Lock-free concurrent van Emde Boas Array

Ziyuan Guo
Reiji Suda (Adviser)
s10e@is.s.u-tokyo.ac.jp
reiji@is.s.u-tokyo.ac.jp
The University of Tokyo
Graduate School of Information Science and Technology
Tokyo, Japan

ABSTRACT

Lock-based data structures have some potential issues such as deadlock, livelock, and priority inversion, and the progress can be delayed indefinitely if the thread that is holding locks cannot acquire a timeslice from the scheduler. Lock-free data structures, which guarantees the progress of some method call, can be used to avoid these problems. This poster introduces the first lock-free concurrent van Emde Boas Array which is a variant of van Emde Boas Tree. It's linearizable and the benchmark shows significant performance improvement comparing to other lock-free search trees when the date set is large and dense enough.

CCS CONCEPTS

• **Computing methodologies → Concurrent algorithms**.

KEYWORDS

concurrent data structures, lock-freedom, van emde boas tree

ACM Reference Format:
Ziyuan Guo and Reiji Suda (Adviser). 2019. Lock-free concurrent van Emde Boas Array. In *Proceedings of SC '19 Research Poster*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This paper introduces the first lock-free concurrent van Emde Boas Array which is a variant of van Emde Boas Tree (vEB tree). It's linearizable and the benchmark shows significant performance improvement comparing to other lock-free search trees when the date set is large and dense enough.

Although van Emde Boas tree is not considered to be practical before [3], it still can outperform traditional search trees in many situations, and there is no lock-free concurrent implementation yet. Thus it has been chosen as the base structure of this study.

SC '19 Research Poster, November 17 - 22, 2019, Denver, CO, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To unleash the power of modern multi-core processors, concurrent data structures are needed in many cases. Comparing to lock-based data structures, lock-free data structures can avoid some potential issues such as deadlock, livelock, and priority inversion, and guarantees the progress [5].

2 OBJECTIVES

- Create a lock-free search tree based on van Emde Boas Tree.
- Get better performance and scalability.
- Linearizability, i.e., each method call should appear to take effect at some instant between its invocation and response [5].

3 RELATED WORK

The base of our data structure is van Emde Boas Tree, which is named after P. van Emde Boas, the inventor of this data structure [7]. vEB tree use some bits from the key as index, and combined with a recursive structure to achieve $O(\log \log u)$ time complexity (u is the upper bound of element key). K. Kułakowski [6] introduced the first lock-based concurrent vEB array, a vEB tree with fixed degree and use bit vector to mark non-empty sub-trees.

The common practice of implementing lock-free data structures is to make threads cooperate with each other. This idea is introduced by G. Barnes [2]. By describing and storing exactly what a thread is doing, other threads can help it to finish its work. Our implementation also uses this technique.

Additional memory required by the cooperate technique is reused to avoid the large amount of memory allocation and release, similar to the method described by M. Arbel-Raviv and T. Brown [1].

4 STRUCTURE

The original plan was to implement a van Emde Boas Tree by effectively using the cooperate technique as locks. However, a vEB tree stores maximum and minimum elements on the root node of each sub-tree, which create a severe scalability bottleneck. Thus, we choose to not store the maximum and minimum element on each node, and use a fixed degree (64 on modern 64bit CPUs) for every node, then use a bit vector *summary* to store whether the corresponding sub-tree is empty or not. In this way, operations about finding the first and last nonempty sub-tree, finding the previous and next nonempty sub-tree, mark a sub-tree to empty or non-empty can be done by only a single instruction.

For better performance and scalability, in our definition the summary and cluster are not strictly related, i.e. when the summary shows the sub-tree is non-empty, it may be actually empty, but

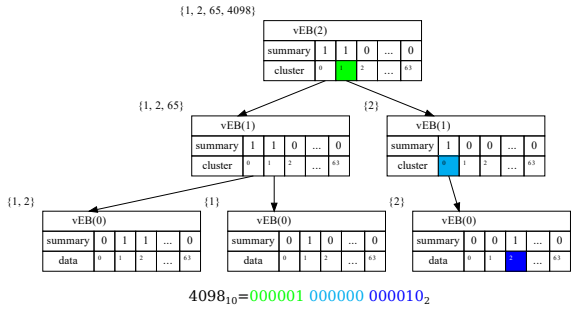

Figure 1: A vEB array that stores 1, 2, 65 and 4098. The color shows the bits that are used to index at each level.

when the sub-tree is non-empty, the summary will always show the right information.

5 OPERATION

Search is the simplest one. As the graph brove shows, once the entire structure is allocated, the memory address of each element can be determined, all it needs to do is to locate 3 elements on 3 arrays and return the data. Search operation is wait-free.

Minimum and Maximum will recursively return the minimum and maximum element of the first and last non-empty sub-tree, and help the unfinished insert operation on the execution path.

