Abstract

For a long time programs with multiple threads of executions have been synonymous with programs relying on synchronization primitives based on mutual exclusion, and thereby locking [4].

Recently desktop computers have been getting multi-core CPUs and multi CPUs, and the focus in the desktop environment has started shifting from utilizing one processor to it's fullest, to how more processors are shared optimally.

Algorithms relying on mutual exclusion are not able to run in parallel; since using a lock makes sure only one thread has access to a given locked object at a time, and thereby forcing sequential execution. If the threads executing constantly has to access common variables, and thereby risk colliding, using locks makes sense since execution cannot be in parallel. In cases where there is a good chance for algorithms to run in parallel, the number of necessary locks that do prevent race conditions might be small. When the necessary number of locks is small, the unnecessary locks are an overhead. If this overhead could be eliminated, the performance could be greatly improved. Using non-blocking algorithms is one way to go.

Some research has gone into developing and proving non-blocking algorithms, and some research has gone into experimenting with the performance of non-blocking algorithms on large scale multithreaded multi processor systems [3] [6] [7] [10]. The questions remains: *Will these algorithms really show improved performance compared to their lock-based variants on desktop computers?*

This paper will perform a performance experiment using an abstract data type implemented as both a lock-based- and non-blocking algorithm. The experiment will show which particular implementation of the selected algorithm shows the best performance. The result of the experiment can be used as guidelines for how to choose between implementing a lock-based or non-blocking variant of an algorithm, at least on the architecture the experiment is performed on.

Table of contents

1 Introduction

This paper will focus on performance differences between lock-based and non-blocking algorithms. Non-blocking algorithms is a relatively new field compared to the lock-based algorithms, and I have only been able to find a few articles on the topic. The articles use a set of terms, but not always with the exact same meaning. The articles I found were mostly focused on developing a non-blocking algorithm, or measuring performance of a data structure I could not believe would perform better in a non-blocking form. When being interested in the performance of the lock-based- versus non-blocking algorithms, and by reading the works of other authors on the topic, the development of a hypothesis started.

In this paper I will briefly introduce the terms I choose to use regarding non-blocking algorithms. The terms I choose are also chosen by the authors of the articles I have read, that have been published since Herlihy et al. published their article [6] in 2003.

After the introduction of the terms, the problems of the two types of multithreading algorithms will be described. That knowledge will be used to formulate a hypothesis, and from the hypothesis an experiment is planned, designed and conducted. The results of this experiment are then analyzed, to either prove the hypothesis or prove it wrong.

1.1 What are lock-based algorithms?

Lock-based algorithms are algorithms relying on mutual exclusion to avoid race conditions. Shared resources used by the algorithms are surrounded by locks, that only allows one thread of executions to progress at a time, so any access to shared resources are conducted in sequence and not in parallel. Lock-based algorithms based on mutexes can be changed to support multiple readers and a single writer by replacing mutexes with shared/exclusive locks.

1.2 What are non-blocking algorithms?

Non-blocking algorithms do not rely on mutual exclusion to avoid race conditions, but uses atomic operations as test_and_set() and compare_and_swap() to test if a race condition occurred. If a race condition did occur the operation will be repeated.

Non-blocking algorithms is used as a broad term describing all algorithms where killing a thread does not result in other threads being able to progress. Any algorithms using locks, spin-locks or similar mechanisms are therefore not non-blocking.

Non-blocking algorithms comes in three variants: Wait-free, lock-free and obstruction-free. The different categories describe what progress algorithms of that category promises to deliver [6].

Obstruction freedom is the lightest guarantee and does guarantee to be deadlock free. Further more it guarantees that any thread running in isolation will be able to complete its operation in a known number of steps [6].

Lock-freedom guarantees no deadlock and no livelock. Every step taken by the algorithm achieves global progress. Lock-freedom can prevent livelock by implementing a helper mechanism. If thread 2 is obstructed by thread 1, thread 2 helps thread 1 finish, and can then carry out its own operation [6].

Wait-freedom is the strongest guarantee of progress; it guarantees no deadlock, no livelock, no starvation and guarantees that every active process will progress in a bounded number of steps [2]. Since each process will progress in a bounded number of steps, there can be no retry loops causing a livelock. If each operation must complete within a bounded number of steps, the resources to carry out an operation must be readily available to avoid starvation.

The stronger the guarantees the algorithm provides, the more careful it has to be designed. Herlihy concluded that introducing obstruction freedom as a guarantee and adhering to it makes the design of non-blocking algorithms easier and makes the algorithms more efficient in typical case low contention situations [6].

1.2.1 The ABA Problem

All non-blocking algorithms that rely on compare_and_swap() are susceptible to the ABA problem. Compare_and_swaps() will by comparing with a previous value, make sure that the value being written has not changed since the original value was obtained. Compare_and_swap() relies on an old value and a new value to check if a swap is valid. The ABA problem covers problem covers problems where the value is modified and later re-modified to a value that can be misinterpreted by another thread of execution. Given the existence of two threads 1 and 2, thread 1 reads a value A from a variable and is switched out. Thread 2 then reads the same variable and writes B into that variable, and changes it back to A. When thread 1 is swapped in again it will do a compare_and_swap() and it will succeed. Using compare_and_swap() does not provide guarantees that data has not been changed, just that the value is the same.

Compare_and_swap() works by taking 3 parameters, an “old value”, a “new value” and a pointer to the integer that needs to be changed.

Compare_and_swap() is carried out as an atomic operation and it succeeds if the value of integer pointed to had the value “old value”. If the value was “old value”, the “new value” is written to the integer, else nothing is written. Algorithms relying on compare_and_swap() will believe that if compare_and_swap() succeeds, their operation succeeded, this however might not be the case.

Making sure the values that are being swapped have not been referenced in the mean time can solve the ABA problem. One way to obtain this is to use load-linked/store conditional instead of compare_and_swap(). Load-linked will read the value from a memory cell and will make sure that store conditional will not succeed, if that memory cell has been modified. Before I start designing algorithms I will describe and analyze ways to solve the ABA problem in the section called [Examining solutions for the ABA problem]. The design phase will then be able to build on the solutions.

1.3 A reason for considering non-blocking algorithms

Standard desktop computers are getting multi-core processors, and some desktop computers have more than one multi-core processor. Desktop computers are moving toward being yesterday’s super computer. By gaining the power of yesterdays super computer, the theory behind algorithms on yesterday’s super computers, applies to today’s desktop computers.

The focus of multithreaded algorithms has been on detecting the critical sections of the code, and making sure no more than one thread entered at a time. The construction of non-blocking algorithms shares the same focus on the critical sections, but instead of aiming for serialization of the critical sections, they aim for parallelization. The construction of non-blocking algorithms has been viewed as a black art, and by many the construction of lock-based algorithms has seemed easier. Having too coarse-grained locking results in serialization of an algorithm, and having serialized execution, the multiple threads of execution will never run in parallel. Trading in coarse-grained locking for a fine-grained locking increases the complexity of the locking algorithm, and the risk of common mutual exclusion pitfalls rise see [Appendix A].

There are multiple reasons for choosing to use threads to implement an algorithm. If the algorithm is very CPU bound, the reason to choose to use more than one thread of execution can be to achieve better performance. If the algorithm is I/O bound, the multiple threads can allow processing while waiting on I/O or allow multiple I/O requests to be performed in parallel without having to resort to asynchronous I/O. If performance is the main concern, the use of non-blocking algorithms should be considered. Non-blocking algorithms will never block thread execution, a thread is only off duty when the operating systems scheduler has told it to be. The summed up theory behind non-blocking algorithms is, that locking is more expensive than detecting collisions and retrying.

1.3.1 Why would locking be more expensive than detecting the collision and retrying?

A possible cause could be the cost of obtaining and releasing locks. The obtaining and releasing of locks do cause an overhead, if obtaining a lock causes a thread to wait; the locking causes a thread context switch. Another possible cause is if threads are rarely requesting access to the same resources at the same time then most of the locking occurring is unnecessary. If on the other hand threads do want the same resource constantly, there really is no performance benefit from threading since only one thread will be running at a time.

Non-blocking algorithms use atomic operations to ensure safe operations in multithreaded environments, while lock-based algorithms marks executable sequences as being locked or only to be carried out atomically. Both types of algorithms aims at protecting shared data by accessing and modifying it atomically. Using locks the programmer decide how large chunks of code should be considered atomic, if the non-blocking approach is chosen a few simple atomic operations are available. The size of the atomic chunks in a lock-based algorithm defines the lock granularity. If the lock granularity is low most of the algorithm cannot be carried out in parallel, if the granularity is high the opportunities for running in parallel rise at the cost of the additional locks and the complexity of the algorithm.

When the lock granularity is decided, it has an impact on the algorithms ability to run in parallel. A linked list could have an integer representing the number of elements. If a lock protects the element count, the lock granularity is high, but the cost of a mutex compared to the cost of an atomic operation used by lock-free algorithms is expensive. Inserting and removing from the linked list would require the number of elements to change, and thereby access via the lock, so ultimately there is only one writer at a time. Mutexes and semaphores can be implemented using atomic operations on integers and a wait queue (Linux 2.6). If a mutex is used to make incrementing and decrementing the number of elements in a linked list an atomic operation, another atomic operation is typically used to decrement and increment a semaphore, and add a thread to a waiting list, making a thread context switch, waking up a sleeping thread. It will potentially be many processor cycles wasted to do a fairly simple task. The problem is these processor cycles probably cannot be reclaimed by running in parallel, since the purpose of the locks is to prevent task being carried out in parallel. Using a lock-based approach, it is hard to get the same granularity as in lock-free algorithms. Settling for a lower granularity means that locks will be held for longer periods, and threads are not able to run in parallel while the locks are held.

A possible improvement of lock-based algorithms is to let the algorithm detect when a lock is necessary, and only lock a resource when it really is necessary. A lock is necessary multiple threads of execution are dependant on their order of execution, a situation named a race condition [2]. Detecting when a lock is necessary would require knowledge of future events, and is therefore still impossible, but it is possible to detect when a lock would have been necessary.

A technique for detecting when a lock would have been necessary has been a known practice in database systems for a long time. Optimistic locking (optimistic concurrency control) or pessimistic locking has been heavily debated in database circles. Optimistic locking has the same characteristics as non-blocking algorithms, they try to execute their transactions and test if they where successful [1]. If a transaction is unsuccessful, it is restarted. Restarting a transaction can be expensive, depending on the contents of the transaction. The success of Optimistic locking depends on the cost of restarting transactions. The cost can be affordable when collisions are rare, or transaction can be redone quickly. Pessimistic locking makes sure that the transaction can run in isolation without being disturbed by other transactions. The pessimistic locking of database systems resembles lock-based algorithms. They have an extra cost involving taking a lock and releasing a lock, and the problems of making sure deadlocks do not occur. Pessimistic locking has the advantage that it can be better at conserving resources than optimistic locking. The cost of trying and failing affects not only the transaction or process retrying, but all other transactions and processes.

Detecting which locks would have been necessary is the same as detecting when threads have collided when accessing a shared resource. If the cost of detecting threads colliding is as low or lower than the cost of obtaining a lock, and the cost of retrying the operation is as low or lower than the cost of a lock wait, it is quite plausible that non-blocking algorithms indeed are faster than lock-based algorithms. There can however be a problem with the number of collisions. If collisions are very frequent, non-blocking threads can work against each other, constantly spoiling each other's work, causing all threads to retry and possibly collide again. This phenomenon is known as a livelock and is very similar to a deadlock, except that threads participating in a livelock are actively trying to progress or get out of the way so other threads can progress [2]. The non-blocking algorithms that are susceptible to livelock are the algorithms that only guarantee obstruction freedom. Lock-free and wait-free non-blocking algorithms guarantee no livelock.

I have summarized the statement in the paragraph above to my hypothesis.

Hypothesis: an algorithm with a small probability of threads colliding causing race conditions would be better of running without locks performance wise, while an algorithm with high probability of threads colliding would benefit from running lock-based.

Maged M. Michael and Michael L. Scott concludes in their paper “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms” [3], that queues using blocking are faster than non-blocking queues. Their conclusion suggests that blocking is faster when threads collide often. An insert in a queue requires modification to one end of the queue, while a delete from a queue requires access to the other end. Multiple threads trying to either insert or delete in the queue will collide. The question is how an algorithm where thread collisions are unlikely will perform.

