Concurrent Patricia Trie

https://github.com/kl4ng/concurrent-patricia-trie

Cole Garner
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, Florida 32816
Email: ColeGarner@knights.ucf.edu

Kevin Lang
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, Florida 32816
Email: klang@knights.ucf.edu

***Abstract*—This paper shows an implementation of a non-blocking Patricia Trie that combines techniques found in numerous recent implementations. It heavily uses the Compare and Swap operation and uses flags to prevent blocking. Additionally, it implements the flags in multiple locations in order to both increase efficiency and reduce the amount of conflicts. It also uses more efficient memory management and decreased overhead to improve upon its predecessors.**

I. Introduction

A Patricia Trie, also known as a Radix tree, is a unique version of a regular tree. The defining feature of any trie, also known as a digital tree, is that the position of a node in a tree defines the key for that node. A Patricia trie takes this and optimizes the tree by merging any parent node with only one child in order to save space and create a more efficient tree. [1] Because a Patricia Trie is a relatively simple data structure and has many practical uses, it is a good data structure to have an efficient parallelization technique for.

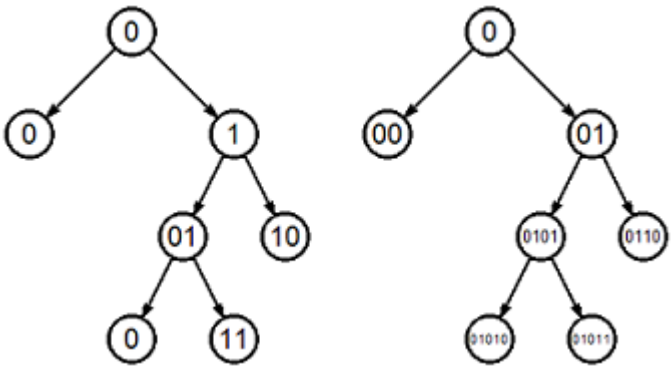

Fig. 1. An example of a Patricia Trie. The left diagram what is seen while transversing the tree and the right diagram shows what the full key value of each node is.

Our implementation of a parallelized, lock-free Patricia Tree will aim to maximize performance and space-efficiency while removing some of the overhead of previous implementations by fine-tuning the memory management. We will combining multiple older approaches and taking the benefits of each and combining them. Our implementation will be focused on using compare-and-swap (CAS) operations which will allow for a blocking-free implementation. [1], [2]

The main changes in our implementation from previous ones is that we will be storing the flags in multiple locations to allow for a smaller portion of the tree to be blocked off for some operations, creating less conflicts overall and increasing performance. [3] The other main improvement we will be making is decreasing the overhead of creating and removing flags by compacting their size as much as possible, and more efficiently handling garbage collection .

We originally planned to use a newer language called Rust to help our implementation by making use of its memory-safety property. However, we decided against this when we discovered that the language would not work in the way we wanted it to for our implementation.

For the actual Patricia tree algorithm we will be using an algorithm similar to the pseudo-code seen in [1]. However, we will be inserting various improvements on this algorithm, with the main one being that flags will also appear in the edges between two nodes to decrease conflicts. [3]

II. Related Works

Creating high-performance, non-blocking data structures has advanced in recent years. There is work into making generalized data structures using CAS operations. [4] This work has further been expanded into making generalized techniques for non-blocking trees. [2] These techniques revolve around using load-link extended (LLX), store-conditional extended (SCX) and validate-extended (VLX) primitives which are generalized techniques of the standard, non-extended versions of the primitives. [2], [4] The techniques used are very powerful and efficient and help form a basis for some techniques used in our work.

Earlier, non-generalized implementation of this technique was seen in a few different data structures. The one related to our work is Shafiei's implementation of non-blocking Patricia Tries. [1] This implementation used a binary tree implementation and handled the parallization by creating flag objects for operations that keep track of what has to be changed. These flags are very powerful because they let multiple threads work on one operation so one thread is not forced to wait. Additionally, since the flag is there there is no chance of a portion of the tree becoming unusable if one thread fails in the middle of an operation. [1], [5] A lot of the basics of

the algoirthms and structure used here can be seen in our techniques, and it provided a good core for us to build upon.