Predecessor and Successor can be done by finding the predecessor (successor) in the same cluster and previous (next) non-empty cluster, then return the larger (smaller) one.

Insert Insert operation will put a descriptor on the first node that it should modify to effectively lock the entire path of the modification. Then help itself to finish the work by execute CAS operations stored in the descriptor.

```
1  bool insert(x) {
2    if (not leaf) {
3      do {
4  start_over:
5        summary_snap = summary;
6        if (summary_snap & (1 << HIGH(x)))
7          if (cluster[HIGH(x)].insert(LOW(x)))
8            return true;
9          else
10           goto start_over;
11       if (needs_help(summary_snap)) {
12         help(summary_snap);
13         goto start_over;
14       }
15       if (empty(summary_snap) && not root)
16         return false;
17       new_summary= create_descriptor() | (1 << HIGH(x));
18       if (!cluster[HIGH(x)].insert_append(LOW(x)))
19         goto start_over;
20       descriptor_append(&summary, new_summary,
               NOHELP(new_summary));
21     } while (!CAS(&summary, summary_snap, new_summary));
22     help(new_summary);
23     return true;
24   } else { ... }
25 }
```

Firstly, a snapshot of summary (combined with a version number and descriptor marker as int128) is taken, and checked if modification is needed on the current level (line 5-10). If no modification is needed, it will insert the lower bits (e.g. cyan and blue part in Fig.1) recursively into its sub-tree. Otherwise, it will try to help other threads and check if the current insertion is legal (line 11-16). Next, a descriptor will be created, and the remaining CAS operation that is needed to perform on the sub-tree is added by *insert_append*. Finally, a CAS operation is performed to put the descriptor on the current node and also mark the corresponding summary bit.

```
1  bool insert_append(x) {
2    summary_snap = summary;
3    while (needs_help(summary_snap)) {
4      help(summary_snap);
5      summary_snap = summary;
6    }
7    if (not empty(summary_snap))
8      return false;
9    descriptor_append(&summary, summary_snap, summary_snap |
         1 << HIGH(x));
10   if (not leaf) {
11     return cluster[HIGH(x)].insert_append(LOW(x));
12   } else {
13     descriptor_append(&data[key], data[key], new_value);
14     return true;
15   }
16 }
```

insert_append just recursively append remaining CAS operations into the descriptor. Since this operation is performed on an empty sub-tree, as long as the root of this sub-tree don't change, all the nodes on this sub-tree will not change.

Delete starts from the bottom to top basically by performing DCSS (Double Compare Single Swap) step by step.

6 RESULT

We compared the thoughput of our implementation with the lock-free binary search tree and skip-lists implemented by K. Fraser [4].

Benchmark Environment

- Intel Xeon E7-8890 v4 @ 2.2GHz
- 32GB DDR4 2133MHz
- qemu-kvm 0.12.1.2-2.506.el6_10.1
- Kernel 5.1.15-300.fc30.x86_64
- GCC 9.1.1 20190503 with flag -O3 -march=native

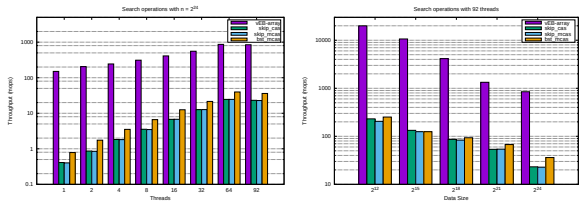

Figure 2: Search with different data size and thread number

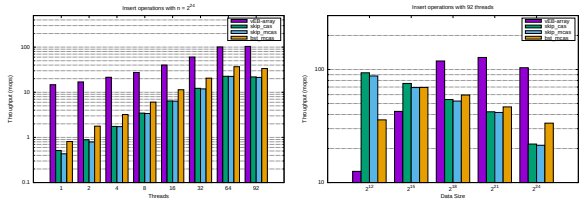

Figure 3: Insert with different data size and thread number

REFERENCES

[1] Maya Arbel-Raviv and Trevor Brown. 2017. Reuse, Don't Recycle: Transforming Lock-Free Algorithms That Throw Away Descriptors. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[2] Greg Barnes. 1993. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel Algorithms and Architectures*. ACM, 261–270.

[3] Roman Dementiev, Lutz Kettner, Jens Mehnert, Peter Sanders, et al. 2004. Engineering a Sorted List Data Structure for 32 Bit Key.. In *ALENEX/ANALC*. 142–151.

[4] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.

[5] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming*. Morgan Kaufmann.

[6] Konrad Kułakowski. 2014. A concurrent van Emde Boas array as a fast and simple concurrent dynamic set alternative. *Concurrency and Computation: Practice and Experience* 26, 2 (2014), 360–379.

[7] Peter van Emde Boas. 1975. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. IEEE, 75–84.