2 Planning the experiment

In order to investigate the differences in performance of lock-based and non-blocking algorithms I will conduct an experiment that will show how similar algorithms implemented as lock-based and non-blocking will perform. I find it interesting to explore how these algorithms behave in a normal desktop computer environment on modern multi CPU machines. If they do provide better performance, a change of focus in the education of multithreaded programming might be considered.

It is often suggested that creating multithreaded applications is hard, and difficult to verify [4]. There has been a tendency to move shared data into thread safe containers, in order to have a set of building blocks that would simplify the creation of error free multithreaded applications. Picking an algorithm for a container therefore seems obvious.

Maged M. Michael found that in the case of queues choosing a blocking variant is faster [3]. Choosing a stack would, if my hypothesis is correct, perform better as a blocking implementation. A stack has much resemblance to a queue, except that insertion and removal will occur from the same end of the stack. Taking what Maged M. Michael found, my hypothesis and applying that to a stack would allow one push or one pop occurring at the same time, meaning stacks will not perform well run in parallel. If the stack will not allow parallel executions, the benefits of running on multi CPU's are non existing.

A container algorithm that is better for examine the part of the hypothesis that favors optimistic locking would be a vector or a hash table. A vector will allow random access to its data, and hash table uses a vector to store its elements. In the case of perfect hashing, the hash table has random access as the vector, but in other hash table implementations the hash collisions must be dealt with. A hash table using chaining is implemented via a vector of linked lists. Having a vector as entry point to the top of the list makes it a good data structures to use for this experiment. Threads can collide when they enter the same hash collision list, but depending on the number of threads, the size of the vector used and the efficiency of the hash function the collisions can become more or less likely.

The progress guarantee supported by the non-blocking algorithm should be obstruction freedom sin the aim is to create an experiment with an algorithm where race conditions seldom occur. Therefore the probability of a livelock should be infinitesimal. The design of the non-blocking algorithm should make the cost of a restart insignificant, and thereby further reduce the chance of a livelock. If a thread can restart and use most of its calculations in the retry, there is not much wasted in retrying, and the risk that the operation fails once more is infinitesimal.

2.1 Defining a case for studying

A hash-table is a data structure that trades in memory for speed. It is usually chosen for its ability to do fast lookups. A hash-table can be implemented as vector of lists. Each item inserted into the hash-table must have a key. This key is used to calculate an elements index in the array via a hash function. A hash table goal is to provide O(1) lookup, but this depends on the quality of a hash function. If a hash function generates the same index for multiple keys, the keys can either be stored in that index, or rehashed with another function. Worst case lookup for a hash table where collisions are kept as a linked list is O(n), worst case for rehashing lookup can be more than O(n) depending on the hashing functions available(hashing functions can hit the same indexes).

Hash-tables can be implemented as a set or as a multiset, as a map or as a multimap. A set or a map stores unique items and multiset and multimap allows storing multiple items with the same key. The difference between a map and a set is that a map stores a key and an item, while a set uses the item as key and only stores items.

The vector of a hash-table contains a number of hash-buckets. A hash-bucket is a collection of items stored under a certain index. The hash-table design we will examine further is a vector where the hash-buckets contain a chained lists items that have collided and ended up in the same bucket. There are other ways to deal with hash collisions, like inserting in the next empty bucket, using another hash function to place the item or adding a new hash table in the bucket. If items can be placed multiple places in he vector, lookup can be costly performance wise. And items placed out of order can cause other items to be less optimally placed. Keeping the hash collisions in a linked list makes sure that the lookup will be O(n) on the list and not O(n) on the vector. If the items are evenly distributed in the buckets, O(n) on the list or on the vector matters for a vector size of 64k.
A hash table typically provides operations for insertion, lookup and deletion of items. The keys of a hash-table are unique. Other typical operations can be returning the number of items in the table, a "contains operation" returning if a key exists in the hash-table and possibly a grow operation that will extend the hash-table and rehash all the keys.

```
void insert(key,item)
bool lookup(key,item&)
void delete(key)
unsigned int count()
bool contains(key)
void grow()
```

Figure 1 Typical interface for a hash-table

Examining the performance of a non-blocking and a lock-based hash-table requires two implementations based on the same general algorithm. Therefore a simple single-threaded hash-table will serve as point of reference for the two different multi-threaded algorithms. This approach has been chosen in order to get the most comparable algorithms.

The next chapter will be about designing the three different algorithms.

3 Designing the algorithms

3.1 Design of a single-threaded hash-table

Showing and explaining the data structure will introduce the design for the hash table. Pseudo-code for the insert, lookup and delete operations will be build upon the explained data structure.

The count, contains and grow operations are not shown as pseudo-code, and will not be part of the experiment. Neither of the operations contributes to understanding the difference between the two different approaches to multithreaded algorithms.

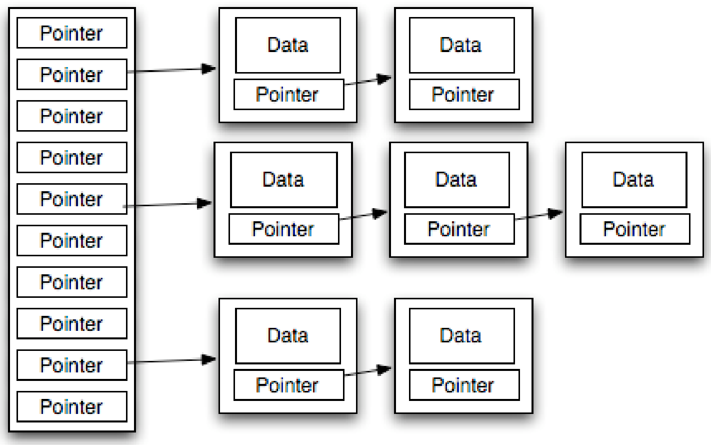


Figure 2 Hash-table overview

Figure 2 Hash-table overview shows the data structures that will be used for the pseudo-code. The table shows a hash-table consisting of pointers to buckets, each bucket can refer to another bucket.

The hash table contains a vector of pointers to the different hash-buckets. Each hash-bucket is a linked list of items with data and a pointer to the next element in the list. The linked list can pose a performance problem if it gets large, but if the lists get large it is not an algorithm problem. The problem is perhaps with the size of the hash vector. The hash vector should be the size of the maximum number of elements in the hash table divided by the maximum number of entries wanted in each bucket. The problem can also be the hash functions ability to distribute the items evenly between the hash buckets. The optimal sizes for the vector and for the hash vector depends on the data set it should hold, therefore a good implementation of a Hash-table will allow both to be selected by the user.

3.1.1 The interface

Looking at code for the hash-table we start by defining the interface. The only change to the interface shown above is a list of template types and the name of the variables used. The interface for the consists of an insert, remove and a lookup method as shown previously. The user can choose the type of Key used for calculating the hash, the Value the container should store, the hash-function used and the size of the vector.

```
template <     class Key ,
               class Value   >

struct Item {
    Value                  value_;
    Key                    key_;
    Item*                  next_;
};

template <     class Key,
               class Value,
               unsigned int (*hashFunctionType)(const Key&),
               unsigned int TableSize=1024>

class HashTable {
public:
    typedef Item<Key,Value> ItemType;

    HashTable();
    ~HashTable();

    bool    insert(const Key& key, const Value& value);
    bool    remove(const Key& key);
    bool    lookup(const Key& key,Value& outvalue);

 private:
    unsigned int    hashTableIndex(const Key& key);
    ItemType*       items_[TableSize];
    unsigned int    itemcount_;
};
```

Figure 3 - Hash table interface

3.1.1.1 Constructor/destructor

The constructor of the HashTable class needs to initiliaze the items_ array with zeroes, and initialize the itemcount_ to zero. The destructor of the HashTable must delete the hash-buckets.

3.1.1.2 Private members

The private function hashTableIndex, needs to call the hashFunctionType supplied as template parameter and make sure the index is within the vectors bounds. To simplify the bounds checking the TableSize is only allowed to be a size that is a power of two. When TableSize is kept as a power of two a bitwise and with TableSize-1 will keep indexes in bounds.

```
unsigned int hashTableIndex(const Key& key) {
    return hashFunctionType(key) & TableSize-1;
};
```

Figure 4 - Hashtable index function

3.1.2 Design of Insert, Lookup and Remove

Having decided on the interface for the class, we can design the pseudo-code for the insert-, remove- and lookup methods. The design of these three methods have a great impact on the design of the multithreaded versions, so this design will not lack any details. The design in form of prose text is accompanied by a very code like pseudo-code where references to are marked by [I] for insert [L] for lookup and [R] for remove. Each reference contains a letter, a line number and a colon. This is done to make the references look different from the ones appearing in the list of references.

3.1.2.1 Insert method

The purpose of the insert method is to insert a value with a unique key in the hash-table. Insert needs to make sure the key suggested is unique [I3:] and this requires a search O(n) through the hash-bucket. If the key is not found in the hash-bucket [I5:] it is considered unique, and the insert function can

```
bool insert( const Key& key,const Value& value ) {
    unsigned int ix = hashTableIndex(key);   // I1:find index
    ItemType*   item = items_[ix];           // I2:find hash-table bucket
    while ( item != 0 && item->key_ != key ) {//I3: check if key is in bucket
        item=item->next_;                    // I4:
    if (items_[ix] && item!=0)               // I5: is key found?
        return false;                        // I6: key was not unique so we exit

    ItemType*   newItemPtr = new ItemType;   // I7:create a new item
    newItemPtr->key_ = key;                  // I8:assign key
    newItemPtr->value_ = value;              // I9:assign value
    newItemPtr->next_ = items_[ix];          // I10:insert it at the top
    items_[ix] = newItemPtr;                 // I11:replace top pointer
     ++itemcount_;                           // I12:increment #items in table
    return true;                             // I13:return success
}
```

create a new node [I7:] and insert it in the list [I11:].

3.1.2.2 Lookup method

The purpose of Lookup is to find a value by a key. If the methods finds a value [L3:]-[L5:], the value is copied to the out parameter value [L6:]. The methods returns true on success [L8:] and false on failure[L7:].

The lookup method resembles the first part of insert, where insert makes sure the key it wants to insert is unique.

```
bool lookup( const Key& key, Value& value ) {
    unsigned int ix = hashTableIndex(key);   // L1:find index
    ItemType*   item = items_[ix];           // L2:find hash-table bucket
    while ( item != 0 && item->key_ != key ) {//L3: check if key is in bucket
        item=item->next_;                    // L4:
    if (item!=0)                             // L5: is key found?
        value=item->value_;                  // L6: copy value
    else
        return false;                        // L7: key was not unique so we exit
    return true;                             // L8:return success
}
```

Figure 5 - insert method

Figure 6 - Lookup method

3.1.2.3 Remove method

The remove method resembles the lookup method, but [R02:] keeps references to an items a previous object [R05:], so it can unlink [R09:] [R10:] the item and delete it [R11:].

The remove method returns true on successful removal of a key, and false in case the key given, as a parameter does not reference an item in the hash-table [R14:].

```
bool    remove(        const Key& key ) {
    unsigned int ix = hashTableIndex(key);      //R01: calculate index
    ItemType*   prev = 0;                       //R02: previous node = 0
    ItemType*   item = items_[ix];              //R03: item = list top
    while ( item != 0 && item->key_ != key ) {  //R04: find item in has bucket
        prev=item;                              //R05: previous=item
        item=item->next_;                       //R06: item = next item
    }
    if (item) {                                 //R07: item found?
        if (prev) {                             //R08: was item at top?
            prev->next_=prev->next_->next_;     //R09: in list unlink item
        } else {
            items_[ix] = items_[ix]->next_;     //R10: unlink item at top
        }
        delete item;                            //R11: delete the item
        --itemcount_;                           //R12: decrement itemcount
        return true;                            //R13: remove was successful
    }
    return false;                               //R14: item was not found
}
```

Figure 7 - Remove method

Having decided on the detailed layout of the methods and the interface for the class, the design and implementation of the multithreaded variants is next.

The most straightforward to port is the multithreaded lock-based algorithm, therefore we will look at that next.

3.2 Design of a multi-threaded lock-based hash-table

Looking at the class interface from Figure 3 there are two instance variables that needs protection. The variables items_ array and the itemcount_ are both written to. A mutex securing the whole items_ array comes at a cost, that multiple operations cannot occur concurrently. If on the other hand, each bucket is protected by a mutex, multiple threads can concurrently insert, remove and lookup in different hash buckets, but are limited to one per hash-bucket. It is possible to allow multiple readers and a single writer using a readers/writer lock. But the lock is more expensive and requires the use of a mutex and a semaphore. The choice is between allowing multiple readers in the same hash-bucket and adding a cost to each lock, or allowing only one in each bucket and keeping the locking the cheapest possible.

