The Rotating Skip List:
A New Non-Blocking Skip List

IAN DICK

SID: 307137341

Supervisor: Dr. Vincent Gramoli

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Information Technology (Honours)

School of Information Technologies
The University of Sydney
Australia

4 November 2013

THE UNIVERSITY OF
SYDNEY

‘

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Academic Board Policy: Academic Dishonesty and Plagiarism can lead to the University commencing proceedings against me for potential student misconduct under the 2012 Academic Dishonesty and Plagiarism in Coursework Policy.

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Ian Dick

Signature: **Date**:

Abstract

As multi-core and many-core machines become more common, there is a growing need for data structures that scale well with the number of concurrent threads of execution. One performance bottleneck currently being addressed by designers of such data structures is mutual exclusion, since locking critical sections of an algorithm has well-known detrimental impacts on scalable performance. Non-blocking data structures offer an alternative method of synchronisation to mutual exclusion. They utilise things like the compare-and-swap atomic primitive to ensure concurrent operations on shared memory are done in a way that scales. Here I present the Rotating Skip List, a new kind of non-blocking skip list that utilises a special trick to avoid contention hot spots. Non-blocking skip lists offer an alternative to the B-Tree as a database indexing structure, since they can maintain a sorted order and allow for logarithmic query times. A C implementation of the Rotating Skip List has been benchmarked against a state-of-the-art non-blocking skip list, and a performance improvement factor of up to 2.65 was seen for certain test workloads.

Acknowledgements

I would like to thank my supervisor Dr Vincent Gramoli for all the help that he has provided me during my honours year. His untiring effort and enthusiasm was instrumental in all that I was able to accomplish. I would also like to thank the University of Sydney School of IT for providing me with excellent facilities and support throughout my honours year and the duration of my undergraduate degree. All of my tutors and lectures were excellent, and it was a great place to learn.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

The skip list [38] is a data structure closely related to the linked list (Figure 1 shows an example of a skip list). The key difference between skip lists and linked lists is that each regular node in a skip list has a height h, where h indicates the number of index nodes that are located above the regular node. Index nodes in a skip list are used during traversals to skip over some parts of the structure in between the start and end of a search. An index node at level k is connected to other index nodes at level k as well as to the index nodes directly below it. The height of any regular node in a skip list is typically decided in a probabilistic fashion, so that a regular node has height k with probability 2^{-k}. Augmenting a linked list in this way allows queries to occur in expected logarithmic time with respect to the number of list elements. Furthermore, skip list implementations can be simpler than implementations of other logarithmic data structures such as the binary tree. In fact, some databases have even begun to use skip lists by default in place of the more traditional B-tree indexing structure [3].

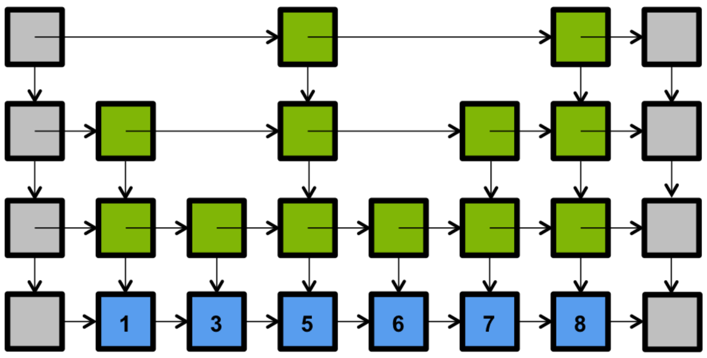

FIGURE 1.1: Example of a skip list - blue nodes represent the bottom-level value-storing nodes, green nodes represent the index-level nodes, and grey nodes represent the head and tail of the list (removing all the nodes in the green levels would transform the structure into a linked list)

Due to the plateauing of processor clock speeds in recent decades there has been a strong focus on exploiting parallelism to allow further performance increases to be made in computational problems. Several concurrent skip list implementations have been proposed that allow parallel operations to be conducted by multiple threads [16, 43, 14]. Concurrent data structures can be distinguished based on the style of synchronisation mechanisms used to manage concurrency. In particular, concurrent data structures are either *blocking* or *non-blocking* [25]. A blocking data structure is one that does not guarantee forward progress, for example if mutual exclusion is used to manage concurrency and deadlock becomes a possibility. In contrast, a non-blocking data structure provides the guarantee that forward progress can always be made, and can be achieved by using atomic primitives such as compare-and-swap instead of mutual exclusion to manage synchronisation.

Non-blocking data structures are attractive alternatives to their blocking counterparts due to the progress guarantees they imply, however using non-blocking techniques to manage concurrency introduces a great deal of complexity into the design process. Depending on the memory model of the computer architecture on which the data structure will be running, it can be much more difficult to ensure correctness of a non-blocking data structure. For example, the linux kernel documentation [27] discusses how special care needs to be taken when using atomic primitives such as compare-and-swap, since weak memory architectures such as DEC Alpha allow for re-ordering of memory operations that can result in incorrect behaviour without the careful use of memory barriers.

One of the reasons that non-blocking algorithms have been gaining momentum is related to recent trends in computer hardware. In particular, in the database community there has been a shift towards databases that operate completely in Random Access Memory (RAM), that only require disk-based memory for persistent backups and not for general operation. Removing the need for disk-based memory accesses during operation eliminates a major performance bottleneck, since database clients do not have to wait for data to be transferred to and from slower disk storage. Without this bottleneck, database designers have begun to consider other places where performance improvements can be made. One such performance issue is the contention induced by locks within database management systems (DBMS's). When multiple threads are accessing a shared database resource protected by a lock, only one thread may enter the critical section protected by the lock at a time, so the lock effectively serialises thread execution and undercuts the performance benefits offered by concurrency. Non-blocking synchronisation mechanisms provide concurrency management without resorting to locks and critical sections, and hence provide a possible solution to this performance bottleneck.

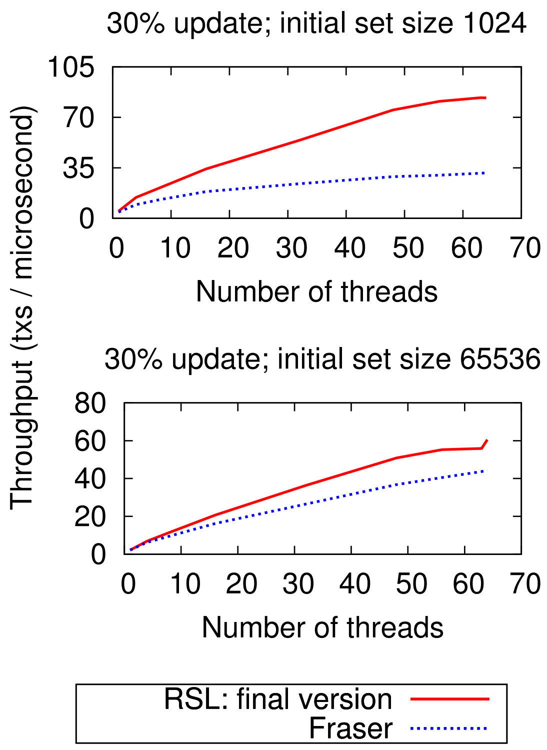

FIGURE 1.2: The final implementation of the Rotating Skip List achieves higher throughput than Fraser's skip list for certain workloads, since the performance of the Rotating Skip List scales better with the number of threads

The aim of this paper is to present the Rotating Skip List: a new kind of non-blocking concurrent skip list that is designed to perform well in high-contention scenarios. The final outcome for this new data structure is that superior performance is achievable under certain test conditions, relative to the performance of a state-of-the-art alternative: Fraser's non-blocking skip list (see Section 3.2.2). Figure 1.2 illustrates this outcome graphically. For a workload where 30 percent of all transactions are updates, the Rotating Skip List exhibits throughout up to 2.65 times greater than Fraser's skip list, since the Rotating Skip List scales better with the number of threads of execution. The performance difference is less noticeable in the case where the initial set size is larger (the Rotating Skip List achieves throughput up to 1.35 times that of Fraser's skip list), since the contention in these tests is diminished due to the wider range of set values, however the Rotating Skip List is still at an advantage.

The contribution of this paper is to showcase the performance benefits of the Rotating Skip List by presenting benchmark results taken from a C implementation developed as part of this work, and comparing these to results taken from the C implementation of Fraser's non-blocking skip list. Chapter 2 of this report will provide background into the problem, Chapter 3 will formalise the problem statement being addressed and outline the methodology used during the research, Chapter 4 will describe the Rotating Skip List in detail including performance results achieved during experiments, Chapter 5 will provide a discussion of the research outcomes and suggestions for future work, and Chapter 6 will conclude.

CHAPTER 2

Background

2.1 The Changing Nature of Database Systems

The traditional model for database management systems (DBMS's) has been under recent scrutiny due to its inability to cope with contemporary usage demands. Stonebraker et. al. [41] argue that the traditional database designs descending from IBM's System R are anachronistic given the drastic changes in hardware that have taken place in recent years, and suggest that instead of trying to modify these designs to suit modern needs it is better to start from scratch and build new systems from the ground up. As an example, traditional DBMS's have been optimised to work well with disk storage, which is a restriction that has impacted the design of software storage and indexing structures. As the cost of RAM decreases, however, new systems are being developed that no longer require disk storage for normal operations, and this has sparked research into new data structures optimised to work well in an in-memory context. The Bw-Tree [33], a data structure contained within an upcoming version of Microsoft's SQL Server product, is one example of a data structure designed without disk storage in mind, since it is designed to operate entirely within main memory and uses SSD storage as a stable secondary medium for logging and backing up.

Another aspect of the movement to make-over the traditional database design model and bring it up-to-date with contemporary hardware trends and usage scenarios is the growing prevalence of multi-core and many-core machines. The uni-core speed of processors is plateauing, and in response to this the number of cores per chip has begun to increase at an exponential rate for many prominent processor manufacturers [28]. Parallel computing offers an alternative way for programmers to pursue ever faster computations without relying on increases in sequential processor speed. Designers of data structures operating on multi-core or many-core machines must consider parallel scalability as a top priority in order to take advantage of this hardware trend. It has been shown that many popular DBMS's when run on a multi-core machine exhibit poor performance as the number of concurrent threads increases [28], illustrating the consequence of design decisions being made in an era when such scalability was not as relevant as it is today.

