Lock-Based CTries

Vered Zilberstein
Tel Aviv University

Tomer Ben-Moshe
Tel Aviv University

Guy Smoilovsky
Tel Aviv University

Abstract

A CTrie is a concurrent hash mapped trie, which implements a key-value map interface. Supported operations include insertion, removal, lookup and their conditional variants. The CTrie design is based on the work of Aleksandar Prokopec et al[1] for their *Concurrent Tries with Efficient Non-Blocking Snapshots*. The original design relies on a special distributed compare-and-swap operation called GCAS (based on the work in [3]). We redesigned their CTrie implementation to use locks instead of GCAS, while keeping the special properties of the original design. Some operations are lock-free but some are lock-based. We compare our CTrie performance with theirs.

1 Introduction

A concurrent hash-trie or Ctrie is a concurrent thread-safe lock-free implementation of a hash array mapped trie. It supports concurrent lookup, insert and remove operations. It has particularly scalable concurrent insert and remove operations and is memory-efficient. It is the first known concurrent data-structure that supports O(1), atomic, lock-free snapshots. Our contributions are the following:

1. We switch the original lock-free implementation with a new, simpler, lock-based one.
2. We streamline the read-only lookup operation.
3. We add recovery operations, that allow a modifying thread to recover from an unexpected state without releasing the lock and retrying.
4. In some failure cases, we optimize operations that can retry from their current location in the tree, wherein previously they had to restart from the root of the tree.
5. In other failure scenarios, we analyze the reason of the failure to potentially return an immediate result, thus preventing a redundant retry of the operation.
6. We run benchmarks to compare the performance of our new implementation with the original.

2 Motivation

In the original implementation, the snapshot mechanism is implemented using a special distributed CAS operation, called GCAS. This operation atomically swaps the contents of a node in the tree with intermediate descriptor objects, which only get committed after checking that the current root of the tree hasn't changed.

- GCAS is complex - multi-phase operation using intermediate objects with potential for deadlocks that have to be carefully handled.
- The GCAS approach forces any other thread concurrently traversing the tree to complete the work of any descriptor object it finds during traversal of the tree, causing potential contention.

By switching to a lock-based approach, we hoped to simplify the implementation, as well as potentially mitigate the contention caused by GCAS. As GCAS operations are complex and non-trivial, it's possible that using a simpler lock-based implementation will yield better performance.

3 GCAS_SYNC

In the original paper, the authors included the theoretical atomic semantics of the GCAS operation:

Algorithm 1: GCAS_SYNC(expectedMain, newMain, root)

```
1 this.synchronized {
2   if (this.gen == root.gen && this.main == expectedMain)
3     this.main = newMain
4     return true
5   else return false
6 }
```

Initially, we implemented this simple approach. However, this introduces a linearizability violation, shown in the following sequence diagram:

T1:[_____Add(k_1,v_1)_____]
T2:[__ReadOnlySnap()__][__SizeOfSnap__][__SizeOfSnap__]

The GCAS_SYNC checks if the generation changed, and then changes the content of the node. Let's consider a situation where T_1 is calling GCAS_SYNC in order to add (k_1,v_1) to the trie. A context switch occurs between lines 2

and 3, and T_2 takes a ReadOnly-snapshot just after the generation check and before the actual change. Such a call to ReadOnly-snapshot upgrades the current CTrie's generation, and no future modifications are allowed to be visible to readers of the previous generation. T_2 reads the snapshot's size, getting s_1, the size of the CTrie before T_1's change, since T_1 still has not changed the actual value. Then T_1 changes the value of the node, although it is now a ReadOnly-snapshot. Then T_2 reads the snapshot's size one more time, and gets a different result.

To solve this issue, we introduce an additional volatile boolean field to the INode object:

```
class INode
    volatile MainNode main
    volatile boolean aboutToWrite
```

When this variable is set to true, no thread can read the content of the node, blocking T_2 in the scenario mentioned above. It is therefore important to do the aboutToWrite check when reading the current main node of an INode:

Algorithm 2: GCAS_SYNC(expectedMain, newMain, root)

```
1  this.synchronized {
2      this.aboutToWrite = true
3      if (this.gen == root.gen && this.main == expectedMain)
4          this.main = newMain
5          this.aboutToWrite = false
6          return true
7      else
8          this.aboutToWrite = false
9          return false
10 }
```

Algorithm 3: READ_MAIN()

```
1  while (true) {
2      var tmp = this.main
3      if (!this.aboutToWrite && this.main == tmp)
4          return tmp
5  }
```

4 Lookup optimization

The original implementation of lookup operations was forced to participate in the following operations:

- Completing intermediate GCAS operations.
- Eagerly renewing the generation of INodes.
- Cleaning up encountered TNodes.