The test for these algorithms will focus on performance, and have one thread plus one thread per CPU. Since we allow multiple threads to access the hash-table, it is desired that the number of times threads try to access the same hash-bucket is infinitesimal. So for the test we rely on the hash-function to distribute the indexes for the data. The test should aim at showing good performance, so the test data should also be selected so threads will not be forced to run in convoy [Convoying].

The interface to the multithreaded implementation bears much resemblance to the single-threaded version, except for the mutex protecting the item count and the array of mutexes protecting the hash-buckets. Between the mutex array and the item array there is a one to one correspondence.

```
template <     class Key,
               class Value,
               unsigned int (*hashFunc)(const Key&) ,
               unsigned int TableSize=1024>

class HashTable {
public:
    typedef Item<Key,Value> ItemType;

    HashTable() {
      for (unsigned int ix=0;ix<TableSize; ++ix)
        items_[ix]=0;
    }
      ~HashTable() {
    }
    bool    insert(const Key& key, const Value& value);
    bool    remove(const Key& key);
    bool    lookup(const Key& key,Value& outvalue);
 private:
    unsigned int    hashTableIndex(const Key& key);
    ItemType*       items_[TableSize];
    Mutex           mutexArray_[TableSize];
    unsigned int    itemcount_;
    Mutex           itemCountMutex_;
};
```

Figure 8 - Interface to the lock-based hash-table

Looking at the interface, it is obvious that construction and destruction is not thread-safe in any way. This means if the HashTable class is used, the client of the HashTable class must secure the construction and destruction.

The Item class has no changes in the lock-based version, since there is no lock on individual items; they are locked using the locks of the hash-buckets.

To avoid deadlocks, the insert, remove and lookup methods will be allowed to take a maximum of two locks, one lock in the mutex array, and one lock for the item count. The locks must be obtained in a predefined order to avoid deadlocks [2].

```
bool    insert(     const Key& key,
                    const Value& value ) {
    unsigned int ix = hashTableIndex(key);           // I01: find index based on key
    MutexLocker rowMutex(mutexArray_[ix]);           // I02: uses a stack-based
                                        //object to lock and unlock hash-bucket

    ItemType*   item = items_[ix];                   // I03: access hash-bucket
    while (item != 0 && item->key_ != key) {         // I04: search for key
        item=item->next_;                            // I05: next item..
    }

    if (items_[ix] && item!=0)                       // I06: was key found?
        return false;                                // I07: no key found

    ItemType*   newItemPtr = new ItemType;           // I08: create new item
    newItemPtr->key_ = key;                          // I09: assign key
    newItemPtr->value_ = value;                      // I10: assign value
    newItemPtr->next_ = items_[ix];                  // I11: newItem's next is top
    items_[ix] = newItemPtr;                         // I12: newItem is top

    MutexLocker itemCountMutex(itemCountMutex_);     // I13: lock itemcount
    ++itemcount_;                                    // I14: increase itemcount
    return true;                                     // I15: success at inserting
}
```

Figure 9 - Insert method in lock-based implementation

Line `[I02:]` and `[I13:]` is special for the lock-based implementation. Locks are obtained using RAII idiom (Resource Acquisition Is Initialization), to be exception safe [5]. Those lines are the only lines different from the single threaded version. The lock-based versions of lookup and remove are equally simple. The lookup method has one MutexLocker instance to lock the Hash-bucket being searched, and the remove method has two MutexLocker instances like the insert method.

The itemcount will not be correct when is not locked before doing the actual insert, but itemcount will quickly become a bottle-neck if that lock is held during the entire insert and remove methods. The consequence of moving the item count mutex to the bottom is that more the item count can be a bit off. The number can be off by the number of threads waiting on the item count mutex.

3.3 Design of a multi-threaded non-blocking hash-table

When designing a non-blocking algorithm it is necessary to decide what kind of guarantee is required. The purpose of this algorithm is to be a competitor to a lock-based algorithm, so the code needs to be simple more than it needs to guarantee a maximum number of cycles spend per method. Herlihy suggested that obstruction freedom is a well performing guarantee in low contention situations. The algorithm to be designed is supposed to work on a data structure where threads seldom clash. Based on the conclusions of Herlihy et al. I choose to implement a non-blocking algorithm that will only guarantee obstruction-freedom [6].

Memory management in non-blocking algorithms can be a challenge. One thread cannot block access to an allocated memory block, so it is not trivial to determine when a block of memory can be freed. The algorithm will be implemented in C++ and in C++ there is no build in garbage collector. One way to implement garbage collection in C++ is by using reference counting, and even though Maged M. Michael warns about the performance problems, I see it as a good option [7].

Reference counting can be implemented on multiple levels, just like locks can be implemented on multiple levels. The more fine-grained the reference counting is, the more flexible and expensive it is, because of the increasing number of counters. If a more coarse-grained reference counting scheme is chosen, the cost of reference counting goes down but you trade in memory for performance. The longer the algorithm must keep dead objects around, the more memory it will consume. There is another cost of making the reference cost more coarse-grained and that is dead objects must be marked or kept in a list, so they can be found and collected when the reference count reaches zero.

This algorithm will use a dead object list called delete list. The advantage is that objects can be torn out of their lists and inserted in a dead object list on removal, so new threads cannot reach these objects, but threads already working on these objects will be able to continue their work.

When an object is unreachable to new threads it is dead. Using a delete list creates a situation where old threads working on a deleted object potentially can keep it alive forever. This means that the data structure looks a bit different depending on which thread is looking. Figure 10 shows an object inserted in a delete list wrapped in another delete list item, so all pointers are kept as they where when the object was alive. This is a temporary inconsistency and it might mean that this algorithm is not useable in all situations. The view of the data structure is consistent again as soon as all threads have dropped their references to the item in the dead object list.

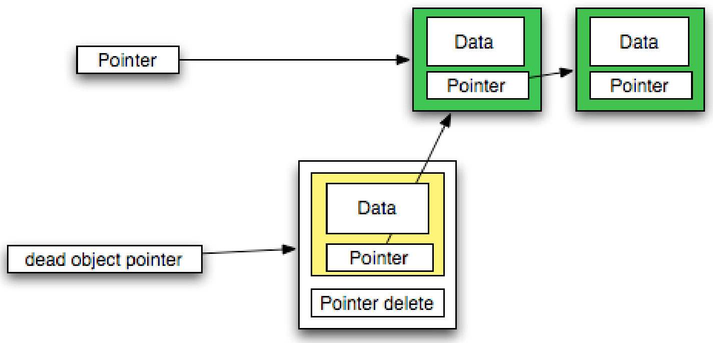


Figure 10 - A list with a dead objects list showing how dead objects are only dead to some threads

3.3.1 Examining solutions for the ABA problem

Before we can start designing and testing algorithms, it is necessary to examine the ABA problem and try to come up with a solution for it.

3.3.1.1 Method 1: Solving the ABA problem by counting references

A common method is to use a part of the value being swapped as a reference count. Compare_and_swap() works with 32 bit integers, 32 bits is the usual size of a pointer. When reference-counting 32 bit integers a pointer cannot be contained, so pointers cannot be used directly. A common method is to use indexes instead of pointers, and using the top 8-bits as reference. Another method could be aligning your storage to 16-bit or 32-bit boundaries, and using the 3-4 lower bits as counters. 3-4 bits is not that much, 8-bits makes the ABA problem more unlikely, but none of these methods solves it.

Adding a counter to each pointer is not from doing actual reference counting. It adds the cost of replacing pointers with indexes, incrementing a counter, shifting it and combining it with an index before each compare_and_swap().

Regular reference counting counts how many references an object currently has, and requires use of incrementing and decrementing a counter.

3.3.1.2 Method 2: Solving the ABA problem using reference-counting

To solve the ABA problem with pointers, it is necessary to make sure that the address of any element that is deleted will not be able to be recycled and reused as a new element, before all threads accessing the element are done with their operation. If reference counting is used, and everyone accessing an element must hold a reference to the element, the element can be reused when no references to it exist.

That is a bold statement that needs a bit of clarification:

Before any thread can experience the ABA problem they need to have a pointer to an object that they will later try to change. So before they can change anything they take this reference. If they hold this reference until they do want to change the items themselves, it is visible to the algorithm that it is not safe to delete that pointer. The typical way for the pointer to be deleted is, it is put in a state where threads accessing can carry on doing so until the last reference to the pointer is gone, then it is safe to delete it. When it is safe to delete it, there are no references to it, and there can be no ABA problem.

Reference counting can be expensive, but just as locking, reference counting gets more expensive the more fine-grained it is. If each element in a linked list is reference counted it is much more expensive to traverse the list, than if the list itself is reference counted. In the example I will show, I have used a very coarse-grained reference counting since the algorithm in the example would not benefit from a more fine-grained reference count.

Example: If we have a simple list of elements with a number of threads performing operations on it. The list is reference counted meaning, that if a thread is using that list, the thread has one reference to the list. So there is not a reference count of each item in the list displayed in Figure 11.

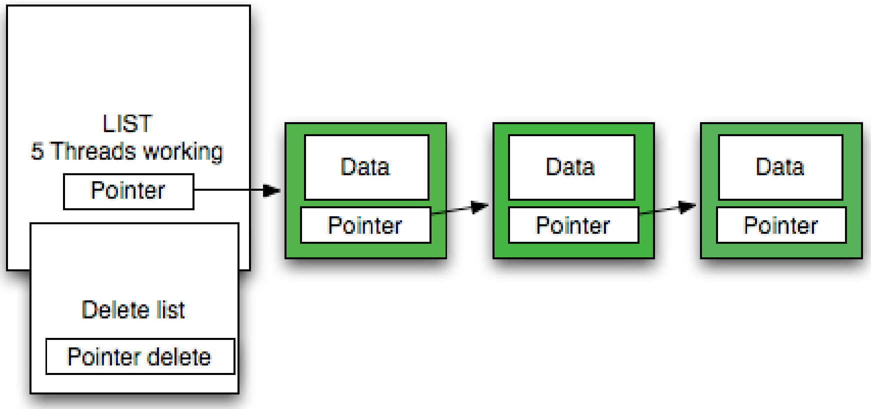


Figure 11 - A reference counted list with five references

When threads work on the items it is not safe to delete any items. So when a thread wants to delete an item from the list, the thread must decouple the item, and put it on a delete list.

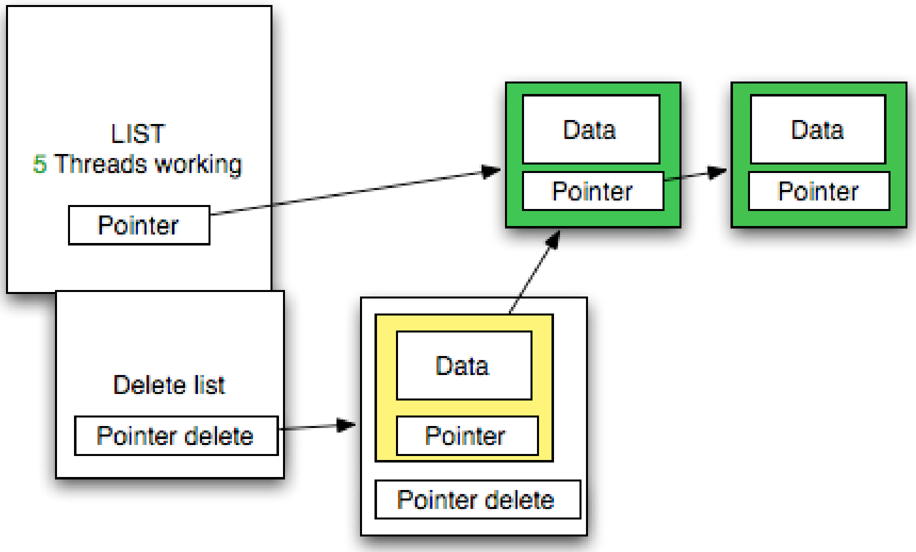


Figure 12 - A thread decides to remove an item from the list

Any thread can carry on doing their job even though they reference a "deleted" item. What it means to the algorithm that an item is deleted varies. If a lookup is being done on the deleted item, and it was found before it was deleted, it would make sense to return a copy of the data. But new operations will not be able to see items in the deleted list.

