Fractal Tree Implementation with Intel Haswell Hardware Supported Transactional Memory

Zhigang Zhong
Comptuer Science Department, Brown University
115 Waterman St. Providence, RI

Introduction

Fractal Tree [1] is a tree data structure proposed by Tokutek as a new type of database index. Compare to widely used index data structure B-Tree, it has better insertion performance by amortizing the insertion cost while keeping the good performance of point queries and range queries. According to Tokutek, it shows great performance in SSDs as well. On the order hand, Intel released its Haswell architecture CPU with hardware transactional memory support. Transactional memory can make the development of concurrent programs easier than using complicated logic of locks or other lock-free synchronization algorithms. However software implementation of transaction memory is usually slow. Built-in hardware transactional memory support aims to speed up transactions in concurrent memory access. We are interested in building an in-memory version fractal tree which takes advantage of the hardware supported transactional memory and than benchmark its performance. According to the experiments we performed on out implementation of fractal tree. Hardware supported transactional memory on this version of Intel Haswell CPU has similar performance as std::mutex which uses Pthreads library under the cover. Transactions that touch a large size of data at high concurrency level are more likely to fail than transactions that touch only a small size of data at low concurrency level.

Implementations

The fractal tree data structure is implemented in C++. The source code can be downloaded at https://github.com/alfredzhong/bft. It requires gcc 4.8 to compile and Intel Haswell architecture CPU to run the code. STL is used in the implementation to support template so that it can be used for multiple data type. The interface looks like B-Tree, it supports insertion and query of key-value pairs. A fallback mutex is used for synchronization if a transaction fails because Intel's hardware transaction is just best effort without guarantee success of the transaction. We use std::mutex in the newest version of C++ which uses Pthread library under the cover.

Experiments and Results

To test the performance of the fractal tree implementation with hardware transactional memory on and off, we set up experiments as following. At the beginning, an object of fractal tree is created on heap so that it is a globally shard object and can be access by any thread. And then a user-specified number of threads are created. They all perform the same task. Insert a user-specified number of normalized unit size of data into the fractal tree. The user can also specify to turn hardware transactional memory on or off during the experiments. If transactional memory support is turned on, each insertion into the fractal tree will be executed as a transaction with best efforts. If for any reason a transaction fails, a fallback mutex will be used for correct synchronization within the concurrent execution environment. The process repeates 1000 times to collect a stable measurement. Fig. 1 shows how data size and concurrent level affects the transaction success rate. As the red line shows, when the data size is small, in this case, 1 normalized size unit, the transactions are likely to succeed although the transaction success drops a little as the concurrency increases. On the other hand, as the blue line shows, when a transaction need

to touch a relatively large size of data, in this case, 20 normalized data size unit, the transaction success rate drop very fast as the concurrency level increases.