2.1.1 The Issues with Mutual Exclusion as a Synchronisation Tool

One reason that some database systems fail to scale as the level of concurrency increases is due to the impact of using mutual exclusion (locking) as a method for managing synchronisation. Protecting a critical section (shared resource) with a lock effectively serialises the execution of all concurrent threads trying to obtain the lock, since all threads waiting for the lock are blocked by the lock owner. On top of this, behaviour such as the cache-line "bouncing problem" [40] can occur when certain locks are used in a highly concurrent system. This can occur for locks such as the test-and-set spin lock, where threads waiting for the lock continuously read and update a shared lock resource, which results in cache invalidations for all other threads waiting for the same lock.

Boyd-Wickizer et al [7] examine the performance collapse that can arise when using ticket-style locks in the linux kernel. They advocate the use of more scalable locks [24], and illustrate experimentally the ability of scalable locks to mitigate performance collapse. Scalable locks improve performance by providing a bound on the level of inter-thread cache activity incurred during the use of a lock, regardless of the number of threads involved. This can be achieved using *atomic primitives* (see Section 2.2.2) to manage some parts of the locking mechanism. Jung et al [29] also use atomic primitives to reduce the number of latches (locks) in the MySQL lock manager, which they show to be a bottleneck restricting the scalability of MySQL. The performance of their augmented version of MySQL scales better with the level of concurrency than any of the other systems involved in the experiment.

The realisation that mutual exclusion is a bottleneck restricting scalability has sparked a trend towards completely lock-free data structures, which use things like the compare-and-swap atomic hardware primitive to synchronise threads without the use of locks. For example, the Hekaton database engine contained within an up-coming version of Microsoft's SQL Server product uses latch-free data structures such as the Bw-Tree exclusively in order to side-step the performance issues related to mutual exclusion [12]. In the research literature the term "lock-free" has come to take on a stronger, more general definition, whereby a data structure is lock-free if certain progress guarantees can be made regarding the data structure during program execution [25]. The term *non-blocking*, formally defined in Section 2.2.1, is used interchangeably to describe the same concept. The data structure produced as part of this work, the Rotating Skip List, is based on Crain, Gramoli and Raynal's No Hot Spot Non-Blocking

Skip List [10], and the following section will provide some background specific to this family of data structures.

2.2 Non-blocking Synchronisation Techniques

The changing computer hardware landscape is causing paradigm shifts to occur not only at the level of database system design, but also at the lower level of data structure design and implementation. As the number of available processing cores increases, so does the opportunity to exploit higher levels of parallelism. Hence, there is a need for data structures that perform well in a parallel context. It is not straight forward how to modify existing sequential data structures to achieve this goal, since data structures that have been optimised to perform well in a single-threaded environment may exhibit performance that scales poorly as the number of concurrent threads increases [28]. In this section we consider *non-blocking* data structures, which are data structures that offer some guarantees about the progress of threads during execution [25], and which provide solutions to some of the issues facing designers of scalable systems.

2.2.1 Definition of Non-Blocking

For the purposes of this paper, a data structure is said to be *non-blocking* if a collection of processes operating on the structure is guaranteed to make forward progress in a finite number of execution steps. In the past, researchers have used the term *lock-free* to convey the same meaning as the one just given for non-blocking [14, 34, 25, 16, 43, 9]. If a data structure is non-blocking then this rules out the use of mutual exclusion for synchronisation, since mutual exclusion can result in data structures that are blocking. An example of this is a thread that faults while owing a lock, which may permanently block other threads waiting to use the lock [25]. Another situation where mutual exclusion can result in a blocking data structure is if the data structure is prone to *deadlock* [37], where all threads operating on the structure are unable to proceed because they are waiting on locks that other threads currently own.

2.2.2 Atomic Primitives

Atomic primitives are hardware mechanisms that offer programmers a way to synchronise concurrent threads without using mutual exclusion. An example of an atomic primitive is the *compare-and-swap* operation (compare-and-swap is offered in the Intel x86-64 architecture via the *compare-and-exchange* instruction [1]). The sequential specification of compare-and-swap is given in Figure 2.1:

```
CAS(x,old,new) {
    if (x == old) {
        x = new;
        return true;
    } else {
        return false;
    }
}
```

FIGURE 2.1: Pseudocode for the compare-and-swap atomic primitive

A compare-and-swap can be used to atomically update a memory location in the following way: first the thread $T1$ reads the memory location x and saves the read value as old. Then $T1$ calculates a *new* value based on the value of old. Finally $T1$ performs $CAS(x, old, new)$, so that the value stored at x is only changed to *new* if it still contains the value old, and otherwise the compare-and-swap fails. This makes the update atomic because an update will fail if another thread $T2$ comes and changes the value stored at x between $T1$'s first read and subsequent compare-and-swap (compare-and-swap is atomic but is prone to the ABA problem, explained in Section 2.2.3). Crain, Gramoli and Raynal's No Hot Spot Non-Blocking Skip List, on which the Rotating Skip List presented here is based, uses single-word compare-and-swap exclusively to manage synchronisation, and this is also the approach taken to implement the Rotating Skip List.

Atomic primitives are useful for building non-blocking data structures because they do not involve mutual exclusion. The only thing that can stop a thread from successfully completing an atomic operation using a primitive is if a contending thread finishes the same operation first, but there will always be at least one thread that can proceed with execution. There are still issues to consider when using primitives, such as the possibility of *starvation* [23], where one or more threads are continuously disallowed from completing an operation because other threads are being favoured. A non-blocking data structure that is prone to starvation is not *wait-free* [22], which means that the non-blocking progress guarantees do not extend to each individual thread (i.e. the threads experiencing starvation).

There are some hardware atomic primitives that are less common, but that offer added benefits. For example, Load-Link/Store-Conditional (LL/SC) [35] is a less common alternative to compare-and-swap that has two steps: first the value is read from memory (LL) and then the value is updated to a new value (SC), but the SC step is only allowed to succeed if no other thread has performed an equivalent SC step in the meantime. The LL/SC primitive is less widely available than other primitives such as compare-and-swap, and is less practical as a candidate for a non-blocking data structure destined for wide-spread use.

2.2.3 The ABA Problem

To illustrate the ABA problem consider the following scenario: thread $T1$ reads memory location x and sets old to equal the value stored there. $T1$ then calculates a *new* value based on old and performs $CAS(x, old, new)$. But before $T1$ performs the compare-and-swap, another thread $T2$ changes the value stored at x from 0 to 1 and then back to 0. When $T1$ conducts the compare-and-swap it succeeds since the value at x is 0, however the resulting execution sequence is not atomic, and may result in undesirable behaviour. The LL/SC operation can be used to stop the ABA problem from happening, since the update step of LL/SC only succeeds if no other thread has changed the value of the memory location in the meantime. Maged Michael illustrated a way to construct the LL/SC primitive using just single-word read, write and compare-and-swap [35], providing a solution to the ABA problem on architectures that do not support LL/SC natively, however the author cautions that these constructions should only be thought of as worst-case backup mechanisms due to the restrictions they impose.

It is common for researchers to ignore the ABA problem during development of non-blocking data structures and algorithms, and assume the semantics of LL/SC are available in general in order to simplify reasoning [35]. This is the approach taken in this work during development of the Rotating Skip List. It is important to remember that the ABA problem can occur in practice, however, and that necessary steps should be taken to mitigate the ABA problem once a non-blocking data structure has reached the stage of being implemented for use in a production environment.

2.2.4 Classifying Primitives

Consensus

Primitives such as compare-and-swap and LL/SC can be classified according to how powerful they are as synchronisation tools. A metric commonly used to rate the power of a primitive is called the primitive's *consensus number* [22]. If a primitive has a consensus number of n, this means that the primitive can be used to construct a *consensus protocol* of n processes. A consensus protocol is a system of processes that each start with some input value and communicate through shared objects in order to agree on a common value chosen from the inputs. A consensus protocol must be consistent (all processes agree on the same value), wait-free (all processes decide on a value in a finite number of steps), and valid (the final agreed-upon value must have been an input to some process).

The implication here is that some primitives are more useful than others when it comes to implementing non-blocking data structures. For example, it has been shown that a primitive with consensus number n cannot be used to construct a primitive with a higher consensus number when there are more than n processes. Maurice Herlihy showed that the fetch-and-add primitive only has a consensus number of 2, whereas the compare-and-swap atomic primitive has an unbounded consensus number [22].

Combinability

Another metric that we can use to rate primitives is *combinability* [31]. Primitives are combinable if we are able to combine several memory accesses to the same location into one memory access, reducing contention in main memory. An example of a combinable primitive is the fetch-and-add primitive. To execute a fetch-and-add, a number stored at a memory location is fetched, an amount is added to this number, and the result is stored back to the original memory location. The fetch-and-add primitive is combinable since $FAA(memory\_location_x, a)$ and $FAA(memory\_location_x, b)$ can be combined into $FAA(memory\_location_x, a+b)$ when the actual access to main memory is made, reducing the number of memory accesses from two to one and thus avoiding issues of main memory contention. In contrast, compare-and-swap is not combinable, since the outcome of the operation depends on that value that is currently stored in memory.

2.2.5 Multi-word Primitives

All of the atomic primitives we have covered so far have been single-word primitives, meaning that they only operate on one memory word at a time. While these primitives can be used to implement more complex structures, there are other more powerful tools emerging that can be used to synchronise concurrent threads. *Multi-word primitives* act just like a single-word atomic primitives but affect more than a single memory word per operation. For example, some hardware platforms offer *hardware transactional memory* as a means for implementing concurrent data structures, whereby the commit and abort semantics typical to database systems are a built-in feature of the hardware (Intel [4] and IBM [20] have both proposed architectures that support this). Such features are not as common as single-word primitives in hardware platforms, but could suggest a direction of future advances in non-blocking parallel computation.

2.2.6 Non-Blocking Data Structures

There have been several instances of non-blocking data structures presented by the research community. Valois [44] was the first to use linked structures to implement a non-blocking dictionary. Harris [21] proposed a technique for decoupling node deletions from physical removals and he used this technique to design a non-blocking linked-list using compare-and-swap. The same decoupling approach was used by Michael to design a non-blocking linked-list and a list-based non-blocking hash table [34].

Some data structures have the advantage that they can maintain a sorted order while still only requiring logarithmic costs for accessing elements and performing updates. Examples of logarithmic data structures satisfying this property include binary search trees [30] (BST) and skip lists [38]. This is an attractive property, and is the reason why the B+-Tree (closely related to the binary tree) has been the index of choice for database systems for many years. Much work has gone on recently to make such logarithmic structures non-blocking and better suited for use in a concurrent environment.