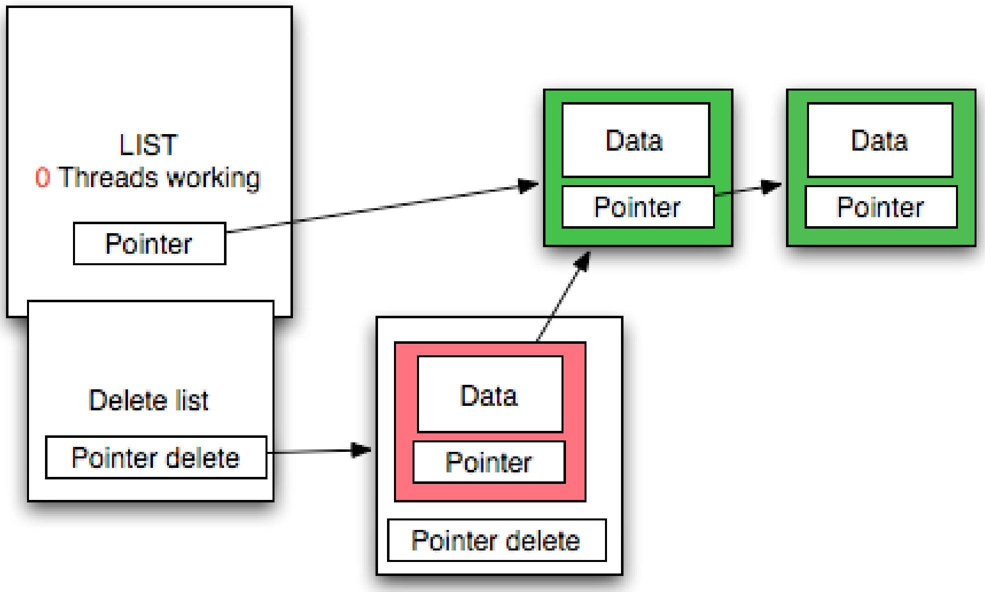


Figure 13 - The last thread exits

When the last thread leaves the list, the list knows it is safe to delete the elements of the delete list. To make sure no new thread can remove elements and put them on the delete list, while the delete list is being deleted, the delete list No new thread can access the data of the delete list. When it is safe to delete the data it is because it is certain there are no references, when there are no references there can be no lurking ABA problem. The correctness of this algorithm depends on the correctness of the reference counting.

The example of the list uses reference counting directly on the list data structure, and that can be implemented using either an internal- or an external reference-count. If the list uses an internal reference-count each public method would increase the reference count when the method was entered and decrease when the method returned. Using external reference count would allow a client to increase the reference count, perform a bunch of operations and then decrease the reference count.

Using internal reference counting hides the bookkeeping, but can be expensive if a client uses many public functions. The external reference count exposes some internal logic to the client, but allows the client to make decisions on when the object in question is not referenced anymore.

Threads entering and leaving will cause the reference count to rise and fall, and there can be a fear that memory will not be reclaimed before the running process is out of address space. To improve on the number of times elements in the delete list will be reclaimed the reference counting can be reconsidered. In order to be safe the list does not need to wait for the reference count to reach 0 before the items in the delete list can be deleted. The only threads that are able to reach the element that was deleted are threads that were actively working when the element was "deleted". So when the last of the threads that was referencing the list when the item was "deleted" have left, the item can be safely deleted.

There is a solution to this, but the bookkeeping is extensive. If each thread entering gets a ticket with an incrementing number, and the item deleted keeps a list of what tickets where active when it was placed on the delete list, it would be possible to delete items from the delete list, without having to wait until all threads are out.

Another possibility is to either reference count all items, or place "gates" that reference count and has a deleted list. The term gate is a term I made up, to describe reference counting for a group of objects. I imagine a gate being like entering a Theme Park. You buy yourself a ticket and enter a gate at the entrance. Each time you take a ride, you find yourself in a queue in front of a gate. The number of people riding on the ride is controlled, and when the ride is finished they leave.

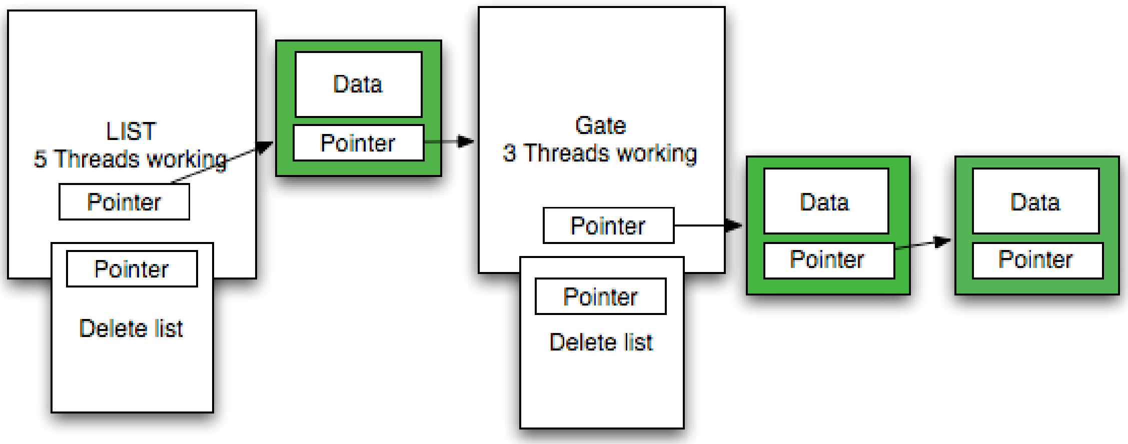


Figure 14 - A list with a gate

If gates are placed in a data structure, they can keep track of how many threads are currently working on a group of objects. And as soon as you can keep track off how many threads are possibly accessing the objects, you will be able to tell when they can be deleted.

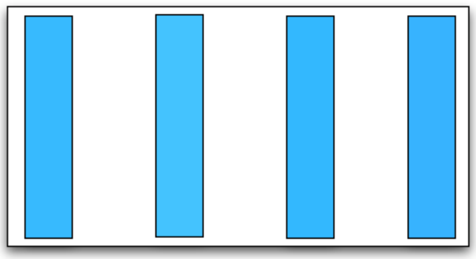


Figure 15 – Gates like a hallway with doors

The gates can be constructed like doors in a hallway. You open one and find yourself in a new part of the hallway, or a set of circumscribed circles you step into. If it is a set of circles, you will enter more and more gates as you get closer to the middle, but you will not leave any of the gates before moving outwards again. Another image of this is a spiral where you enter gates moving in- and exits gates moving outwards.

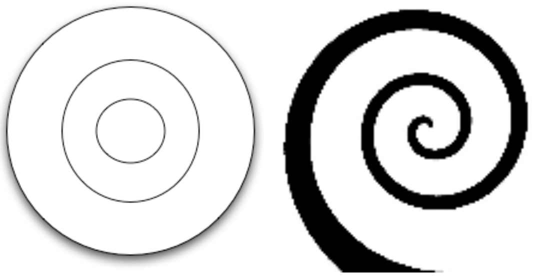


Figure 16 – Gates like circumscribed circles or like a spiral

Reference counting has problems with cyclic references, and therefore the last suggestion has problems with cyclic references. If the elements in the list example were cyclic and entering another circle will keep adding to the reference count, the reference count will continue to grow and be useless. Using the hallway with doors principle, the reference counts lost to cycles will be steady and can therefore be detected. Cyclic references can be solved, and suggestions to how can be found in the book "Garbage Collection" [9].

This section has described two different ways to solve the ABA problem without relying on automatic garbage collection. Since the non-blocking algorithm I am designing uses reference counting, I will solve the ABA problem in the algorithm using reference counting as suggested in method 2. The variant used will be the one using external reference counting. This will let clients of the algorithms choose how often cycles should be spent on reference counting.

3.3.2 Designing the methods

Designing the methods for the non-blocking algorithm has required a different mind set than the blocking methods. In some ways it has forced the code to be as simple as possible to wrap my head around them. Therefore the design of the non-blocking methods has got more details than the methods for the other algorithms.

3.3.2.1 The insert method

Like the single-threaded and the lock-based algorithm, the insert method has responsibility for inserting an item in the list, guaranteeing the key of the item is unique and updating the item count. The insert algorithm is presented in Figure 17.

When inserting in the list of a hash-bucket without blocking, the safest place to insert is at the top of the list. The top of the list is the most safe, since the top pointer cannot be removed and put on the dead objects list, any next pointer in the list can bring cause that situation.

Guaranteeing uniqueness of keys in a hash-bucket can be expensive, since it means that the list has to be traversed and no insert may occur while or after it is checked. Only inserting at the top of the list [NB-I06:]gives the guarantee that if the top pointer has not changed since the traversing of the algorithm started[NB-I11:], and if there was no key found matching the one to be inserted, the key is unique. That means, if the item is successfully inserted at the top of the list, the key is also guaranteed to be unique.

The item count in the non-blocking version relies on the atomic operation Add_Atomic. Both insert and remove will use the Add_Atomic command to increase or decrease item count. Here the item count is no bottle-neck like it is in the lock-based implementation, but the number is still off a bit depending on how many threads have performed an insertion or deletion and are about to increase or decrease the item count.

```
bool    insert(   GateType& gate,const Key& key, const Value& value ) {
    unsigned int ix = hashTableIndex(key);   //NB-I01: find hash_bucket index

    bool    success = false;                 //NB-I02: no success yet
    ItemType*   newItemPtr = new ItemType;   //NB-I03: create new item
    newItemPtr->key_ = key;                  //NB-I04: assign key
    newItemPtr->value_ = value;              //NB-I05: assign value

    do {
        ItemType* itemTopPtr = items_[ix];   //NB-I06: take a fresh copy of the top pointer
        if (findFirst( key, itemTopPtr ) )  {//NB-I07: check if key is unique
            delete newItemPtr;               //NB-I08: another thread beat us to it...
            return false;                    //NB-I09: not unique (anymore)
        }
        newItemPtr->next_=itemTopPtr;       //NB-I10: current top is next_ cannot fail
        success = Compare_And_Swap( itemTopPtr , newItemPtr, &items_[ix])) {   //NB-I11:
     } while (!success);                    //NB-I12: check if we need to restart/retry

    Add_Atomic( 1, &itemcount_);            //NB-I13: add to the item count
    return true;
}
```

Figure 17 - Non-blocking insert

The gate reference parameter is not used in the function, but is taken to make sure the client of the algorithm has instantiated a gate before calling. The algorithm will not function without the gate since there would be no reference counting and thereby no protection against deletion of memory and the ABA problem.

A thread can be prevented in progressing if other threads have success in modifying the hash-buckets top pointer. But if one thread is prevented from progressing, the top pointer has been successfully modified by another thread. This algorithm is not lock-free since other threads constantly changing the top pointer of a hash-bucket [NB-I06:] can starve the insert method.

3.3.2.2 The lookup method

Having done the insert method, the lookup method can use a part of it. The lookup method is very simple in comparison.

```
bool lookup(   GateType& gate, const Key& key, Value& value ) {
// performs an insert in the top of the chained list
    unsigned int ix = hashTableIndex(key);
    ItemType* oldItemTopPtr = items_[ix];

    // check if unique
    if (oldItemTopPtr) {
      if (ItemType*  item = findFirst( key, oldItemTopPtr ) ) {
        value = item->value_;
        return true;
      }
    }
    return false;
}
```

The lookup method can find item that is removed from the list by another thread, during the copying of its value. A thread started shortly after will never find this item, but that is not a problem. At the time it was there, and the thread found it successfully, and therefore returning the value after the item is deleted is correct.

3.3.2.3 The remove method

It is hard to remove items from a non-blocking list, therefore many non-blocking implementations uses tombstones, and some that avoid to e.g. the algorithm suggested by Chris Purcell and Tim Harris [10]. A tombstone is an easy way to mark an item as dead. The item still lingers and can be accessed, but it is marked as being dead. The disadvantage of leaving tombstones is that they are not really deleted and they are traversed as any valid item. For some uses tombstones are a smart solution of a tricky problem, but for structures with a high throughput, like a system event queue, leaving tombstones is not acceptable. So when designing the remove method I will try to make a remove method that removes the item, so the system eventually can reclaim its resources tied to that item.

