Contention in Concurrent Data Structures

JOEL GIBSON
SID: 311199828

Supervisor: Dr Vincent Gramoli

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Science (Advanced Mathematics) (Honours)

School of Information Technology
The University of Sydney
Australia

June 9, 2015

Student Plagiarism: Compliance Statement

Name: Joel Gibson

Signature: **Date**:

Abstract

Mutual-exclusion locks are a common mechanism for protecting shared data in a concurrent environment. However, as machines with tens or hundreds of cores become more common in database systems, the poor scaling properties of locks become problematic. Various lock-free data structures have been proposed as a solution to these problems which work well in practice, however the analysis of their asymptotic complexities has not kept pace with their practical development. Most lock-free algorithms have no accompanying proof of running time, and whenever proofs are given, they are often very complicated, partly in due to authors striving to achieve a bound in terms of the tightest possible notion of contention.

I present new theoretical results about contention as it is used in the analysis of lock-free data structures, showing two commonly used measures of contention equivalent. I use these results to provide a simplified proof of a lock-free linked list, and suggest a modification which increases performance without changing the asymptotic complexity. I implement four state-of-the-art concurrent linked list algorithms in C and experimentally evaluate their performance under a wide range of conditions, showing that contention has an observable impact on performance, and exploring the scaling properties of both lock-based and lock-free structures. Finally, I suggest directions for future work in both practical and theoretical areas.

Contents

List of Figures

Chapter 1

Introduction

Traditionally, systems built to handle a high level of concurrency have used sequential data structures protected by mutual exclusion locks to ensure data consistency. Wherever this has impacted performance, more concurrency-friendly locking strategies have been applied, such as readers-writer locks, or by pushing the locks inside the data structure itself to achieve a more fine-grained locking approach. However, it is becoming common today to see machines with tens or hundreds of cores and multiple terabytes of shared memory, so completely in-memory databases or caches are becoming viable. At this scale, the inherent costs and complications of locking become problematic: a better approach to concurrency is needed.

Lock-free data structures offer an alternative approach to synchronisation, relying on atomic primitives rather than mutual exclusion to ensure consistency and safe access to shared data. The absence of locks makes lock-free structures immune to common problems such as deadlocks, priority inversion, and convoying, while the strong progress guarantees of lock-free structures makes them resilient to poor scheduling decisions, and tolerant of thread failures. These progress guarantees also make it possible to, in some cases, put an upper bound on the total number of steps the system must execute in order to complete a series of operations. This bound includes a *contention* parameter, a measure of the number of operations simultaneously accessing the data structure. These types of bounds are not achievable for lock-based structures, since the scheduler could at any point suspend a thread, blocking other threads indefinitely.

As lock-free data structures become more common, there is a growing need for good theory backing up their observed practical performance. Furthermore, lock-free data structures should be compared against recent lock-based data structures in order to evaluate the merits and drawbacks of each.

1.1 Contributions

My main contributions to this area of research are in the analysis of lock-free data structures, and the experimental evaluation of recent high-performance concurrent linked lists. I take existing measures of contention used in the (amortised) analysis of lock-free data structures and analyse them, showing the point contention, process contention, and interval contention are equivalent. I use this theoretical result to simplify the proof of complexity of an existing data structure, and also make practical improvements to the algorithm itself. I conduct a survey of existing concurrent linked-list algorithms, testing their performance under varying workloads to show that contention is a factor that cannot be ignored. Finally, I suggest directions for future work based off the ideas presented here.

In Chapter 2, I go over some necessary background about lock-free data structures and their implementation. Following this, I introduce the definitions of contention that are currently used in the analysis of lock-free data structures, and comment on how the point and interval contention have been used recently. I finish by performing a survey of the literature, focusing on practical lock-free linked lists and their lock-based alternatives.

In Chapter 3, I take the interval and point contention, two measures of contention that are commonly used today in the analysis of non-blocking data structures, and show their equivalence in an amortised context. This is shown by relating an execution to its corresponding *interval graph*, and using properties of interval graphs to put a lower bound on the overall point contention in terms of the overall interval contention. I show the tightness of these bounds, and briefly discuss related bounds for other definitions of contention.

In Chapter 4, I select five concurrent linked list algorithms from the literature and outline their design, commenting on their complexities where possible. Of these five, four are state-of-the-art high-performance linked lists, and the other is a linked list produced from a universal construction. I use the theoretical results developed in Chapter 3 to provide a simplified proof of the running time of one list, and create my own modification which performs better in practice.

In Chapter 5, I present results from the experiments I conducted on the lists, showing the differences between locking and lock-free data structures. I show that the effects of contention are observable and non-negligible in practice, and finish by evaluating my modifications to the linked list from Chapter 4.

In Chapter 6, I discuss the impact of my work, and outline future directions in which this work could be taken before concluding.

Chapter 2

Background

In this chapter, I review the current literature on concurrent data structures. I begin by looking at desirable properties of concurrent data structures, including the linearisability correctness condition, several non-blocking progress guarantees, and degree of concurrency. I then explain how these data structures are usually implemented using atomic primitives on shared-memory systems, along with some common implementation considerations. Next, I address different definitions of contention used currently in the literature on lock-free data structures, which motivated the main theorems of Chapter 3. Finally, I survey a range of both blocking and non-blocking concurrent data structures presented in the literature, some of which are discussed in detail in Chapter 4 and experimentally evaluated in Chapter 5.

2.1 Desirable properties of concurrent data structures

Correctness

Classical data structures execute in a sequential environment, where their correctness is relatively easy to reason about because of the single thread of execution. This ease of reasoning extends to simple concurrent data structures created by taking an existing sequential data structure and serialising access to it by using a lock. However, in a highly concurrent data structure it is desirable to have multiple readers *and* writers inside the data structure at the same time, causing the execution intervals of operations to overlap, or even properly contain each other. In this setting it becomes less clear as to what a "correct" execution is.

Linearisability [HW90] was proposed as a correctness criterion which allows programmers to carry over their intuition and reasoning about abstract data types from a sequential setting into a concurrent one. Informally, a data structure is *linearisable* if every operation appears to have taken place atomically at some point between its invocation and its response, such that these *linearisation points* can be identified and used to identify an equivalent legal sequential history.

Linearisable operations have intuitively ”correct” behaviour, in the sense that a linearisable data structure will behave (from a correctness standpoint) as if it were a sequential data structure protected by a mutual exclusion lock [Fra04]. Linearisability is also *compositional*: using linearisable data structures, new linearisable data structures can be built [HS12]. This is an important property, since it allows (for example) the designer of a lock-free hash table to use any lock-free linearisable set in the place of separate chaining.

There have been alternative correctness criteria suggested, for example Howard and Walpole [HW13] propose a data structure which is *relativistic*, in the sense that it is permissible for each concurrent reader to observe a different sequence of updates (each reader is in a different “frame of reference”). They argue that for data structures which do not have an inherent time order such as a key-value store, with independent or commutative updates, this model is acceptable. However, this criterion is only appropriate in certain circumstances, is not compositional, and can be confusing to reason about, whereas linearisability appears to be a more useful and intuitive criterion in general.

Progress guarantees

Mutual exclusion, or locking, is one of the most common ways of allowing safe access to data on a shared-memory computer. Unfortunately, the use of mutual exclusion comes at a cost. There are known problems such as deadlocking, priority inversion, and convoying, leading to disadvantages in fault-tolerance and scalability [GC96]. Recently, non-blocking data structures with much stronger progress guarantees have become popular, partly due to their inherent immunity to these problems. The main progress criteria used in non-blocking[1] data structures are outlined below.

The first property is *lock-freedom*: a data structure is lock-free if some operation will always complete after a finite number of steps have been executed system-wide. This ensures that even in the case where all but one process is suspended (through, for example, pre-emption) the remaining process can still complete its operation, no matter what state the data structure was left in by the incomplete operations. Usually this requires operations to leave enough information in the data structure that another operation, obstructed by the partially completed changes, can perform *helping* to complete the partial changes and proceed. The running time of a single lock-free operation can be arbitrarily long if it is forced to help new operations or retry repeatedly.

The second property is *wait-freedom*: a data structure is wait-free if every operation will finish

[1]There a few different common uses of the terms “non-blocking” and “lock-free’ in the literature, I use the definitions from [HS12].

after a finite number of steps, ensuring that no process is ever live-locked or starved. If a strict upper bound can be placed on the running time of every operation, it is called *bounded wait-free*. Wait-freedom is a strictly stronger property than lock-freedom, in the sense that every wait-free data structure is automatically lock-free, but not vice-versa. Wait-freedom is a much fairer condition than lock-freedom, but is difficult to implement since fair access to memory is usually not guaranteed, and is further complicated by underlying factors such as cache coherence algorithms, over which the programmer has no control.

The third property is called *obstruction-freedom*, which guarantees progress to any process eventually executing in isolation [HLM03]. This is weaker than the two previous criteria, since it does not guarantee progress when two or more processes are executing concurrently. It was introduced as a more practical alternative to lock-freedom, however it requires that some other scheme external to the algorithm itself, such as exponential backoff, is employed during conflicts.

All three of the above properties preclude the use of locks, but apart from that have mixed guarantees about progress. Wait-freedom is ideal, but usually very hard to achieve efficiently, so is not used often in general data structures. Obstruction-freedom is easier to achieve than lock-freedom, but relies on some out-of-band mechanism of ensuring that threads can execute in isolation to do their work. Lock-free algorithms perform well in practice while not relying on out-of-band progress mechanisms, or paying heavy overhead costs to ensure wait-freedom.

Degree of concurrency

The above progress guarantees are quite strong, but by themselves may not be indicative of good performance. For example, in most universal constructions (discussed later), any two processes executing updates concurrently will always conflict, and so there is a lot of wasted work, and throughput will in general not be higher than an equivalent sequential data structure protected using a mutual exclusion lock.

One possible idea for a concurrency metric proposed in [GKR12] is to fix some base *sequential specification* of a data structure, and extract the subsequence of steps (reads and writes) a concurrent algorithm takes which are part of the sequential specification. An interleaving of these subsequences for one or more processes is called a *schedule*, and the concurrency metric is the set of all schedules a concurrent algorithm *accepts* (would be observable in some execution).

For example, a schedule accepted by a sequential data structure protected by a mutual exclusion lock would just be a number of sequential executions concatenated together. A finer grained locking scheme would accept strictly more schedules, as it would permit some interleaving of individual reads and writes within high-level operations, so it would have a higher degree of concurrency.

2.2 Implementation of non-blocking data structures

On shared memory systems (such as most commodity hardware), any synchronisation between different processor cores or processes is done implicitly, through the cache control protocol implemented by the manufacturer [Sco13]. The underlying architecture makes available some *synchronisation primitives* to the programmer which usually take the form of atomic read-modify-write instructions, such as compare-and-swap, whose pseudocode is shown in Figure 2.1. In addition to this, certain instructions are guaranteed to execute atomically, for example on the x86 architecture, loads and stores on aligned memory words are atomic.