A slightly different work is an implementation of a lock-free binary search tree by Natarajan and Mittal. Their work also heavily involves CAS operations but the largest difference is that instead of marking the nodes they mark the edges between the nodes. [3] This has interesting applications in that it allows a smaller portion of the tree to be flagged during insert and delete operations and allows for less conflicts on the whole.

Similar to the previous work is another edge-based algorithm for a concurrent binary search tree by Ramachandran and Mittal. This is not a lock-free solution so it is not completely applicable to ours, however it involves edges and has relatively few nodes locked for each operation. [6] We will be using the general technique seen in the previous two works in order to reduce conflicts inside the tree and to increase the thoroughput of modify commands on our trie.

Shun and Blelloch showed another alternative approach to parallization of trees with a multiway Cartesian tree. Theres is slightly unique in that they first create an array and then convert it into a tree. However, despite being different from out project, the algorithms they show in order to generate the tree from the array using parallization techniques warranted study. We looked into their techniques of differentiating what part a particular node is protected in, but ultimately decided the techniques were too far from our own to be much use. [7]

We performed a large amount of research towards the viabiility of Rust as a tool for a concurrent patricia trie, however we ultimately decided against it due to various reasons which will be seen later. [8]

III. Difficulties with Rust Language and Libraries

Rust is a new programming language developed by Mozilla. Developed in parallel with their new Servo web rendering engine, it is developed from the ground up to support safety, concurrency, and parallelism, and was thus a very attractive candidate for our implementation of the concurrent patricia trie. [9] Specifically, in terms of safety it guarantees no data races, buffer or stack overflow, and null pointer exceptions for most use cases. Due to its heavily static nature, it can validate the compiled program to be free of such errors and thus allows us to leverage this in creating a concurrent Patricia trie that has more simple aspects of memory management handled for us. Similarly, it is a language that focuses on speed, which will also help us achieve our desired result of having a Patricia trie implementation that out-performs its counterparts.

However, Rust's approach to memory management, namely it being handled almost entirely by the language without a separate runtime for a garbage collector, has its limits. While this is true for most use cases, it becomes increasingly difficult for the compiler to guarantee this when the complexities of parallel lock-free algorithms come into play. The very nature of these algorithms relies more heavily on complex linearizability reasoning than the compiler can deduce, which means that more complex concurrency problems must have their own memory management, as is in our case.

Do to, at the time, the alpha-stage status of the core language, in addition to our inexperience with the language, we ended up choosing to use Java instead. These difficulties included the fact that the main HazardPointer library we were intending to use did not even compile on the latest version of Rust [8]. Due to the fact that we would have to manually do a large portion of our memory management by using unsafe constructs in Rust, the main advantages of the language disappeared with our use case. However, these difficulties led us to choose to implement a new algorithm (an edge-flagging patricia trie), instead of the original plan of simply doing a node-flagging patricia tree with managed memory.

IV. Algorithm Description

Our algorithm is devised from two state of the art techniques for concurrent trees both relying on CAS operations. The two techniques we will be using are the generalized non-blocking tree approach using flags inside nodes [1], [2] and the similar technique to have the flags be inside the edges instead of the nodes. [3] By combining these two methods we will be able to take the advantages of both with some minor drawbacks.

The main advantage to come the edge-based technique is that it allows for insert and delete operations to flag a smaller portion of the tree and use fewer atomic instructions. [3] By doing this, the implementation will create fewer conflicts and will allow for more concurrent insert and delete operations, allowing for great speedups for trees with heavy emphasis on insert and delete. However, the method also causes seek operations to perform slightly worst sometimes by increasing the time it takes for them to reach their target node when there is a marked branch in their path. Because of this, we will combine this approach with the more traditional node-based approach for certain operations so that we can further improve the edge-based approach by only using it when necessary.

Go into detail about what changes (if any) we make from the original pseudocode due to the nature of the Rust language (for better or for worse). Go into detail about what we do new (hopefully able to implement it with more than two children, or have less Flag objects).