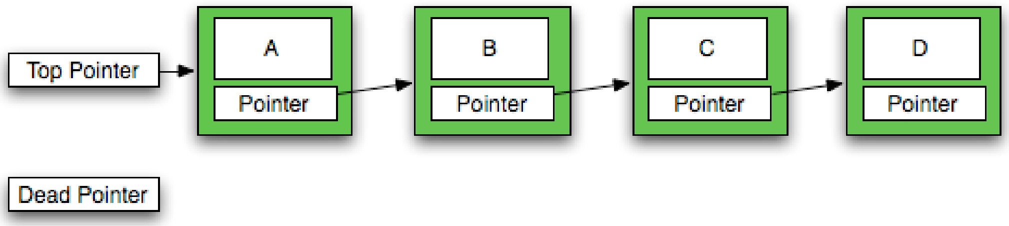


Figure 18 - List before delete

The insert method avoided problems by only adding items to the top pointer of the hash-bucket; this is not possible for the remove method. The remove method must be able to remove an item no matter where it is in the list. This makes the remove method trickier to design.

One example of a tricky part is doing a dual delete in the same hash-bucket list: Consider a list like the list shown in Figure 18 two threads want to remove two different items A and B. Only one compare_and_swap() can occur atomically so reading a pointer and doing a successful compare_and_swap() does not guarantee that the pointer read has not changed, just that the pointer written to has not changed. Figure 19 shows a fragment of code used for removal. If the to threads use the same piece of code and B is first to read the Prevnext, and thread A is switched in at the ellipsis, the removal will not be complete.

```
Prevnext= previous->next
…
Next = next
If ( CAS( Prevnext, next , & previous->next))
```

Figure 19 - A code fragment of removing from a list

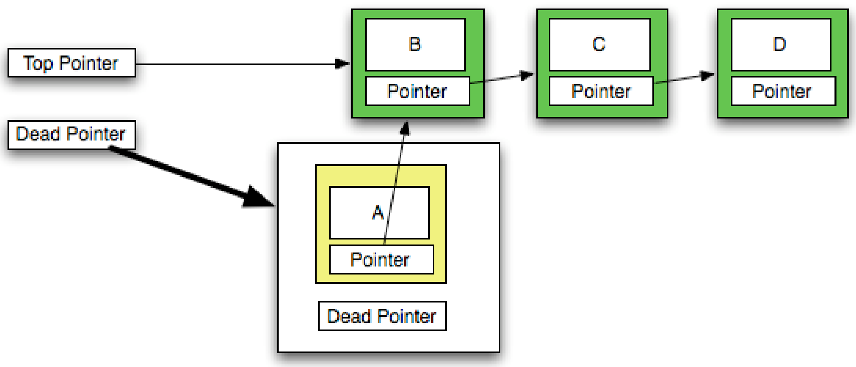


Figure 20 - removing A while removing B

A will be properly deleted in the first shot, and makes the top pointer point to B, but B will still remember A as it previous item.

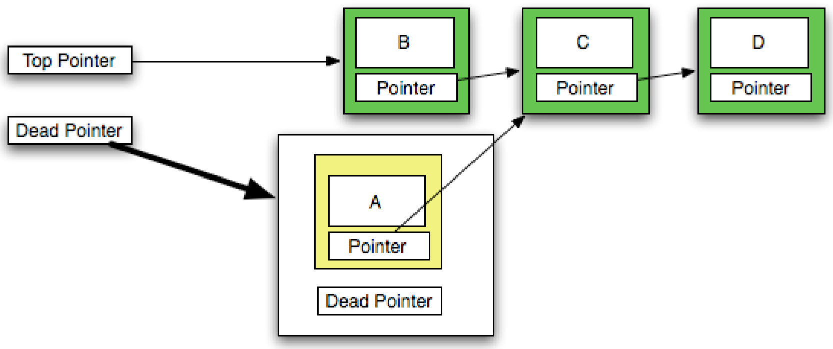


Figure 21 - Item B deletion makes it's previous point to it's next

And now B is accidentally left alive in the top of the list, so the thread deleting must after the compare_and_swap make sure that a lookup will not find the item it has tried to delete. It is completely valid if another thread has inserted a new item with the same key, so the lookup must check if the item found points to the same item as it was trying to delete, before the delete is retried.

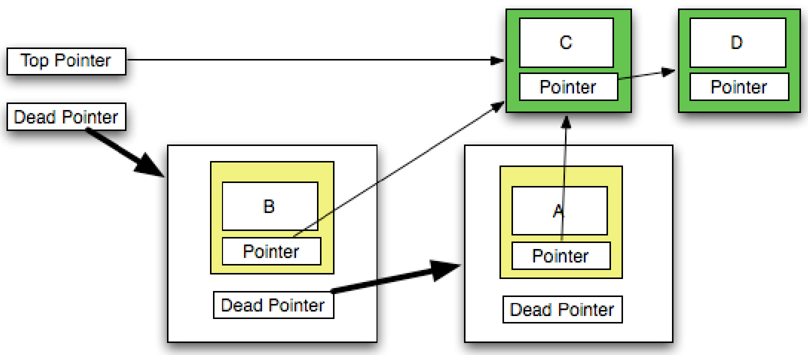


Figure 22 - A and B deleted

The top hash-bucket pointer is not the only pointer to remove from; when we remove from the top pointer we know that if we change that pointer, that pointer will be okay.

Figure 23 shows the pseudo-code for the remove method, the line [NB-R09]shows how the remove method separates the handling of deletion in the top of from the deletion any other place in the list.

Another problem with the remove pseudo-code is the handling of removing items while the item after is removed and the items next pointer is changed. This problem is an intermittent problem, since it requires two values to be in sync while storing that value somewhere else with compare_and_swap() [NB-R08]. It is possible to detect the situation, since line [NB-R10] does a compare_and_swap() and right after we should check if the next pointer `itemTypePtr->next_` is equal to `typesNext` the copy of the next pointer. If they are equal there is no problem. If they are not equal a new compare_and_swap() can be performed to try to rectify the situation, but it can fail.

```
bool    remove(  Gate<ItemType>& gate,  const Key& key ) {
    unsigned int ix = hashTableIndex(key);   //NB-R01: find hash-bucket
    bool success = false;                    //NB-R02: no success yet
    do {                                     //NB-R03:
        ItemType* itemTypePtr = items_[ix];  //NB-R04: take a copy of the top pointer
        ItemType* itemTypePrevPtr=0;         /NB-R05: previous pointer = no previous

        while (itemTypePtr) {                //NB-R06: iterate
            if (itemTypePtr->key_==key) {    //NB-R07: is the key the right key
                ItemType* typesNext = itemTypePtr->next_;
                                        //NB-R08: store a copy of the next pointer
                if (itemTypePrevPtr) {       //NB-R09: if item has a previous pointer
                    success = Compare_And_Swap( itemTypePtr , typesNext ,
                              &itemTypePrevPtr->next_); //NB-R10: no success yet
                    if (success) {
                         if (itemTypePtr != findFirst(key)) {//NB-R11: no success yet
                              gate.addGarbage(itemTypePtr);   //NB-R12: item was deleted
                         } else {
                              success = false;         //NB-R13: item was partly deleted
                         }
                    }
                    break;                       //NB-R14: exit while loop
                } else {                         //NB-R15: if item is first pointer
                    success = Compare_And_Swap( itemTypePtr , typesNext , &items_[ix]);
                                                 //NB-R16: CAS
                    if (success)
                         gate.addGarbage(itemTypePtr);//NB-R17: item was deleted
                    break;                       //NB-R18: exit while loop
                }
            }
            itemTypePrevPtr = itemTypePtr; //NB-R19: update previous pointer
            itemTypePtr=itemTypePtr->next_;//NB-R20: advance itemTypePtr
        }
        if (itemTypePtr==0)                //NB-R21:(itemTypePtr==0)key is not in bucket
          return false;                    //NB-R22: key was not in bucket
    } while (!success);                    //NB-R23: do{}while no successful remove

    if (success)                           //NB-R24: successful?
        Add_Atomic(-1,&itemcount_);        //NB-R25: decrement itemcount
    return success;
}
```

Figure 23 - Pseudo code for the remove function

The remove method has issues, some I have not found a workaround for, since they were discovered very late in the process. So for the purity of the test, I will leave out the remove method. I cannot time the method properly if it has code to detect the flaw described above, and without the code I cannot trust the result of a test.

A possibility could be turning to the tombstones, or disallow collisions in the hash-table. Neither of the options are attractive since tombstones will hurt the performance and make large claims on memory and disallowing collisions will leave the algorithm useless for anything else than perfect hashing. Therefore I will leave the remove method in the state it is in.

4 Executing the experiment

The purpose of this experiment is to investigate the algorithms for performance, and test if my hypothesis is correct. The methods that can be tested are Insert and Lookup.

For the experiment I have a plain text file of 173.528 words. This file I will memorymap and let multiple threads read and insert words from this file. Each thread will get a part of the file for processing, and the insert part is done when it has processed its own part. After Insert has been tested, another thread will take over that will lookup all words stored by the Insert thread. So when I execute with a number of threads, I create the same number of threads for Insert as I do for lookup. I time the executions of all insert threads and all lookup threads, so I end up with two execution measurements in nanosecond resolution.

The insert test will start with an empty hash-table and when the insert part is done; the lookup will take each of the words and look it up. The two parts must be measured independently.

I will perform the experiment on the single-threaded algorithm, the lock-based algorithm and the non-blocking algorithm on hash-tables with different vector sizes. I have chosen the sizes 256,4096 and 65536 (hex 0x100, 0x1000 and 0x10000). Of the threaded algorithms I expect the multithreaded algorithm to perform best on the small vector sizes and the non-blocking best on the large sizes.

I will use the hash-function called DJBHash by Professor Daniel J. Bernstein. It is a fast hash-function and it distributes the keys well, so the number of hash-collisions will be at a minimum.

I will perform the experiment with different numbers of threads too, since the number of threads can affect performance. I expect to see differences in the performance of the algorithms when changing the number of executing threads. If the number of threads gets very high, the non-blocking algorithm might experience a live-lock, the number of CPU's do limit how many threads can be running concurrently, and as long the threads are allowed to carry out more than one complete operation before they are switched out, the risk of colliding with another thread is slim. The lock-based algorithm has one lock that is shared between all threads, and that is the item count lock. I suspect that might degrade the performance of inserts with many threads running. I will test with 1,2,3,4,5,10,25 and 50 threads. Each test is repeated 10 times to get an even measurement.

5 The results of the experiment

The gathered data from the experiment has been collected as spreadsheets, and are available on CD. There is too much data to present it as an appendix.

5.1 Experiment with a vector size of 256

The first test where the vector size was 100, my hypothesis was that the lock-based implementation would perform better.

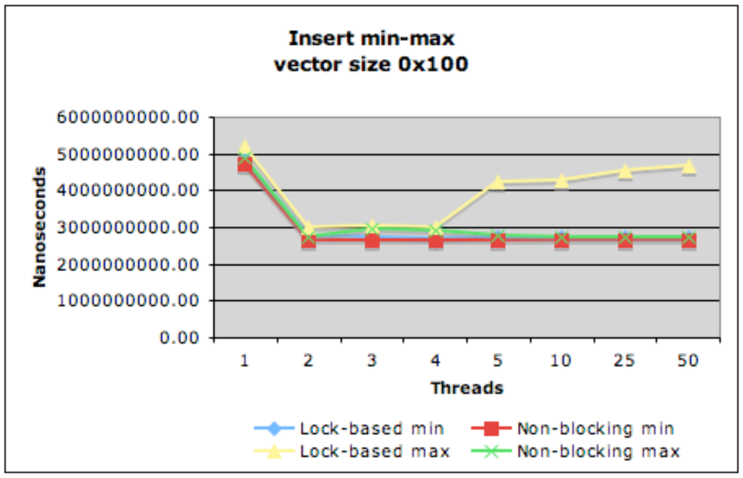


Figure 24 - Insert min-max vector size 0x100

Looking at the results of the experiment, the lock-based and non-blocking algorithms had very similar performance, but the lock-based insert had some peaks when adding more than four threads, which could be caused by the item count mutex. The lookup lock-based and non-blocking implementations showed very little differences.

