Unexpected Scaling in Path Copying Trees

Ilya Kokorin
ITMO University, Russia
kokorin.ilya.1998@gmail.com
Trevor Brown
University of Waterloo, Canada
trevor.brown@uwaterloo.ca
Vitaly Aksenov
ITMO University, Russia
aksenov.vitaly@gmail.com
Alexander Fedorov
ISTA, Austria
afedorov2602@gmail.com

Motivation

How to make a data structure concurrent?

- Use techniques designed specifically for that data structure (e.g., hand-over-hand locking, descriptors)
- Use coarse grained techniques (e.g., Universal Construction, global lock)
- Use a "functional" approach, such as path-copying

Simple synchronization for path-copying trees:

- Maintain a pointer to the current version of the tree
- Read-only operation: fetch current version, execute the sequential operation on it
- Update operation: fetch current version, copy entire path from the root to the target node, replace the current version using *CAS*.

Can this approach scale on write-dominant workloads?

Workload description

Performance evaluated using two write-only workloads:

- `P` processes concurrently insert and then remove disjoint batches of elements to a single search tree
- `P` processes concurrently insert or remove random elements from a search tree

Theoretical result

Some scaling of updates is possible, in theory, thanks to processor caches. Failed update attempts can benefit future successful attempts. Some values read during the first attempt remain in the cache during subsequent re-tries.

Proposed a simple model for these caching effects.
Result: path-copying trees of size N can have $\Omega(\log N)$ scalability for write-only workloads with many processes.

Implementing a concurrent path-copying tree

```
struct Tree<T> {
  TreeNode<T>* Root_Ptr
}

fun <T> Find(Tree<T> tree, T key):
  // read linearization point
  root := tree.Root_Ptr
  return Find_Sequential(root, key)
```

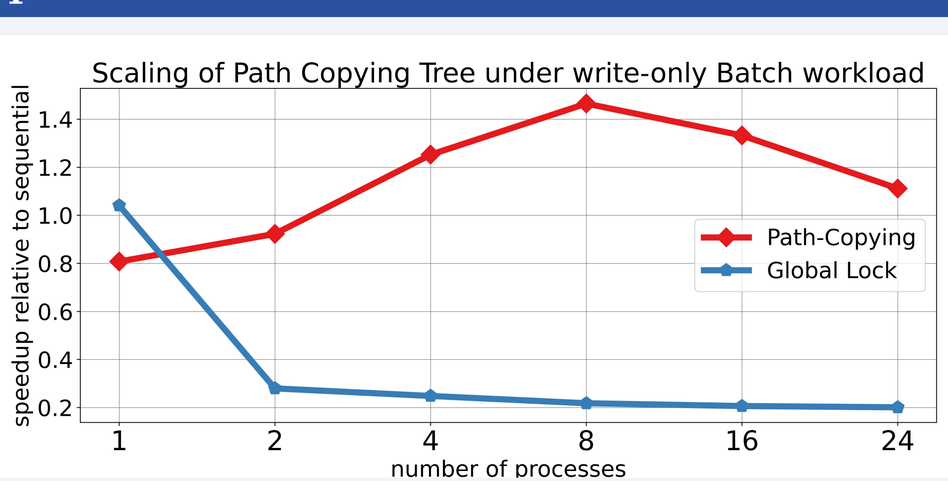
```
fun <T> Insert(Tree<T> tree, T key):
  while true:
    root := tree.Root_Ptr
    new_root := Persistent_Insert(root, key)
    if CAS(&tree.Root_Ptr, root, new_root):
      // insert linearization point
      return
    // else retry
```