Compare-and-swap allows a caller to update a memory location on the condition that it currently contains a certain value. If it does not (for example the location was modified by another thread), the compare-and-swap will report a failure, and the caller can handle it appropriately. This all-or-nothing action makes compare-and-swap the main synchronisation primitive used in most current lock-free algorithms.

```
procedure CAS($addr, oldval, newval$)
    Do the following atomically:
    if $*addr = oldval$ then
        $*addr \leftarrow newval$
        return $oldval$
    end if
    return $*addr$
end procedure
```

Figure 2.1: Pseudocode of the compare-and-swap primitive. An asterisk $*$ denotes a pointer dereference in the style of C.

Compare-and-swap (henceforth *CAS*) is a *universal primitive*, meaning it can be used to solve the n-process consensus problem for any n [Her91]. Another such primitive is *load-link/store-conditional*, which are a pair of instructions, one performing a load from a memory location, and the other one performing a store to the same location only if that memory location has not been modified since the load-link. Such primitives are important, since various universal constructions rely on their existence. However, on current hardware the primitives operate only on single memory words, which makes more refined implementations of non-blocking data structures difficult. All of the lock-free data structures presented in Chapter 4 are built using single-word CAS.

Helping and retrying

Lock-free data structures often have more complex operations than their sequential counterparts. If there is more than one modification (a modification being an atomic step changing the data structure, like a CAS operation) necessary for an operation to complete, the operation is written in a re-entrant style so that if a process is suspended or pre-empted, other processes can notice the partially completed operation and carry out the required modifications to complete it. When a process does this on behalf of another, this is called *helping*.

Similarly, a process may be pre-empted by another between reading a memory location and attempting a CAS operation, causing a failed CAS. After this, the operation must restart from some earlier point, such as the start of the data structure. This is called *retrying*.

Because a single high-level operation in a lock-free data structure may be forced to retry or help other operations repeatedly, we cannot put an upper bound on the number of steps a single operation takes. Instead, the *amortised* step complexity[2] of an operation is given, which is the total number of steps taken by all operations, divided by the number of operations in the execution.

The work necessary when performing helping and retrying affects the asymptotic complexity of the data structure. Operations which are concurrent with only a small number of other operations will be affected much less than operations concurrent with many other operations, and this must be taken into account in any bound on the running time of a lock-free data structure. To do this, several measures of *contention* have been suggested.

2.3 Contention

When attempting to bound the overall running time of non-blocking data structures, often the size of the data structure n is not enough. For example, some structures logically delete elements before physically removing them from the structure, so although the structure contains n nodes, it may only contain $n - p$ elements if there are p concurrent processes. Furthermore, operations may have to perform extra work in the form of retrying or helping, due to the effects of concurrent operations. Another *contention* parameter is needed, which will provide some measure of the number of concurrently executing processes or operations. There are several existing notions of contention currently used in the analysis of lock-free data structures.

The *interval contention* c_I of an operation is the number of operations concurrent with a given operation. An amortised bound on step complexity in terms of the interval contention is relatively easy to obtain by making operations charge each other for steps that would usually be unnecessary in a sequential execution. Provided each operation charges any other concurrent operation a constant amount at most a constant number of times, this leads naturally to an additive amortised $O(c_I)$ term. Unfortunately, the interval contention of an individual operation can be arbitrarily large in a system involving as few as two processes.

[2]*Step complexity* is the number of steps required to complete a computation, given an unbounded number of processors.

The *point contention* c_P of an operation is the maximum number of processes active at any time during the operation [AF03]. An amortised bound on the step complexity in terms of the point contention of a data structure is usually quite involved, for example the proofs in [FR04; Ell+14] rely on reasoning about the interleavings of individual CAS steps inside concurrent operations. Some authors [OS13; CNT14] have provided modifications of their algorithms which perform extra helping so that they can tighten what would otherwise be a bound in terms of the interval contention to a bound in terms of the point contention. Since for any given operation, the interval contention is always at least as large as the point contention, this seems like a better bound.

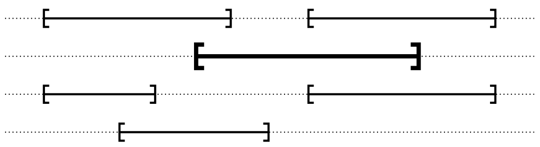


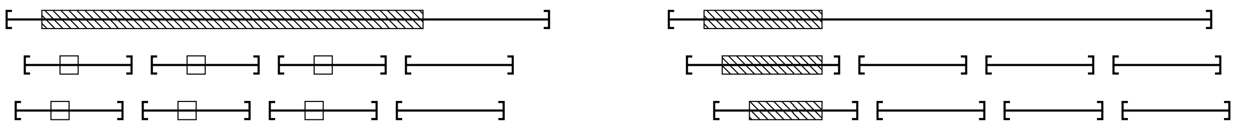
Figure 2.2: An example execution on 4 processes. The bolded operation has point contention 3, process contention 4, and interval contention 5.

The *process contention* c_K of an operation is the number of processes ever active during the operation. This was the original definition of interval contention [AST02]. The *overlapping-interval contention* c_{OI} of an operation is the maximum interval contention over all operations overlapping with that interval. It was introduced in [OS13], but seems relatively unused in the literature otherwise. It is included here for completeness.

Relating helping to contention

My result, that point contention and interval contention are amortised equivalent, challenges a popular view that increasing the amount of helping done by operations can reduce the asymptotic step complexity of data structures. I will briefly outline why this view might be held.

Proofs of the asymptotic complexity of lock-free data structures are often done by getting operations to "charge" each other for steps that would be unnecessary in a sequential execution, for example helping or restarting. During the analysis of the skiplist given in [OS13], the authors reason that an inconsistency caused by an update may be encountered by every traversal concurrent with that update, and so the update may be charged $O(c_I)$ times. This leads to an additive term of $O(c_I)$ in the amortised step complexity of the operation. The authors then argue that by making the traversals perform extra helping so as to resolve this inconsistency, the update will be charged at most $O(c_P)$ times (since a traversal may not proceed without first resolving the inconsistency). The analysis of the lock-free binary search tree in [CNT14] follows a similar argument. A sketch of this idea is illustrated in Figure 2.3.



(a) Without helping. (b) With helping.

Figure 2.3: In this figure, intervals represent operations, shaded boxes represent helping or trying to fix an inconsistency, and clear boxes represent operations taking extra steps because of an inconsistency. The top long-running operation introduces an inconsistency (such as a logical deletion) and attempts to resolve it. Without helping, $O(c_I)$ operations may observe this inconsistency. With helping, at most $O(c_P)$ operations may observe (and try to fix) this inconsistency.

2.4 Concurrent search structures

Universal constructions

There are several known *universal constructions* for creating lock-free or wait-free data structures from existing sequential specifications, using only atomic loads, stores, and a universal synchronisation primitive (such as CAS). Usually these constructions result in very inefficient implementations in practice, due to their generality.

A method of taking a sequential specification of a data structure and producing a lock-free data structure using CAS is described in [Her93]. The core idea is that all accesses to the data structure go through a shared pointer, and any updates will read the pointer, make a full local copy of the data structure where they apply their update privately, then attempt to swing the shared pointer to their local copy using CAS. If the pointer has been modified in the meantime by a concurrent update, the process will restart its update operation.

There is an obvious overhead of copying the data structure for every update. This can sometimes be overcome: for example if the data structure is a balanced tree on n nodes, then an update operation need only make copies of $O(\log n)$ nodes along the path from the modification site to the tree root. Despite optimisations such as this, the construction suffers in practice because all updates conflict at the root pointer, which effectively serialises accesses to the data structure. While this construction is lock-free, it exhibits a poor degree of concurrency.

A different kind of universal construction is described in [TSP92], which can take any deadlock-free lock-based data structure, and transform it into a lock-free data structure. The idea of this construction is to replace locks with lists of virtual instructions which must be completed before the lock is relinquished, so that other processes can perform helping by taking ownership of the "lock" and performing these tasks. This addresses the problem of serialised access, and the resulting data structure has the same degree of concurrency as the original lock-based structure. However, in performance is poor in practice because of the high overhead of creating and interpreting these lists of instructions.

Linked lists

There are are a few very efficient lock-free linked list implementations known which use CAS. The first lock-free linked list using only single-word CAS was presented in [Val95]. It has a high degree of concurrency, but involves leaving "auxiliary nodes" between regular nodes of the list to cope with problems arising from concurrent deletion. These auxiliary nodes can grow into long chains, leading to poor performance.

Harris refined these ideas to create a lock-free linked list implementing an ordered set [Har01]. His approach does not require auxiliary nodes, instead using a "tagged reference" (a word holding both a pointer and a boolean) to indicate logical deletion, and separating the logical deletion of a value from the set from the physical removal of its node from the list. The Harris list is one of the best performing lock-free lists known.

Fomitchev and Ruppert expanded on Harris's ideas to create a lock-free linked list with an amortised step complexity per operation of $O(n + c_P)$, where n is the number of values stored and c_P is the *point contention* of an operation [FR04]. It uses a more complicated deletion process than the Harris list which sets backlinks, so that operations which encounter logically or physically deleted nodes can backtrack through the list rather than restarting from the front.

On the other hand, there are some very efficient concurrent *lock*-based linked lists. The Lazy list [Hel+06] uses per-node locks in a different way than standard hand-over-hand locking, achieving a much higher degree of concurrency. Traversals proceed without acquiring locks until they reach nodes to be modified, at which point they acquire the locks on adjacent nodes, then *validate* that the data in those nodes has not been changed since it was examined before locking. The contains operation acquires no locks, and is wait-free.

Recently, a new linked list has been proposed which uses *versioned locks*, which are locks that carry an accompanying "version number" indicating how many times they have been locked. The versioned list [Gra+15] is quite similar to the lazy list, but locks are acquired only if work can be performed, and this structure achieves very good performance by keeping its critical sections very small.

Logarithmic search structures

A skip list is a probabilistic search structure first described by Pugh [Pug90], which has expected logarithmic time for lookup, insertion, and deletion. Skip lists are easier to implement than self-balancing binary search trees (such as AVL trees or red-black trees), and in a concurrent environment this seems to be doubly the case, where many efficient lock-free skip lists are known, and no efficient lock-free self-balancing trees are known.

In his thesis, Fraser describes an implementation of a lock-free skip list, loosely based on Harris' linked list [Fra04]. Fomitchev and Ruppert also describe a skip list based on their own linked list. Although it is clear that these structures have expected $O(\log n)$ operation times when operating sequentially, no analysis was provided by Fraser or Fomitchev and Ruppert to determine the complexity in a concurrent environment. Oshman and Shavit describe a very similar skip list as part of their *SkipTrie* data structure [OS13], and sketch a proof of it having expected amortised time of $O(h + c_I)$, where h is the height of the skip list.

