# A Lock-Free Concurrent Hash Table Design for Effective Information Storage and Retrieval on Large Data Sets

Steven Feldman, Pierre LaBorde, and
University of Central Florida
Orlando, FL 32816
{Feldman, pierrelaborde}@knights.ucf.edu

Damian Dechev
*Scalable Computing R&D Department*
*Sandia National Laboratories, Livermore, CA 94550*
ddechev@sandia.gov

## Abstract

The purpose of this work is to develop a lock-free hash table that allows a large number of threads to concurrently insert, modify, or retrieve information. Lock-free or non-blocking designs alleviate the problems traditionally associated with lock-based designs, such as bottlenecks and thread safety. Using standard atomic operations provided by the hardware, the design is portable and therefore, applicable to embedded systems and supercomputers such as the Cray XMT. Real-world applications range from search-indexing to computer vision. Having written and tested the core functionality of the hash table, we plan to perform a formal validation using model checkers.

## 1. Introduction

The ISO C++ Standard [5] does not mention concurrency or thread-safety (though it's next revision, C++0x, will [1]). Nevertheless, ISO C++ is widely used for parallel and multi-threaded software. Developers writing such programs face challenges not known in sequential programming: notably to correctly manipulate data where multiple threads access it. Currently, the most common synchronization technique is to use mutual exclusion locks. A mutual exclusion lock guarantees thread-safety of a concurrent object by blocking all contending threads except the one holding the lock. This can seriously affect the performance of the system by diminishing its parallelism. The behavior of mutual exclusion locks can sometimes be optimized by using fine-grained locks or context-switching. However, the interdependence of processes implied by the use of locks -- even efficient locks -- introduces the dangers of deadlock, livelock, and priority inversion. To many systems, the problem with locks is one of difficulty of providing correctness more than one of performance.

The widespread use of multi-core architectures and the hardware support for multi-threading pose the challenge to develop practical and robust concurrent data structures. The main target of our design is to deliver good performance for such systems (see Performance Evaluation). The use of non-blocking (lock-free) techniques has been suggested to prevent the interdependence of the concurrent processes introduced by the application of locks [4]. By definition, a lock-free concurrent data structure guarantees that when multiple threads operate simultaneously on it, *some* thread will complete its task in a *finite* number of steps despite failures and waits experienced by other threads. The creation of a lock and wait free concurrent hash table will improve database performance due to the commonplace occurrence of multiple threads and users accessing the same tables at the same time.

This paper presents a design for a lock-free, wait-free, extendible hash table which avoids new array allocation through the use of a bounded number of indirections. The design includes dynamic hashing, meaning each element has a unique final and current position.

## 2. Design Principles

In designing the algorithm for the hash table we wanted to achieve a variety of design goals.

(a) Lock-Free/Non-Blocking: insure that no single thread can prevent another from accessing information.

(b) Wait-Free: guarantee that at least one thread is making progress at any point in time, which implies that all tasks will finish in a set amount of operations.

(c) Perfect Hashing: each element has a unique final position in the table, and this position lets the algorithm insert, find, or delete elements concurrently.

(d) Atomic Operations: available on all modern architectures, are the only operations used when modifying the table.

(e) Thread Death Safety: if a thread were to suddenly die, regardless of the point during its execution, no data would be lost, with the sole exception of the threads own operation.

## 3. Implementation

In order to achieve our design goals, we are developing an innovative approach to the structure of the hash table, inspired by Ori Shalev and Nir Shavit, Split-Ordered Lists: Lock-Free Extensible Hash Tables [6].

The hash function we use is a one-to-one hash, where each key produces a unique value. The hash function that we use reorders the bits in the key, promoting a more even distribution of keys. The length of the memory array used in the table is a power of two. By taking the first X bits of the hash, where X is a value such that $2^X$ is equal to the length of the memory array, we determine the location to place the key-value pair.

The hash table can be composed of multiple arrays, where a position in one points to another. However the total number of arrays is bounded by the key length and the size of each array. When referring to indirections we mean the number of entries one might need to check before a key is found. The maximum number of indirections caused by going from one array to another is equal to the number of bits in the key divided by X, where X is the number of bits taken from the key, as described earlier. An example for a 32-bit key with an array of length 64, would have at most seven indirections. Users could choose a much larger length for the memory array, which will further decrease the number of indirections caused by going from memory array to memory array.

In the event that the position that a keys hashes to is occupied, and no valid spot can be found whilst probing. Then a new memory must be made. When the new memory array is added to the table, then all elements will that belong at the position that points to this memory array, will be moved into it.

By allowing concurrent table expansion this structure is free from the overhead of an explicit resize that involves copying the whole table, thus facilitating concurrent operations. Moreover, some related algorithms, such as the implementations of Cliff Click [2] and that of Gao, Groote, and Hesselnik [3], allow stacked resizes; which can lead to starvation of one or more threads. Similar algorithms have been devised; however, their respective performances have not provided enough incentive for widespread adoption.

By allowing concurrent table expansion this structure is free from having to allocate space for a new table and copying over the information from the previous table, thus facilitating concurrent operations.