A. Data Structures

The data structure used to store the Patricia Tree is simply a tree that stores a grand root node and a logical root node as well as all the functions to search and modify inside the tree. The two root structure is used for creating searching by forcing enough nodes to make a proper seek record. There is also a small data structure called SeekRecord used for storing a seek record during a seek operation.

The individual node structure is also relatively simple. It contains a key to store for comparison purposes, which is technically unneeded due to the nature of a trie, but very helpful, as well as a mask which is used for. It contains a value and a mask, and perhaps the most important part is that instead of classic pointers to the children, it makes use of an AtomicStampedReference that allows for CAS operations on the edge between the two nodes and allows for both a node

pointer and an integer to be stored. By allowing the integer to be stored inside the edge, it enables the edge to be flagged on the edge and is a large part of how our algorithm works.

B. Algorithm

The Patricia Trie initially contains a Grand root node linking to a dummy node and a logical root node. The logical root node will link to two dummy nodes representing. One of these represents all 0s, where the key is equal to 0, and the other represents all 1s, where the key is equal to the max value of an integer. With this the tree is created and we will now show you how the various operations will work on our algorithm.

1) Get and Contains Operations: These two operations are slightly different but accomplished in the same way. The operations work by starting at the logical root and going down the tree in much the same way that a normal trie would do it. The only unique feature here is that instead of using pointers, they are using the getReference function of the AtomicStampedReference.

```
pubic boolean contains(int key)
{
   return get(key) != null;
}

public T get(int key)
{
  Node node = rootNode;

  while(!isLeaf(node) && isPrefix(node,key))
  {
    if(node.key > key)
      node = node.left.getReference();
    else
      node = node.right.getReference();
   }

  if(isLeaf(node && key = node.key)
    return node.value;
  else
    return null;
}
```

2) Insert Operation: The insert operation is different from the previous two operations in that it is remarkably different from the linear version of the operation. This is one of the two operations that we implement that make use of the edge-based technique in order to implement the concurrency.

The algorithm for the operation is relatively simple. First it cycles through the tree storing the current and previous node until it either finds the node it is trying to insert, in which case it returns false, or the correct position for the node to be inserted. Once it has reached this part it then uses the current nodes key to generate the new key and mask for the new nodes. it then creates a new internal node and links it to both a new node that your inserting and the node originally in that position. Then at this point, it uses a CAS operation to ensure that the original node is still unchanged, and if it is, it then links the previous node to the new internal node which is linked to the new node and the original node. After this, the thread handles the cleanup.

Because the CAS operation is done after the internal node is properly linked to everything, the operation is linearizabile. From the view of other threads the entire operation takes place during the CAS and because of this, the operation is completely concurrent. And since we use the edge, no unrelated nodes are unavailable during the operation.

```
public boolean insert(int key, T value)
{
  Node node;
  Node prevNode;

  while(true)
  {
    prevNode = root;
    node = root.left.getReference();

    while(!isLeaf(node) && isPrefix(node,key))
    {
      prevNode = node;
      if(node.key > key)
        node = node.left.getReference();
      else
        node = node.right.getReference();
    }

    if(isLeaf(node) && node.key == key)
      return false;

    int bitwiseXOR = node.key ^ key;
    int counter = 0;
    while(bitwiseXOR != 0)
    {
      bitwiseXOR >>= 1;
      counter++;
    }

    int inKey = key >>> i << i;
    inKey = inKey | (1 << (i-1));
    int inMask = all1 >>> i << i;

    Node inter;
    Node newNode = new Node(key, value)
    if(node.key > key)
      inter = new Node(inkey,inMask,newNode,
          node)
    else
      inter = new Node(inkey,inMask,node,
          newNode)

    if(prevNode.key > key)
    {
      if(prevNode.left.compareAndSet(node,
          inter,Node.UF_UT,Node.UF_UT)
        return true;
      else if(node == prevNode.left.
          getReference())
        cleanUp(key, seek(key));
    }
    else
    {
      if(prevNode.right.compareAndSet(node,
          inter,Node.UF_UT,Node.UF_UT)
        return true;
      else if(node == prevNode.right.
          getReference())
```

```
        cleanUp(key, seek(key));
    }
  }
}
```

3) Delete Operation: Similar to the insert operation, the delete operation also takes advantage of the edge-based approach we are using. Unlike the insert operation, the delete operation has to keep track of multiple nodes to function correctly, and for this it uses both a seek record and the the grand root.