Lock-free binary search trees appear to be extremely hard to create, even when unbalanced. The first correct lock-free binary search tree was published in 2010 [Ell+10], and uses an external tree structure, where data is stored in external nodes and internal nodes are used only for routing purposes. A more recent binary search tree using external nodes includes an amortised bound on the step complexity of any operation of $O(h + c_P)$, where h is the height of the tree at the start of the operation and c_P is the point contention of the operation [Ell+14]. Another recent binary search tree uses the technique of threading backlinks through the tree so that restarting operations do not have to backtrack far [CNT14]. They prove an amortised bound on the step complexity of $O(h + c_I)$, but point out that it can be "tightened" to $O(h + c_P)$ by introducing extra helping into their algorithm. Note that none of the binary search trees above are balanced, and so they may be expected to have logarithmic height only when the input values are drawn uniformly at random.

Chapter 3

Measures of contention

In this chapter, I analyse the four definitions of contention stated in Chapter 2 in an amortised context. Surprisingly, the point contention, process contention, and interval contention are all amortised equivalent, in the sense that when summed across every operation the interval contention is no larger than twice the point contention.

To show this, I use the theory of *interval graphs*: graphs formed from a set of real intervals by replacing intervals with vertices, and connecting vertices whenever their corresponding intervals overlap. By using the existence of at least one special *simplicial* vertex which forms a clique with its neighbourhood, I construct a lower bound on the point contention and use this to bound the ratio of the interval to the point contention.

3.1 Model

I assume a standard asynchronous model of computation, where any number of processes take steps which may be interleaved arbitrarily. These processes execute high-level operations (for example, add, remove, and contains on a concurrent set structure) all of which have an invocation time and a response time. There are no bounds on the relative speeds of the processes. What follows is a model which captures this, and is useful for my purposes.

Define a *finite execution* $\alpha = (\mathcal{O}, P, I, \pi)$ involving P processes to be a finite set $\mathcal{O}$ of *operations*, along with two mappings I and π. The *interval function* $I : \mathcal{O} \to \mathbb{R} \times \mathbb{R}$ maps operations to the compact real intervals representing their execution times, with the left endpoint being the invocation time and the right endpoint being the response time. The *process function* $\pi : \mathcal{O} \to \{1, \ldots, P\}$ assigns a unique process to each operation, representing the process which executed that operation.

If for two operations $op, op' \in \mathcal{O}$ we have $I(op) \cap I(op) \neq \emptyset$, then we say that op and op' are

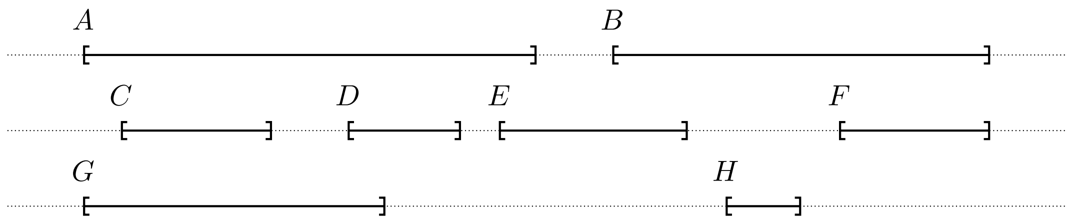


Figure 3.1: An example execution involving 3 processes and 8 operations. The operation B has point contention 2, process contention 3, and interval contention 4.

concurrent, or that they *overlap*. For any execution we require that I is injective[1], and that the execution should be *well-formed*: any two operations mapping to the same process should not be concurrent. Figure 3.1 shows an example of a finite execution.

I will briefly recall the definitions of contention. The point contention c_P is the maximum number of processes active at any point in time during the operation. The process contention c_K is the number of processes ever active during an operation. The interval contention c_I is the number of operations concurrent with the given operation, and the overlapping-interval contention c_{OI} is the maximum interval contention over all operations overlapping with the given operation. In terms of the model given above, these may be formally defined as follows:

Definition 1. *In a finite execution $\alpha = (\mathcal{O}, P, I, \pi)$, the point contention c_P, process contention c_K, interval contention c_I, and overlapping-interval contention c_{OI} are functions $\mathcal{O} \to \mathbb{Z}_+$ defined by:*

$$\begin{aligned}
c_P(op) &= \max_{x \in I(op)} \left|\{op' \in \mathcal{O} \colon x \in I(op')\}\right| \\
c_K(op) &= \left|\{\pi(op') \colon op' \in \mathcal{O} \wedge op' \text{ overlaps } op\}\right| \\
c_I(op) &= \left|\{op' \in \mathcal{O} \colon op' \text{ overlaps } op\}\right| \\
c_{OI}(op) &= \max_{\substack{op' \in \mathcal{O} \\ op' \text{ overlaps } op}} c_I(op')
\end{aligned}$$

It should be clear from this definition that for any operation op, both $c_P(op) \leq P$ and $c_K(op) \leq P$. In fact, there is a chain of inequalities involving the four definitions of contention given above.

Proposition 1. *For any operation $op \in \mathcal{O}$, $1 \leq c_P(op) \leq c_K(op) \leq c_I(op) \leq c_{OI}(op)$ and this bound is tight.*

Proof. Fix some operation $op \in \mathcal{O}$. Let $S = \{op' \in \mathcal{O} \colon op' \text{ overlaps } op\}$, and for any $x \in I(op)$,

[1]This is not restrictive: in any finite execution in which two intervals are identical, they may be perturbed slightly such that they are not, without affecting contention.

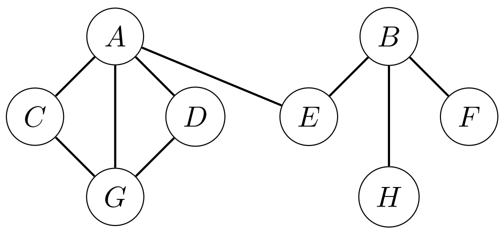


Figure 3.2: The interval graph of the set of intervals shown in Figure 3.1

let $S_x = \{op' \in \mathcal{O} \colon x \in I(op')\}$. The definitions of contention for the operation op now become:

$$c_P(op) = \max_{x \in I(op)} |S_x| \qquad\qquad c_I(op) = |S|$$

$$c_K(op) = |\pi(S)| \qquad\qquad c_{OI}(op) = \max_{op' \in S} c_I(op')$$

By these characterisations we find $c_{OI}(op) \geq c_I(op)$ because $op \in S$, and $c_I(op) \geq c_K(op)$ because a set is at least as large as its image under a map. Note that since the execution is well-formed, $|S_x| = |\pi(S_x)|$, and since we have $S_x \subseteq S$ for all $x \in I$, it follows that $c_P(op) \leq c_K(op)$. Finally, all of these bounds are tight by considering an execution containing one operation and one process. □

Putting aside the process contention momentarily, it is easy to see that all that is needed to calculate the interval contention and the overlapping-interval contention is information about which operations overlap. In fact, as we will soon see, this is true for the point contention as well. Hence a natural setting to analyse these measures of contention is as an *interval graph*, which contains precisely this information, while hiding the complications of specific processes and points in time.

3.2 The interval graph and simplicial vertices

Any graphs $G = (V, E)$ considered here are finite, undirected, and without multiple edges or loops. V denotes the vertex set and E denotes the edge set. n always refers to the number of vertices $|V|$ and m always refers to the number of edges $|E|$. For any vertex subset $U \subseteq V$, $G[U] = (U, E \cap (U \times U))$ is called the *subgraph induced by* U. A vertex subset $U \subseteq V$ forms a *clique* if the subgraph $G[U]$ is complete. For any vertex v, its *neighbourhood* $N(v)$ consists of all vertices incident to v. A vertex v is called *simplicial* if the subgraph induced by its neighbours and itself $G[\{v\} \cup N(v)]$ is complete.

Definition 2. *The interval graph of a finite set of real intervals S is the graph with vertex set S, and an edge between two intervals $I, J \in S$ if $I \neq J$ and $I \cap J \neq \emptyset$.*

For an execution $\alpha = (\mathcal{O}, P, I, \pi)$, define the interval graph of that execution to be the interval graph of the set $\{I(op) \colon op \in \mathcal{O}\}$. It is convenient for the discussion to have the vertex set of this graph being the set of operations $\mathcal{O}$, which is unambiguous since I is injective.

From the perspective of an interval graph, the interval contention of an operation becomes remarkably simple: $c_I(op) = 1 + \deg op$, since the degree counts every $op' \in \mathcal{O} \setminus \{op\}$ overlapping with op, and 1 is added to count op overlapping itself. The point contention $c_P(op)$, on the other hand, is the size of a maximum clique containing the vertex op. This should be clear by the following lemma:

Lemma 1. *Let S be a finite set of compact real intervals. There is a point $x \in \mathbb{R}$ common to every interval in S if and only if all of the intervals in S intersect pairwise.*

Proof. The only if direction is simple: if x is contained in every interval, then $x \in I \cap J$ for every $I, J \in S$, and so every pair of intervals have nonempty intersection. For the other direction, let the set of intervals $S = \{[a_i, b_i] \mid 1 \leq i \leq n\}$. Let $a = \max_i a_i$ be the latest starting time and $b = \min_i b_i$ be the earliest finishing time amongst the intervals. Then $a = a_l$ and $b = b_r$ for some r and l. The intervals $[a_l, b_l]$ and $[a_r, b_r]$ have nonempty intersection and so for any $x \in [a_l, b_l] \cap [a_r, b_r]$ and $1 \leq i \leq n$ we have $a_i \leq a \leq x \leq b \leq b_i$, so x is common to every interval. □

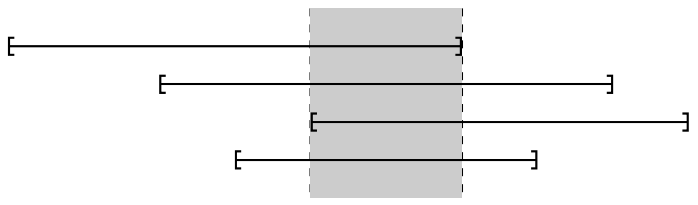


Figure 3.3: For a set S of pairwise intersecting intervals, any point x between the latest starting time and the earliest finishing time (shown shaded) belongs to every interval.

Interval graphs belong to a larger class of graphs called *chordal graphs*, in which any cycle of length 4 or more always contains a *chord*, an edge connecting two nonadjacent vertices of the cycle. Chordal graphs are characterised completely by the existence of a *perfect elimination order*, defined below. In the case of interval graphs, the existence of such an order is easy to see, so I will give a short proof.