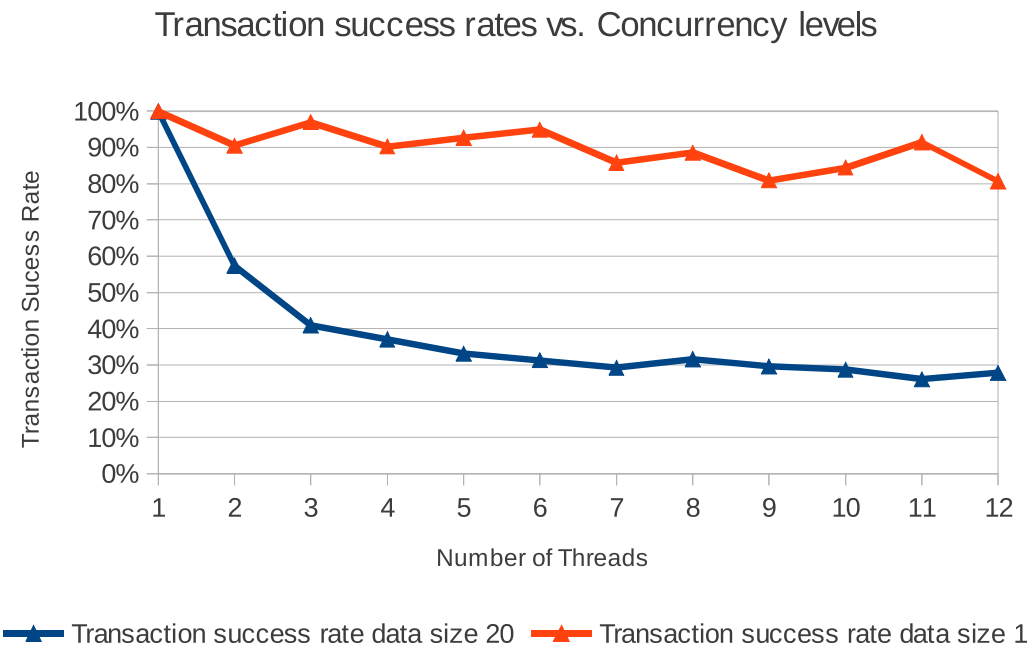

Figure 1: Transaction success rates vs. Concurrency levels

For example, when 12 threads all try to insert 20 normalized unit of data to the fractal tree, only less than 30% transactions succeed. Therefor, more than 70% insertions use the fall back mutex. The is as expected. When a transaction is too big, it has higher change to fail in higher concurrency level. Notice that, however, when there is only one thread executes the insertion, the transaction success rate is closed to 100% for both small and relatively large data size.

Next, let us look at the execution time. As Fig. 2 shows, the total execution time is generally proportional to the total size of data to be inserted. Both synchronization scheme, transactional memory and mutex show similar scalability trends. For data size of 20 normalized units, using mutex only performs better than using hardware transaction with fall back mutex. This may due to low transaction success rate with this data size and the extra overhead of falling back to use mutex. A more interesting topic to explore is if hardware transaction itself is faster than mutex. When the data size is small, say, 1 normalized data unit, it has higher transaction success rate. The comparison to execution with mutex with small data size can be used as measurement of the relative speeds of the two.

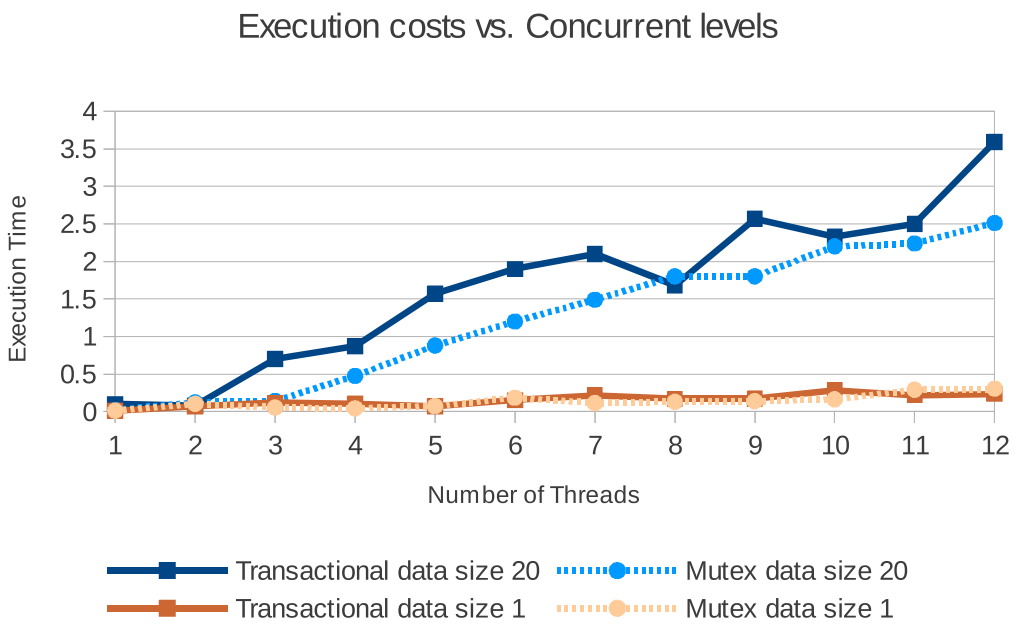

Figure 2: Transaction success rates vs. Concurrency levels

Fig. 3 is a zoomed in view of execution time vs. concurrency levels with data size 1. The line curves cross each other multiple times which indicates the speed transaction memory is comparable, if not worse, to the that of using mutex only. Although the data size is small, insertion to a fractal tree is still a relative complicated process. To eliminate other potential factors that may affect the measurement, we perform another experiment targeting in compare the speeds of transaction memory and mutex directly.

