ELB-trees - Efficient Lock-free B+trees

Bonnichsen, Lars Frydendal; Karlsson, Sven ; Probst, Christian W.

Published in:
Proceedings of the 10th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES 2014)

Publication date:
2014

Efficient Lock-free B+trees

Lars F. Bonnichsen, Sven Karlsson,
Christian W. Probst

ABSTRACT

As computer systems scale in the number of processors, data structures with good parallel performance become increasingly important. Lock-free data structures promise improved parallel performance at the expense of higher complexity and sequential execution time. We present ELB-trees, a new lock-free dictionary with simple synchronization in the common case, making it almost 30 times faster than sequential library implementations at 24 threads.

1 Introduction

Modern computer systems routinely make use of multiple processors. However, software generally scales poorly to multiple processors, due to the way underlying data structures are designed or used. The current best practice is to use mutual exclusion to keep shared data structures consistent. Mutual exclusion is a source of many problems, including deadlocks, priority inversion, serialization of critical regions, and starvation. Lock-free data structures remedy these problems, trading mutual exclusion for complex synchronization algorithms using atomic operations. As a result of their complexity and overhead for atomic operations, lock-free data structures tend to have high execution time overhead.

In this abstract, we introduce ELB-trees, a fast lock-free dictionary. The main contributions being the ELB-trees and their performance evaluation. An early form of ELB-trees, supporting only priority queue operations were originally introduced in [2].

2 Background

Modern multiprocessors support synchronization with atomic operations, such as *compare-and-swap* (*CAS*). *CAS(a, b, c)* atomically performs `*a == b ? (*a = c, b) : *a`. *CAS* is powerful enough to implement any data structure in a lock-free manner [1], but naive use of *CAS* can lead to the ABA problem [4]. Specifically, if a node is deallocated and reused for a new node, then other threads may mistake the old and new nodes.

Hazard pointers solve the ABA problem by ensuring the nodes are only deallocated when *unaccessed* through a memory-access policy: before *accessing* nodes, threads must specify that they will *access* them and verify that the they are *reachable*. Before deallocating nodes, threads must ensure that the nodes are *unreachable* and not *accessed* by other threads. Threads specify which nodes they *access* through a list of *Hazard pointers*, ie pointers to nodes. The only complication with Hazard pointers, is checking whether nodes are *reachable* or not.

3 The design of ELB-trees

ELB-trees are structurally similar to B+trees, with separate representations for leaf and internal nodes. Key-value pairs (entries), are stored unordered in leaf nodes, while internal nodes store child pointers and separator entries sorted to guide search. Keeping leaf nodes unordered enables insertion and removal with a single *CAS* operation; simply locate a relevant entry and modify it with *CAS*. Internal nodes can only be modified by changing a single child pointer at a time, replacing the child. To enable replacing of the root, ELB-trees keep a static "fake root" which points to the real root. While replacing entire nodes is expensive, it is only neccessary for infrequent rebalancing.

As with B+trees, ELB-trees rebalance when nodes are too dense or sparse. Rebalancing is achieved by splitting the node, merging it with its siblings, or evenly distributing the entries between the node and its siblings. Such operations involve multiple nodes, which is handled with a lock-free "help-locking" scheme: modification to the node and its closest relatives is prevented, while providing details on how to finish rebalancing. If another thread is prevented from making progress, it can help finish the rebalancing, ensuring lock-freedom.

Leaf node modification is prevented by setting a special freeze bit in all of its entries, similar to the work of Bragisky *et al* [5]; entries cannot be modified when the bit is set, forcing other threads instead to help rebalance. Modification of internal nodes is prevented by setting a "status" field in the node, prior to modification of internal nodes, similar to the work of Ellen *et al* [3]. The status field stores the details required to help a rebalancing. Specifically it contains a pointer to the parent node, an entry stored in the unbalanced node, the unbalanced nodes height, and the current nodes relation to the unbalanced node.