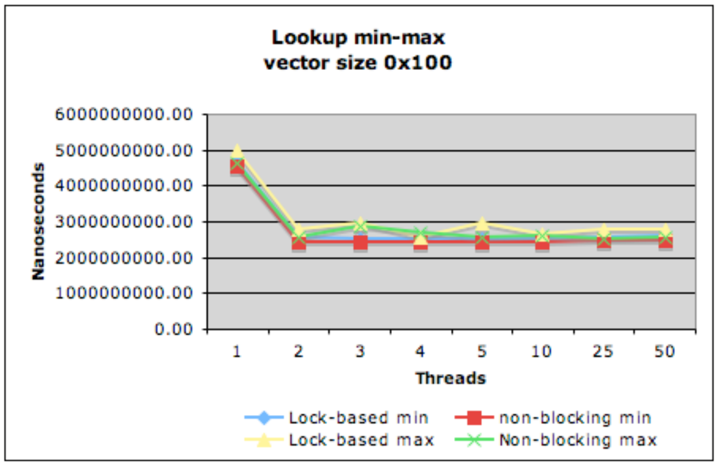


Figure 25 - Lookup min-max

There seems to be very small differences between the lock-based and the non-blocking version.

During the experiment I wondered why the measurements on executing a specific executable would give so different result when executing it again, and why some executables would give almost the same results over and over again.

I decided to create a graph for the average deviation for the measurements of the 10 runs of the two algorithms.

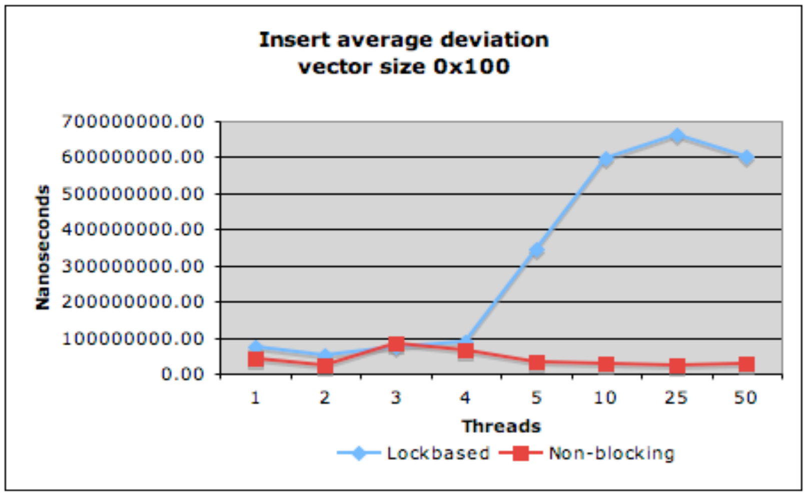


Figure 26 - Insert average deviation for a vector of size 0x100

The non-blocking insert method Figure 26 shows very steady performance, while the lock-based implementation starts to vary when the number of threads is above 4.

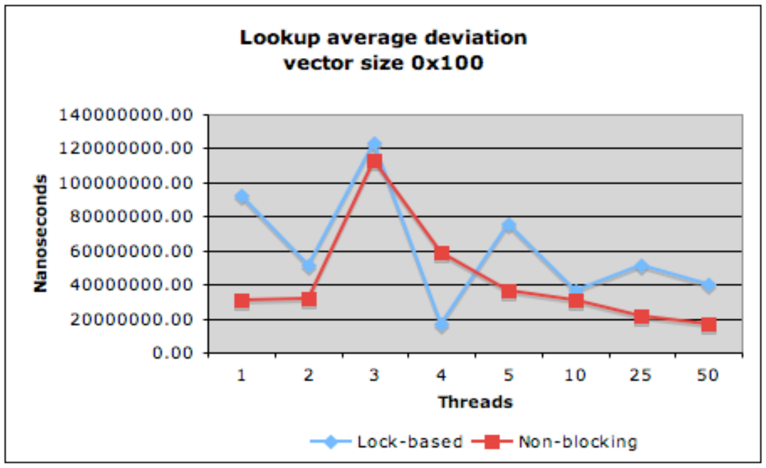


Figure 27 - Lookup method average deviation for a vector size of 0x100

The performance shown for the lookup method in Figure 27 does not vary as much as the performance shown for the insert method in Figure 26.

The average deviation shows how good the algorithms are at giving the same performance time after time, the lower their value the better.

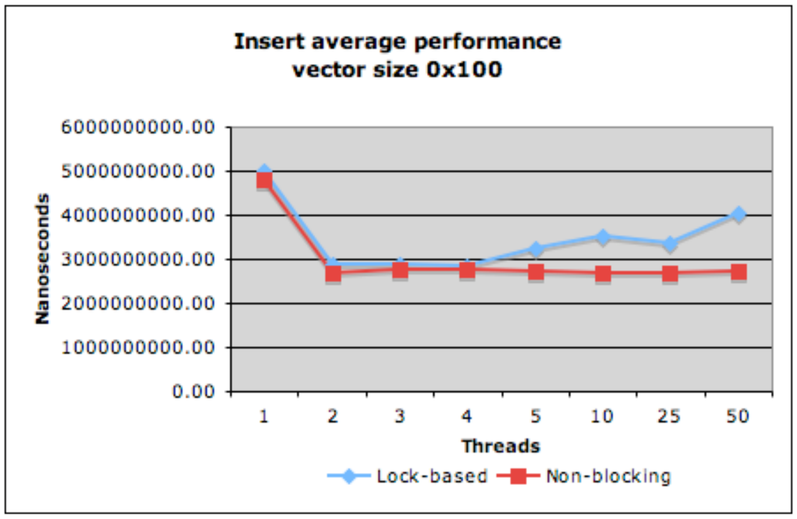


Figure 28 - Average insert performance for a vector of size 0x100

My experiment with vector size 256 (0x100) shows that even with an average of 173528/256=677 collisions per bucket, the non-blocking algorithm shows equivalent performance when the number of threads are low, and better performance when the number of threads are high. The non-blocking insert has another benefit; it is more consistent than the performance of the lock-based algorithm.

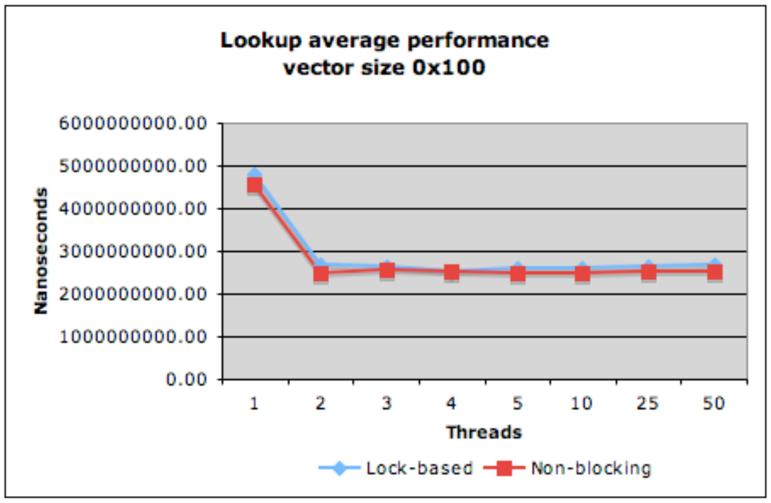


Figure 29 - Average lookup performance for a vector of size 0x100

The lookup methods measurements in Figure 29 showed very small differences, so small that I will call the results inconclusive. One algorithm might be faster than the other, but it has not been shown in my experiment. The experiment can not conclude that the last part of my hypothesis is correct, since the experiment shows the algorithms are on par.

5.2 Experiment with a vector size of 4096

The hash-table with a vector size of 4096 gives around 173528/4096=42 collisions per hash-bucket. I considered the size of 4096 for a middle ground, where I expected the algorithms two show equal performance, but the results where not as I expected.

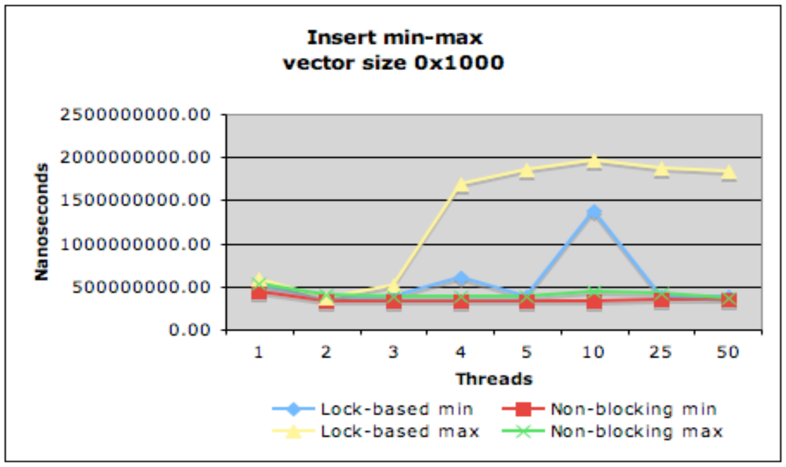


Figure 30 - Insert min-max with a vector of size 0x1000

Figure 30 shows that the lock-based algorithm showed very varying performance, one run would show the same performance as the lock-based while next run would be far off.

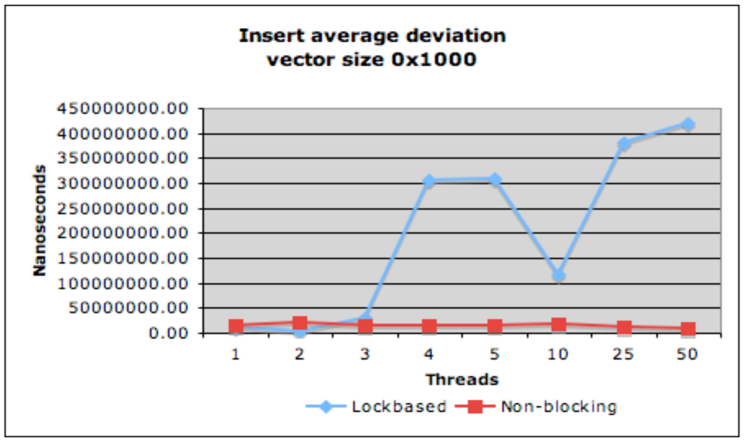


Figure 31 - Insert average deviation with a vector size of 0x1000

The average performance show in Figure 32 of the insert algorithms shows that the non-blocking method is better dealing with multiple threads.

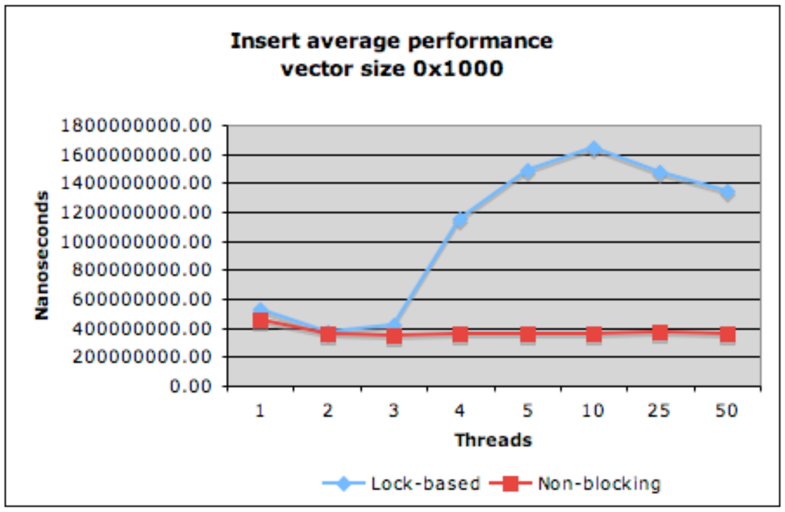


Figure 32 - Average insert performance

The lookup method performance measurements show a slight tendency towards the non-blocking algorithm but as with the vector size of 256, the performance of the two algorithms are similar. The average insert performance seems to be the same for the two algorithms at 2 threads and already at 3 threads a small difference shows, at 4 threads there is a huge different in performance. At vector size 256(0x100) the problem showed at 5 threads (Figure 28).

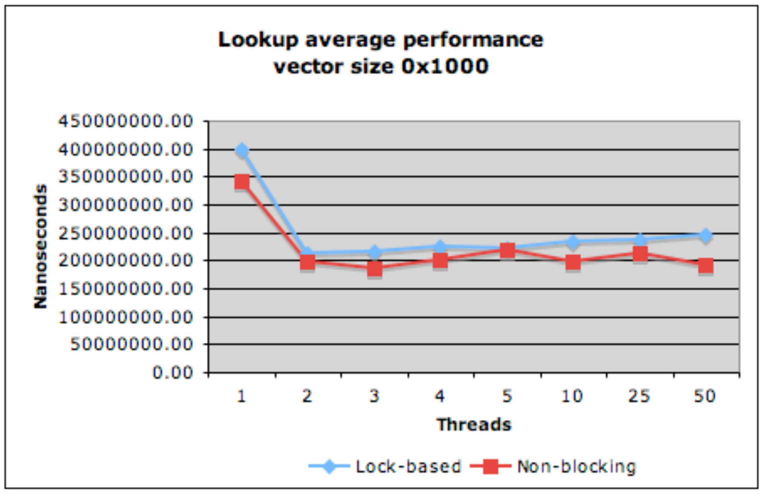


Figure 33 - Average lookup performance

The average deviation of the lookup methods shows that the lock-based algorithm has the most predictable lookup performance.