2.2.6.1 Non-Blocking Binary Trees

Ellen et. al. [13] produced the first complete, non-blocking, linearisable BST using just single-word read, write and compare-and-swap (*linearisability* is a correctness condition and is explained in Section 3.1.3). This work built on several prior works on concurrent BST's, for example there are several examples that require mutual exclusion to manage concurrency [18, 32, 36, 6]. The tree proposed by Ellen et. al. is not self-balancing, however, and so only offers logarithmic query times in the average case. The Rotating Skip List presented here is balanced deterministically and so is able to be made arbitrarily well-balanced. Fraser [16] also illustrated a method for constructing a non-blocking BST, although his method requires the use of a multi-word compare-and-swap primitive.

Braginsky and Petrank [8] propose a non-blocking B+-Tree that is implemented using just single-word compare-and-swap operations. They report results of better contention handling and higher scalability in comparison to a lock-based B+-Tree. They also offer a proof of correctness of the linearisability of their implementation. Furthermore, since the B+-Tree is self-balancing, logarithmic query times are guaranteed in the worst-case. The Rotating Skip List developed as part of this work decouples structural maintenance from regular operations in order to reduce contention, and this could provide a performance advantage in comparison to Braginsky and Petrank's B+-Tree, although this was not tested experimentally.

Gramoli et. al. [11] build on the notion of decoupling the rebalancing of a balanced BST from insertions and deletions, which has been used in previous works to reduce contention [18, 36]. They use a transactional approach in order make their tree *speculation-friendly*, which offers another interesting approach to the problem of transforming the balanced BST into a scalable concurrent data structure. The transactional memory approach implicitly uses mutual exclusion to manage concurrency and so is not strictly non-blocking.

Levandoski et. al. [33] propose a new type of data structure, called a "Bw-Tree", which is cache-friendly version of a B+-Tree implemented using compare-and-swap to reduce blocking (the structure only blocks when it needs to fetch pages from stable storage). The B+-Tree is optimised to work with flash-based stable storage, and is organised around a mapping table that virtualises the location and size of pages. Updates are done by prepending update deltas to the prior page state, and this has the effect of reducing the number of cache invalidations. The authors report that their implementation outperforms BerkleyDB and a latch-free skip list implementation, although their data structure is subject to proprietary constraints and could not be benchmarked against the Rotating Skip List presented here.

2.2.6.2 Non-Blocking Skip Lists

Non-Blocking skip lists have received much recent attention from the research community. Since JDK 1.6 the standard Java API has included classes *ConcurrentSkipListMap* and *ConcurrentSkipListSet* [2], which were both implemented by Doug Lea. The skip list implementation of these classes involves a decoupling of node deletions into logical deletion and physical removal phases. Decoupling node deletions has also been used by Fraser [16] to design a non-blocking skip list (used for performance comparison in this work). Sundell and Tsigas [42] have also proposed a non-blocking skip list, which is used to implement a dictionary abstraction. Their approach also uses Harris logical deletion technique, and three atomic primitives: compare-and-swap, test-and-set and fetch-and-add. Fomitchev and Ruppert [14] proposed a non-blocking skip list where lookups physically remove nodes that have been marked for removal, however they do not offer an implementation of their approach.

Gramoli et. al. reuse the idea of decoupling structural changes that they applied to their speculation-friendly BST, and apply the same approach to their No Hot Spot Non-Blocking Skip List [10] (the basis of the Rotating Skip List presented here), which is implemented using single-word compare-and-swap. Their skip list avoids contention hot spots by postponing changes to the index levels of the skip list and leaving them to be carried out by a sole background thread. This way, multiple abstract modifications can be coalesced into possibly fewer modifications that cause less interruptions to other threads, and the abstract modifications can be made deterministic as opposed to the probabilistic balancing techniques common to most skip list implementations. They compare their implementation to Doug Lea's ConcurrentSkipListMap in the JDK, and show that their implementation performs better as the number of threads increases and also as the number of concurrent updates increases.

CHAPTER 3

Problem Statement and Methodology

The performance benefits of Crain, Gramoli and Raynal's No Hot Spot Non-Blocking Skip List have perviously previously been showcased experimentally using a Java implementation of the data structure [10]. The problem addressed here is that this Java implementation cannot be used to benchmark the data structure against others written in languages such as C. To address this problem, a version of the No Hot Spot Skip List was developed in C and tested against another non-blocking skip list written in C using a micro benchmark. It is more accurate to say that the skip list produced as part of this work is *based* on the No Hot Spot Skip List, since several changes to the initial specifications were necessary in order to achieve the end product. In particular, a memory reclamation module needed to be added to the program, since C programs do not have memory management by default. The skip list presented here has been renamed *The Rotating Skip List* to differentiate it from the original No Hot Spot Non-Blocking Skip List, and the reason for this name will be explained in Chapter 4.

3.1 Implementing a Non-Blocking Key-Value Store

The Rotating Skip List is an implementation of a *key-value store*, which is a simple map abstraction where arbitrary objects are stored and indexed (mapped) by distinct keys. Key-value stores have been receiving attention recently as they are well-suited to coping with modern demands such as the efficient storage of big data, the need for high scalability and availability, and the increasing level of concurrency brought about by the internet and cloud computing [19]. The key-value store abstraction can be implemented in a variety of ways. A hash table, a linked list, and a binary tree can all be used as the underlying data structure for a key-value store. As explained in Section 2.2.6, however, some data structures have the added benefit that they can maintain a sorted order of elements and still offer logarithmic query times. Binary trees and skip lists are two data structures that fall into this category. For this work the underlying data structure is a skip list.

3.1.1 The Key-Value Store Interface

Since The Rotating Skip List is a simple key-value store, it supports the following operations:

- *insert(k, v)* - insert the key *k* into the map so that it can be used to access the value *v*, and if *k* is already present in the map then do nothing
- *delete(k)* - if the key *k* exists in the map then remove it and its corresponding value, and if *k* is not already in the map then do nothing
- *contains(k)* - return true if the key *k* exists in the map and false otherwise

3.1.2 Non-Blocking Synchronisation

The fact that The Rotating Skip List is non-blocking means that special care must be taken to synchronise concurrent operations, since the non-blocking condition precludes the use of mutual exclusion for the reasons outlined in Section 2.1.1. This implies the use of tools such as atomic primitives as described in Section 2.2.2. Synchronisation of the Rotating Skip List is implemented using single-word compare-and-swap, as well as compiler memory barriers to guard against compiler re-orderings of program statements that would otherwise cause the program to be incorrect.

Hardware memory barriers are sometimes also necessary to ensure that reordering does not occur at the hardware level. Different computer architectures provide different guarantees in terms of what re-orderings of statements are allowed to occur, and for a program to be truly portable it must assume the guarantees are no better than the weakest computer architecture [27]. For this work the single assumed target architecture is Intel x86-64, which provides guarantees strong enough that no hardware barriers are necessary to ensure correctness. Compiler barriers were necessary to ensure that the gcc compiler did not re-order critical sections of the code.

3.1.3 Linearisability: A Correctness Condition

When designing concurrent data structures we need some way of ensuring that our implementation is correct, since the interleaving of multiple threads can cause unexpected results. *Linearisability* [26] is a necessary and sufficient condition for correctness of concurrent data structures. Informally, the operations invoked by concurrent objects in a linearisable system can be thought to have occurred instantaneously at some point between the their invocation and response, implying that these operations can have meaningful pre and post-conditions. Linearisability differs from other concurrency correctness conditions such as *serialisability* [39], since linearisability is a local property, where a system of objects is linearisable if each individual object is linearisable (a proof of serialisability involves considering all objects together and illustrating a serialisable sequence of execution). Linearisability is also a non-blocking condition, which makes it a good candidate for testing correctness of non-blocking data structures [10, 16, 21, 34, 13].

Crain, Gramoli, and Raynal declare linearisability to be the correctness condition achieved by the No Hot Spot Non-Blocking Skip List, and they offer a proof of linearisability of their data structure in the technical report version of their paper. Linearisability is also a characteristic desirable for the Rotating Skip List. No proof of linearisability is offered as part of this work as time did not permit this, however the proof is likely to be similar to the one for the No Hot Spot Non-Blocking Skip List, since the insert, delete and contains operations are implemented in a similar way.

3.1.4 Memory Reclamation for Unmanaged Languages

The Rotating Skip List presented here is designed to work in the context of an unmanaged language such as C. Managed programming languages such as Java simplify program development by providing memory management, so that unused memory resources are automatically reclaimed by the system without requiring explicit instructions from the programmer. Unmanaged languages such as C do not provide memory management, and these services must be provided explicitly by the programmer. The non-blocking memory reclamation module from Fraser's skip list (see Section 3.2.2) was modified so that it could be used with the Rotating Skip List. The memory reclamation portion of the program is non-trivial, since it is complicated by the need to be non-blocking, in order for the entire data structure to remain non-blocking.

3.2 Testing Performance in the Face of Contention

3.2.1 Synchrobench

Vincent Gramoli of the University of Sydney has created a micro-benchmark suite called Synchrobench [17] for benchmarking non-blocking data structures. The program is freely available on Github, and was used to benchmark the Rotating Skip List in this work. Synchrobench allows users to create custom benchmarking experiments by adjusting parameters, so that for example the level of contention (defined later in this section) can be increased or decreased.

3.2.2 The Comparison Skip List: Fraser's Non-Blocking Skip List

Fraser's non-blocking skip list [16] was used as a comparison skip list for this work. The JDK ConcurrentSkipListMap written by Doug Lea, which is a variant of Fraser's skip list, was used as the comparison data structure in the Not Hot Spot Non-Blocking Skip List paper [10]. Fraser's skip list is considered to be a state-of-the-art non-blocking skip list well-suited as a benchmark [5], and the original C implementation of Fraser's skip list is publicly available [15].

3.2.3 Definitions of Performance Metrics

In order to formally compare the performance of Fraser's skip list and the Rotating Skip List, it is necessary to provide some definitions of the performance metrics used. The main metric to be used in this paper is *throughput*, which shall be precisely defined as the number of transactions executed per second. A *transaction* in this instance is defined to be either a read-only transaction or an update, where an update is either an insertion or a deletion performed on the specified data structure. A read-only transaction simply returns the value of an element without conducting any modifications on the structure. In order to generate some throughput, Synchrobench was used to conduct transactions on the two data structures, and the respective levels of throughput were documented.