At a high level, rebalancing of a node is done as follows:

1. If an ancestor of the node is unbalanced, then balance the ancestor first.
2. Prevent modification of the grandparent and parent of the node, as well as the node and its sibling, in that order.
3. Create balanced replacements of the node, its sibling, and their parent.
4. Using *CAS*, replace the parent and clear the status field of the grandparent.

```
help(step, child, parent, gParent):
goto step
STEP1:
  get hazard pointer to sibling.
  CAS grandparent's status field to
    {key, parent, cHeight, STEP2}.
  if status field changed, goto FIND_HELP.
  if grandparent no longer contains parent,
    clear grandparent's status field.
STEP2:
  CAS parent's status field to
    {key, parent, cHeight, STEP3}.
  if status field changed, goto FIND_HELP.
STEP3:
  if node is leaf, freeze it, rebalance it,
    replace parent and return.
  CAS node's status field to
    {key, parent, cHeight, STEP4}.
  if status field changed, goto FIND_HELP.
STEP4:
  if node is dense, split the node
    replace its parent and return.
  CAS sibling's status field to
    {key, parent, cHeight, STEP5}.
  if status field changed, goto FIND_HELP.
STEP5:
  even out number of children in the node and
    its sibling.
  replace parent and return.
FIND_HELP:
  find nodes relevant to preventing operation.
  if the grandparents status field does not
    match or the parent does not match status
    field or sibling unreachable, then the
    preventing operation has finished, return.
  goto step indicated by preventing operation.
```

Figure 1: High-level description of the helping scheme.

Table 1: Platforms used for the experiments. PC_A supports hyper-threading.

Microprocessor and speed			Cores	L1 Size	L2 Size	L3 Size
PC_A	2x Intel Xeon X5570	2.933 GHz	2x 4	8x 32 KB	8x 256 KB	2x 8 MB
PC_B	2x AMD Opteron	1.9 GHz	2x 12	24x 64 KB	24x 512 KB	4x 6 MB

The helping scheme is implemented in a manner similar to Figure 1. Along the way, there are a number of special cases that has to be dealt with, such as dense nodes, rebalancing being finished by other threads, leaf nodes, the root node, and deciding on siblings. Nodes are rebalanced with the sibling that permit the lowest entries. We use Quickselect when rebalancing leaf nodes into two leaf nodes [6]. Nodes cannot be sparse, if they are the only child of the real root.

Searching ELB-trees is similar to other trees, but with hazard pointers. The *reachability* criterion is: The fake root is always *reachable*. The real root is *reachable* if the fake root points to it. Other nodes are *reachable* if they are not being rebalanced, or the parent to a node being rebalanced. When searching through nodes that are being rebalanced, the searching thread helps the rebalancing.

4 Results

We evaluate ELB-trees on two systems summarized in Table 1. Both platforms run Linux; Debian 6.0.6 with kernel 2.6.38.6 on PC_A, and Scientific Linux 6.1 with kernel 2.6.32 on PC_B. We compile with GCC 4.6.1 using the flags: `-Ofast -flto -fwhole-program -fopenmp`. All memory is preallocated to prevent memory allocation overhead from influencing the results. Each data point is the average of 160 runs presented with 95% confidence intervals.

The experiment is laid out as follows: p threads each perform n/p operations on a dictionary with n entries. 20% of the operations are insertions, 20% are removals, and 60% are searches. The keys used for the operations and initial entries are sampled from the discrete key distribution $U(1, 2^{\lceil 1+log_2(N) \rceil})$. Nodes have 32 entries or child pointers, and are considdered sparse when containing 4 or fewer entries or child pointers. Leaf nodes are considdered dense when they have more than 26 entries. Allowing such sparse nodes may seem inefficient, but we found that the ELB-trees used significantly less memory than the competing data structure, likely due to B+trees having few internal nodes.

Fig. 2 displays the results of this experiment. When compared to the single-threaded case, PC_A achieves a peak speedup of 9.9, 10.9, and 12.1 for $n = 10^4$, $n = 10^5$, and 10^6, respectively. On PC_B the same figures are 8.8, 13.7, and 17.6. This is almost 30 times faster than GCC's STL multimap implementation.