If any of the above operations fail, the lookup is forced to restart from the trie's root.

Due to the lock-based approach, the first item becomes unnecessary.

After analyzing the possible scenarios, we concluded that there is no linearizability violation if we simply remove the other items.

Therefore, we also never have to restart the lookup operation from the root.

This makes the lookup operation more passive and much simpler. We predicted that these changes should translate to much faster lookup operations during concurrent modification of the CTrie.

Algorithm 4: lookup(k, hashCode)

```
1  val m = READ_MAIN()
2  m match
3      case cn: CNode ⇒
4          if (cn does not contain hashCode) return NOT_FOUND
5          else if (cn[hashCode] is an INode)
6              // Go down one level in recursion
7              return cn[hashCode].lookup(k, hashCode)
8          else if (cn[hashCode] is an SNode sn &&
9                  sn.hc == hashCode && sn.k == k)
10             return sn.value
11         else return NOT_FOUND
12     case tn: TNode[K, V] ⇒
13         if (tn.hc == hashCode && tn.k == k)
14             return tn.value
15         else return NOT_FOUND
16     case ln: LNode[K, V] ⇒
17         return ln.get(k)
```

5 Recovery operations

In the original implementation, and in the new implementation shown so far, any unexpected main node discovered inside the monitor lock will cause the operation to fail. In many cases, this is unnecessary. For example, insertions into different locations in a CNode don't conflict, but will still cause GCAS_SYNC to fail, as a new CNode is created and written to this.main for every mutation operation (CNodes are immutable).

To prevent spurious failures, we introduce a mechanism called "Backup Generators". These are anonymous functions which are passed to GCAS_SYNC. When GCAS_SYNC detects an unexpected main node, the backup generator is called with the unexpected main as an argument. The generator can optionally return a new main node, computed from the unexpected main. In the above example, the generator will return an updated CNode with the new value inserted at the appropriate location.

In some cases, the failure is irrecoverable - for example, when detecting that the main node is now a TNode. In these cases, the generator will return nothing and the GCAS_SYNC operation will fail. If the generator returns some new main node, the GCAS_SYNC succeeds.

Note that the recovery operation is performed *without* releasing the monitor lock. We predicted that this will decrease the contention on the monitor lock and prevent the overhead of releasing and re-acquiring it. Also, if we release the lock while generating the backup value, we risk another thread concurrently changing the main node, which will force us to repeat the process.

Algorithm 5: GCAS_SYNC(expectedMain, newMain, root, backupGenerator)

```
1  this.synchronized
2    val currMain = this.main
3    this.aboutToWrite = true
4    if (this.gen ≠ root.gen)
5      this.aboutToWrite = false
6      return false
7    else
8      if (currMain == expectedMain)
9        WRITE_MAIN(newMain)
10       this.aboutToWrite = false
11       return true
12     else
13       backupGenerator(currMain) match
14         case Some(backup) ⇒
15           WRITE_MAIN(backup)
16           this.aboutToWrite = false
17           return true
18         case _ ⇒
19           this.aboutToWrite = false
20           return false
```

6 Comparison failure analysis

In the original implementation, no distinction is made regarding the reason for a GCAS failure. There are multiple very different failure scenarios:

- The generation of the CTrie was updated while we're traversing the tree, due to a concurrent snapshot operation (this.gen $\neq$ root.gen). This error is unrecoverable without restarting the operation from the CTrie's root.
- An unexpected TNode was found in this.main. This scenario is also unrecoverable, since INode has the invariant that a TNode as the main node will never be replaced.
- An unexpected subtree was detected. For example, if we want to $insert(k_5, v_5)$ into cell i. When we search the trie for the correct node to add k_5 we find cell i with the main pointing to an SNode as seen in figure 1. But then when locking the node, we detect that cell i already contains a CNode subtree(as seen in figure 2). In the original implementation, this would have caused a restart of the operation from the root. Optimally, we only need to release the monitor lock on the current level of the tree, and recurse into the subtree to insert the value there.