Definition 3. *A perfect elimination order is an ordering $\{v_i\}_{i=1}^n$ of vertices in a graph such that for all $1 \leq i \leq n$, v_i is simplicial in $G[v_1, \ldots, v_i]$.*

Lemma 2. *Every interval graph G on n vertices admits a perfect elimination order.*

Proof. The case for $n = 1$ is clear. Proceed by induction: assume the claim holds for interval graphs with $n-1$ vertices. Take the vertex v corresponding to the interval with earliest finishing time: this vertex is simplicial since the finishing time intersects every interval which overlaps v. The graph $G - v$ is again an interval graph and so by hypothesis there exists a perfect elimination ordering $\{v_i\}_{i=1}^{n-1}$ of $G - v$. Setting $v_n = v$ gives a perfect elimination order $\{v_i\}_{i=1}^{n}$ of G. $\square$

A perfect elimination order can be viewed as a process of reducing the graph to nothing by deleting one vertex at a time, in such a way that every vertex is simplicial just prior to its deletion. The clique a vertex forms with its neighbourhood at the time of its deletion will be a clique in the original graph, and so by using this process we are able to construct a lower bound on the overall point contention in an execution.

Lemma 3. *Let G be a graph with a perfect elimination order $\{v_i\}_{i=1}^{n}$, and let $M(v)$ be the size of a maximum clique containing the vertex v. Then $\sum_{v \in V} M(v) \geq n + m$.*

Proof. Take a perfect elimination order $\{v_i\}_{i=1}^{n}$ of the vertices of G, and define $G_i = G[v_1, \ldots, v_i]$. This gives a family of graphs $G_n, \ldots, G_1$, such that $G_n = G$, G_1 is a single vertex, and $G_j = G_{j+1} - v_{j+1}$ for all $1 \leq j < n$. Let d_i be the degree of v_i in G_i. Since v_i is simplicial in G_i, $\{v_i\} \cup N(v_i)$ forms a clique in G_i and hence also in G, so $1 + d_i \leq M(v_i)$. Finally, note that d_i is the number of edges removed when removing v_i from G_i, so $\sum_{i=1}^{n} d_i = m$. So $\sum_{v \in V} M(v) \geq \sum_{i=1}^{n}(1 + d_i) = n + m$. $\square$

3.3 Equivalence of overall point and interval contention

For any finite execution α, let $c_P(\alpha) = \sum_{op \in \mathcal{O}} c_P(op)$, and likewise for the other measures of contention.

Theorem 1. *In any finite execution α, $c_P(\alpha) \leq c_I(\alpha) < 2c_P(\alpha)$.*

Proof. Form the interval graph $G = (V, E)$ of the execution α, with n vertices and m edges. By the definitions of contention given before, the interval contention of a single operation op is $c_I(op) = 1 + \deg op$, where $\deg$ denotes the degree of the operation's interval in the graph. Summing across all operations, $c_I(\alpha) = \sum_{v \in V}(1 + \deg v) = n + 2m$. As discussed previously, the interval contention $c_P(op)$ is the size of the largest clique containing op in the graph, so by Lemma 3 we have $n + m \leq c_P(\alpha)$.

Putting these together with the inequality in Proposition 1, we find that

$$n + m \leq c_P(\alpha) \leq c_I(\alpha) \leq n + 2m$$

and so by taking the ratio of $c_I(\alpha)$ to $c_P(\alpha)$,

$$1 \le \frac{c_I(\alpha)}{c_P(\alpha)} \le \frac{n+2m}{n+m} = 1 + \frac{m}{n+m} < 2 \qquad \square$$

Had we defined the point and interval contention of an operation op to not include op itself, so that in an uncontended execution $c_P(op) = c_I(op) = 0$ instead of 1, a similar result holds. In that case, the sum of interval contention with respect to the interval graph is $c_I(\alpha) = 2m$, and the lower bound on the point contention is $m \le c_P(\alpha)$, so we find that $c_P(\alpha) \le c_I(\alpha) \le 2c_P(\alpha)$.

Although Theorem 1 alone says that in amortised terms, $c_P = \Theta(c_I)$ and so the point contention and interval contention are equivalent, it is interesting to examine what a "worst-case" execution is. Intuitively, it has a very restricted point contention, while intervals overlap as many times as possible. Such a construction is given in the proof of Theorem 2 and illustrated in Figure 3.4, and shows that the bound given above is tight.

Theorem 2. *For any $0 < \epsilon < \frac{1}{2}$, there exists a family of executions $\{\alpha_n\}_{n \ge 1}$ where each α_n has n operations and ϵn processes, such that*

$$\lim_{n\to\infty} \frac{c_I(\alpha_n)}{c_P(\alpha_n)} = 2 - \epsilon$$

Proof. Let α_n be an execution containing n operations labelled op_i for $0 \le i < n$, and let $1 \le k \le n/2$. Define the mappings $\pi(op_i) = i \pmod k$ and $I(op_i) = [i, i + k - \frac{1}{2}]$ for all $0 \le i < n$. It is easy to check that at the start or end point of each operation there are k operations active at that point in time and so $c_P(op) \ge k$ for all operations. Since there are only k processes, $c_P(op) = k$ for all operations, so $c_P(\alpha_n) = nk$.

By the length and placement of operations, for every operation op the set of operations intersecting its left endpoint is disjoint to the set of operations intersecting its right endpoint, and the union of these is every operation concurrent with op. Hence every operation but the first $k-1$ and the last $k-1$ operations have interval contention $2k-1$. The first operation has interval contention k, the next $k+1$, and so on until the kth operation has interval contention $2k-1$, and by symmetry the same goes for the last k operations. By overcounting the interval contention overall and subtracting off the start and end deficits, we find that $c_I(\alpha_n) = n(2k-1) - 2(0 + 1 + \ldots + (k-1)) = 2nk - n - k(k-1)$. The ratio of interval to point contention then becomes $c_I(\alpha_n)/c_P(\alpha_n) = 2 - \frac{1}{k} - \frac{k-1}{n}$. By letting $k = \epsilon n$, we get $c_I(\alpha_n)/c_P(\alpha_n) = 2 - \epsilon - \frac{1-\epsilon}{\epsilon n}$. $\square$

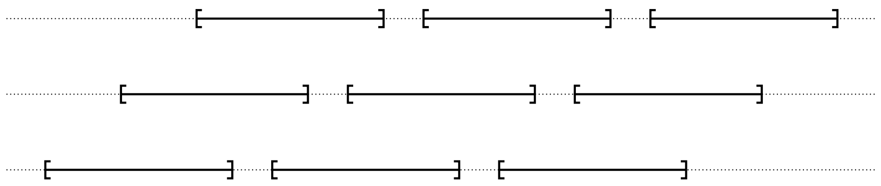


Figure 3.4: An example worst-case construction with $n = 9$ and $k = 3$.

3.4 Bounds for overall process and overlapping-interval contention

Theorem 1 together with Proposition 1 gives the chain of inequalities $c_P(\alpha) \leq c_K(\alpha) \leq c_I(\alpha) < 2c_P(\alpha)$, and so the point, process, and interval contention are interchangeable additive terms when doing amortised analysis.

The overall overlapping-interval contention, on the other hand, cannot be bounded within a constant factor of the point contention. Consider an execution of two processes, where the first process performs one long-running operation and the second process runs $n - 1$ short operations, all of which execute inside the interval of the long-running operation. Figure 3.5 shows an example of such an execution. The point contention of every operation is 2, and since every operation overlaps with the long-running operation op which has interval contention n, the overlapping-interval contention of every operation is n. So in this execution $c_P(\alpha) = 2n$ and $c_{OI}(\alpha) = n^2$.

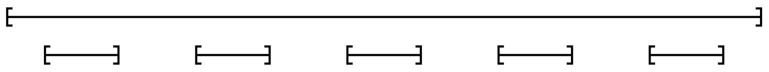


Figure 3.5: An execution on 2 processes and n operations, with $c_P = 2$ and $c_{OI} = n$ for every operation.

3.5 Conclusion

In this chapter, we have seen that three definitions of contention which are in use in the literature today, the point contention, process contention, and interval contention, are equivalent in an amortised context. This result has practical impact, as time bounds in terms of the point contention are usually much harder to work with than interval contention.

This result allowed me to simplify Fomitchev and Ruppert's proof of amortised step complexity in their linked list, and also led me to come up with a modification which has the same step complexity as the original list but performs much better in practice. The proof and modification are shown in the next chapter.

Chapter 4

Linked Lists

In the previous chapter, I examined different definitions of contention from a purely theoretical perspective. In order to determine what impact contention has in practice, I selected five different concurrent linked list algorithms from the current literature to implement and test experimentally. In this chapter, I present these algorithms, accompanied by bounds on their step complexity where possible. I also provide a simpler proof of the step complexity of Fomitchev and Ruppert's linked list, and a modification of their list with the same step complexity but better performance in practice.

All of the linked lists described in this chapter implement a set abstraction which stores integer keys. Every list has a dummy head node storing the key $-\infty$, and a dummy tail node storing the key $+\infty$. All of these lists can easily be extended to a dictionary abstraction, storing keys coming from any totally ordered set.

4.1 Harris' List

The Harris linked list [Har01] is a well-known lock-free linked list, regarded as one of the most efficient concurrent linked lists. The list structure resembles a regular linked list, with each node having a key field and a $next$ pointer. However, in the Harris list the $next$ pointer also stores a $marked$ bit to indicate logical deletion. On most modern architectures (such as x86, x86_64, ARM), pointers suitable for CAS are always aligned on 4-byte or 8-byte boundaries, so this marked bit can be stored in one of the unused low-order bits of the pointer. This does, however, require the marked bit to be masked off every pointer before it is dereferenced, which has a non-negligible performance impact.

Every operation uses a common *traversal* subroutine, which will traverse the list to find the node with the desired key, or otherwise the insertion point if the key does not exist. The traversal operation is "not allowed" to ignore logically deleted nodes: it must attempt to remove them from the list. If a traversal encounters a chain of one or more consecutive marked nodes, it will attempt to remove the chain from the list by performing a CAS on the node immediately preceding the chain. If this CAS was successful, the traversal may proceed, otherwise the traversal must restart from the beginning of the list.

The insertion of a key k is relatively simple: a traversal is performed to attempt to find two nodes $pred$ and $succ$ such that $pred.key < k \leq succ.key$. If $succ.key = k$, the insertion returns false. Otherwise, a new node $node$ is allocated, with $node.key = k$, $node.next = succ$, and then a CAS is performed on $pred.next$ to attempt to change it to point to $node$. If the CAS succeeded, the insertion returns true, otherwise the insertion restarts from the front of the list.