3.2.4 Experimental Settings

This research has focused on achieving high performance in situations where contention is high. *Contention* in this setting is defined to be the percentage of conflicting transactions that result when executing a benchmark with multiple concurrent threads. Inducing a particular level of contention can be done indirectly by manipulating parameters in Synchrobench, for example we could limit the range of possible transaction values. The smaller the range of values, the more likely it is that concurrent threads will be conducting transactions involving the same value, leading to higher contention. The relevant parameters and their experimental settings were as follows:

- *update percentage* - The percentage of update transactions (as opposed to read-only transactions) was varied between 0, 10 and 30 so that low, medium and high contention settings could be tested respectively (this parameter could have been made higher than 30 to induce even more contention, however it is more realistic for the update percentage to be at 30 or below).
- *initial set size and range* - The initial set size was varied between 2^{10} and 2^{16} elements and the range of possible values was fixed at twice the initial set size for all tests. The smaller the set size and range the higher the contention, since threads are more likely to compete with each other trying to update the same elements.
- *duration* - The duration of all tests was fixed at 5 seconds
- *number of threads* - the number of simultaneous threads of execution was varied between $\{1,4,8,16,32,48,56,63,64\}$. The higher the number of threads the higher the contention, since more threads are competing with one another to update the same set of elements.
- *effective updates* - Synchrobench allows users to specify whether or not update transactions should be *effective*: updates are effective if they must be successfully carried out to be considered processed. If updates are not effective and the update percentage parameter is set to 10, then 10 percent of transactions will be update transactions but there is no guarantee of these succeeding (e.g. an insert that fails to insert anything due to contention still contributes to this 10 percent). With effective updates enabled and the update percentage set to 10, 10 percent of all transactions must be successfully completed updates. Effective updates were used for all tests.

3.2.5 The Test Machine

The test machine used for all experiments carried out during this research was a Krypton Quattro R6010, which features 64 AMD Opteron processing cores. This meant that for all experiments, the number of threads of execution was less than or equal to the number of available cores, and it was assumed that the scheduler of the computer would schedule each thread onto its own core.

CHAPTER 4

The Rotating Skip List

As mentioned in Chapter 3, the problem addressed in this work is that Crain, Gramoli and Raynal's No Hot Spot Non-Blocking Skip List [10] has thus far only been implemented in Java. This Java implementation is not suitable for benchmarking the data structure against others that have been implemented in languages such as C. To address this, the Rotating Skip List was developed, which is based on the No Hot Spot Non-Blocking Skip List but which has a few unique qualities. These unique qualities were incorporated during implementation in order to improve performance, however, the core idea behind the Rotating Skip List is still the same as for the No Hot Spot Non-Blocking Skip List: structural skip list maintenance is decoupled from insert, delete and contains operations through the use of a background maintenance thread.

The Rotating Skip List is a non-blocking concurrent implementation of the key-value store abstract data type. The name "The Rotating Skip List" reflects the design of the underlying components, and this is explained in more depth in Section 4.3. As outlined in Section 3.1.1, the key-value store interface supports operations for inserting and deleting elements and testing for element membership. The low-level specifications of the Rotating Skip List are actually quite complicated in contrast to the simple interface they underpin. This is due to the fact that the data structure uses non-blocking synchronisation techniques (see Section 2.2), provides its own memory reclamation module, and uses Crain, Gramoli and Raynal's trick to avoid contention hot spots.

4.1 Overview of the No Hot Spot Non-Blocking Skip List

The No Hot Spot Non-Blocking Skip List is designed to perform well in high-contention situations, when there are many threads of execution performing updates and there is a high possibility of conflicts. Crain, Gramoli and Raynal realised that a major bottleneck in such situations is the contention that occurs during modifications to the upper index levels of the skip list. They decided that it would be beneficial for all index-level maintenance to be single-threaded, since this would remove the need for costly synchronisation mechanisms, and they proved the efficacy of this approach by comparing performance of the No Hot Spot Non-Blocking Skip List to a state-of-the-art alternative non-blocking skip list.

Index-level maintenance refers not only to the pointer manipulations made necessary by insert and delete operations, but also to re-balancing of the skip list. A side-effect of single-threaded maintenance is that balancing of the skip list can be done deterministically rather than probabilistically, and the No Hot Spot Non-Blocking Skip List employs deterministic balancing through raising and lowering routines carried out by the maintenance thread. For full design specifications of the No Hot Spot Non-Blocking Skip List, refer to the original paper [10].

4.2 The Naive Design

The No Hot Spot Skip List provided the starting point for the Rotating Skip List. The design of the Rotating Skip List in terms of the inherent algorithms and data structures initially matched the design of the No Hot Spot Non-Blocking Skip List. In particular, the index elements of the list were represented using distinct memory objects, which were connected to one another using rightwards and downwards pointers (see Figure 1.1). This design contrasts another common design technique for skip lists, which is to represent the index items above each bottom-level node as an array, implicitly connected to one another and accessible from one another using array manipulations.

The performance of the first iteration of the Rotating Skip List is shown in Figure 4.1. The Rotating Skip List at this stage exhibits worse performance than Fraser's skip list in all tests. This contradicted the hypothesis that using the no hot spot decoupling technique would reduce contention and provide an advantage over Fraser's algorithm. The problem is that there are other differences between Fraser's algorithm and this version of the Rotating Skip List that are contributing to this performance deficit. In particular, the Rotating Skip List at this stage uses distinct objects for index items, whereas Fraser's skip list uses the array-based approach described earlier for representing index towers.

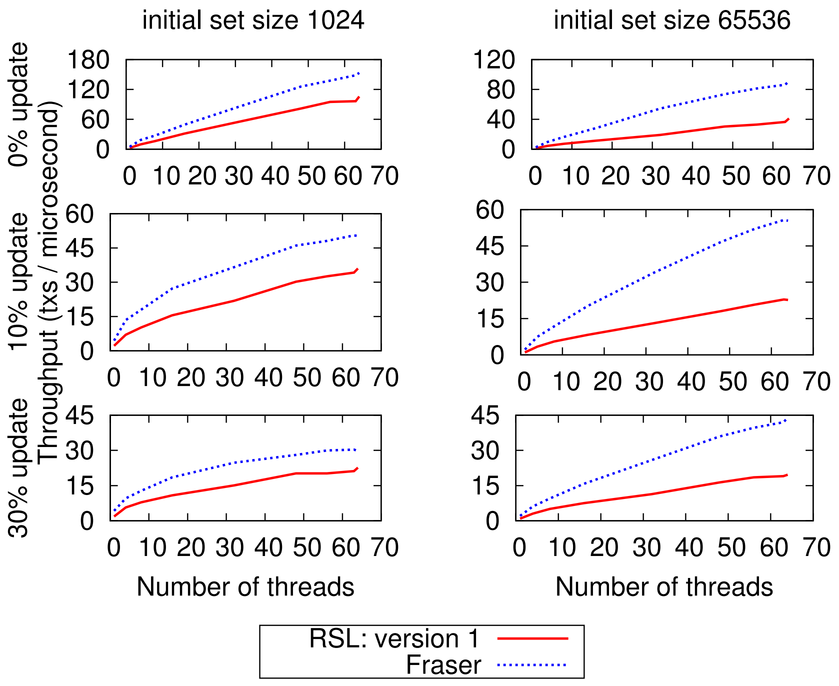

FIGURE 4.1: Naive implementation of the Rotating Skip List where all index nodes are distinct structures results in poor performance

4.3 Consolidation of Index Nodes into Rotating Arrays

In the first alteration to the Rotating Skip List, the data structure was modified so that the distinct index nodes were consolidated into arrays, similar to the approach used in Fraser's skip list (see Figure 4.2 for a graphical depiction of this change, and refer to Algorithm 1 for a pseudocode description of the Rotating Skip List node structures). Rotating arrays were used instead of regular arrays in order to incorporate the lowering algorithm from the No Hot Spot Non-Blocking Skip List. By decoupling skip list maintenance from regular operations and delegating it to a background thread, this allows the balancing of the skip list to become deterministic. The method devised by Crain, Gramoli and Raynal for balancing the No Hot Spot Skip List was to periodically *lower* the skip list by nullifying the bottom level of index nodes. This results in a shallower skip list that can be raised again by the background thread in order to achieve better balancing.

A lowering mechanism is necessary in order to rebalance a skip list that has become unbalanced, but the Rotating Skip List was unable to use this approach after the index levels were consolidated into arrays. It would be too costly to delete the bottom level of several arrays in order to carry out lowering, as this

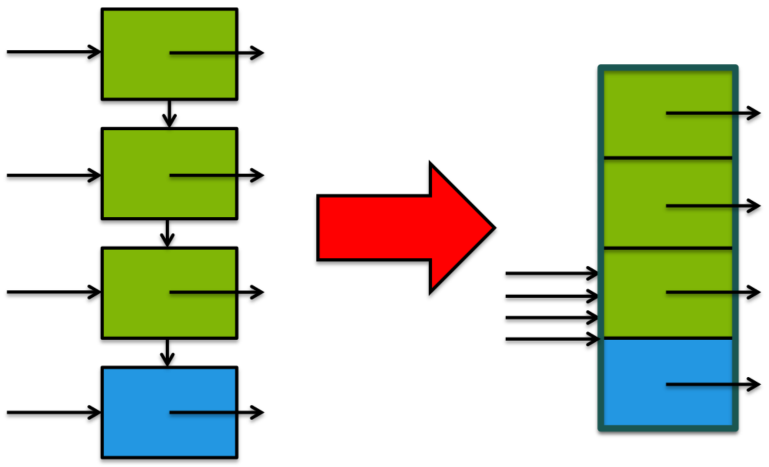

FIGURE 4.2: Nodes were consolidated into an array to improve memory locality

```
// get the ith element of the index array
node_t *next = index_array[ (ZERO + i) % N ];
```

FIGURE 4.3: The Rotating Skip List uses special array indexing: *N* is the length of *index_array*, *i* is the index element being accessed, and *ZERO* dictates which array level is currently being treated as the lowest

would necessitate the deletion of all the arrays and the creation of new array copies with the lowest level missing. To address this issue, a new style of lowering was devised for the Rotating Skip List. The arrays in the Rotating Skip List are *rotating* arrays as they are accessed using modulo arithmetic from within the program, so that increasing indexing past the end of an array wraps around to the beginning of the array (see Figure 4.3). This allows lowering to be done by simply increasing the index of the lowest logical level of all the arrays. A global variable *ZERO* is used to keep track of the current lowest level, and when a lowering occurs the lowest index level is invalidated by incrementing the *ZERO* variable. This causes other threads to stop traversing the index levels before they reach the previous lowest level of pointers (Figure 4.4 illustrates this graphically). If *ZERO* is increased above the length of the arrays this will not compromise the program since the arrays are accessed with modulo arithmetic. Pseudocode for the Rotating Skip List lowering mechanism is given in Algorithm 2 starting at line 7.

