Red black trees are hard to parallelize because the *balancing* operation requires blocking entire areas of tree (some cases area increases).

- FLAT COMBINING: new paradigm ONE thread does EVERYTHING.

- Makes sense as long as an operation taking O(n) time takes less than O(n*k) when k operation are done together. Red black trees do not match this pre-requisite.

Scope

- Only red black trees. Not AVL or B-Trees. INSERT(10%), DELETE(10%), GET(80%)
- Could not find lock - free red black tree. Some papers did not describe all the steps of building the tree:
  - *Lock-Free Red-Black Trees Using CAS :* (Kim, Cameron, Graham): The method to combine several moveUpStruct is not provided.
- Thanks to Arun, found Herlihy's red black trees as benchmark:
-

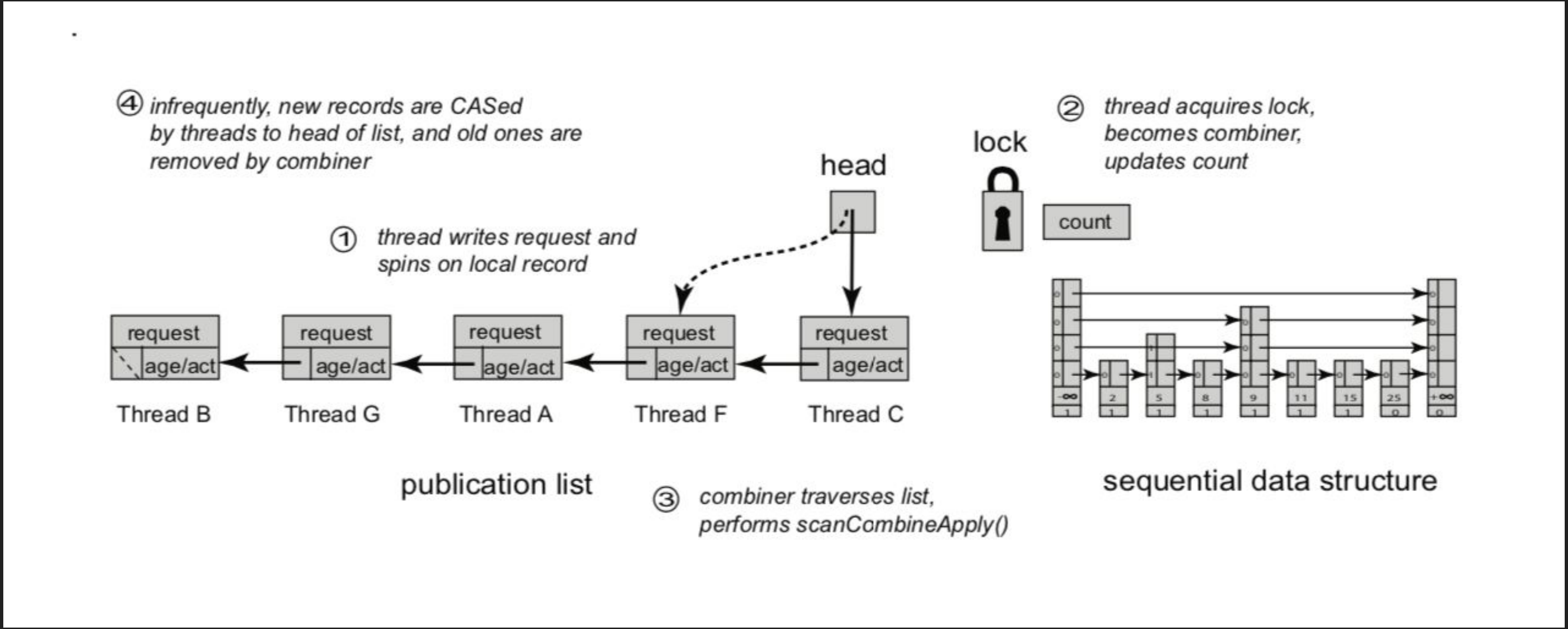
RBTCompositional	RBTTransactional

https://github.com/gramoli/synchrobench/blob/master/java/src/trees/transactional/CompositionalRBTreeIntSet.java

https://github.com/gramoli/synchrobench/blob/master/java/src/trees/transactional/TransactionalRBTreeSet.java