Deletion, however, can run into problems if done naively. Suppose the following approach is taken to deletion: locate the node $node$ to be deleted, along with its predecessor $pred$ and its successor $succ$. Then, perform a CAS on $pred.next$ to attempt to change it from $node$ to $succ$. If two operations attempt to delete two consecutive nodes simultaneously, then the deletion will only take effect for one of them, as illustrated in Figure 4.1.

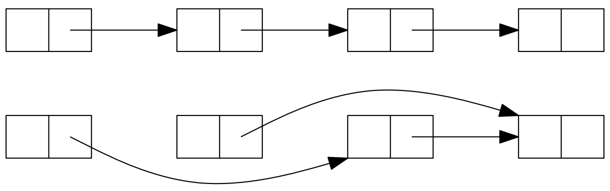


Figure 4.1: A naive deletion scheme leads to lost deletions.

Harris' approach to fixing this problem is to break deletion into two CAS steps. The node to be deleted $node$ is located, along with its predecessor $pred$ and successor $succ$. Firstly, $node.next$ is *marked* by setting the pointer's low bit. This is done using CAS, and if it fails, the delete operation restarts from the front of the list. After the marked bit has been successfully set, $node.key$ is considered logically removed from the set, and $node.next$ is never allowed to change. The delete operation will then optimistically attempt to remove $node$ using CAS on $pred$, and if this fails, will run a traversal operation from the start of the list to the node's successor, guaranteeing that the marked node has been physically removed from the list.

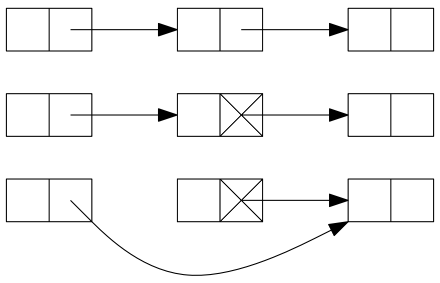


Figure 4.2: The two-step deletion of Harris' linked list: first a node is *marked* by using CAS on its $next$ pointer, then the node is physically removed by using CAS on its predecessor's $next$ pointer.

Since operations restart from the beginning of the list on failed CAS operations, the Harris list cannot have a good asymptotic complexity. It was pointed out by Fomitchev & Ruppert [FR04] that there are executions involving m operations on a list of length n where the Harris list performs $\Omega(nm)$ work per operation. The example they present is as follows: consider an initial list of length n, and m processes labelled P_1 through to P_m. The process P_m repeatedly deletes the last element in the list, while processes $P_1, \ldots, P_{m-1}$ all attempt to insert nodes at the end of the list. Every time $P_1, \ldots, P_{m-1}$ are all ready to perform a CAS at the end of the list, P_m performs a CAS to mark the last node, causing all $m - 1$ processes to restart from the front of the list.

In that execution, the point contention of each operation is m, so it is clear there are executions requiring $\Omega(nc_P)$ work per node. Despite this, the Harris list performs well in practice in cases when the elements being accessed are spread throughout the list.

4.2 Fomitchev & Ruppert's List

Fomitchev and Ruppert [FR04] proposed a linked list based on Harris', but with improved worst-case performance. They add to the $next$ field of each node a $flag$ bit, in addition to the $marked$ bit. They also add a $backlink$ field to each node which is set during deletion and points to the node's predecessor, to assist operations that "get stuck" on a logically or physically deleted node.

The main difference from the Harris list is in how nodes are deleted. Once a node $node$ is located, along with its predecessor $pred$ and successor $succ$, the predecessor is *flagged* by setting the flag bit of $pred.next$. Once $pred$ is flagged, its $next$ field is never allowed to change until $node$ has been physically removed from the list and $pred$ has been unflagged. Next, $node$ is marked by setting the mark bit of $node.next$. Once $node$ has been marked, its $next$ field is never allowed to change. Finally, $pred.next$ is changed to point to $succ$ using a CAS, removing the flag bit at the same time.

Thus the deletion is broken into three steps: *flagging*, *marking*, and *removal*. Just prior to marking, $node.backlink$ is set to point to $pred$. This ensures that any operations that "get stuck" on $node$ following its marking or removal can backtrack to $pred$ and continue from there, rather than restarting from the front of the list like in the Harris algorithm, and is what gives Fomitchev & Ruppert's list a much better asymptotic complexity. Setting the backlink can be done with an atomic store, since the predecessor's $next$ field is not allowed to change until the $node$ is physically removed from the list.

Aside from deletion, operations in Fomitchev and Ruppert's list are very similar to Harris' list, although they may have to perform helping on flagged nodes, as well as marked nodes. The largest improvement is that due to backlinks, retrying is very cheap, and the number

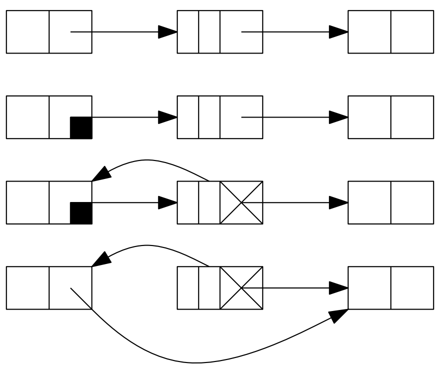


Figure 4.3: The three-step deletion of Fomitchev's linked list. The predecessor is flagged using CAS. The node's backlink is set, then the node is marked using CAS. Finally, the node is removed using CAS, unflagging the predecessor at the same time.

of backlinks which need to be traversed by an operation can be shown to be amortised $O(c_P)$, where c_P is the point contention of an operation. By using a scheme where operations "charge" each other for extra work, Fomitchev and Ruppert showed that their list has an amortised step complexity of $O(n + c_P)$ for all operations.

A simplified proof of time complexity

Fomitchev and Ruppert's original proof of the amortised step complexity of an operation in their list was quite involved. Because they were aiming to achieve an additive factor of the point contention, a complex charging scheme was needed to make sure that the net amount charged to any CAS belonging to an operation *op* was $O(c_P(op))$. Here I present a simpler proof that the amortised step complexity of an operation *op* is $O(n(op) + c_I(op))$, where $n(op)$ is the number of logically present elements in the set at the invocation time of *op*.

Firstly, note that it is only important to count the number of forward pointer traversals, backlink traversals, and CAS attempts: the total work done by the algorithm is at most a constant factor away from this. If an operation *op* executes in isolation, it will take $O(n(op))$ steps to complete. Call all of the steps an operation would have taken in isolation *necessary*, and any other steps it needs to take due to interference from other operations *extra*. Any extra work that an operation has to do will be "charged" to another operation. By showing that any operation *op* may be charged at most $O(c_I(op))$ times, the bound follows.

An insert operation op_i may cause another operation *op* to have a failed CAS attempt, but will not cause it to backtrack or help as a result. So op_i may be charged at most $c_I(op_i)$ times due to any failed CAS attempts it has caused. The insert operation op_i may also cause another operation *op* to traverse a new node that was not present at *op*'s invocation time. Since operations will only traverse pointers once, op_i may be charged at most $c_I(op_i)$ for this. So an insert operation op_i may be charged a total of $2c_I(op_i)$.

A remove operation op_r begins by successfully flagging a node. Once the flag has been successfully set, other operations may help set a backlink, mark the node to be removed, and physically remove it from the list: all of these operations (along with failed attempts trying to accomplish them) are charged to op_r. Furthermore, any backlink traversals an operation performs from the node that op_r is removing are charged to op_r. Since the same backlink is never traversed twice by the same operation, and from the time that a node is flagged it only takes a constant number of steps to remove its successor from the list, a remove operation op_r may be charged at most $O(c_I(op_r))$ times.

This accounts for all extra steps in an operation. So any update operation op may be charged an extra cost of $O(c_I(op))$, in addition to its necessary cost of $O(n(op))$, so the amortised cost of an operation in Fomitchev and Ruppert's list is $O(n + c_I)$. Due to Theorem 1, this bound is equivalent to Fomitchev and Ruppert's original bound of $O(n + c_P)$.

4.3 A modification of Fomitchev & Ruppert's list

Note that in the simplified proof, a contains operation never needed to be charged. This is partly due to the fact that the interval contention is a coarser, easier to deal with quantity than the point contention. However, one realisation we can draw from this is that if we "turn off" the helping that the contains operation would usually do (cleaning up marked nodes), it will have no effect on the amortised step complexity of a list operation, and remain lock-free.

I modified Fomitchev and Ruppert's list to have a contains operation which is wait-free: it performs no helping and makes a single pass through the list. The pseudocode for the operation is shown in Figure 4.4.

```
1:  procedure CONTAINS($k$)
2:      $current \leftarrow head$
3:      $marked \leftarrow false$
4:      while $current.key < k$ do
5:          $succ \leftarrow current.succ$
6:          $marked \leftarrow succ.mark$
7:          $current \leftarrow succ.right$
8:      end while
9:      return $(current.key = k) \wedge (marked = false)$
10: end procedure
```

Figure 4.4: The wait-free Contains operation

The new contains operation to search for key k traverses the list while performing no helping, until it reaches some node $node$ such that $node.key \geq k$. If $node.key = k$ and $node$ is not marked, the contains operation return true. Otherwise, it returns false.

What remains is to show this operation is linearisable, by identifying linearisation points within each operation's interval of execution. A successful contains operation linearises at the point when the successor field of the matching node is read. Finding a linearisation point for an *unsuccessful* contains operation on the other hand, is less straightforward. Because the new contains operation ignores node markings while traversing the list, it is possible for the contains operation to be traversing a chain of physically removed nodes, and be oblivious to the effects of concurrent insertions.

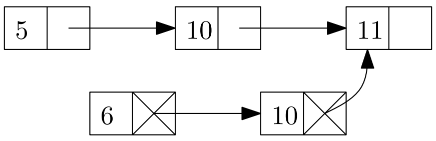


Figure 4.5: If a $\mathrm{Contains}(7)$ operation is currently observing the recently removed nodes 6 and 10, it will be oblivious to the effects of a concurrent $\mathrm{Insert}(7)$. This example shows that it is incorrect to linearise a unsuccessful contains operation at the time it last read a node.

To resolve this, I use the same scheme as in the lazy list. An unsuccessful $\mathrm{Contains}(k)$ operation is linearised at whichever of these points comes first:

- The point where a node $node$ satisfying $node.key \geq k$ is found.
- The point immediately before a new node with key k is added to the list.

4.4 The Lazy List

The lazy list [Hel+06] is a linked list which uses locking in a novel way to achieve good performance. Each node in the list is protected by a lock, and any modifications to that node's data may only be done while holding the lock. Each node, in addition to key and $next$ fields, also stores a $marked$ field which indicates logical deletion. (Note that this marked field is a regular boolean field, not hidden in the low bits of a pointer).