Benchmarking results gathered after the consolidation of index nodes are shown in Figure 4.5. The consolidation of index towers into arrays was expected to improve overall performance of the Rotating

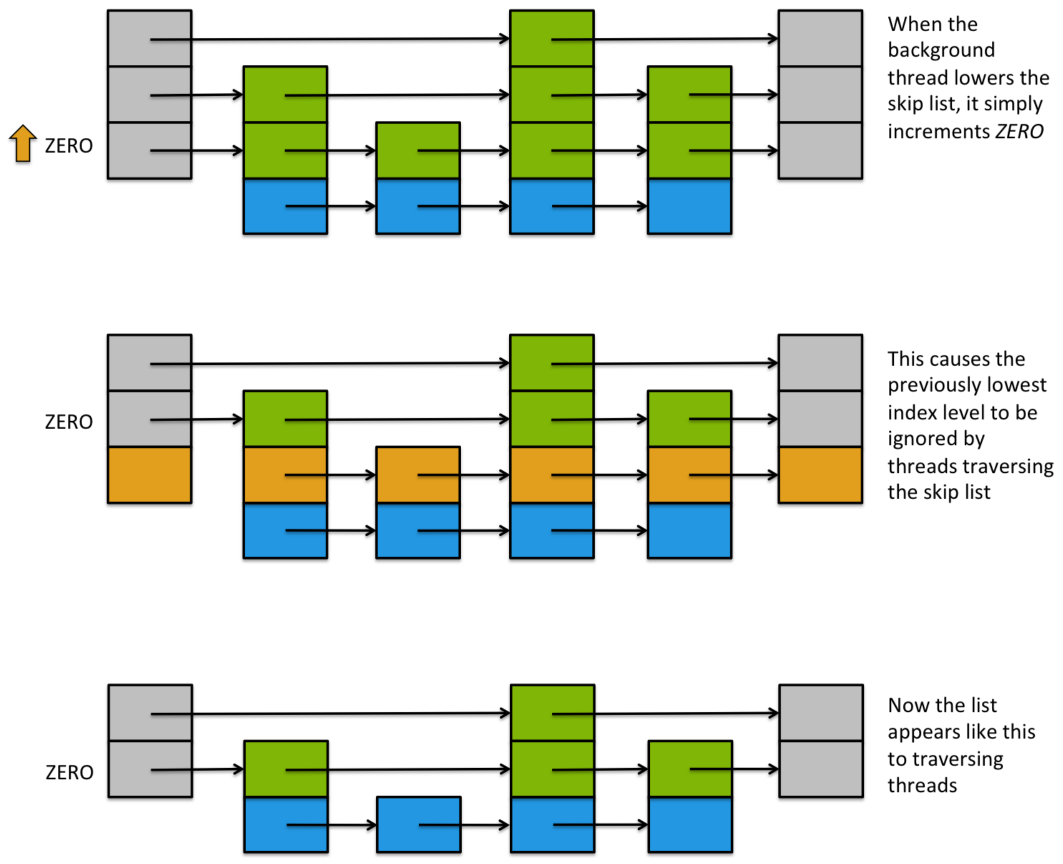
ZERO
When the background thread lowers the skip list, it simply increments *ZERO*
ZERO
This causes the previously lowest index level to be ignored by threads traversing the skip list
ZERO
Now the list appears like this to traversing threads

FIGURE 4.4: Lowering the Rotating Skip List

Skip List. Representing index towers using arrays has the side-effect of improving memory locality. To illustrate how this can improve performance, consider line 77 of Algorithm 1, which involves testing to see if the current node *next* in the traversal has a key greater than the search key. If the index nodes are represented using distinct objects, then *next.k* at line 77 would need to be changed to *next.node.k*, reflecting the fact that the key-value information is being stored in a distinct node object that the index *next* has a reference to. This extra layer of indirection can hurt performance, especially since *do_operation*() is the most commonly executed function of the Rotating Skip List. The improved memory locality that comes from consolidation of memory segments is likely to reduce the number of cache misses incurred during program execution, which can greatly improve performance as the upper cache levels of memory are much quicker to access than lower cache levels and main memory.

An improvement is noticeable when the initial set size is small, but when the set size is increased the performance gap between Fraser and the Rotating Skip List widens. Analysis of these results led to

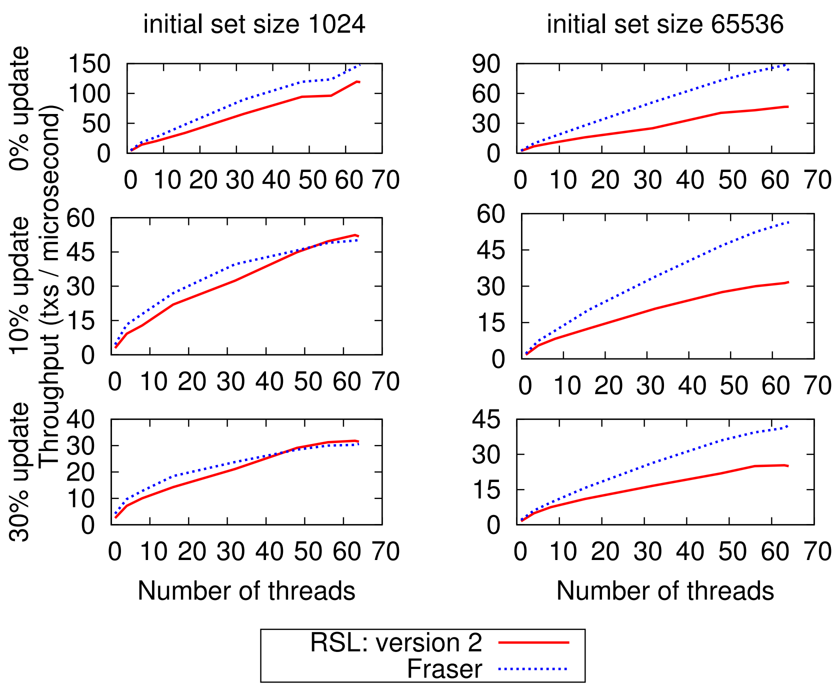

FIGURE 4.5: The Rotating Skip List with consolidated array node structures exhibits better performance for the smaller set size, but Fraser's skip list dominates when the set size if larger

the following observation: consolidation of nodes into arrays does provide a performance improvement by improving memory locality of list items, but this improvement alone is not enough to match the performance of Fraser, and some other difference between Fraser and the Rotating Skip List is creating this performance gap. The only other major difference between the two data structures is the fact that the Rotating Skip List uses a background thread for maintenance, which Fraser's skip list does not. Optimising the background thread became the focus of the next iteration of experiments.

4.4 Reducing the Frequency of Background Maintenance

To address the performance deficit between the Rotating Skip List and Fraser's non-blocking skip list, the Rotating Skip List background thread was set to sleep between maintenance iterations (see line 51 of Algorithm 2). This was done since it was discovered that the background thread can adversely affect performance if it runs too often. If the balancing of the skip list is not changing too much during execution then the background thread does not always need to be running. The adverse impact of the

Algorithm 1 The Rotating Skip List Interface and Helper Functions

```
state of a node:
    $node$ is a record with fields:
    - $level$: number of index items
    - $prev$: points to node preceding $node$
    - $next$: points to $node$'s successor
    - $k$: is used to retrieve $node$
    - $v$: is the value stored at $node$
    - $succs$: array of $level$ index pointers
    - $marker$: flag to indicate marker

finish_contains$(k, node, node\_v)$:
    $res \leftarrow 0$
    // check for logical deletion
    if $k = node.k \wedge node\_v \neq \bot$ then
        $res \leftarrow 1$
    return $res$

finish_delete$(k, node, node\_v)$:
    $res \leftarrow -1$
    if $k \neq node.k$ then
        $res \leftarrow 0$
    else
        if $node\_v \neq \bot$ then
            // loop until node logically deleted
            while $TRUE$ do
                $node\_v \leftarrow node.v$
                if $node\_v = \bot \vee node = node\_v$ then
                    $res \leftarrow 0$
                    break
                else if $CAS(\&node.v, node\_v, \bot)$ then
                    // don't fully remove the node
                    $res \leftarrow 1$
                    break
        else
            $res \leftarrow 0$
    return $res$

contains$(k)$:
    return $do\_operation(CONTAINS, k, \bot)$

delete$(k)$:
    return $do\_operation(DELETE, k, \bot)$

insert$(k, v)$:
    return $do\_operation(INSERT, k, v)$

finish_insert$(k, v, node, node\_v, next)$:
    $res \leftarrow -1$
    if $k = node.k$ then
        // k is already present in the skip list
        if $CAS(\&node.v, node\_v, v)$ then
            // insertion succeeded
            $res \leftarrow 1$
        else
            // insertion failed
            $res \leftarrow 0$
    else
        // key is not already in the list
        $new \leftarrow new\_node(k, v, node, next)$
        if $CAS(\&node.next, next, new)$ then
            if $next \neq \bot$ then
                // update the prev pointer
                $temp \leftarrow next.prev$
                $CAS(\&next.prev, temp, new)$
            // insertion succeeded
            $res \leftarrow 1$
        else
            // insertion failed due to
            // contention and -1 will
            // be returned so that we try again
            $delete\_node(new)$
    return $result$

do_operation$(optype, k, v)$:
    // get the logical zero index
    $zero \leftarrow GLOBAL\_ZERO$
    $i \leftarrow set.head.level$
    $item \leftarrow set.head$
    $res \leftarrow 0$
    // find entry point to node level
    while $TRUE$ do
        // get the next logical index item
        $next \leftarrow item.succs[IDX(i, zero)]$
        if $next = \bot \vee next.k > k$ then
            $next \leftarrow item$
            if $i = zero$ then
                // done traversing index levels
                $node \leftarrow item$
                break
            else
                // move down a level
                $i \leftarrow i - 1$;
        $item \leftarrow next$
    // find the correct node
    while $TRUE$ do
        // backtrack till we reach active node
        while $node = (node\_v \leftarrow node.v)$ do
            $node \leftarrow node.prev$
        $next \leftarrow node.next$
        if $next \neq \bot$ then
            $next\_v \leftarrow next.v$
            if $next = next\_v$ then
                // help with the removal of
                // nodes marked for deletion
                // and then try again
                $helpremove(node, next)$
                continue
        // finish the operation
        if $next = \bot \vee next.k > k$ then
            if $optype = CONTAINS$ then
                $res \leftarrow finish\_contains(k, node, node\_v)$
            else if $optype = DELETE$ then
                $res \leftarrow finish\_delete(k, node, node\_v)$
            else if $optype = INSERT$ then
                $res \leftarrow finish\_insert(k, v, node, node\_v, next)$
            if $res \neq -1$ then
                break
    return $res$
```

background thread on performance was not anticipated, since the main reason for incorporating the background thread is to decouple index level modifications from normal query operations, so that the two do not conflict with one another. In the Java implementation of the No Hot Spot Non-Blocking Skip List the background thread is set to run continuously without intermission, so it was surprising that using the same approach was not working once the algorithm had been ported to the C language.