The delete operation functions by first getting the seek record which stores 4 nodes and then it uses a CAS operation to mark the edge from the parent to the node to be deleted and handles the deletion and cleanup. This is once again linearizabile because the entire deletion seems to take place during the CAS operation to other threads and since the cleanup only handles unreferenced nodes it is completely parallelizable.

```
public final void delete(int key)
{
  boolean isCleanup = false;
  SeekRecord<t> sRecord;
  Node parent;
  Node leaf = null;

  while(true)
  {
    sRecord = seek(key)
    if(!isCleanup)
    {
      leaf = sRecord.leaf;

      if(leaf.key != key)
        return;
      else
      {
        parent = sRecord.parent;
        if(parent.key > key)
        {
          if(par.left.compareAndSet(leaf,leaf,
              Node.UF_UT,Node.F_UT)
          {
            isCleanup = true;
            if(cleanUp(key, sRecord))
              return;
          }
          else if(leaf == parent.left.
              getReference())
            cleanUp(key, s);
        }
        else
        {
          if(par.right.compareAndSet(leaf,leaf
              ,Node.UF_UT,Node.F_UT)
          {
            isCleanup = true;
            if(cleanUp(key, sRecord))
              return;
          }
          else if(leaf == parent.right.
              getReference())
            cleanUp(key, s);
        }
      }

      else if(s.leaf == leaf)
      {
        if(cleanUp(key, s))
          return
      }

      else
        return;
    }
  }
}
```

V. Experimental Evaluation

In this section, we will describe the results of evaluating our edge-based-flagging patricia trie with other similar concurrent data structures.

A. Other Implementations

The other data structures we will be using will be a lock-free binary search tree based on [3] which is also implemented in Java [10]. The second and final data structure we will be testing our results with is a lock-free patricia trie with node-based flagging as described in [1]. For the implementation, we reached out to the author who provided us with the Java source code of his non-replace implementation.

B. Experimental Setup

We conducted our experiments on a basic desktop computer with a 3.0GHz Q6600 Intel Quad Core, with 4GB of ram, and running the x86_64 version of Fedora 21. As mentioned, all other implementations were also written in Java, and were executed under Java 8. Additionally, for those data structures that could accept any type of stored value, we made sure that this object was also an Integer, to minimize the difference object allocation/deallocation would have on the performance. Each simulation was run for a large enough number of iterations so that each test ran for approximately five seconds, and the results were then averaged and the throughput was calculated for each data point. For each test, we varied the following parameters, much in the same way as [3]:

1) **Key Range:** We varied the key range in order to test how the various data structures behaved under ranges of possible inserted values. We considered four different ranges: 1K, 10K, 100K, and 1M.
2) **Operation Distribution:** Considering that performance of a data structure depends heavily on how often certain instructions were executed, we were sure to consider three different workload distributions. For the write dominated workload, we had 50% insert and 50% delete operations. For the mixed workload, we had 70% search, 20% insert, and 10% delete. Lastly, the read dominated workload consisted of 90% search, 9% insert, and 1% delete.
3) **Number of Threads:** This varied the amount of contention in the data structure. We used the values of 1, 2, 4, 8, and 16.

C. Results

As mentioned before, each simulation was run for around five seconds. The results are shown in Figure 1. It is clear that the node-flagging implementation of the patricia trie (CPT-NODE-FLAG) has the best throughput for nearly every configuration, still performing better than our new edge-flagging implementation of the patricia trie (CPT-EDGE-FLAG). Another interesting data point is the fact that CPT-NODE-FLAG also outperformed the lock free binary search tree (BST-EDGE-FLAG) that uses the same edge-based flagging method as our implementation (CPT-EDGE-FLAG). Another important observation is the fact that our implementation has about equal throughput in comparison to the BST-EDGE-FLAG implementation, when averaged across all the configurations.