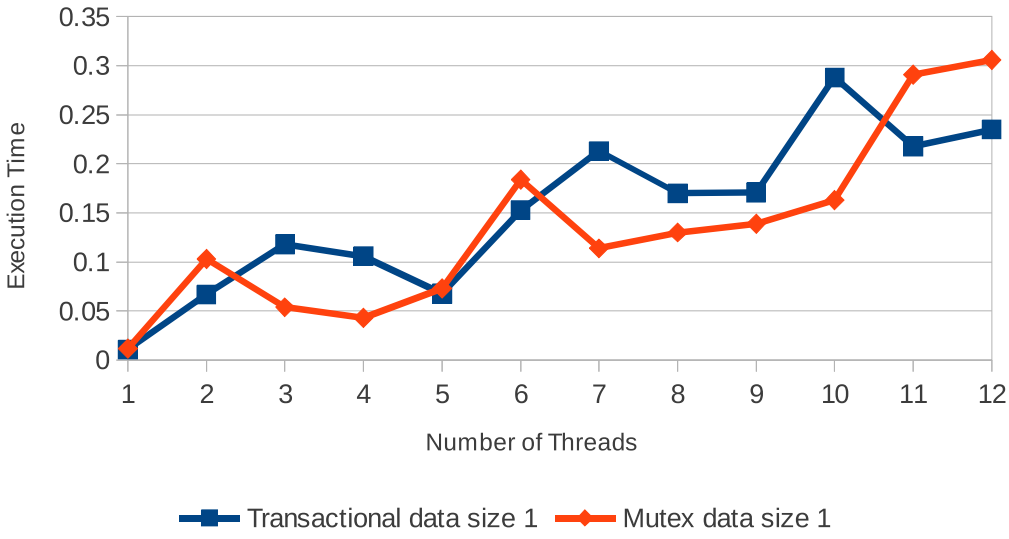

Figure 3: Transaction success rates vs. Concurrency levels with Data size 1

In the last experiment, a C++ vector of integer object is created on the heap with preallocated space. A user-specified number of threads are created. Each each thread modifies every value in the vector using transactional memory with fallback mutex or mutex only. Because the data size is small and the data structure is relatively simple, as Fig. 4 shows, 98% - 100% transaction success rate is achieved at almost all concurrency levels. The execution time of the two can now be used as direct measure of speeds of transactional memory and mutex.