The background thread modifies the skip list by removing references to deleted list items, and by raising and lowering index towers. These modifications result in index-level pointers in the data structure changing, and these pointers are also accessed by other threads as they traverse the skip list. When the pointers change, this could result in cache invalidations for other threads. Setting the background thread to sleep between iterations resulted in an increase in achievable throughput for the Rotating Skip List, and one explanation of this is that the number of cache invalidations was reduced, however this is merely a hypothesis and was not proved conclusively.

The results from benchmarking shown in Figure 4.6 were collected after the Rotating Skip List background thread was set to sleep for 1 second between iterations. Performance has again improved on previous attempts, with performance of the the Rotating Skip List for a small set size exceeding performance of Fraser by up to a factor of 1.69 when 10% of transactions are updates. There is still room for improvement, however, and in particular when the initial set size is large and the update percentage is 30 Fraser's performance overtakes The Rotating Skip List's performance when the number of threads is increased past 40. Another modification to the Rotating Skip List was needed for performance to exceed Fraser in all test scenarios.

4.5 Delegating Physical Deletes to the Background Thread

For the final major modification to the Rotating Skip List the style of deletions for normal threads was modified. In the initial specifications for the No Hot Spot Skip List [10] normal threads conducting a delete operation try to fully remove the item from the list using compare-and-swap operations. In order for these compare-and-swap operations to work correctly it is necessary to use memory barriers so that the compiler does not reorder the operations relative to surrounding program statements, and this adversely affects performance. As a result, the less compare-and-swap statements and barrier statements there are in the program the better performance will be, as long the program is still correct. Using this insight, the Rotating Skip List algorithm was changed so that threads conducting deletes never try to fully remove a deleted item from the list. Instead, when a delete is processed the corresponding thread simply conducts a logical deletion by nullifying the value of the item without actually removing it. This minimises the number of compare-and-swap operations needed to process a delete. Deleted items are still eventually removed from the list, they are simply removed later by the background thread during a maintenance iteration. Line 29 of Algorithm 1 shows where the physical deletion would occur if physical deletions were processed by threads conducting deletes; a call to the *remove()* routine from Algorithm 2

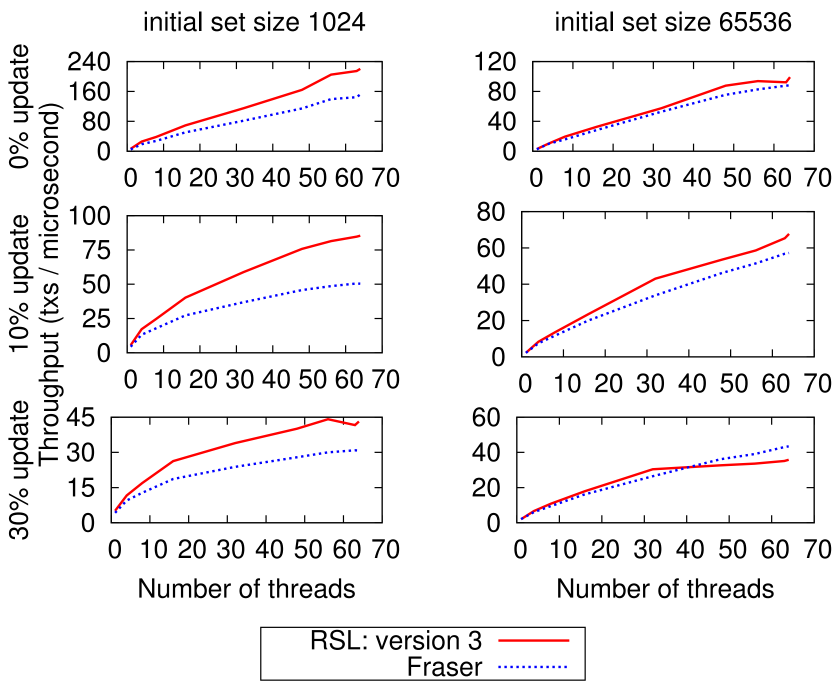

FIGURE 4.6: Setting the background thread to sleep for 1 second per iteration in the Rotating Skip list helped to improve performance, however when the initial set size is large and the update percentage is 30 Fraser's skip list exhibits better scaling

would be placed here to initiate a full physical deletion (this is the approach taken for the No Hot Spot Non-Blocking Skip List).

It may seem as though the number of costly compare-and-swap operations has not decreased at all, since they are only being postponed for the background thread to complete later. The key here is that if a thread tries to insert an item that has recently been logically deleted and not yet fully removed, the inserting thread can simply toggle the value for logical deletion off again so that the item is reactivated and considered a member of the list again. If full deletions are postponed then more instances of this style of insertion are likely to occur, which will reduce the amount of costly compare-and-swap and barrier operations and hence improve performance.

Leaving list items as logically deleted without fully removing them results in 'junk' nodes being left in the skip list that are not logical members of the structure but that still have to be traversed during a query. Hence, there is a performance tradeoff between the benefits provided by postponing full deletions

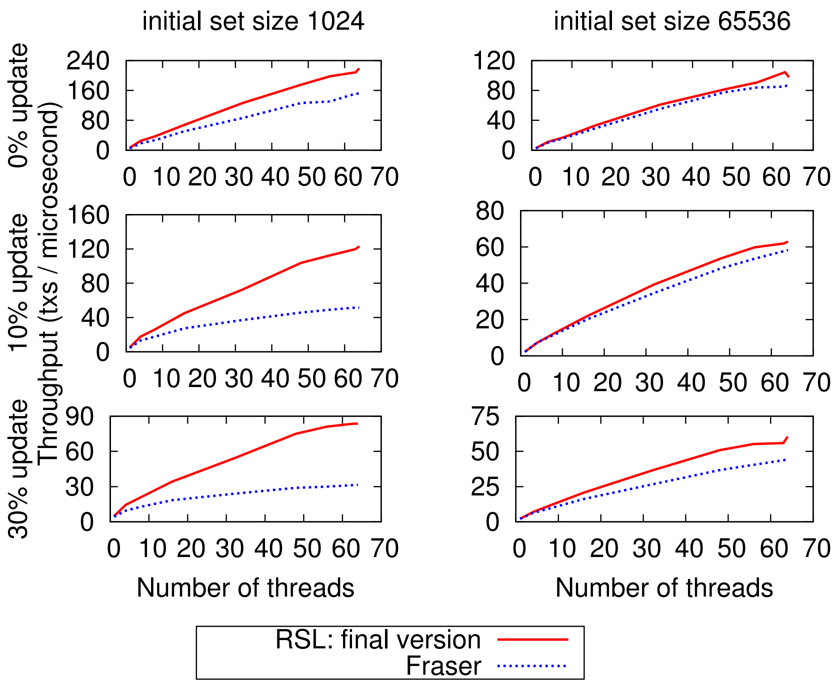

FIGURE 4.7: Delegating physical deletion of list items to the background thread was the final performance improvement, and now performance of the Rotating Skip List is superior in all tests

and the cost imposed by leaving logically deleted nodes in the list, however in this case the benefits of postponing physical deletions outweigh the costs.

The results from benchmarking after delegating physical deletions to the background thread are shown in Figure 4.7. It is clear that requiring all threads to fully remove deleted list items imposes a cost on the data structure, and postponing full deletions effectively mitigates this cost. The performance of the Rotating Skip List exceeds performance of Fraser's skip list in all test scenarios, with an improvement factor of 2.65 when the update percentage is 30 and the initial set size is 1024. As the update percentage increases, the performance gap between the Rotating Skip List and Fraser's skip list increases, indicating that the Rotating Skip List is coping better with the increased level of contention. The performance difference is less when the initial set size is increased to 65536 (with an improvement factor of up to 1.35 for an update percentage of 30), since the contention is diminished by the wider range of set values, but the performance difference is still noticeable. The pseudocode for the final version of the Rotating Skip List after all of the above modifications is given in Algorithm 1 and Algorithm 2.

Algorithm 2 The Rotating Skip List: Maintenance Operations Processed by Background Thread b (Some function definitions omitted for brevity)

```
remove($prev, node$)$_b$:
    if $node.level = 1$ then
        // only remove short nodes
        $CAS(\&node.v, \perp, node)$
        if $node.v = node$ then
            $helpremove(prev, node)$

lower_skiplist()$_b$:
    $node \leftarrow set.head$
    $zero \leftarrow GLOBAL\_ZERO$
    while $node \neq \perp$ do
        // nullify the index item
        $node.succs[IDX(zero, 0)] \leftarrow \perp$
        // decrement this node's level
        $node.level \leftarrow node.level - 1$
        $node \leftarrow node.next$
    // invalidate the bottom index level
    $GLOBAL\_ZERO \leftarrow$
            $GLOBAL\_ZERO + 1$

helpremove($prev, node$)$_b$:
    // check that this node
    // should be removed and is
    // not a marker
    if $node.v \neq node \wedge node.marker$ then
        return
    // loop till a marker succeeds node
    $n \leftarrow node.next$
    while $n = \perp \vee n.marker \neq TRUE$ do
        $new \leftarrow new\_marker(node, n)$
        $CAS(\&node.next, n, new)$
        $n \leftarrow node.next$
    if $prev.next \neq node \vee prev.marker$ then
        return
    // try to CAS node and marker out
    // of the list
    $res \leftarrow CAS(\&prev.next, node, n.next)$
    if $res$ then
        // free memory for node and marker
        $delete\_node(node)$
        $delete\_node(n)$
    // fix prev pointer
    $prev\_next \leftarrow prev.next$
    if $prev\_next \neq \perp$ then
        $prev\_next.prev \leftarrow prev$

background_loop()$_b$:
    $node \leftarrow set.head$
    $zero \leftarrow GLOBAL\_ZERO$
    while not done do
        $sleep(SLEEP\_TIME)$
        $raised \leftarrow raise\_bottom\_level()$
        if $raised = TRUE$ then
            for $i \in \{0..top\_level\}$ do
                $raise\_index\_level(i)$
        // if the skip list has become
        // too unbalanced then lower
        if $LOWERING\_NECESSARY$ then
            $lower\_skiplist()$

raise_bottom_level()$_b$:
    $node \leftarrow set.head$
    $next \leftarrow node.next$
    while $node \neq \perp$ do
        if $node.val = \perp$ then
            // help with removals
            $remove(prev, node)$
        else if $node.v \neq node$ then
            // only raise non-deleted nodes
            $raise\_node(node)$
        $node \leftarrow node.next$
```