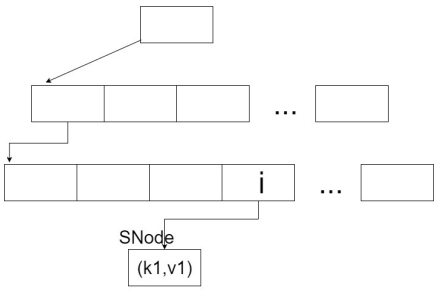

Figure 1: Trie when Searching the node to add (k_5, v_5)

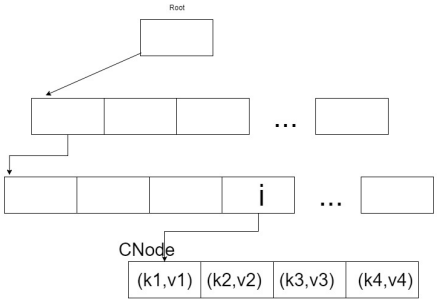

Figure 2: Trie when the lock is obtained

- In various insert-if operations (e.g. insert if key is absent), we can fail due to a violation of the condition (e.g. some other concurrent thread inserted the key before we entered the monitor lock). The original implementation would restart the operation from the root. Optimally, by analyzing the unexpected main node, we can immediately detect that the condition is violated and return the operation's result without restarting and without any additional work.

  For example, let's consider the previous demonstration of $insert(k_5, v_5)$, but instead, $insert - if - key - is - absent(k_4, v_4)$. The original implementation will fail after the CAS fails and restart from the root. Our implementation will immediately see that the key is not absent and return the appropriate result, without the unnecessary restart of the operation.

To implement the above optimizations, we need to differentiate the various possible results of a GCAS_SYNC operation. We also need to return the unexpected main, if one is encountered, so that its contents can be analyzed.

Below, we give the full pseudocode of the $insert - if - key - is - absent$ operation as an example, including the required modifications to GCAS_SYNC and the handing of all the possible cases:

Algorithm 6: GCAS_SYNC(expectedMain, newMain, root, backupGenerator)

```
this.synchronized
  val currMain = this.main
  this.aboutToWrite = true
  if (this.gen ≠ root.gen)
    this.aboutToWrite = false
    return GcasGenFail
  else
    if (currMain == expectedMain)
      WRITE_MAIN(newMain)
      this.aboutToWrite = false
      return GcasSuccess
    else
      backupGenerator(currMain) match
        case Some(backup) ⇒
          WRITE_MAIN(backup)
          this.aboutToWrite = false
          return GcasCompareRecovery(currMain)
        case _ ⇒
          this.aboutToWrite = false
          return GcasCompareFail(currMain)
```

Algorithm 7: insert-if-key-is-absent-backupGenerator(k, v, hashCode)

```
return (unexpectedMain) => match
  case tn: TNode[K,V] => return None
  case ln: LNode[K,V] =>
    if (ln.contains(k)) return None
    else return ln.inserted(k, v)
  case cn: CNode[K,V] =>
    if (cn does not contain hashCode)
      return cn.inserted(k,v,hashCode)
    else cn[hashCode] match
      case INode => return None
      case sn: SNode[K, V] =>
        if (sn.hc == hashCode && sn.k == k)
          return None
        else
          // Hash collision, create new tree level or LNode
          return cn.inserted(k,v,hashCode)
```

Algorithm 8: attempt-insert-if-key-is-absent(expectedMain, newMain, root, k, v, hashCode)

```
backupGenerator ←
  insert-if-key-is-absent-backupGenerator(k, v, hashCode)
gcasResult ← GCAS_SYNC(expectedMain, newMain, root,
  backupGenerator)
gcasResult match
  case GcasSuccess => ReturnExpected
  case GcasGenFail => Restart
  case GcasCompareRecovery => Return(None)
  case GcasCompareFail(unexpectedMain) =>
    getPreviousValue(unexpectedMain, k, hashCode) match
      case SomeValue(prevV) => Return(prevV)
      case SubTree | TombNode => Retry
```