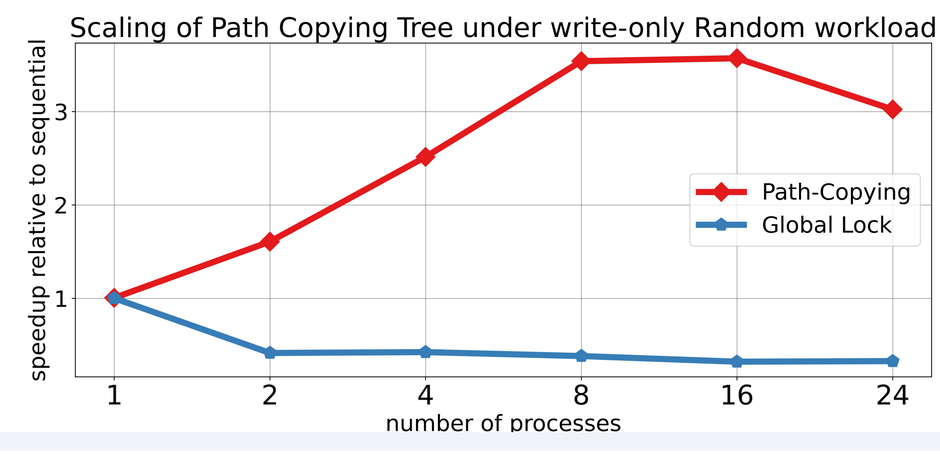
Performance measurements

Experiments on a path-copying treap confirmed our theoretical results. Surprisingly, we do see (limited) scaling on write-only workloads.

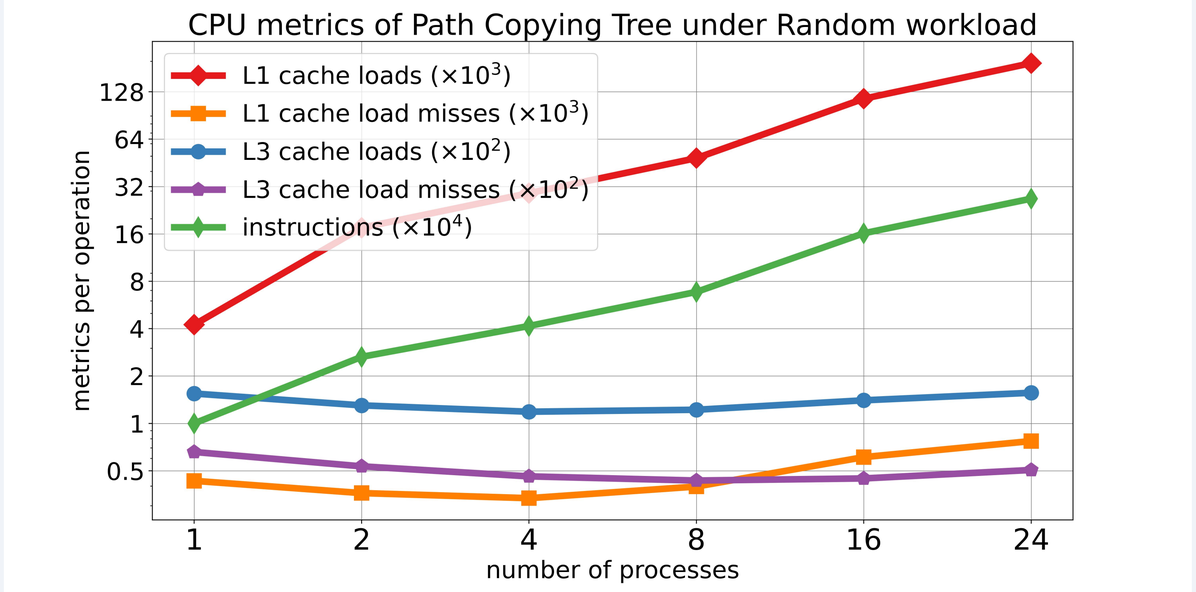
Batch workload: up to **1.88x** single threaded performance.
Random workload: up to **3.55x** single threaded performance.
Outperforms sequential tree by **1.47x** and **3.54x**, resp.

Experimental results





Experimental metrics



As the number of processes increases:
L1 cache loads grow substantially, but L1 misses do not. Shared L3 cache's behaviour is relatively stable. Operation retries increase loads and instructions, but do not increase cache misses meaningfully.

Remarks

- On workloads with read operations, scaling is even better. Read operations are mostly uncontended.
- Similar results are expected for other path-copying trees: e.g., AVL trees, Splay trees, AA trees, Red-Black trees, B-trees