Advanced Parallel Computing: Exercise #7

Due on Tuesday, June 9, 2015

Svend Dorkenwald, Günther Schindler

Reading

Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory

McKenney et. al discuss the strength and weaknesses of locking and transactional memory schemes. They analyse those and give suggestions for further improvement.
Their work shows that locking solutions might not be perfectly suited, but since it is widely used many workarounds were developed for certain problems. Transactional memory on the other side might look like very elegant at first glance, but leads to many ugly problems. Moreover, a large community working on these problems is missing. The authors do not doom TM to the bin, but argue that it need to focus on applications which preexisting machanisms poorly serve to grow a support.
Surprisingly, this paper might be mostly correct today showing how little progress was reached in the last two decades. Both schemes might have improved since then, but few of their main problems are overcome - even their supposed remaining challenges for locking are still not solved today. We give this paper a strong accept.

Stretching transactional memory

Dragojevic, Guerraoui and Kapalka present their Software Transactional Memory implementation (Swiss TM). To proof that Swiss TM is superior to other implementations they compare their performances in different benchmarks.
The key asset of their implementation is their two-phase contention manager. Unlike other STMs they use eager and lazy strategy. SwissTM detects write/write conflicts eagerly and read/write conflicts lazily. This makes it effective for mixed workloads containing non-uniform, dnyamic data structures and various transaction sizes.
We like the way the authors try to improve existing schemes by combining different strategies to use them at what they are best. However, it remains unclear how their implementation stacks up against HTM or even current Mutex implementations. If STM should be the standard one day they have to compete with currently used systems, otherwise their work is only meant to be recognized by a small community. Additionaly, they name problems like that their implementation is not privatization-safe which "did not affect us (they), as none of the benchmarks we (they) use requires privatization-safe ST".
Nevertheless, we give this paper a weak accept.

Red-Black Tree – Implementation of sequential version

Our red-black tree will implement an associative array, so we will store both the key and its associated value.

```
typedef struct rbtree_node_t {
    void* key;
    void* value;
    struct rbtree_node_t* left;
    struct rbtree_node_t* right;
    struct rbtree_node_t* parent;
    enum rbtree_node_color color;
} *rbtree_node;
```

Each node also stores its color, either red or black, using an enumeration.

We will at all times enforce the following five properties, which provide a theoretical guarantee that the tree remains balanced.

- Each node is either red or black
- The root node is black
- All leaves are black and contain no data
- Every red node has two children, and both are black
- All paths from any given node to its leaf nodes contain the same number of black nodes

We will have a helper function

```
void verify_properties(rbtree t)
```

that asserts all five properties in a debug build, to help verify the correctness of our implementation.

An empty tree is represented by a tree with a NULL root. The function

```
rbtree rbtree_create()
```

provides a starting point for other operations.

Searching for values is straightforward, by finding the node and extracting the data if lookup succeeded.

```
void* rbtree_lookup(rbtree t, void* key, compare_func compare);
```

We return NULL if the key was not found.

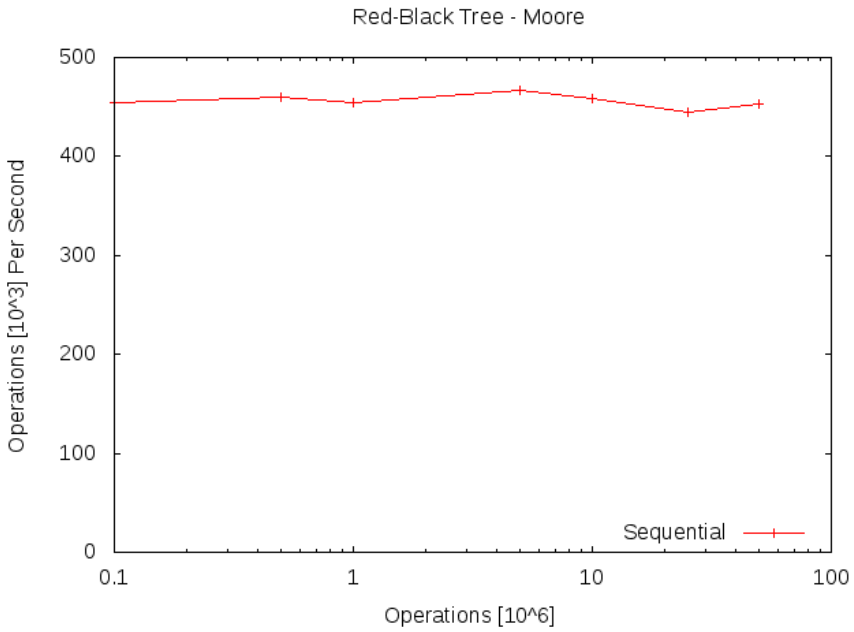
When inserting a new value, we first insert it into the tree as we would into an ordinary binary search tree. If the key already exists, we just replace the value.

```
void rbtree_insert(rbtree t, void* key, void* value, compare_func compare)
```

Otherwise, we find the place in the tree where the new pair belongs, then attach a newly created red node containing the value.

Red-Black Tree – Sequential Performance

Since the searching and inserting time is $\mathcal{O}(log(n))$, where n is the total number of elements (constant 10M) in the tree, we expect constant OPS for different lengths of the update stream.



As the graphic shows, we actually get a almost constant number of OPS for the sequential Red-Black Tree.