Algorithm 9: insert-if-key-is-absent(k, v, hashCode, root, startGen)

```
m ← this.READ_MAIN()
m match
  case TNode =>
    cleanParent()
    Restart from root
  case ln : LNode =>
    if (ln.contains(k)) return ln.get(k)
    else
      requiredAction ← attempt-insert-if-key-is-absent(
        ln, ln.inserted(k,v,hashCode), root, k, v, hashCode)
      requiredAction match
        case ReturnExpected => return None
        case Return(someExistingValue) => return
                                          someExistingValue
        case Restart => Restart from root
        case Retry => return this.insert-if-key-is-absent(
                              k, v, hashCode, root, startGen)
  case cn : CNode =>
    if (cn does not contain hashCode)
      requiredAction ← attempt-insert-if-key-is-absent(
        cn, cn.inserted(k,v,hashCode), root, k, v, hashCode)
      requiredAction match
        case ReturnExpected => return None
        case Return(someExistingValue) => return
                                          someExistingValue
        case Restart => Restart from root
        case Retry => return this.insert-if-key-is-absent(
                              k, v, hashCode, root, startGen)
    else cn[hashCode] match
      case in : INode =>
        if (startgen == in.gen)
          return in.insert-if-key-is-absent(
                              k, v, hashCode, root, startGen)
        else attemptGenerationUpdate(
                                cn, startgen, root) match
          case Retry => return this.insert-if-key-is-absent(
                              k, v, hashCode, root, startGen)
          case _ => Restart from root
      case sn : SNode =>
        if (sn.hc == hashCode && sn.k == k)
          return sn.v
        else //Hashcode collision
          newMain ← cn.inserted(k,v,hashCode)
          requiredAction ←
  attempt-insert-if-key-is-absent(
                              cn, newMain, root,
                              k, v, hashCode)
          requiredAction match
            case ReturnExpected => return None
            case Return(someExistingValue) => return
                                          someExistingValue
            case Restart => Restart from root
            case Retry => return
                        this.insert-if-key-is-absent(
                              k, v, hashCode, root, startGen)
```

7 Snapshot

There are two main improvements in the lock-based CTrie over the original CTrie. First, the original implementation made use of RDCSS which allocates an intermediate object. Second, whenever the call to RDCSS fails a recursive call is made. Our implementation doesn't make use of RDCSS and therefore there is no need for intermediate objects and recursive calls, which makes our implementation simpler and more memory efficient.

8 Evaluation

We performed experimental measurements on a JDK8 configuration with two Intel Xeon 2.20 GHz processors, each with 22 cores and 44 hyperthreads (total of 88 hyperthreads). We followed established performance measurement methodologies [2]. We compared the performance of the original CTrie data structure against our implementation.

All of the benchmarks show the number of threads used on the x-axis and the mean time of (M = 100000) measurements in milliseconds on the y-axis. Some experiments were done directly on the CTrie and some on a snapshot.

The first benchmark called **insert** starts with an empty data structure and inserts N = 1000000 entries into the data structure. The work of inserting the elements is divided equally between P threads, where P varies between 1 and the maximum number of hyperthreads (x-axis).

The benchmark **lookup** does N = 1000000 lookups on a previously created data structure with N elements. The work of looking up all the elements is divided between P threads, where P varies as before.

The **remove** benchmark starts with a previously created data structure with N = 1000000 elements. It removes all of the elements from the data structure. The work of removing all the elements is divided between P threads, where P varies.

To evaluate **snapshot** performance, we do **3 kinds** of benchmarks:

The **snapshot - remove** benchmark is similar to the remove benchmark - it measures the performance of removing all the elements from a snapshot of a Ctrie and compares that time to removing all the elements from an ordinary Ctrie.

The benchmark **snapshot - insert** is similar to the insert benchmark - it measures the performance of inserting all the elements to a snapshot of a Ctrie and compares that time to inserting all the elements to an ordinary Ctrie.

The benchmark **snapshot - lookup** is similar to the last one, with the difference that all the elements are looked up once instead of being removed.

Our implementation appears in red, the previous implementation in blue.