## 4. Operations

This section presents the generalized pseudo-code of the implementation and omits code necessary for a particular memory management scheme.

For the purpose of explaining the algorithm, keys will be written as (A-B-C...), where A is the position that the key belongs in on the top-most memory array, B is the memory array pointed to by position A in memory on top-most array, and so on. In addition the word ``local" will refer to the memory array that the thread is currently examining. It will, unless otherwise stated, always start as being the main array.

**Insert** There are several types of nodes. The two basic types of nodes are a ``spine node," which is a memory array that can consist of pointers to other spine nodes or data nodes, and a ``data node" which is a node that holds a key-value pair. There are also several other types used for special cases. When a thread requests that a node be deleted, the type of that node is changed from data Node to ``deleted node" or ``soft deleted node," depending on its location. An ``unreachable node" is a node that if searched for, by its key, cannot be found. The case wherein this can happen is generally rare, and it will be explained in detail later. A ``spine node," is initially a ``new spine node," which is used to allow other threads working in the same area to work concurrently. The reasons for changing between the states will be explained later in this section.

In order to insert a node, the key is hashed and the first X bits are retrieved from the hash value. Remember that X is such that $2^X$ equals the length of the memory array. Let ``significant node" be the node pointed to by position X on local. If it is a ``spine node," it will set local equal to ``significant node," and look at the next X bits, and get a new ``significant node." It will continue this until ``significant node" is a non-``spine node." If a ``new spine node" is reached then the thread will help finish creating the ``new spine node," which will be described in its own section. After this, the ``significant node" will no longer be a ``new spine node," and as such it will set local equal to ``significant node," look at the next X bits, and get the new ``significant node." At this point, if ``significant node" is ``null," ``deleted node," ``soft deleted node," ``unreachable," or a key match we will simply CAS ``significant node" for our node. If the CAS fails then we re-examine the ``significant node." If it passes, then we return out of the insert function.

**Creating a Spine Node** The process of creating a ``spine node" starts by the thread allocating a ``new spine node," placing the node currently at the location we want to make the spine, and if its node belongs at that location, its node as well. It then Compare-and-Swaps the current node for the ``spine node." If this CAS fails, and the new current node is a ``new spine node," then it will help create the spine node, returning the result of this. If it is a spine node, then it will return false, letting the thread that called this know that its node was not inserted.

**Find** The retrieval of a value using a key is a simpler process. Not only is it natural, that find is the simplest of the operations, but it is also necessary because find operations are the most pervasive in real-world applications of this kind of data structure. If the ``significant node" is a ``spine node," it will set ``local" equal to ``significant node," and look at the next X bits, and get a new ``significant node." It will continue this until ``significant node" is a non-``spine node."

In the event of a ``new spine node" there are two options. The first option is that the thread will help finish creating the ``new spine node." After this, the ``significant node" will no longer be a ``new spine node," and as such it will set local equal to ``significant node," look at the next X bits, and get the new ``significant node." The second option is that we will probe down, looking for a key match. If a key match is found, the thread will return out of the find method. If it is not found then it will check the ``new spine node." We chose the second option, because we felt it would improve performance of the find function, and the insert function will have less contention on the CAS operation.

If ``significant node" is a key match, then it will simply return the value. However if it is not a key match, the the thread will probe the memory array, until a key match is found or it has circled back to the start position. If ``significant node" is ``null," ``deleted," ``unreachable," or not at its proper position, then the thread returns false, because we have the guarantee that if there is a node that is not at its proper position then the node that is at our node's proper position is at its proper position. Furthermore, if a node is to be deleted, and it is at its proper position, then its type will be changed to ``soft deleted node." This will be explained in depth in the section on the deletion process.

**Deletion** The delete method is similar to the find method.

**Cleanup** It is possible for a thread to insert a value into the table, and have that thread find a valid spot before it reaches a key match; which could lead to inconsistencies. In order to prevent this, we implement a bit that indicates to other threads whether or not this node is ``in clean up." ``Clean up" is the procedure of probing down from the position that the node was inserted at, until it reaches the proper position, or the bit has been changed to false, all the while removing key matches.

## References

[1] P. Becker. Working Draft, Standard for Programming Language C++, ISO WG21N2009, April 2006. URL http://www.open-std.org/JTC1/SC22/WG21/.

[2] C. Click. A lock-free hash table, 2007.

[3] H. Gao, J. F. Groote, and W. H. Hesselink. Almost wait-free resizable hashtables. In Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, page 50. IEEE, 2004. ISBN 0-7695-2132-0. doi: http://dx.doi.org/10.1109/IPDPS.2004.1302969.

[4] M. Herlihy. A methodology for implementing highly concurrent data objects. ACM Trans. Program. Lang. Syst., 15(5):745{770, 1993. ISSN 0164-0925.

[5] ISO/IEC 14882 International Standard. Programming languages C++. American National Standards Institute, September 1998.

[6] O. Shalev and N. Shavit. Split-ordered lists: lock-free extensible hash tables. In PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing, pages 102{111, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-708-7. doi: http://doi.acm.org/10.1145/872035.872049.