The update methods (add and remove) of the lazy list will proceed through the list without acquiring locks to locate two consecutive nodes $pred$ and $curr$ such that $pred.key < key \leq curr.key$, where key is the key being inserted or removed. When found, the update operation will lock both $pred$ and $curr$. Once the locks have been acquired, the operation then *validates* that the information it read prior to locking has not changed: it turns out all that needs to be checked is that $pred$ and $curr$ are unmarked and adjacent. If the validation fails, the update restarts from the front of the list. If the validation succeeded, the update will proceed and do any necessary work to the structure before releasing both locks.

The contains method of the lazy list is wait-free: it acquires no locks, and only makes a single pass through the list. The logical deletion markings are essential to having a wait-free contains operation, as well as a linearisation argument similar to the one given in the last section.

Because the update operations in this list restart from the front when they fail to validate, I expect this list to perform similarly to the Harris list when many updates are taking place in the same location.

4.5 The Versioned List

The versioned list [Gra+15] is a lock-based list similar to the lazy list but using the novel idea of *versioned locks*, locks which count the number of times they have been acquired. The version number on the lock acts as a kind of "modification timestamp" for the data it protects: if the version number is read and then the protected data examined, the operation can at some later time attempt to acquire the lock at that specific version number. If the locking succeeded, the operation is guaranteed that the data in that node has not changed in the meantime.

A versioned lock l exposes an interface with the following semantics:

- $\text{GetVersion}(l)$ returns the current version number of the lock.
- $\text{TryLockAtVersion}(l, v)$ acquires the lock if and only if the lock is unlocked with version number equal to v, otherwise it returns false.
- $\text{Unlock}(l)$ atomically releases the lock and increments its version number.
- $\text{LockAtCurrentVersion}(l)$ will spin to acquire the lock at the earliest available opportunity.

These methods can be easily be implemented by treating the lock as an atomic integer l, with even numbers meaning unlocked states and odd numbers being locked states. The version number is the value of l with the least significant bit set to 0 (version numbers are always even). To attempt to acquire the lock at version v, perform $\text{CAS}(l, v, v + 1)$. Unlocking can be done by incrementing l, and $\text{LockAtCurrentVersion}(l)$ can be implemented using $\text{GetVersion}(l)$ and $\text{TryLockAtVersion}(l, v)$.

The versioned locks, therefore, allow *pre-locking validation*, as opposed to the *post-locking validation* used in the lazy list. The version number can be recorded and then a validation in the style of the lazy list can be performed, and the operation partially aborted if that validation fails. If the validation passes, the operation attempts to acquire the relevant lock at the recorded version number. If the lock was acquired, the operation is guaranteed that no data has changed since validation, so it can perform its work quickly before releasing the lock. Otherwise, the operation partially aborts and attempts to re-validate. If this validation fails, the operation must perform a full abort and restart from the front of the list.

The first advantage of this is that an operation will acquire a lock only if there is a guarantee that the operation needs that lock and can do useful work while holding it. The second advantage is that it keeps critical sections as small as possible: all preparation can be done without holding locks. Since all synchronisation is done through the locks, read-only operations on the versioned and lazy linked lists are very fast: they are wait-free, and unlike the Harris list or Fomitchev and Ruppert's list, they have unmarked pointers and do not waste time masking bits off pointers.

4.6 The Universal Construction

For completeness (as well as my own interest) I also implemented a linked list based on the universal construction technique presented in [Her93]. Because all updates are serialised through a single pointer, and because all update operations must make a full copy of the list, this should have very poor performance compared to the other highly concurrent linked lists, and this was observed in practice. However, it did have some very interesting behaviour under highly contended workloads, which is discussed in Chapter 5.

The pseudocode operations for this list are simple enough to state here. There is one shared pointer $head$, which points to the head of a regular, sequential linked list (with the head node storing $-\infty$ and the tail node storing $+\infty$). The update operations follow this basic format:

1. Read the shared pointer $head$, and store its value locally.
2. From this local value, make a full copy of the linked list.
3. Apply the update privately to the local copy. If that update was ineffective, return false.
4. Attempt to use CAS to swing the $head$ pointer to the local copy. If the CAS was successful, return true. Otherwise, restart the whole operation.

A successful update linearises when its CAS is successful, while an unsuccessful update linearises when it most recently read the shared $head$ pointer. A contains operation linearises at the point when it reads the shared pointer $head$. The contains operation is wait-free: it simply reads the shared pointer $head$, then checks for the existence of the given key in the list. Pseudocode for the contains and add operations are given in Figure 4.6. The remove operation is analogous to the add operation.

The "universal list" is easily seen as both correct and inefficient. If multiple unrelated (in the sense they operate on different keys) updates attempt to modify the list, they will always con-

```
**procedure** FIND($h$, $k$)
    $prev, curr \leftarrow h, h.next$
    **while** $cur.key < k$ **do**
        $prev, curr \leftarrow curr, curr.next$
    **end while**
    **return** $(prev, curr)$
**end procedure**

**procedure** CONTAINS($k$)
    $h \leftarrow head$
    $prev, curr \leftarrow$ FIND$(h, k)$
    **return** $curr.key = k$
**end procedure**

**procedure** ADD($k$)
    **loop**
        $h \leftarrow head$
        $copy \leftarrow$ COPY$(h)$
        $prev, curr \leftarrow$ FIND$(h, k)$
        **if** curr.key = k **then**
            **return** 0
        **end if**
        $prev.next \leftarrow$ NEWNODE$(k, curr)$
        **if** CAS(head, h, copy) = 1 **then**
            **return** 1
        **end if**
    **end loop**
**end procedure**
```

Figure 4.6: The universal linked list operations. The procedure Copy(h) makes a new copy of the linked list starting at the node h. The procedure NewNode$(key, next)$ returns a newly allocated node with the specified key and next fields.

flict at the root pointer, forcing all but one to restart. So it has a very poor degree of concurrency, but is still an interesting example to study, based on its behaviour under high contention, as explored in the next chapter.

Chapter 5

Experimental Evaluation

In this chapter, I evaluate the linked lists detailed in Chapter 4 experimentally. I start by comparing all of the lists under a uniform workload to show the difference between locking and lock-free structures in practice. I proceed to then test the lists under highly contended workloads to bring out their worst-case behaviours. I also compare Fomitchev and Ruppert's original list to my modification, showing the modification performs better in practice.

5.1 Experimental settings

Synchrobench

All of the linked lists in Chapter 4 were implemented in C and built into Synchrobench. Synchrobench [Gra15] is a micro-benchmarking suite for non-blocking data structures and software transactional memory, which allows a user to create their own benchmarks by setting different parameters, such as the structure size, proportion of updates, and so on. It also has detailed output about the number and type of operations performed by each thread, and whether they were effective or not. (Here, an update that inserts a value that is already in the set, or removes a value not present in the set, is *ineffective*).

The synchrobench options used during my testing are detailed below:

- Range $r \in \mathbb{N}$. Any randomly generated elements are drawn uniformly at random from the range $[1, r]$.
- Initial size $i \in [1, r)$. Before the benchmark starts running, the data structure is initialised to a size of i, by repeatedly inserting random elements from the update range.
- Duration: how long the each individual benchmark is run for.

- Update ratio $u \in [0, 1]$. Throughout the benchmark, a proportion u of operations will be updates (such as add or remove), and $1 - u$ will be read-only operations (such as contains).
- Threads $t \in \mathbb{N}$. The benchmark is run using t independent threads of execution. The threads share no data (aside from the common data structure) throughout the benchmark.
- Effective updates $f \in \{0, 1\}$. Turning on effective updates means that threads will reattempt updates if they were not effective, to try to get the proportion of effective updates to total operations equal to u.

Throughout a benchmark, Synchrobench will try to keep the size of the data structure as close to the initial size i as possible. Each thread remembers whether its last effective update was an insert or a remove, and will always attempt the opposite operation next. So, over the course of a benchmark, the actual data structure size can vary from the initial size i by at most $\pm t$, in practice it does not vary much at all.

I found that, in practice, setting the effective updates parameter f led to difficult-to-understand results. For example, the update ratio u could be set to 0.5, but more than 90% of operations performed were updates, due to most updates being ineffective in the uncontended case. For this reason, I have disabled updates ($f = 0$) in all of my experiments, and I note the observed proportion of observed updates for each experiment.

I also extended Synchrobench with my own options, a *bias range* and a *bias offset*. These options were used to test the behaviour of the linked lists in scenarios with high contention, where all threads attempt to insert or remove the same two values.

- Bias offset $B \in [1, r]$.
- Bias range $b \in \mathbb{N}$.

Switching on both of these flags causes Synchrobench to generate random numbers in the range $[B, B + b]$ instead of $[1, r]$ during the benchmark. When these bias flags are enabled, updates are chosen to be inserts or remove with equal probability, rather than based on the last effective update a thread performed.

Hardware and libraries

I used two machines for my experiments. The first was a 2 Socket Intel Xeon E5-2450 2.1GHz 8 cores (16 cores in total) with 2 hyperthreads per core (32 hyperthreads in total) machine running Ubuntu 12.04.5 LTS.. The second was a 4 socket AMD Opteron 6378 2.4 GHz 16 cores

(64 cores in total) machine running Fedora Linux 18. GCC 4.9.2 was used to compile the C code: the same binaries were run on both machines. The optimisation options passed to GCC were `-O3 -m64`.

All of the tests shown were run using the jemalloc[1] allocator. The standard allocator in Glibc uses locking internally, and thus becomes a bottleneck in these sort of experiments due to threads serialising by repeatedly calling `malloc`. Newer allocators such as jemalloc and tc-malloc allocate large chunks of memory per-thread every so often, after which many allocation calls on that thread are served from a thread-local cache, and can proceed without locking. The tests were run using jemalloc 3.6.0 compiled to a shared library, and loaded into each test run by using the `LD_PRELOAD` Linux environment variable.

The code for each of the linked lists I have implemented use the new C11 atomics library defined in `stdatomic.h`, and so will only compile with recent versions of GCC and Clang. The new atomics library allows for writing code which is portable between machines with different memory consistency guarantees, by explicitly specifying memory ordering guarantees per atomic operation.

5.2 A standard workload

To investigate the difference in performance between lock-based and lock-free data structures, I used both machines to test all of the linked lists I implemented under a fairly standard workload. The workload has an initial size i and an update ratio u listed above each graph. The other parameters used were the range $r = 2i$, and effective updates disabled ($f = 0$). Because the choice of the range, the proportion of effective updates can be expected to be $u/2$, and this was observed in practice. Each datapoint shown is the average of 20 runs of 5 seconds each, and the error bars shown are $\pm$ the sample standard deviation. The data for the 32 hyper-threaded Intel machine is shown in Figure 5.1, and the data for the 64 core AMD machine is shown in Figure 5.2.