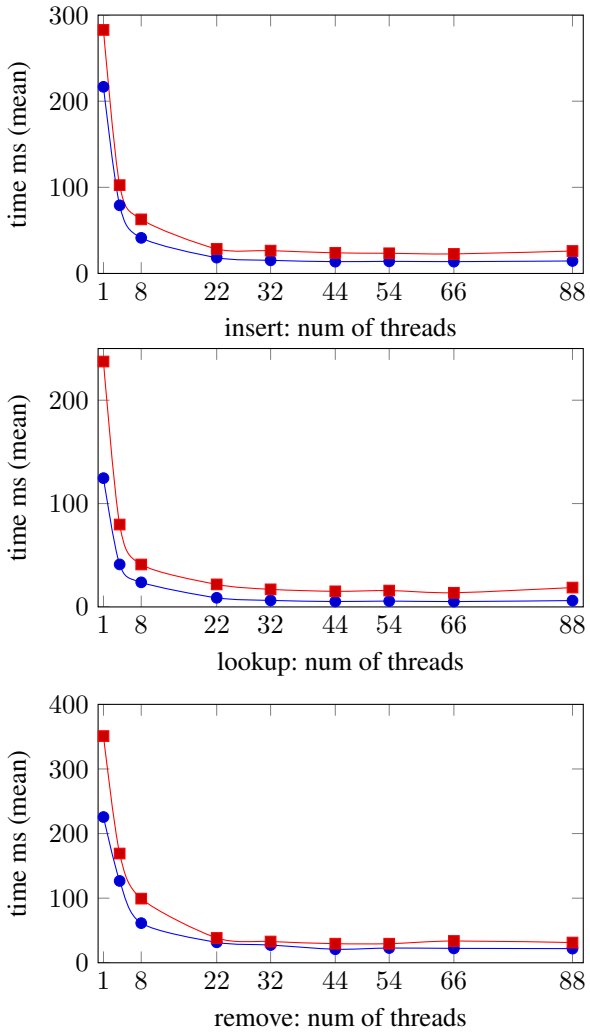

Figure 3: basic operations

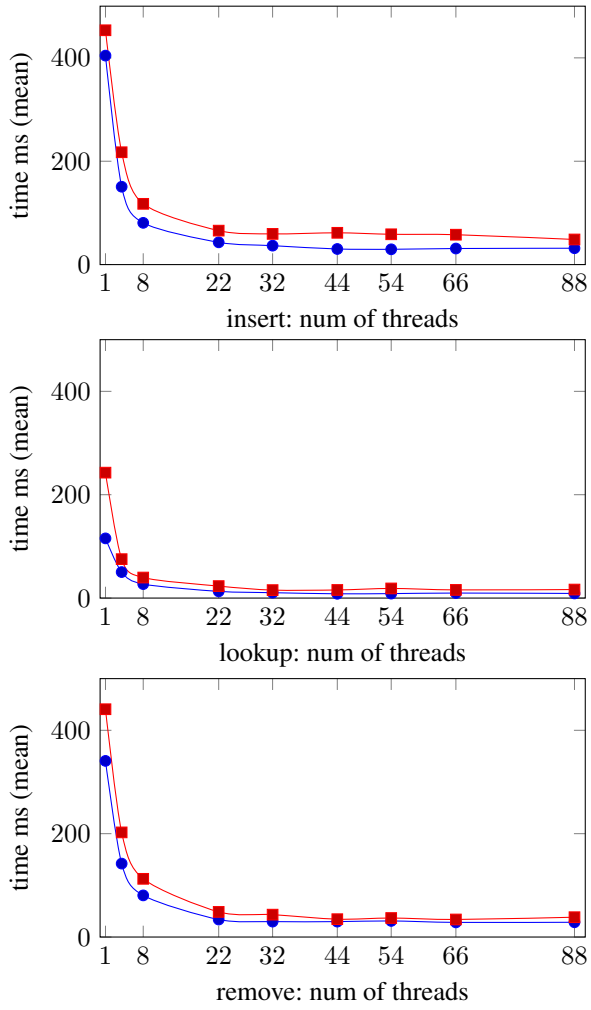

Figure 4: basic operations /w snapshot

Then we ran a benchmark for memory occupancy. It seems that our design required slightly more memory than the original design.

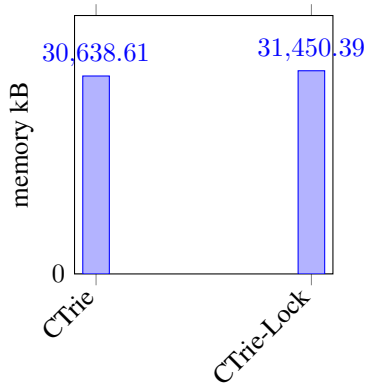

Figure 5: memory occupancy

Finally we ran benchmarks testing the performance of the data structures when the operations are invoked in a respective ratio. The ratio resembles: lookups-inserts-removes. The tested ratios were: 90-9-1, 80-15-5 and 60-30-10. Starting from an empty data structure, a total of N = 1000000 invocations are done. The work is divided equally among P threads.