Relation to literature

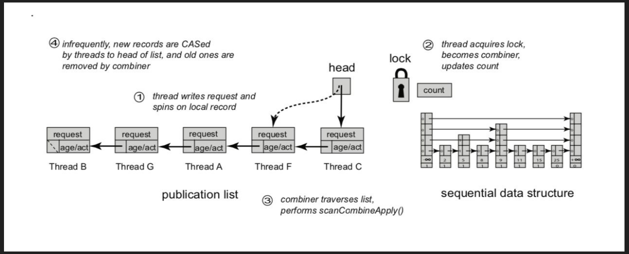
Flat Combining and the Synchronization-Parallelism Tradeoff:


④ infrequently, new records are CASed by threads to head of list, and old ones are removed by combiner
lock
② thread acquires lock, becomes combiner, updates count
head
count
① thread writes request and spins on local record
request
age/act
Thread B
Thread G
Thread A
Thread F
Thread C
-∞
2
5
8
9
11
15
25
+∞
publication list
③ combiner traverses list, performs scanCombineApply()
sequential data structure

Relation to literature

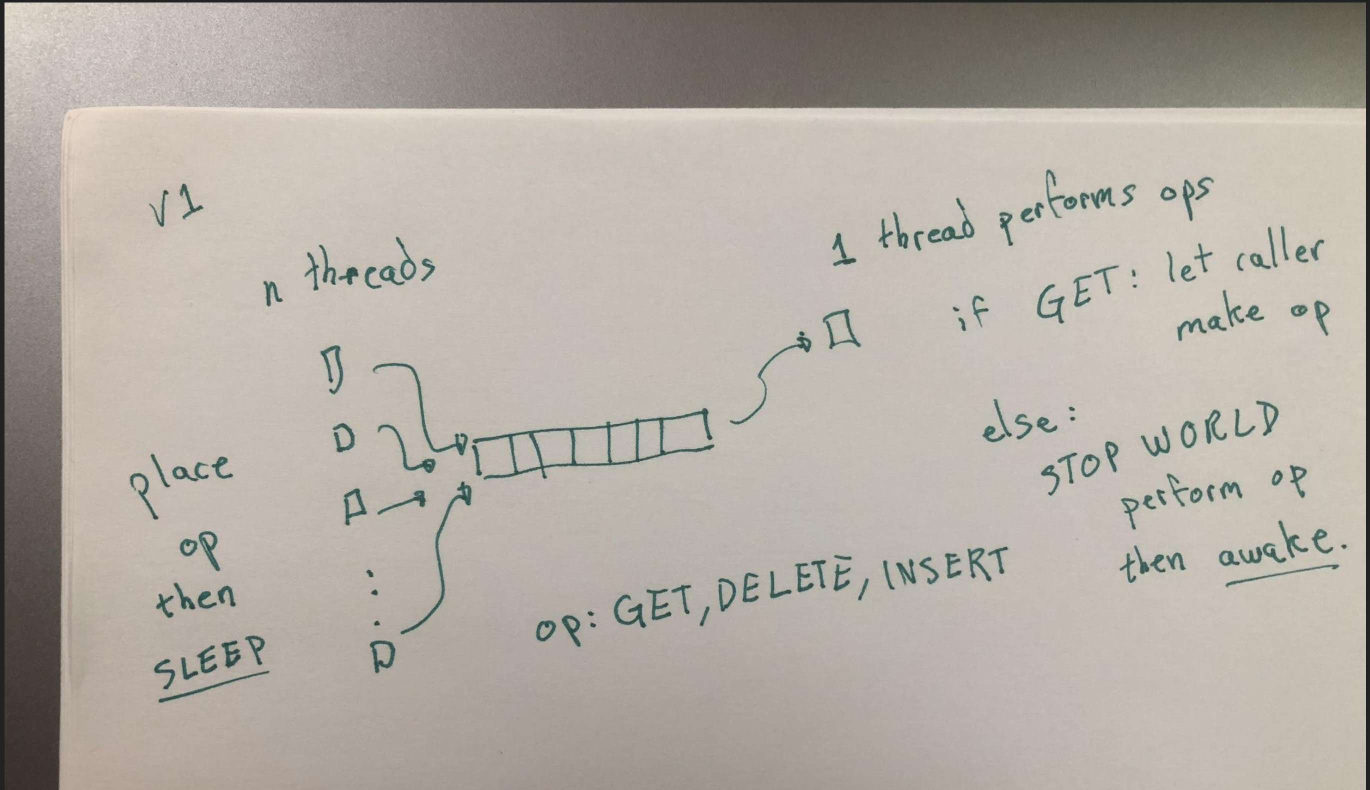
Flat Combining and the Synchronization-Parallelism Tradeoff:

- *Do not use multiple threads as combiner: only one thread is the combiner always. In literature, multiple threads as combiner.*
- *Not all work is done by combiner.*


④ infrequently, new records are CASed by threads to head of list, and old ones are removed by combiner
head
lock
② thread acquires lock, becomes combiner, updates count
count
① thread writes request and spins on local record
request
age/act
Thread B
Thread G
Thread A
Thread F
Thread C
publication list
③ combiner traverses list, performs scanCombineApply()
sequential data structure

Solutions: V1, V2, V3, V4

- V1:



Solutions: V1, V2, V3, V4

- V2: Same as V1 but GET operations will not sleep, GET will SPIN.
  - Use AtomicInteger to know whether all GET threads have finished processing when STOP WORLD operations appear.

- V3: Same as V1 but GET operations will not sleep, GET will SPIN. And DELETE and INSERT use backoff lock using manual delays (1, 3, 10, 20, 50, 100, 200, 500, 1000, 3033, 5000).
  - V3 performs too poorly to measure.

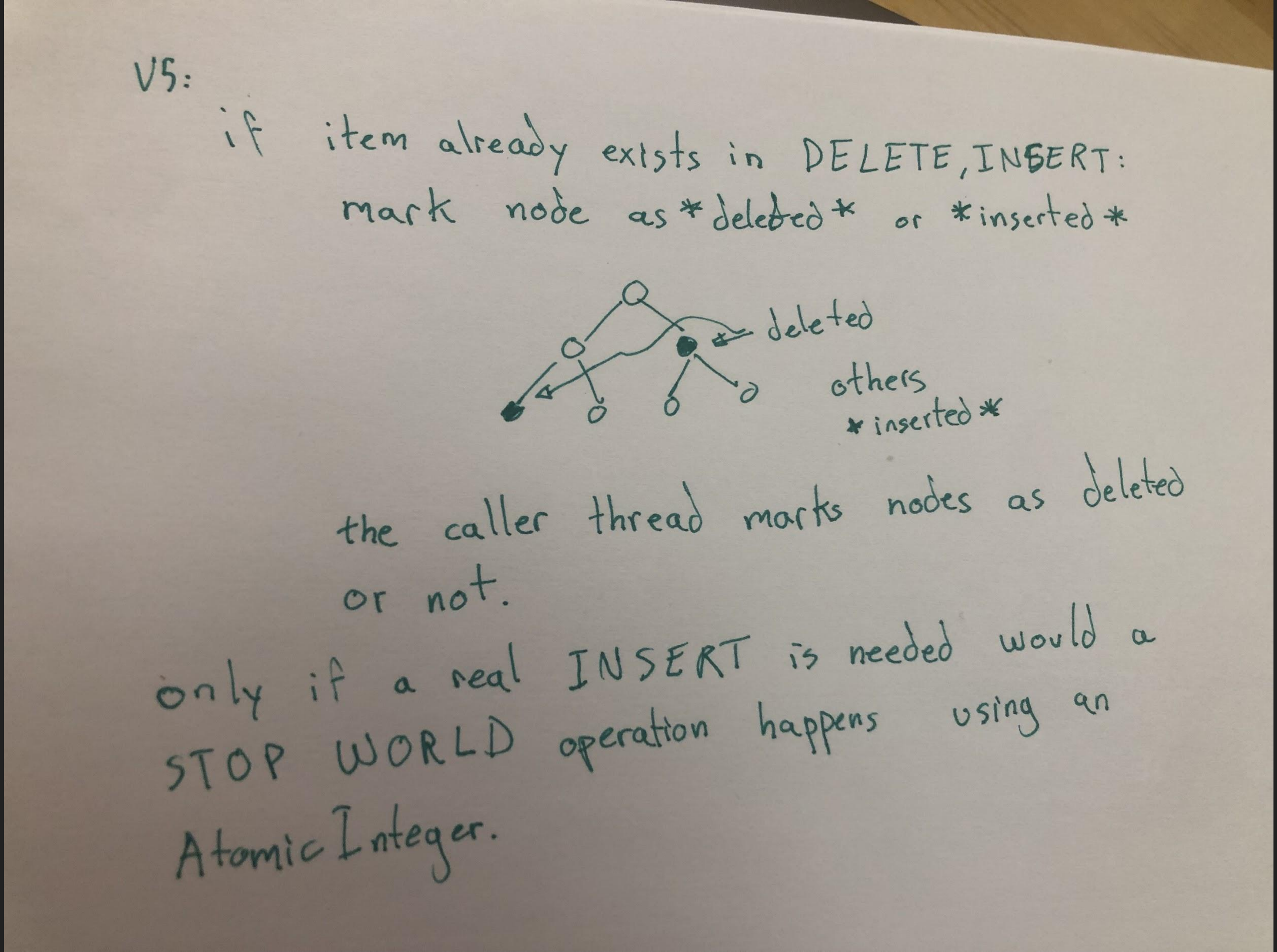
- V4: Uses SPIN for all three operations: GET, DELETE, INSERT