Firstly, the universal linked list has poor performance in every setting considered in this standard workload. This is to be expected as it serialises all updates, whether or not those updates are operating on disjoint locations within the list. The throughput of the list increases slightly going from 1 to 4 threads, but then remains relatively constant all the way up to 80 threads on both the Intel and AMD machines. So while it has poor performance to begin with, its performance does not deteriorate when more threads are added.

The Harris list is the most efficient non-blocking list in the small 128-element cases on both machines. On the Intel machine, it scales almost linearly with the number of threads up to the

[1] `http://www.canonware.com/jemalloc/`

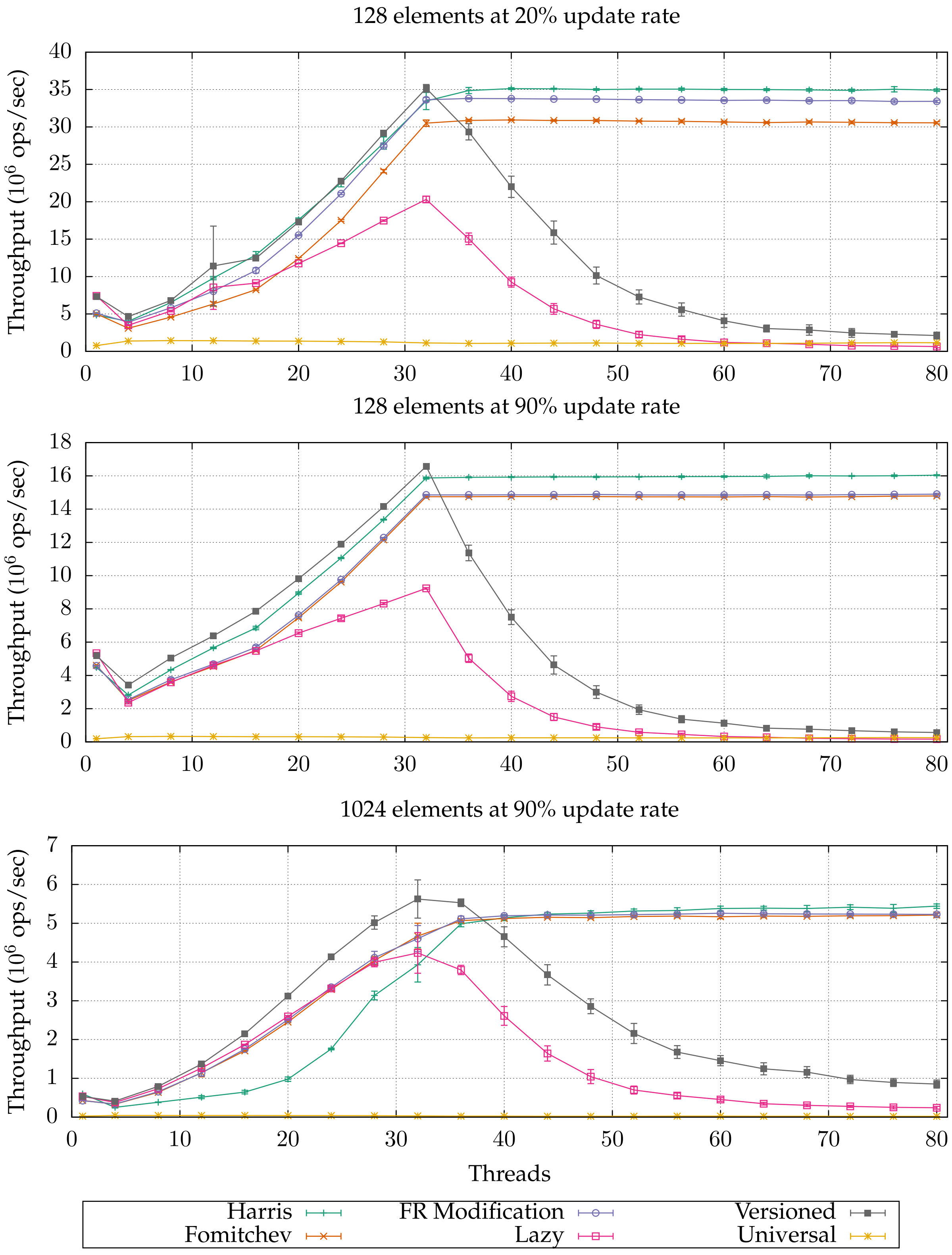


Figure 5.1: A benchmark showing a uniform workload at lists of different sizes, run on the 32 hyperthread Intel machine. The lock-based lists performance decreases as the number of threads exceeds the number of hyperthreads. The lock-free data structures suffer no performance penalty, and their throughput remains constant as the number of threads increases.

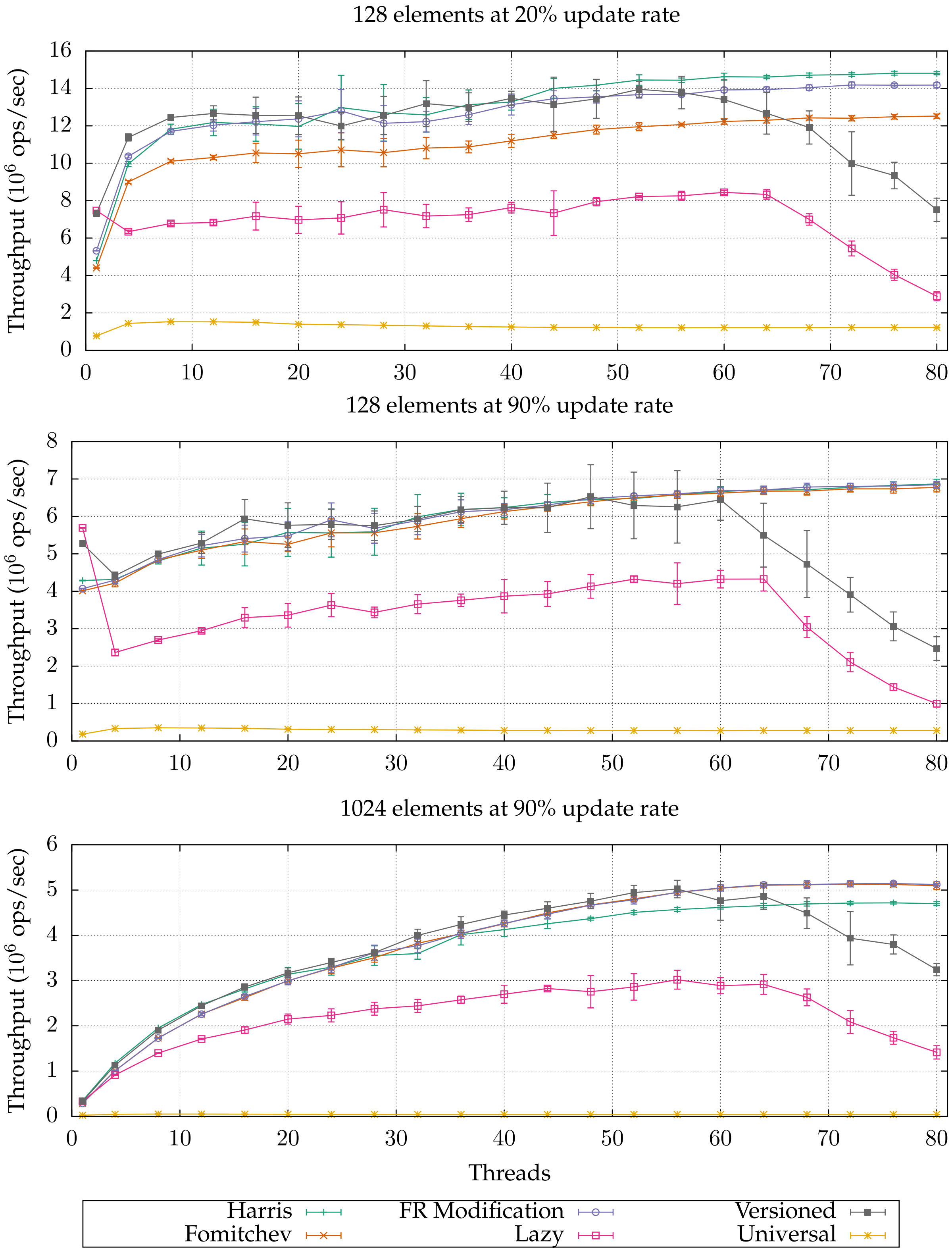


Figure 5.2: The same benchmark as shown in Figure 5.1, run on the 64 core AMD machine. Again, the performance of the lock-based lists decreases as the number of threads exceeds the 64 physical cores.

limit of 32 hyperthreads, from which point its performance remains constant. However, the 1024 element 90% update case on the Intel machine shows poor performance of the Harris list for small numbers of threads.

Fomitchev and Ruppert's list shows similar behaviour to the Harris linked list in most cases. It also has the property that when the number of threads exceeds the number of hardware threads or cores, its performance stays constant rather than deteriorating. We can see that the performance of the modification to Fomitchev and Ruppert's list is a strict improvement on the original algorithm when the update ratio is low, and the two are practically indistinguishable when the update ratio is high. Fomitchev and Ruppert's list is unaffected by the performance problems of Harris' list in the 1024 element 90% update ratio case on the Intel machine, and on the AMD machine it outperforms the Harris list in this case. This is consistent with the fact that the operations in Harris' list pay an $O(n)$ penalty for retrying, wheras in Fomitchev and Ruppert's list the penalty for retrying is constant (traversing a backlink), so the Harris list is impacted more by the large cases than Fomitchev and Ruppert's list.

Despite the worst-case complexity of $O(n + c)$ for Fomitchev and Ruppert's list, Harris' list usually performs better. There are a few reasons for this. Firstly, CAS steps are expensive as they require processor threads to lock cache lines before operating on data, which can be tens or hundreds of times slower than a regular read or write to shared memory. An insertion in either list only takes one (successful) CAS, but a deletion in the Harris list takes two successful CAS steps, while a deletion in Fomitchev and Ruppert's list takes three successful CAS steps. The large time impact of a CAS operation, along with the fact that more CAS operations means a larger window in which a thread could be pre-empted and have its next CAS fail, means that the Harris list will perform better in small cases than Fomitchev and Ruppert's list. Secondly, the size of a node in the Harris list is 16 bytes (on the 64-bit architechtures considered here), wheras the size of a node in Fomitchev and Ruppert's list is 24 bytes. Smaller nodes mean that more of the list can fit into a processor's local cache, and can also have effects on the allocation strategy chosen by the allocation library. However, we can see that a lot of the performance gap between the Harris list and Fomitchev and Ruppert's list can be closed by the modification with the wait-free contains operation, especially in cases with a low update ratio.

Moving on to the lazy list, we immediately see the dramatic performance difference of a lock-based list compared to a lock-free list. Once the number of threads exceeds the number of hardware threads, the performance of the lazy list falls off sharply, on both the Intel and AMD machines. When the number of threads does not exceed the number of hardware threads, the lazy list scales well, but in general is slower than the other lists.