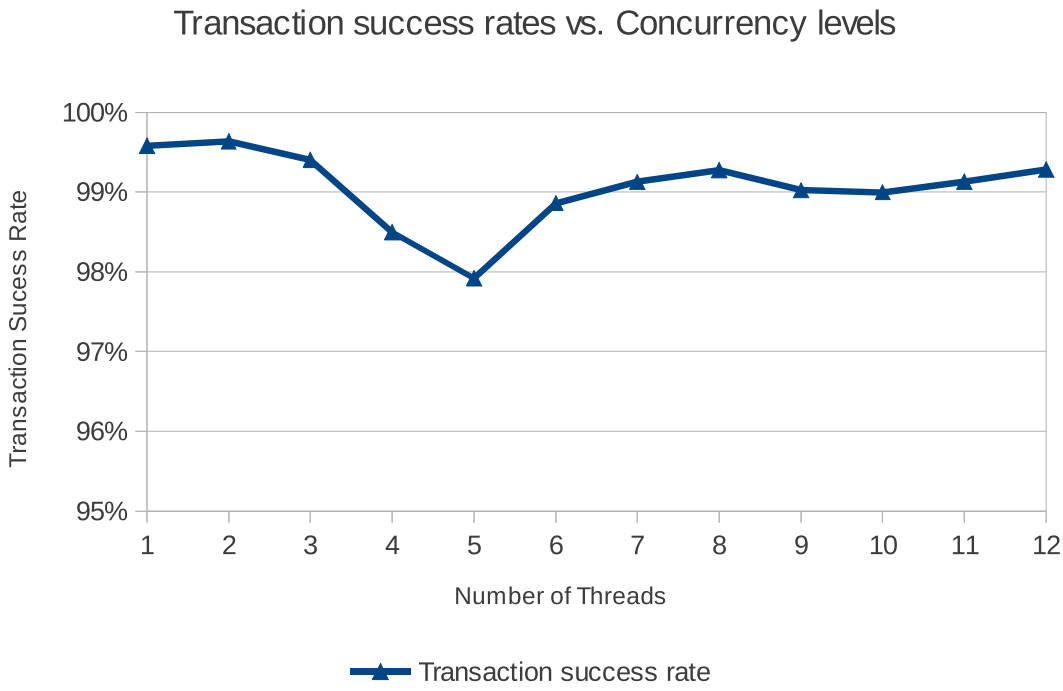

Figure 4: Transaction success rates vs. Concurrency level at data size 1

As Fig. 5 shows, the two curves cross each other multiple times which is consistent with the behavior observed in small data size insertion to the fractal tree. Therefore, we can conclude that the performance hardware transactional memory in this version of CPU is comparable to mutex only in the experiments we performed.

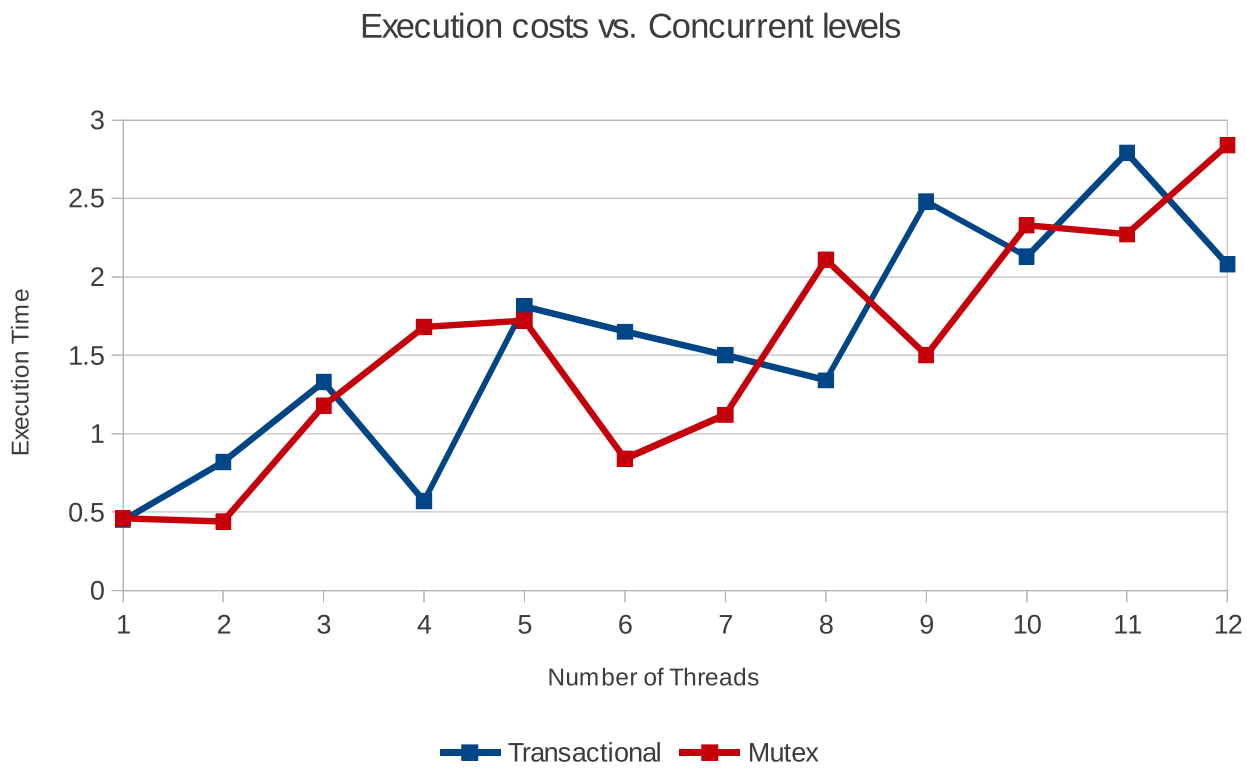

Figure 5: Transaction vs. Mutex execution for simple and small data structure

Conclusion

We implemented an in-memory version of Tokutek's fractal tree in C++ with hardware transactional memory support on a Intel Haswell CPU. According to our experiments, the speed of transaction memory is comparable to mutex only. Transactions are likely to fail at high concurrency level and large data size.

Reference

[1] http://tokutek.com/downloads/mysqluc-2010-fractal-trees.pdf