V5:

Soft delete

&

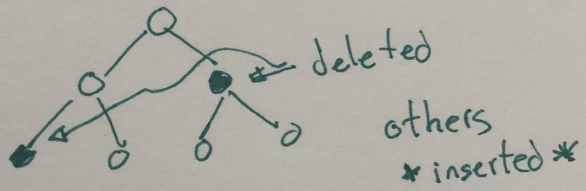
soft insert



V5:

if item already exists in DELETE, INSERT:

mark node as *deleted* or *inserted*



the caller thread marks nodes as deleted or not.

only if a real INSERT is needed would a STOP WORLD operation happens using an AtomicInteger.

Solutions: V6

- Same as V5, but STOP WORLD synchronization happens in ThreadLocal variables.

Implementation - V6: GET

Combiner	Caller

Nothing	`if(threadLocal.stopWorld.get()){`
	    `threadLocal.stopped.set(true);`
	    `threadLocal.stopWorld.wait()`
	`}`
	
	`Value v = actualTree.get();`

Implementation - V6: DELETE

Combiner	Caller
`if(actualTree.exists(key)){`	`if(threadLocal.stopWorld.get()){`
`    caller.softOperation = true;`	`    threadLocal.stopped.set(true);`
`    caller.waitingInBlockingQueue.set(false);`	`    threadLocal.stopWorld.wait()`
`    continue; // below lines do not continue.`	`}`
`}`	
	`queue.enq(new Operation(DELETE, key));`
`caller.waitingInBlockingQueue.set(false);`	`while(threadLocal.waitingInBlockingQueue.`
	`get());`
	`if(threadLocal.softOperation){`
	`    actualTree.softDelete(key);`
	`}`
	`// if key is not in tree no need to delete it.`

Implementation - V6: INSERT

Combiner	Caller
if(actualTree.exists(key)){	if(threadLocal.stopWorld.get()){
caller.softOperation = true;	threadLocal.stopped.set(true);
caller.waitingInBlockingQueue.set(false);	threadLocal.stopWorld.wait();
continue; // below lines do not continue.	}
}	
	queue.enq(new Operation(INSERT, key, value));
// STOP WORLD and INSERT	
Continue next page.	while(threadLocal.waitingInBlockingQueue.get());
	if(threadLocal.softOperation){
	actualTree.softInsert(key, value);
	}
	// if key is not in tree combiner will add it.