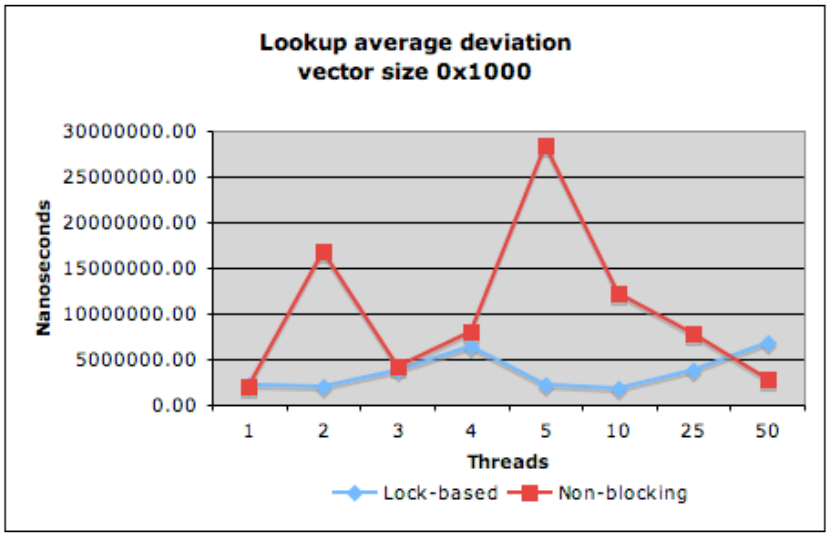


Figure 34 - Average deviation of the lookup methods

My experiment with a hash-table with a vector of size 4096(0x1000) shows an apparent distinction between the two insert methods and very little difference between the lookup methods.

I did not expect that the lock-based algorithm would provide the most predictable performance for lookup,

5.3 Experiment with a vector size of 65536

The experiment with a vector size of 65536 would give an average of 173528/65536=2 collisions per bucket. The risk of threads colliding would be small, so the necessity for synchronization is small.

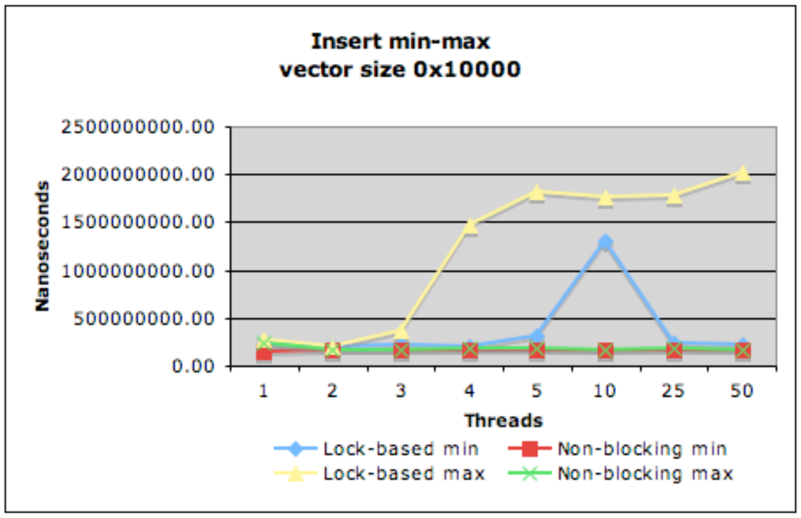


Figure 35 - minimum and maximum values for insert

If the first part of my hypothesis is correct, the performance of the non-blocking algorithm should be better, so from this experiment I expect to see a clear difference in both the insert and the lookup methods.

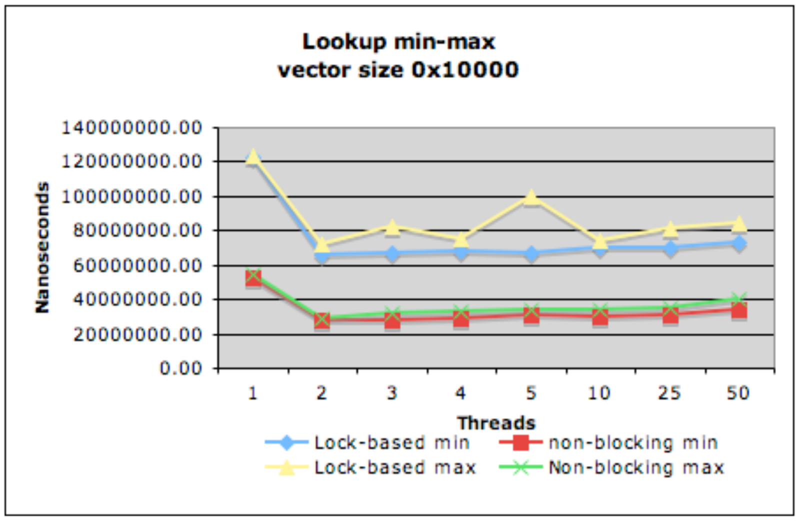


Figure 36 - minimum and maximum for lookup

Figure 35 and Figure 36 shows the minimum and maximum performance for insert and lookup. The shape of the graph of the insert performance resembles the shape of the graph shown in Figure 30 showing the inserts in a hash-table with a vector size of 4096.

The graph shows that with 2 threads, the performance of the insert method of the algorithms is almost identical. The average performance of the lock-based version is 206745860.30 nanoseconds and the average performance of the non-blocking algorithm is 166108600.90, so the non-blocking algorithm is measurably faster.

Figure 37 shows the average performance of the lock-based algorithm divided by the performance of the non-blocking algorithm. It indicates how much faster the non-blocking algorithm is on inserts.

Vector size:0x100

1 thread	1.05
2 threads	1.07
3 threads	1.05
4 threads	1.04
5 threads	1.20
6 threads	1.31
7 threads	1.25
8 threads	1.49

0x1000

1 thread	1.15
2 threads	1.01
3 threads	1.18
4 threads	3.19
5 threads	4.15
6 threads	4.59
7 threads	3.99
8 threads	3.79

0x10000

1 thread	1.47
2 threads	1.24
3 threads	1.57
4 threads	4.17
5 threads	7.08
6 threads	9.38
7 threads	6.71
8 threads	6.58

Figure 37 – showing differences in the average insert performance

The non-blocking insert method is marginally faster up to 4 threads using a vector of size 256. It is marginally faster using up to 3 threads with a vector size of 4096. The tendency is that the non-blocking insert method generally is faster, but it is not noticeable before the thread count is high, or the vector size of the hash-table has a low collision count.

Vector size:0x100

1 thread	1.05
2 threads	1.08
3 threads	1.03
4 threads	1.01
5 threads	1.05
6 threads	1.04
7 threads	1.05
8 threads	1.06

0x1000

1 thread	1.17
2 threads	1.09
3 threads	1.16
4 threads	1.12
5 threads	1.02
6 threads	1.18
7 threads	1.11
8 threads	1.27

0x10000

1 thread	2.31
2 threads	2.38
3 threads	2.42
4 threads	2.38
5 threads	2.34
6 threads	2.31
7 threads	2.33
8 threads	2.10

Figure 38 - showing differences in the average lookup performance

Figure 38 shows the difference in the lookup performance. The number of threads does not affect the lookup performance.

The number of collisions in a hash-bucket affects the lookup performance. As long as the number of collisions is high; the penalty of locking and unlocking a mutex is low. As soon as the number of collisions is low; the cost of locking a mutex affects performance.

Changing the vector size to 65536 also had an impact on the predictability of the performance. The non-blocking algorithm produced very consistent result while the lock-based algorithm produced different result for every run.

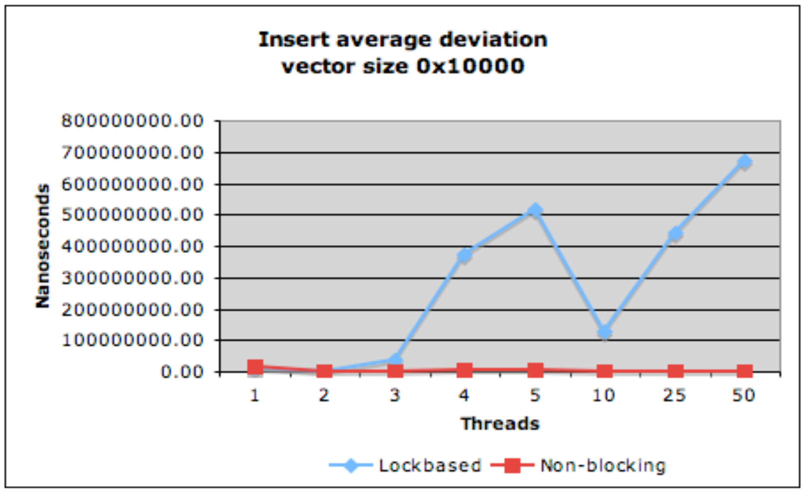


Figure 39 - average deviation for insert methods

The insert methods average deviation in Figure 39 is almost similar to the one of Figure 31 with a vector size of 4096. So the performance of the lock-based algorithm is not predictable.

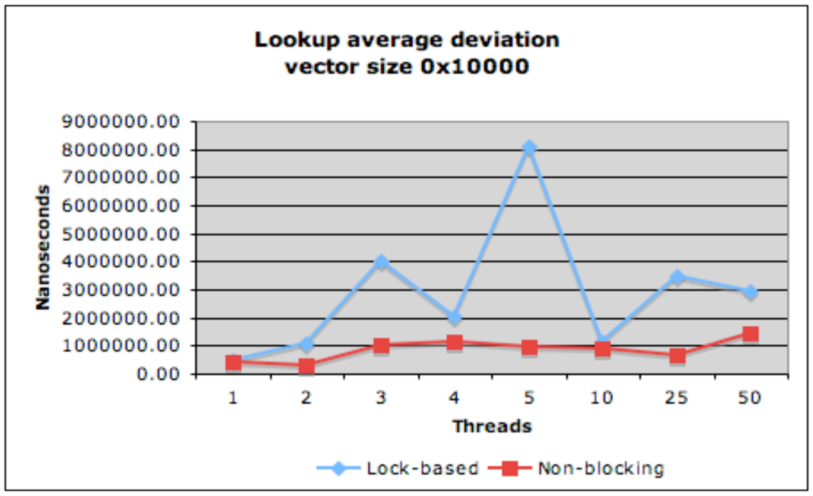


Figure 40 - The average deviation for lookup

The average deviation of the performance of the lookup methods has changed. When the vector size was 4096 the non-blocking algorithm provided the least predictable performance, with a vector size of 65536 the tables have turned

The graph shows a difference between 3,4 and 5 threads that is hard to explain, and the difference between 5,10 and 25 threads is just as hard to explain. I suspect that dividing the lookups between threads depending on the number of threads causes more or less threads to wait on mutexes.

My experiment with a vector size of 65536 shows that the non-blocking algorithm is faster than the lock-based algorithm. The insert method is faster with a few threads and much faster with many threads (Figure 37). The lookup method is consequently more than two times as fast (Figure 38). The suspected cause of the measurable better performances from the ”vector size 4096 experiment” to this is, that the number of collisions has been reduced and the cost of locking and unlocking a mutex suddenly is a larger part of the lookup method.

The experiments have shown that none of the situations I have setup resulted in the lock-based hash-table algorithm showing better performance. The insert method of the non-blocking algorithm showed better performance, and showed more predictable performance.

The lookup method showed performance improvements when there are few hash-bucket collisions. An algorithm using rehashing or another way of dealing with hash-bucket collisions might yield better results.

6 Multithreaded- versus singlethreaded algorithms

When comparing the two multithreaded algorithms, the question whether or not to use threads was never posed. When examining the performance, a valid question is, do we get more performance using multithreaded algorithms?

To compare the algorithms I have chosen to compare the single-threaded algorithm with the number of threads that gave the overall best performance using the 3 different vector sizes. All the numbers are measured in nanoseconds and are presented as such.