ELB-trees are $\approx 25\%$ slower in the single-threaded case than the multimap for $n = 10^4$, but roughly 60% faster for $n = 10^6$. The performance at $n = 10^4$ is due to synchronization and ELB-trees executing 5.9 times as many instructions per operation. The STL-multimap causes 6.9 times as many L1 cache misses, which helps explain why ELB-trees are only 25% slower. The improved performance when $n = 10^6$ is due to ELB-trees only executing 3.4 times as many instructions, and the STL-multimap causing 5 times as many TLB misses. The

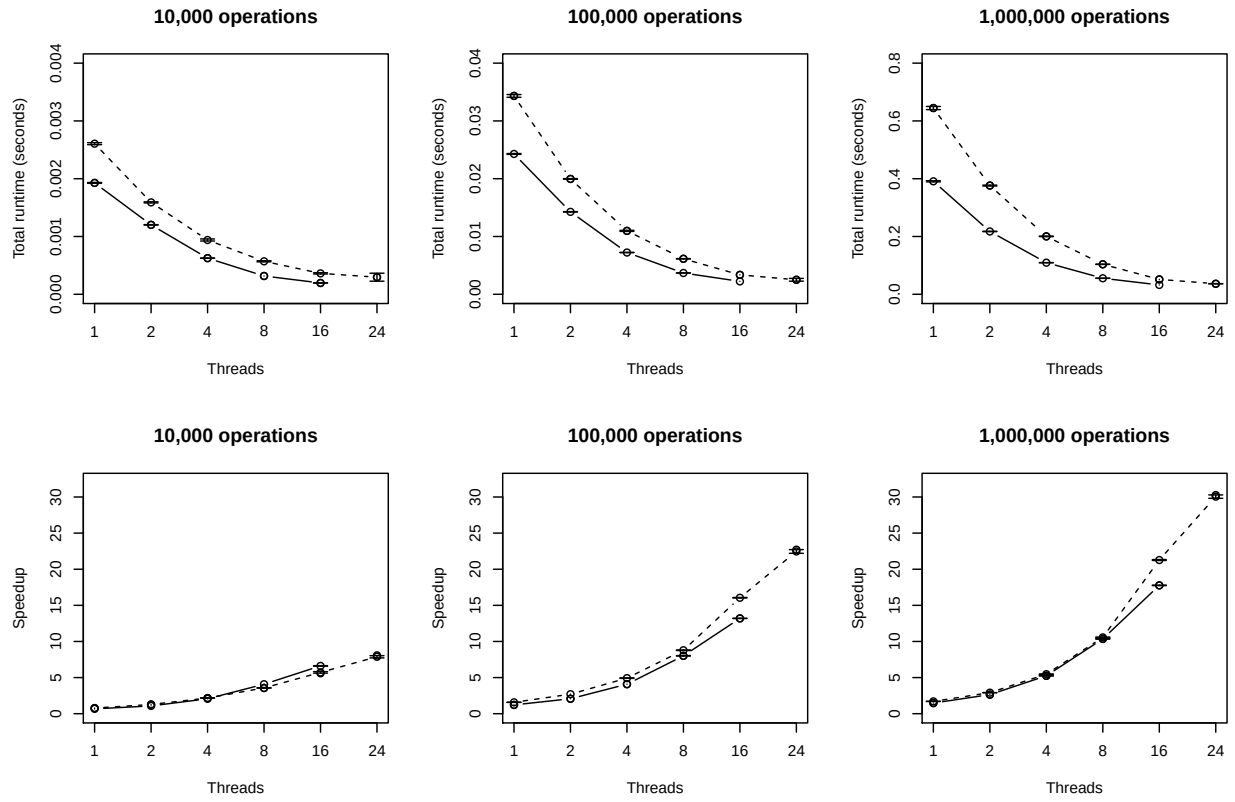

Figure 2: Dictionary runtime and speedup relative to libstdc++-v3 multimap. Solid line is from PC_A, dashed is PC_B.

improved performance for large trees occurs, as searching unordered leafs is less expensive, and cache locality is more important.

5 Conclusion

We have presented and evaluated ELB-trees, a new lock-free dictionary. ELB-trees are almost 30 times faster than sequential library implementations at 24 threads, corresponding to a 17.6 times speedup over the single threaded case.

References

[1] M. Herlihy. Wait-free synchronization. *TOPLAS '91*.

[2] L. F. Bonnichsen. Contention resistant non-blocking priority queues. MSc thesis 2012.

[3] F. Ellen *et al*. Non-blocking binary search trees. *PODC '10*.

[4] M. M. Michael. Hazard Pointers: Reclamation for Lock-Free Objects. *TOPLAS '04*.

[5] A. Braginsky, E. Petrank. A Lock-Free B+tree. *SPAA '12*.

[6] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM '64*.