As the key range varies with the workload kept constant, there is no clear change in throughput for any of the algorithms, relatively speaking. However, a larger key range actually led to worse throughput, regardless of the workload type or number of threads. This can be seen in the graphs: in the write dominated configuration, e.g., there is a digression from around $7 * 10^6$ operations per second when at 8 threads at a 1K key range down to $1.4 * 10^6$ operations per second with the same amount of threads at a 1M key range.

The difference between the differing workloads was also not very pronounced, in terms of relative performance. In terms of absolute performance of all the data structures, we saw that as we reduced the amount of write-oriented instructions, we saw an expected increase in throughput. One can look across any given row in Figure 2 in order to see that this is the case. For example, as we reduce the proportion of write instructions in the 1K key range, we see an improvement of max throughput from $7.5*10^6$ to $3*10^7$ operations per second, a fairly significant improvement.

D. Discussion

The first thing that comes apparent when looking at the results is the similarity of throughput between BST-EDGE-FLAG and CPT-EDGE-FLAG (our implementation). This shows that the added checks performed at every branch during any seek within the patricia trie that are utilized to exit the search early have too much overhead to lead to significant throughput improvements over BST-EDGE-FLAG. Indeed, it shows that the added performance of being able to exit searching early equals the loss of performance that all those extra checks add.

Additionally, its clear that CPT-NODE-FLAG, the node-based flagging method for the patricia trie, is the clear throughput winner in almost all configurations. This becomes more surprising when one looks at the results from other tests, such as in Natarajan and Mittal's [3], which found a clear improvement over node-based-flagging BST techniques. We speculate that this unexpected result is due to the nature of the Java runtime environment, where the excess amount of helper objects created by CPT-NODE-FLAG may not result in any loss of throughput due to the garbage collector never being called during the short time of our tests, for example. This may explain the discrepancy between our Java-based results and Natarajan and Mittal's C-based results.

Another possible explanation for these results is the possibility that the advantages of using edge-based flagging for a patricia trie are only more readily apparent in heavily parallel environment with 32 or more logical cores. Natarajan and Mittal's testing results [3] seem to confirm this, as any significant improvements in throughput only become apparent with 16 or more threads within their 64 logical cores testing environment.