The ST/X factor is calculated by dividing the average singlethreaded performance for a method by the average performance of a threaded version of the same algorithm. This factor shows how many times faster the algorithm is than the singlethreaded version.

Table 1 – Single- versus multithreaded performance

	ST 1 thread		**NB 2 threads**		**LB 2 threads**	
	Insert	**Lookup**	**Insert**	**Lookup**	**Insert**	**Lookup**
0x100 Vector size						
avg	4732295662	4572246119	2678607070	2475515652	2878824986	2669828908
ST/X factor	1.00	1.00	**1.77**	**1.85**	1.64	1.71
0x1000 Vector size						
avg	425465908	342240817.1	361997488.9	196653453.1	366535246.9	214483906.1
ST/X factor	1.00	1.00	**1.18**	**1.74**	1.16	1.60
0x10000 Vector size						
avg	127013655.6	53800640.8	166108600.9	28215340.9	206745860.3	67052433.7
ST/X factor	**1.00**	1.00	**0.76**	**1.91**	**0.61**	**0.80**

For each of the 3 vector sizes I have marked the method of the algorithm that has the best performance as **bold**, and I have marked algorithms with worse performance than the singlethreaded with **bold** and red.

The measurements of Table 1 shows that in the first to cases where the, vector size is 256 and 4096; the non-blocking algorithm provides the best performance. But in the last case with few hash-bucket collisions, the singlethreaded algorithm outperforms both threaded algorithms when it comes to the insert method. The non-blocking algorithm has much better lookup speed than the singlethreaded and the lock-based algorithms.

The size of the vector had great impact on the performance of the singlethreaded insert method. When the of the size of the vector changed from 256 to 4096 the performance improved by a factor 11 while the threaded algorithms only improved by a factor 7. From 4096 to 65536 the performance of the singlethreaded version changed by a factor 3, while the threaded versions improvement was a factor 2 for the non-blocking and about a factor 1.5 for the lock-based.

The lock-based implementation shows it has an overhead using the locks, and it clearly visible in the lookup method, when the vector size is 65536. The lookup method only uses one mutex per hash-bucket, but using 2 threads the algorithm can handle 80% of what a singlethreaded algorithm can. Running lookups from a lock-based multithreaded environment eats up around 120% of the performance.

Looking at the non-blocking algorithms performance, it is more scaleable and performs almost twice as good as the single-threaded algorithm, and more than twice as good as the lock-based algorithm. The non-blocking algorithm's insert method does seem to have problems with scaling. When the vector size is large, it is not able to outperform the singlethreaded insert method, even though it outperformed it massively when the vector size was 256.

Having profiled the non-blocking algorithm I found that the multithreaded algorithms can perform the search for if keys are unique in parallel, and searching through a long list of collisions is faster, if 2 CPU's can handle each half. When the vector size increases, there are fewer hash-collisions. And when there are fewer hash-collisions, the benefit of using two processors to skim through the keys diminishes.

6.1 Why are the singlethreaded- and the non-blocking algorithm not on par then?

The non-blocking algorithm uses compare_and_swap() with a memory barrier. This is necessary to make sure that all previously written data is written when the compare_and_swap() succeeds. On insert the compare_and_swap() takes place after the next pointer is set to point to the next element in the list, but modern CPU's allow writes to be reordered for aggressive caching purposes. From one thread of execution these reordering are not noticeable, but with multiple threads of execution memory barriers are necessary. The compare_and_swap() with memory barrier is not cheap, and I have profiled the non-blocking algorithm and can see that the percentage of time spent in compare_and_swap() climbs drastically as the size of the hash-table vector declines.

6.2 How can the non-blocking algorithm be improved?

Storing collisions in chained lists is not a good idea, a fixed size array of pointers to items would avoid storing a next pointer in the item, and the memory barrier can be avoided. This has another benefit; all items can be removed without the risk of a race condition during the compare_and_swap of a previous elements pointer.

7 Conclusion

This experiment was carried out to examine the differences in performance of a lock-based- and a non-blocking algorithm. The 3 implementations had 3 functions they supported, insert, lookup and remove. I successfully tested the insert and lookup methods of 3 different types of hash-table implementations, namely a lock-based, a non-blocking and a singlethreaded.

I have been unable to test remove method since I found problems in the non-blocking implementation late in the process. This leaves the question of how remove would perform non-blocking.

An alternative solution to the ABA problem was suggested and implemented, but without the remove method, the ABA problem will not occur. The correctness of the algorithm has not been proved; therefore the suggested solution to the ABA problem could be a topic for another project.

My hypothesis was, that an algorithm with a small probability of threads colliding causing race conditions would be better of running without locks performance wise, while an algorithm with high probability of threads colliding would benefit from running lock-based.

I suspected there would be clear cases where one algorithm would perform better than the other, but among the algorithms tested the non-blocking implementation has superior performance in most cases. There was one case where the singlethreaded algorithm outperformed the non-blocking algorithm, but I think it due to my lack of experience with non-blocking algorithms and a will to create very identical implementations of the algorithms to make them comparable.

Access to queues and stacks is innately serial while access to hash-tables is widely parallel. Therefore I suspected that the abstract data structure hash-table is well suited for non-blocking algorithms, while data structures such as stacks and queues are most suitable for lock-based algorithms.

During the experiment I discovered that the non-blocking algorithm not only showed good performance, but also a more reliable performance than the lock-based implementation. This can only be explained by a low number of thread collisions, resulting in a low number of retries necessary by the non-blocking algorithms, this strengthened my hypothesis.

The reason the non-blocking algorithm has done so well in the experiment and that I have been unable to prove the last part of my hypothesis, is that I have not worked with vector sizes small enough to create enough problems for threads to run in parallel on the data structure. Maged M. Michael demonstrated how queues perform better blocking than non-blocking. So with the help of another author the last part of the hypothesis is made probable, without my experiments indicating it directly [3]. The hash-table is innately parallel in its design, but also very reliant on the hash-function that distributes the keys. To falsify the hypothesis a cleverly chosen hash-function that resulted in many hash collisions might have shown a different picture.

This has been my first encounter with non-blocking algorithms, therefore I am not surprised that I did have a hard time getting the algorithms right. I believe this field will gain much more interest in the years to come and it in time will get its own set of idioms and patterns to ease the implementation.

Appendix A – Mutual exclusion pitfalls

The number of problems caused by use of locking makes lock-based algorithms more vulnerable to other problems. Lock-based algorithms are vulnerable to deadlock, priority inversion and convoying, while lock-free algorithms do not have these problems, therefore writing lock-based programs requires a great deal of planning and thought.

What is a deadlock? And why is it a problem?

A set of threads are deadlocked if each thread is waiting for an event, that only another thread in the set can generate. A deadlock can be illustrated as a cyclic graph, where each thread is waiting for a resource and by itself contributes to the deadlock by holding a resource wanted by another thread. Figure 41 shows a deadlock in the simplest form where two threads each holds a resource required by the other.

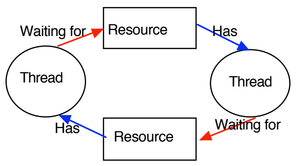


Figure 41 Deadlock where to threads have acquired a resource each, that the other needs to proceed.

To be part of the deadlock, a thread must have a resource required by another thread, so thread 3 in Figure 42 is not part of the deadlock.

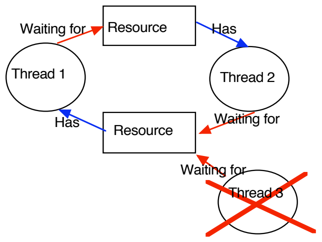


Figure 42 Thread 1 and 2 as part a the deadlock, while thread 3 just waits for a resource to be available

Why are deadlocks a problem? The problem is it grinds the threads to a halt. The threads tasks are not being carried out; they will never have any progress. And the threads will hold the resources they have obtained forever, so other threads cannot use them. If the resource held is memory, it might be negligible. But if the resource held is a hospital heart monitor, or a space shuttles landing gear, the consequences can be fatal.

Priority inversion

Priority inversion can occur when threads with different priorities shares resources and demand exclusive access for those resources. If a priority 1 (1=low,10=high) thread is using the resource a high priority thread requires, any medium priority thread ready to run will cause the low priority thread to be switched out by the scheduler and prevent the high priority thread from running [2].

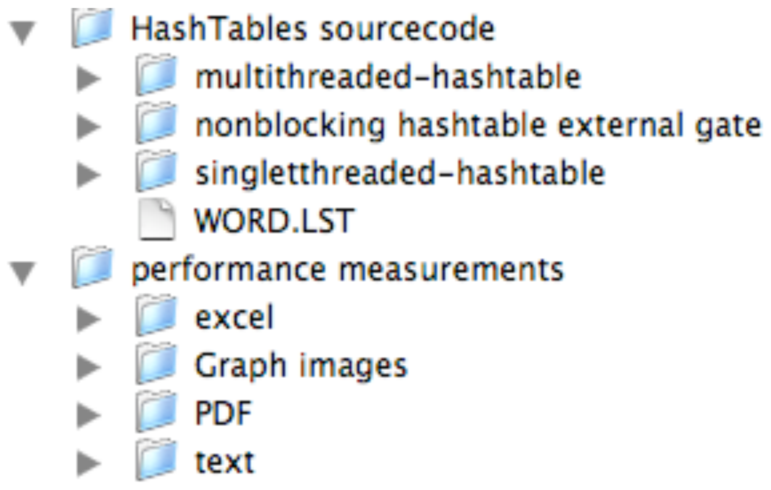
Convoying

A problem of threads blocking other threads by holding locks, while being preempted, can cause convoying. Convoying will make one or more threads wait for another thread effectively causing threads to form a convoy like cars following a tractor on a single-track road. If threads are used to achieve high performance or some threads cannot be allowed to block for security reasons, lock-free algorithms are a better choice.

Appendix B – CD Contents

The CD included contains the source code for the hash-tables. To each algorithm is a project file for version 2.2.1 of XCode.

There is a directory called performance measurements, that includes the text output gathered from the executables. The processed performance measurements in the form of excel documents, PDF documents and images of the graphs produced.



```
▼ HashTables sourcecode
    ▶ multithreaded-hashtable
    ▶ nonblocking hashtable external gate
    ▶ singletthreaded-hashtable
      WORD.LST
▼ performance measurements
    ▶ excel
    ▶ Graph images
    ▶ PDF
    ▶ text
```

Bibliography

[1]	The essence of Databases By F.D.Rolland,, book Publisher: Prentice Hall PTR; 1st edition (November 10, 1997) ISBN: 0137278276
[2]	Introduction to Operation Systems By John English,, book Publisher: Palgrave Macmillan (August 10, 2004) ISBN: 0333990129
[3]	Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms By Maged M. Michael & Michael L. Scott, 1996, article Source: Annual ACM Symposium on Principles of Distributed Computing ISBN:0-89791-800-2
[4]	Modern Multithreading By Richard H. Carver & Kuo-Chung Tai, book Publisher: Wiley-Interscience (October 19, 2005) ISBN: 0471725048
[5]	C++ Coding standards: 101 rules, Guidelines, and Best Practices By Herb Sutter and Alexandrei Alexandrescu Publisher: Addison-Wesley Professional (October 25, 2004) ISBN: 0321113586
[6]	Obstruction-Free Synchronization: Double-Ended Queue as an example By Maurice Herlihy, Victor Luchangco, Mark Moir, article International Conference on Distributed Computing Systems, pages 522--529. IEEE, 2003 http://www.cs.brown.edu/people/mph/HerlihyLM03/main.pdf
[7]	High Performance Dynamic Lock-Free Hash Tables and List-Based sets By Maged M. Michael, 2002, article, Source: Annual ACM Symposium on Principles of Distributed Computing ISBN:1-58113-529-7
[8]	Modern C++ Design: Generic Programming and Design Patterns Applied By Andrei Alexandrescu, book Publisher: Addison-Wesley Professional; 1st edition (February 13, 2001) ISBN: 0201704315
[9]	Garbage Collection, Algorithms for Automatic Dynamic Memory Management By Richard Jones & Rafael Lins,, book Publisher: John Wiley & Sons (September 17, 1996) ISBN: 0471941484
[10]	Non-blocking Hashtables with Open Addressing By Chris Purcell and Tim Harris, 2005, article ISSN: 1476-2896 UCAM-CL-TR-639 http://www.cl.cam.ac.uk/Research/SRG/netos/papers/2005-disc-hashtables.pdf