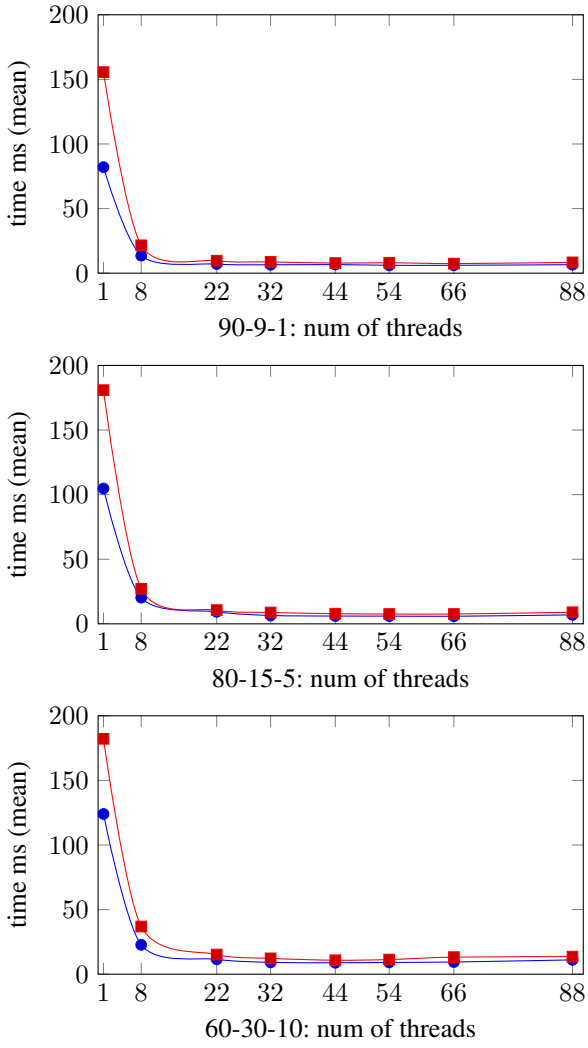

Figure 6: basic operations with ratios

The preallocated-5-4-1 benchmark is slightly different - it starts with a data structure that contains all the elements.

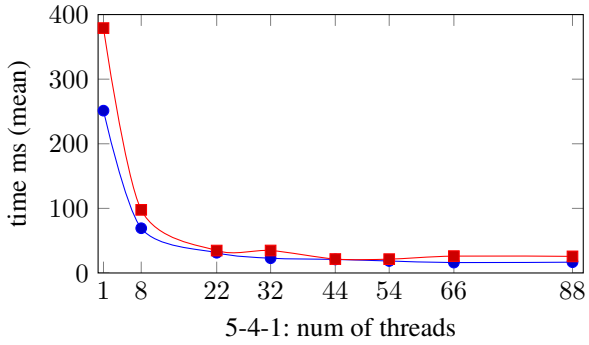

Figure 7: basic operations with ratios

One can notice that the overall time of our design did not succeed in reaching better results, although we did maintain the same scalability curve. The performance degradation in mutation operations is not unexpected, and can probably be explained by the overhead of locking vs. CAS operations. The overhead of the lock operation seems to be much more significant when there are few threads.

Surprisingly, it seems we also experienced performance degradation in lookup operations, even though we feel that the lookup operation was streamlined significantly and should perform better than the previous implementation, regardless of locks. We currently have no explanation for this. This requires further investigation.

9 Future work

It's possible to implement additional operations on the data structure. For example: an atomic replaceKey, which given a key and value already present in the map, atomically replaces them with another key and value. Another example is an atomic union operation which takes two CTrie data structures and atomically produces their union.

Our lock-based approach could be used to more easily implement such operations, at the possible price of blocking sections of the trie.

Although our evaluation shows degradation of performance compared to the original implementation, we suspect that it is still possible to improve performance by taking only parts of our design while omitting other parts. For example, the comparison failure analysis can be performed without switching GCAS to a lock based implementation. It's also possible that recovery operations while holding the lock degrade performance instead of improving it.

10 References

[1] Concurrent Tries with Efficient Non-Blocking Snapshots
A. Prokopec, N.G. Bronson, P. Bagwell, M. Odersky. *PPOPP* pp. 151-160 (2012)

[2] Statistically Rigorous Java Performance Evaluation
A. Georges, D. Buytaert, L. Eeckhout. *Blackboard Systems.* OOPSLA, 2007

[3] DCAS-Based Concurrent Deques
O. Agesen, D. L. Detlefs, C. H. Flood, A. Garthwaite, P. A. Martin, N. Shavit , G. L. Steele Jr. *SPAA* 2000.