4.6 Memory Reclamation for The Rotating Skip List

Since C is an unmanaged language and there are no memory management facilities provided by default, a memory reclamation module must be provided in order for a C program to be able to reuse memory segments that have been decommissioned. Up to this stage in the research all tests were conducted without memory reclamation enabled, so that the performance characteristics of the two skip lists could be analysed without influence from the memory reclamation code. Once desirable performance for the Rotating Skip List was achieved the memory reclamation modules for the two skip lists were enabled, and performance was again evaluated to see how the data structures perform in a more realistic setting. Fraser's implementation provides a memory reclamation module, which can be turned on or off from within the code. To provide memory management for the Rotating Skip List, Fraser's memory reclamation was modified so that it could be used. Enabling memory reclamation was not expected to alter the performance outcomes from previous experiments.

The results shown in Figure 4.8 were taken after memory reclamation was enabled for both skip lists. The results are largely the same as those shown in Figure 4.7, however there is a noticeable drop in

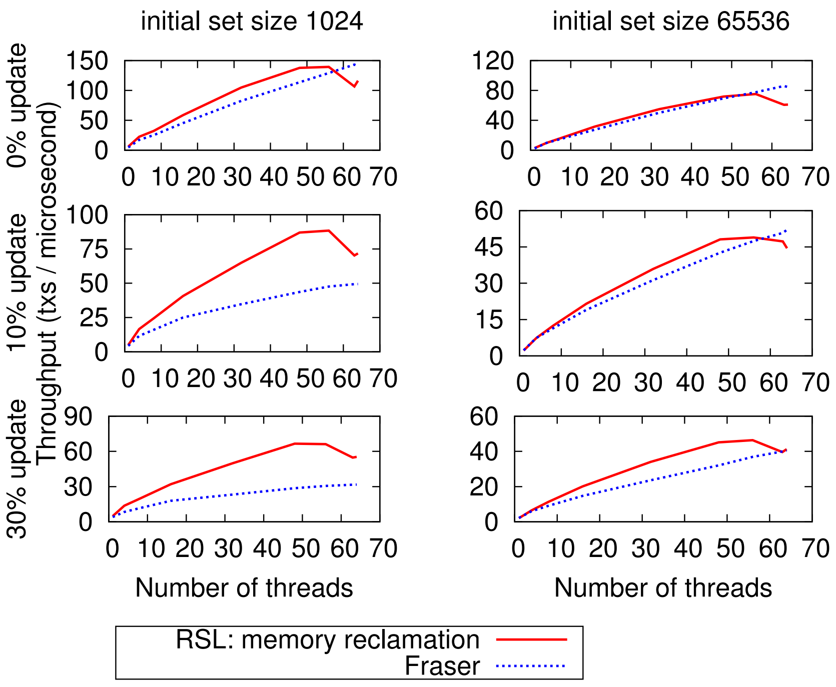

FIGURE 4.8: Turning on memory reclamation resulted in the Rotating Skip List throughput dipping for a high number of threads

performance for the Rotating Skip List when the number of threads of execution is high. These results prompted an investigation into what differences between the two data structures could have led to this performance degradation in the Rotating Skip List and not in Fraser's skip list.

In the case of Fraser, the memory reclamation module is partitioned into sections responsible for managing nodes of a particular skip list level. Partitioning the memory manager like this means that requests to the memory manager regarding different skip list levels do not need to conflict with one another, and can be processed by distinct portions of the memory manager. In contrast, the Rotating Skip List does no such partitioning of memory management responsibilities, since the Rotating Skip List uses only one node size for all list elements regardless of their level. This means that it is likely that there is contention in the memory reclamation module when a large number of threads are making memory requests simultaneously. In order to deal with this issue the parameters of the memory reclamation module in the Rotating Skip List were modified to see if they could mitigate the performance drop.

4.6.1 Reducing the Frequency of Memory Reclamation

The memory manager in the Rotating Skip List was modified so that the frequency of memory reclamation was reduced. It was discovered through experimentation that the reclamation routine of the memory manager incurs noticeable performance costs whenever it is invoked, since threads need to be synchronised so that they can communicate and all the memory marked for reclamation can be coalesced. Reducing the frequency of memory reclamation means that memory segments marked for reclamation take longer to be collected, so it is important to make sure that the memory usage of the program remains reasonable.

The results shown in Figure 4.9 were taken after the frequency of the Rotating Skip List's memory reclamation was reduced. With the frequency of memory reclamation reduced, the performance of the Rotating Skip List has improved so that it looks similar to the results from Figure 4.7. The performance drop for a high number of threads seen in Figure 4.8 is no longer noticeable, and the Rotating Skip List's performance scales well with the number of threads of execution and also with the level of contention influenced by the percentage of update transactions. The frequency of memory reclamation was not reduced for Fraser's skip list, and this is a source of bias for the results in Figure 4.9, since the same advantage was not granted to Fraser for the tests. The goal of this last iteration of experiments was less concerned with comparing performance of the two skip lists, and more concerned with establishing that the source of the performance drop in Figure 4.8 was simply due to the memory reclamation module requiring tuning, rather than a significant problem with the Rotating Skip List data structure.

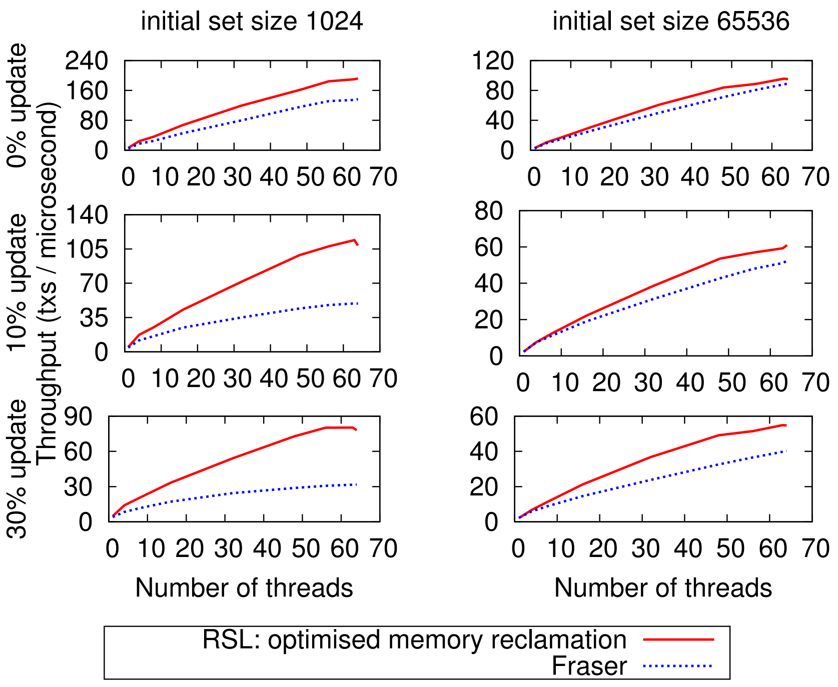

FIGURE 4.9: After adjusting a parameter to reduce the frequency of memory reclamation the performance dip noticeable in Figure 4.8 was removed, and the Rotating Skip List now performs well with memory reclamation enabled

CHAPTER 5

Discussion and Suggestions for Future Work

5.1 Insights Reached During Development of The Rotating Skip List

The goal of this research was to showcase the performance benefits of the No Hot Spot Non-Blocking Skip List [10] once it has been implemented in an unmanaged language. In particular, the goal was to test the efficacy of using a background thread to conduct skip list maintenance. At the beginning of this research it was hypothesised that the design specifications of the No Hot Spot Non-Blocking Skip List would not need to change substantially for an implementation in an unmanaged language to be developed, since the performance benefits were believed to be language agnostic. During development of a C implementation, however, enough changes to the original design were necessary that the data structure presented here was renamed the Rotating Skip List. This section will recap on the changes that were necessary and summarise the insights gained during experimentation.

The differences between the No Hot Spot Non-Blocking Skip List and the Rotating Skip List are summarised in Table 5.1. After considering these differences together and the results from Chapter 4 the main insight to be gained is the following: decoupling skip list maintenance from regular operations can provide performance benefits in both the Java and C contexts, however these performance benefits do not override other design considerations, such as the internal structures used to implement the skip list, and the use of synchronisation tools and their impact on performance.

In order to achieve performance superior to the comparison skip list for the transaction workloads being tested, several changes to the initial Rotating Skip List design were necessary. Using distinct nodes to represent index items proved ineffective in the C context, and node consolidation was necessary to improve memory locality and reduce the penalty paid by dereferencing pointers during traversals. Second, for transaction workloads that do not substantially change the balancing of the skip list it is beneficial to postpone skip list maintenance to reduce the contention on the data structure. A side-effect

The No Hot Spot Skip List	The Rotating Skip List
distinct index nodes connected using pointers	index nodes represented using arrays
background thread runs continuously	background thread sleeps between iterations
all threads process physical deletes	physical deletes delegated to the background thread

TABLE 5.1: Differences between the Rotating Skip List and The No Hot Spot Non-Blocking Skip List

of restricting the range of possible values to twice the initial set size was that the Rotating Skip List did not become too unbalanced during testing, and it was preferable to postpone maintenance to reduce contention. Lastly, even though a value proposition of non-blocking data structures is that they perform well in high-contention scenarios, this does not preclude the need for judicious use of the associated non-blocking synchronisation primitives. This was illustrated when node deletions were modified in the Rotating Skip List to reduce the amount of compare-and-swap primitives executed by threads, and performance increased noticeably.

