Split-Ordered Lists: Lock-Free Extensible Hash Tables (a summary)

Gunnar Johansson

April 4, 2007

1 Introduction

The hash table is a common data structure for applications requiring constant time insert, delete and find of certain items. The basic idea is to use an array of buckets, where each bucket stores a list of items. A hash function is used to determine which bucket a certain item belongs to. Finding or deleting a specific item amounts to a linear search in a bucket list pointed to by the hash function, which can be considered a constant time operation if the list is small. When the number of items grows, constant time operations cannot be guaranteed, so the hash table needs to be extended. A common way of doing this, is to increase the size of the bucket array and redistribute the items among the new buckets.

In the context of concurrent programming, the operations find, delete and resize requires special care since several parallel threads may be accessing the data structure simultaneously. Previous work on concurrent hash tables have used lock-based schemes to allow safe operation on individual buckets, or sets of buckets. These implementations have shown reasonable performance, but they do not scale well for increasing number of concurrent accesses. Also, in order to maintain the average constant time complexity, the lock-based schemes tend to be very complex, especially for the resize operation.

2 Lock-free extensible hash tables

Building on the success of many lock-free data structures, especially lock-free linked lists, the authors in [Sha06] propose an elegant solution for dealing with the resize problem. They first note that the main problem with lock-free resizing/extension lies in the process of moving, or re-locating, items to new buckets. Current architectures support basic compare-and-swap (CAS) atomicity, which does not support atomic movement of even a single item. To circumvent this, the authors propose a solution based on the following idea; rather than *moving the items among the buckets*, they *move the buckets among the items*.

In practice, this means that instead of storing separate lists in each bucket, each bucket points to a *sub-list* in one *common list* as depicted in Figure 1. The list itself can be made lock-free by using existing implementations, meaning that the find, delete and insert operations should not introduce any concurrency problems. Now, if we construct a resize scheme which does not move the list items, but only direct additional bucket pointers into the list, we should be

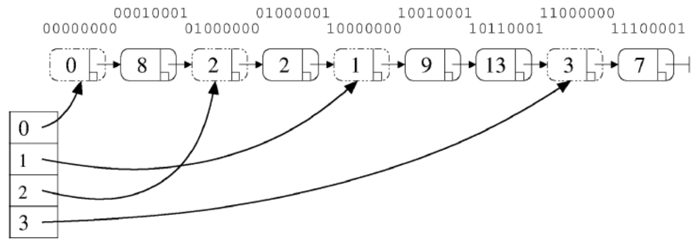

Figure 1: Split-ordered hash table

able to build a lock-free hash table. Note that the pointer assignments from bucket-to-list can be made atomic using CAS.

3 Recursive split-ordering

The key to success lies in the correct *ordering* of the list items. To simplify, the authors assume a modulo 2^i hash function and a hash table size of 2^i. For example, given a table of size 4, this means that key 13 will be in bucket 13 modulo $4 = 1$. Note that this is equivalent to saying that the i:th least significant bits (LSB) of the key determines the bucket. It turns out that the so called *recursive split-ordering* is achieved by binary reversal of the keys. For example, the split-order of key 13 is $\mathbf{reverse}(13_{10}) = \mathbf{reverse}(00001101_2) = 10110000_2$, as shown in Figure 1. For technical reasons, it is convenient to include a *dummy node* at the start of each sub-list. One can distinguish between dummy nodes and regular nodes by the LSB of the sort order.

Using this hash and sort strategy fulfills the requirements for our resize scheme as outlined in the previous section. That is, it performs the resize only by directing additional pointers into the list, and does not move any items. By studying the process of increasing the hash table in Figure 1 from size 2 to 4 you note that the list of items is *recursively* split among the 2 new buckets. This idea readily generalizes for new resize steps, each doubling the size of the table. Note also, that even if you have increased the size and recursively split parts of the list, you can still access items from higher levels of recursion. For example, key 2 in Figure 1 can be accessed both from bucket 0 and bucket 2. This means that parallel threads can operate on the hash table even if they work on different logical sizes, meaning that you have a completely concurrent resize.

4 Results

To validate the solution, the authors compared it to the currently most efficient lock-based counterpart of an extensible hash table. The first benchmark showed that the lock-based version reaches maximum throughput at 24 threads, thereafter it remains at a constant throughput level. At the same time the lock-free version has two times the amount of throughput at 24 threads, and continues to improve as the number of threads increase. This shows that the data structure

scales reasonably well up to 70 threads, albeit with higher deviation between measurements. The remainder of the benchmarks point out the same trend - under low levels of concurrency, the lock-based and lock-free data structures performs equally well. You note generally, that the lock-free version wins when you have more than 10 concurrent threads.

5 Conclusion

To summarize, the authors have presented a lock-free extensible hash table based on the idea of recursive split-ordering. The implementation is simple, and can be based on existing lock-free linked list data structures. The evaluation shows that their solution outperforms the current best lock-based counterpart under high levels of concurrency (more than 10 threads). Moreover, the recursive split-ordering can possibly also benefit a sequential hash table implementation.

References

[Sha06] Ori Shalev, Nir Shavit, "Split-Ordered Lists: Lock-Free Extensible Hash Tables", J. of the ACM 53(3), May 2006, pp. 379-405