Implementation - V6: INSERT

Combiner

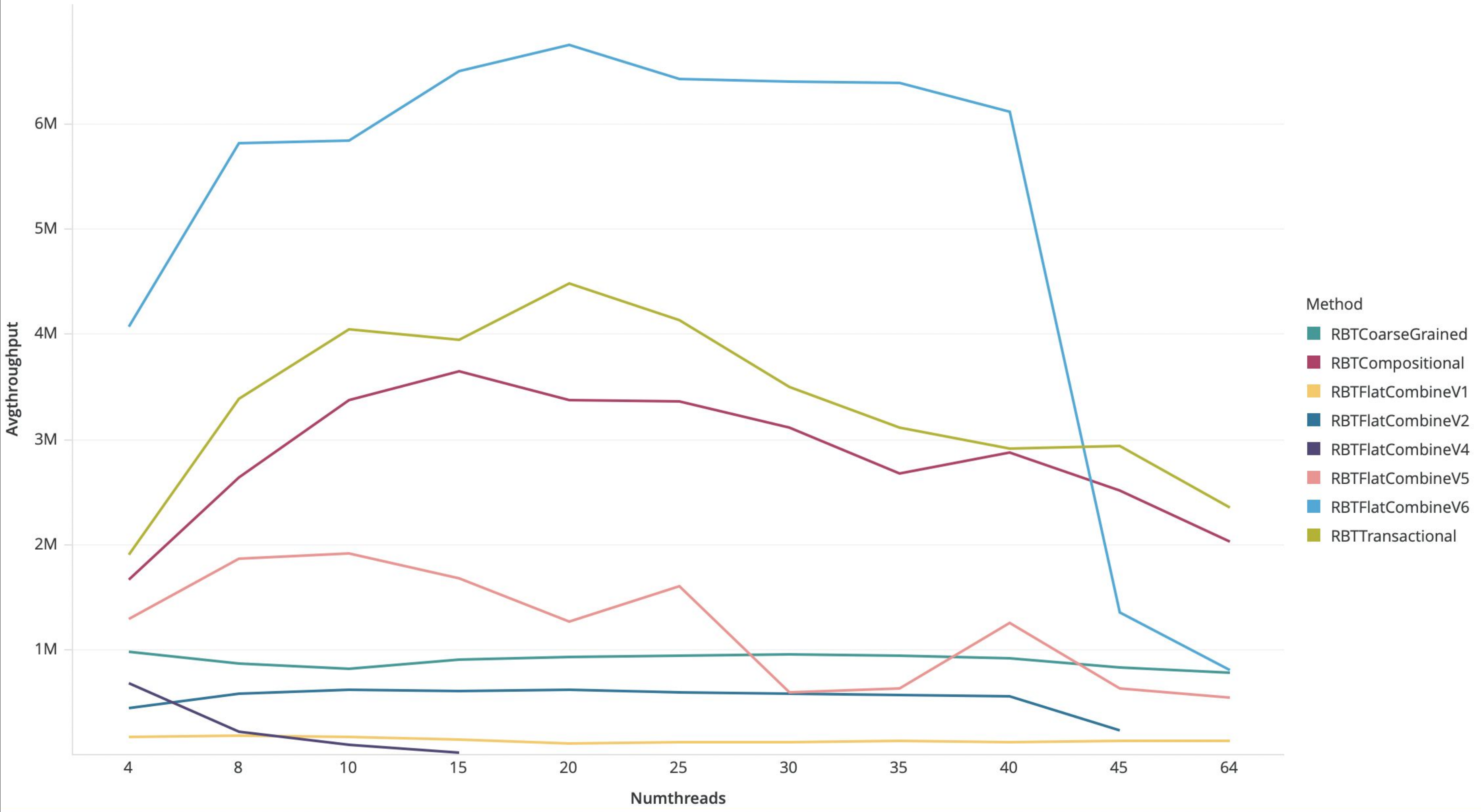
```
// STOP WORLD
for(each threadLocal ){
   threadLocal.stopWorld.set(true);
}

while(! allStopped){ // Wait for all threads to be either waiting in queue or stopped.
   allStopped = true;
   for (each threadLocal){
       if( threadLocal.stopped || threadLocal.waitingInBlockingQueue){
             allStopped &= true;
        }else{
             allStopped &= false;
        }
   }
}
// continue next page
```

Implementation - V6: INSERT

Combiner

```
// remove stop world flag
for(each threadLocal ){
   threadLocal.stopWorld.set(false);
}
if(tree.size() > MAX_SIZE){  // Check whether actual tree size is larger than max limit
    List<Node> deletedNodes = actualTree.getDeletedNodes();
    for (each deletedNode){
         actualTree.delete(key); // actual delete
    }
}
actualTree.insert(key, value); // actual insert….
for(each threadLocal ){
   if(threadLocal.stopped) threadLocal.stopWorld.notifyAll(); // Awake stopped
threads.
}
// END
```

Flat combine vs Transactional / Coarse grained and Compositional RBT (10% adds / 10% put / 80% get)



Limitations

- Needs initialize step for all threads to add their threadLocal variables to the list (for alter use by combiner).
- One thread is waiting always as combiner (in case there are no callers).
- Depends on the MAX_SIZE limit… If too small then it will be called often reverting back to V4.

Future Ideas:

- We see from results that for threads > 45 performance degradates greatly. Consider dynamically sleep threads when waiting time is beyond threshold.

Conclusions

- Flat combining useful even for data structures that naively cannot “batch” operations. Although cost is increase memory.
- Spinning in ThreadLocal and signaling in Atomic variables in ThreadLocal really made a difference between version 5 and version 6. Best idea :)
- Linearization points for soft delete and soft insert are when the node’s isDeleted flag is set/unset.
- No easy accessible lock-free red black tree.