5.2 Suggestions for Future Work

The Rotating Skip List presented here has successfully illustrated that the design principles of the Not Hot Spot Non-Blocking Skip List can provide performance benefits in unmanaged languages like C, however there are still unanswered questions related to this research. Further improvements to the Rotating Skip List could be made so that the data structure performs well in test scenarios other than the ones focused on in this report, and the interface of the Rotating Skip List could be expanded so that it is more suitable as a data structure for use in a production database setting.

5.2.1 Improving Performance in Other Test Scenarios

In Section 3.2.4 the experimental settings for benchmarking were listed. In particular, the range of possible set values was said to be fixed at twice the initial set size in all tests. As aforementioned, this caused the Rotating Skip List to remain balanced during testing, which is why reducing the frequency of the maintenance thread was so effective. Having the range of set values much larger than the initial set size causes the Rotating Skip List to temporarily become unbalanced, as new elements are added to the set across a wide range of positions and the skip list becomes very shallow. The Rotating Skip List maintenance thread sleeps during maintenance iterations, and although this improved performance for

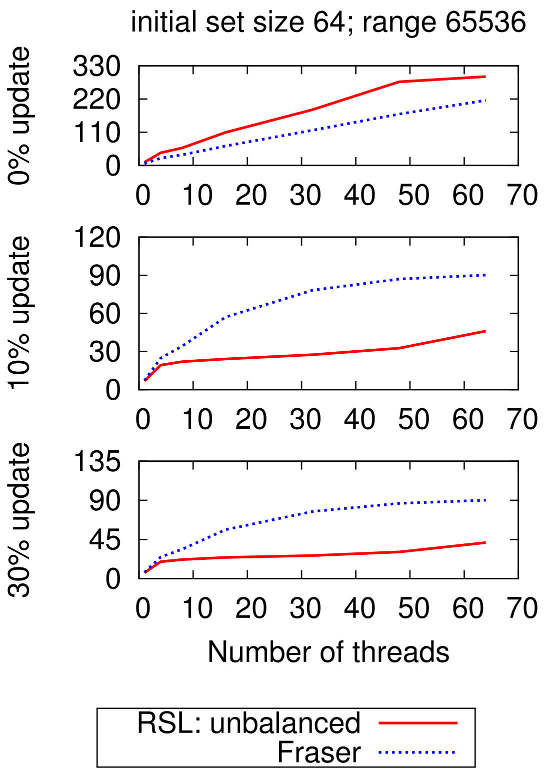

FIGURE 5.1: The Rotating Skip List becomes unbalanced for certain workloads, and as a result performance drops

the tests in Chapter 4 it means that it takes longer for the structure to become balanced again after it has become unbalanced.

Poor performance was seen for tests where the range of values was much larger than the initial set size. Figure 5.1 shows performance results when the initial set size is just 64 and the size of the range of possible values is 65536. As soon at the percentage of updates rises above 0, the Rotating Skip List becomes unbalanced due to the wide range of set values and as a result performance drops significantly in comparison to Fraser's skip list. A variety of different sleep times were tested for the maintenance thread, for example it was set to sleep for 100 milliseconds per iteration and also 1 millisecond per iteration, to see if reducing the sleep time would have a positive impact for workloads such as the one corresponding to Figure 5.1, and some improvements were noticed by reducing the sleep time.

One possible solution for dealing with this issue is to change the background module of the Rotating Skip List to allow it to respond dynamically to changes in transaction workload. There is a trade-off between the benefits of reducing the background thread frequency and the increased time for the skip list to become balanced again. While the balancing of the skip list is not being overly affected by the transaction workload it is beneficial to have the background thread running infrequently, but as the skip list becomes unbalanced the tradeoff begins to swing in the opposite direction, and it makes sense to increase the frequency of the background thread so that the skip list can be re-balanced quickly. A dynamic background thread could be achieved by developing an heuristic for testing the balancing of the skip list during execution, which the background thread could use to test whether or not it needs to increase or decrease its frequency. The heuristic could include things such as the number of elements at each skip list level.

5.2.2 Expanding the Interface of The Rotating Skip List

At the commencement of this research one of the goals was to expand the Rotating Skip List interface to handle more complex queries such as range queries. This would make the data structure more suitable as a component in a production database, and would allow the Rotating Skip List to be benchmarked in a wider variety of scenarios to provide further evidence of its efficacy. Providing a range query mechanism for a non-blocking concurrent skip list is a non-trivial exercise. The most simple approach would be to somehow lock the data structure during range queries so that a consistent snap shot of the structure could be taken, however this style of solution would violate the non-blocking condition and would adversely affect performance.

Avni, Shavit, and Suissa recently proposed a new data structure named the “Leaplist” [5] that utilises Software Transactional Memory (STM) in order to provide range query functionality in a consistent manner. The Leaplist stores arrays of *k* key-value objects at each individual node, which allows linearisable range queries to be processed *k* times faster than an equivalent query performed non-linearisably on a skip list. The authors benchmark an implementation of their Leaplist against Fraser’s non-blocking skip list (the same skip list used for comparison in this work) and illustrate that for mixed workloads involving range queries the Leaplist is able to achieve twice the throughput of Fraser’s non-blocking skip list. For workloads that do not involve range queries, however, the performance of the Leaplist drops below that of Fraser’s skip list due to the higher overhead of insert and remove operations in the Leaplist.

A possible approach to range queries tailored to the Rotating Skip List would be to utilise the background thread to minimise processing time. The background thread could take a snap shot of the skip list during a maintenance traversal, cache this snap shot, and return it to threads executing range queries. There is no guarantee that such a snap shot would reflect a consistent past state of the skip list, since other threads may be modifying the skip list while the snap shot is being taken, and a more complicated approach would be necessary to allow for consistent and efficient range queries.

CHAPTER 6

Conclusion

In this report I have presented a novel data structure named the Rotating Skip List, which is a non-blocking concurrent skip list designed to scale well with the number of concurrent threads of execution. The Rotating Skip List is based on a previously proposed data structure named the No Hot Spot Non-Blocking Skip List [10], which avoids contention hot spots by decoupling structural maintenance from insert and delete operations. A C implementation of the Rotating Skip List was developed and benchmarked against a state-of-the-art non-blocking skip list [16], and performance improvements of up to a factor of 2.65 were seen for certain transaction workloads.

Although the Rotating Skip List has been shown to perform well in certain scenarios, there are improvements that can be made to the data structure in its current form. The Rotating Skip List maintenance thread could be augmented to respond dynamically to changes in transaction workload, which would allow the Rotating Skip List to re-balance itself more effectively. The Rotating Skip List could also be extended to handle range queries. This would make the Rotating Skip List more suitable as an indexing structure in a modern database system, however this is a non-trivial extension to implement due to the non-blocking nature of the data structure.

Non-blocking synchronisation techniques offer an alternative to mutual exclusion for managing concurrency in parallel applications. Mutual exclusion is known to exhibit poor scalability as the number of concurrent threads of execution is increased on modern multi-core and many-core machines, and as a result there is a growing trend towards the adoption of non-blocking data structures in highly parallel and performance-critical applications. The Rotating Skip List proposed here builds on previous research efforts that aim to utilise non-blocking synchronisation techniques to create scalable and efficient data structures for use in applications such as high performance in-memory database engines.

Bibliography

[1] Intel 64 and ia-32 architectures software developer's manual, combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. http://download.intel.com/products/processor/manual/325462.pdf.

[2] Java 7 api documentation. http://docs.oracle.com/javase/7/docs/api/.

[3] Memsql indexes. http://developers.memsql.com/docs/1b/indexes.html.

[4] Transactional synchronisation in haswell (intel blog post). http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell.

[5] Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: Lessons learned in designing tm-supported range queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC)*, 2013.

[6] Joan Boyar, Rolf Fagerberg, and Kim S Larsen. Amortization results for chromatic search trees, with an application to priority queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997.

[7] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. Citeseer, 2012.

[8] Anastasia Braginsky and Erez Petrank. A lock-free b+ tree. In *Proceedinbgs of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67. ACM, 2012.

[9] Daniel Cederman, Anders Gidenstam, Phuong Ha, Håkan Sundell, Marina Papatriantafilou, and Philippas Tsigas. Lock-free concurrent data structures. *arXiv preprint arXiv:1302.2757*, 2013. To appear in "Programming Multi-core and Many-core Computing Systems", eds. S. Pllana and F. Xhafa, Wiley Series on Parallel and Distributed Computing.

[10] Tyler Crain, Vincent Gramoli, and Michael Raynal. No hot spot non-blocking skip list. Technical report, 2012. To appear in the 33rd International Conference on Distributed Computing Systems (ICDCS), 2013.

[11] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. *SIGPLAN Not.*, 47(8):161–170, February 2012.

[12] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql serverâĂŹs memory-optimized oltp engine. Sigmod, 2013.

[13] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140. ACM, 2010.

[14] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, pages 50–59. ACM, 2004.

[15] Keir Fraser. C implementation of fraser's non-blocking skip list. `http://www.cl.cam.ac.uk/research/srg/netos/lock-free/`.

[16] Keir Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[17] Vincent Gramoli. Synchrobench: A microbenchmark for non-blocking data structures. `https://github.com/gramoli/synchrobench`.

[18] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on the Foundations of Computer Science, 1978.*, pages 8–21, 1978.

[19] Jing Han, E. Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363–366, 2011.

[20] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48–60, 2012.

[21] Timothy Harris. A pragmatic implementation of non-blocking linked-lists. *Distributed Computing*, pages 300–314, 2001.

[22] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[23] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[24] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kauffmann, 2008.

[25] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Principles of Distributed Systems*, pages 313–328. Springer, 2011.

[26] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[27] David Howells. Linux kernel memory barriers. `https://www.kernel.org/doc/Documentation/memory-barriers.txt`.

[28] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 24–35, New York, NY, USA, 2009. ACM.

[29] Hyungsoo Jung, Hyuck Han, Alan D Fekete, Gernot Heiser, and Heon Y Yeom. A scalable lock manager for multicores. In *Proceedings of the 2013 international conference on Management of data*, pages 73–84. ACM, 2013.

[30] Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[31] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, October 1988.

[32] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, September 1980.

[33] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *The 29th IEEE International Conference on Data Engineering (ICDE), April 2013*, 2013.

[34] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[35] Maged M Michael. Aba prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004. http://www.research.ibm.com/people/m/michael/RC23089.pdf.

[36] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic binary search trees. *Acta Informatica*, 33(5):547–557, 1996.

[37] Peter S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[38] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[39] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[40] C. Schimmel. *NIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*. Addison-Wesley, 1994.

[41] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[42] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1438–1445. ACM, 2004.

[43] Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2004.

[44] John David Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1996.