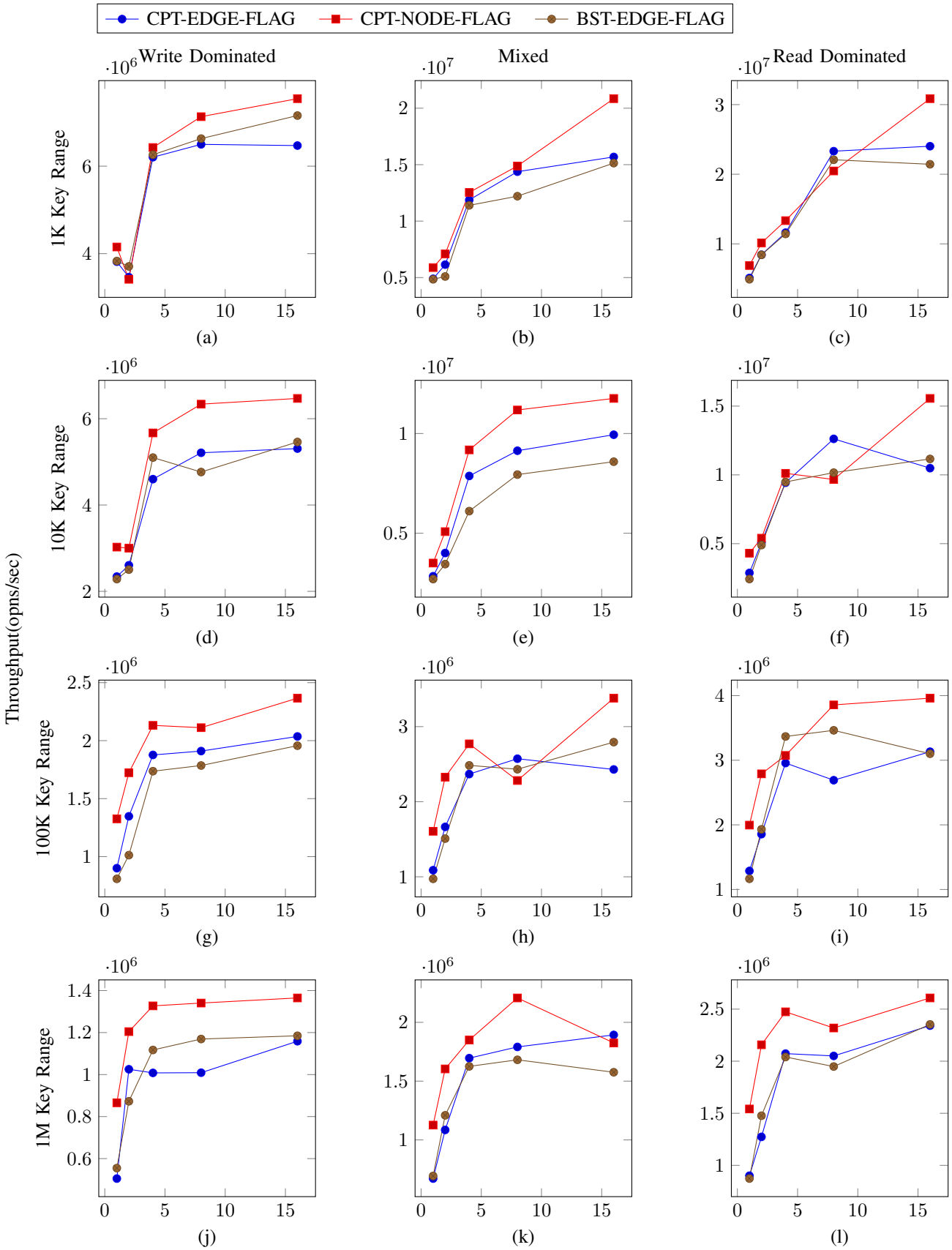

Fig. 2. Each row represents a different key range. Each column, a workload type.

VI. Conclusion and Future Work

In this paper, we present a new lock-free algorithm for concurrent manipulation of a binary patricia trie that uses compare-and-swap and bit-test-and-set atomic instructions. While our algorithm does not out-perform the node-flagging implementation that we sought to improve upon, it does have its own advantages that may be more evident in a non-garbage-collected environment.

Further work to be done consists of implementing this algorithm in a managed memory environment (such as C) and see the degree to which that affects the throughput relative to the algorithms described in [1] [3]. Additionally, the performance difference between ours and other similar algorithms should be tested within a more parallel environment to see how it performs comparatively within a highly parallel testing environment. Other future work includes adding a replace operation to our algorithm and comparing the ease of implementation and performance of that with a node-flagging patricia trie that also implements a replace operation [1].

VII. Acknowledgment

We would like to thank Niloufar Shafiei for providing her source code for her Java implementation of her paper's algorithm [1] and her willingness to help us. We would also like to thank Arunmoezhi Ramachandran for his publicly available Java implementation of lock free binary search trees with edge-based-locking [10].

References

[1] N. Shafiei, "Non-blocking patricia tries with replace operations," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, July 2013, pp. 216–225.

[2] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," *SIGPLAN Not.*, vol. 49, no. 8, pp. 329–342, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2692916.2555267

[3] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," *SIGPLAN Not.*, vol. 49, no. 8, pp. 317–328, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2692916.2555256

[4] T. Brown, F. Ellen, and E. Ruppert, "Pragmatic primitives for non-blocking data structures," in *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, ser. PODC '13. New York, NY, USA: ACM, 2013, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2484239.2484273

[5] S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 161–171. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312036

[6] A. Ramachandran and N. Mittal, "Castle: fast concurrent internal binary search tree using edge-based locking," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015, pp. 281–282.

[7] J. Shun and G. E. Blelloch, "A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction," *ACM Transactions on Parallel Computing*, vol. 1, no. 1, p. 8, 2014.

[8] B. Allard, "Lock-free hash array mapped trie in rust," https://github.com/ballard26/concurrent-hamt/, 2014.

[9] "Mozilla research projects," https://www.mozilla.org/en-US/research/projects/, accessed: 2015-03-24.

[10] R. Arunmoezhi, "Lock-free edge-based-flagging bst implemented in java," https://github.com/arunmoezhi/LockFreeBST, 2014.