Finally, the versioned linked list appears to have very good performance, outperforming every other list on the Intel machine, provided the number of threads used is less than the number of hyperthreads. When the number of threads exceeds the number of hardware threads or cores, its performance deteriorates rapidly, similarly to the lazy list.

There are two more features of the graphs which are interesting. The first one is that on the Intel machine the jump from 1 to 4 threads causes a significant performance drop, after which performance starts increasing again. The performance drop is more pronounced in the cases with a higher update ratio. I conjecture that these effects are due to Linux's default policy of running threads on separate NUMA (non-uniform memory architecture) nodes, which means that threads will be placed on different sockets and have to lock cache lines across the interconnect between processors. However, the same effect is not observed on the AMD machine, which again could be to do with the different caches and interconnects used in its architecture. The second feature is that raising the number of threads past the 16 physical cores doesn't always give a performance boost. As seen in the graphs, the lazy list seems to not be able to take advantage of the hyperthreads as much as the other data structures.

5.3 The effects of contention

In order to measure the cost due to contention in the linked lists, I created an artificial workload using the bias parameters I added to Synchrobench. A list of some initial size i was created, then when performing updates, every thread had to insert or remove the key with value $\lfloor i/2 \rfloor$ or $\lfloor i/2 \rfloor + 1$. This was done by setting the bias offset $B = \lfloor i/2 \rfloor$ and the bias range $b = 2$. Effective updates were not forced ($f = 0$), though in this case the effective update ratio varied between lists. I ran this experiment on both the AMD and the Intel machines, however here I only give results for the AMD machine, shown in Figure 5.3. The Intel machine gave chaotic and unpredictable results, and I could not track down why this was the case.

The behaviours of each list vary, so again I will go over each list in turn.

Firstly, the performance of the lazy list is very poor throughout these experiments, mostly due to the very contended locks on the middle three elements of the list. The performance of the lazy list drops off sharply as the number of threads jumps from 1 to 4, and continues to deteriorate as more threads are added. The combination of mutual exclusion locks, and the behaviour of retrying from the front of the list upon a failed validation, conspire to give the lazy list a very low throughput in this highly contended workload.

The performance of the universal list *increases* as more threads are added, which is a surprising result. In some of the benchmarks, it performs better than both the lazy list and the Harris list. The explanation of this good performance is the tendency of the universal list to "batch" together updates with the same key. Suppose there are 10 $\mathrm{Insert}(100)$ and 10 $\mathrm{Remove}(100)$ operations invoked simultaneously. Assuming operations proceed at relatively the same speed, each update will spend a while making a copy of the list, and then attempt to locally modify it. At this point, half the operations will instantly return false because they could do no work (if

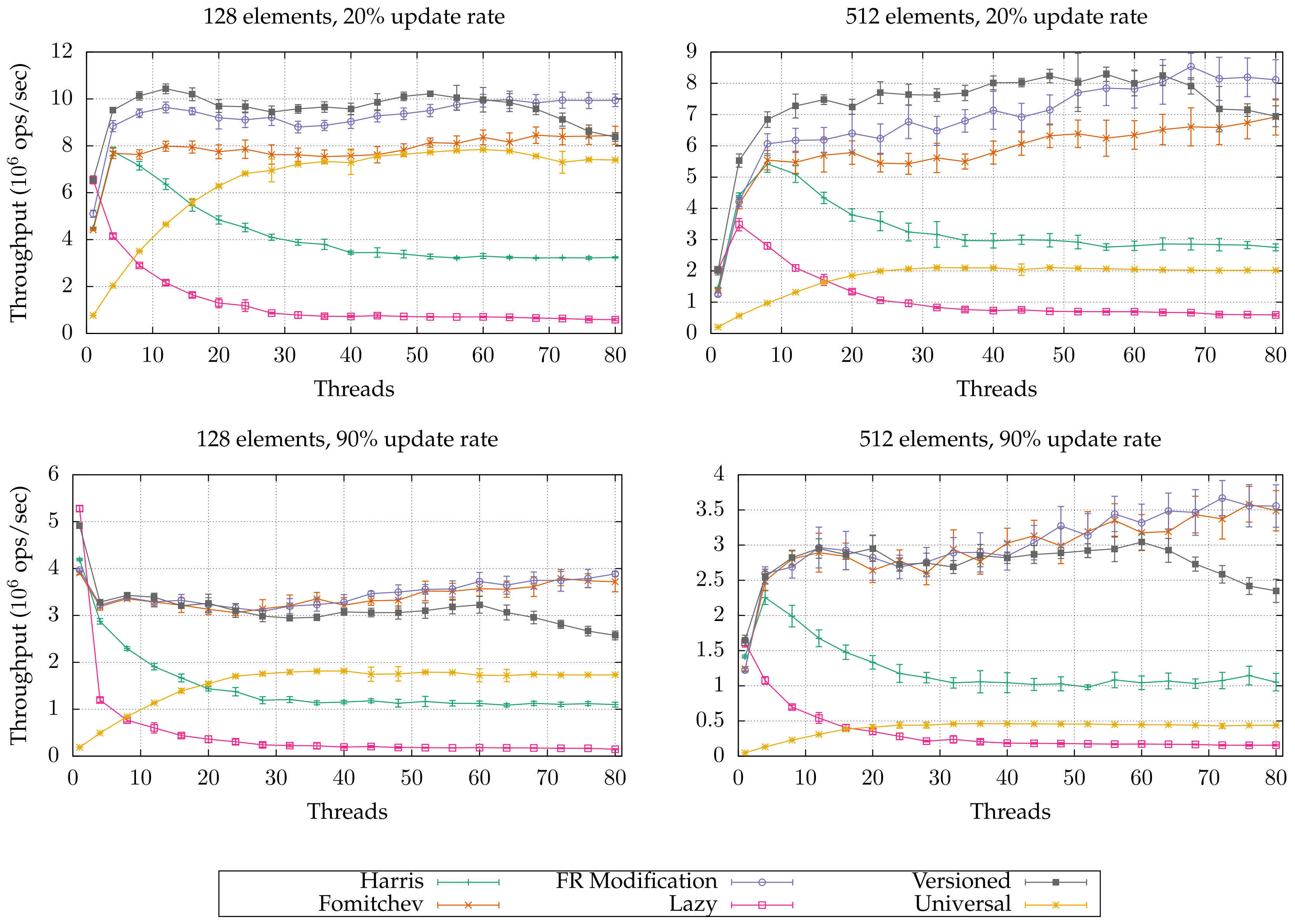


Figure 5.3: These graphs show the throughput of highly contended workloads on the 64 core AMD machine, where every update operation targets one of the middle two elements of the list.

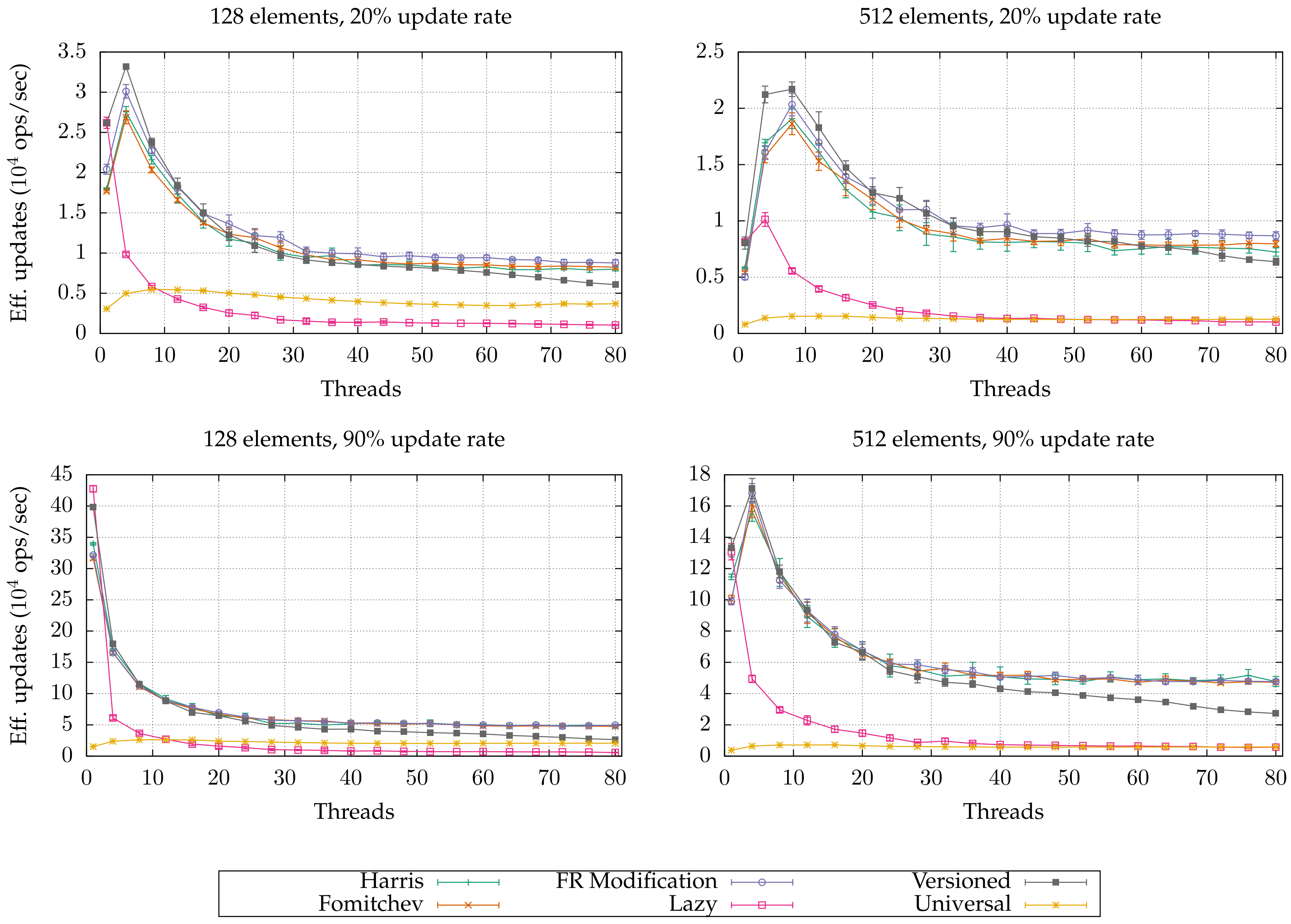


Figure 5.4: The same benchmark as shown in Figure 5.3, only showing the number of effective updates per second. The scale is in thousands, rather than millions, of updates per second.

100 was already in the set, the $\text{Insert}(100)$ operations return false, otherwise the $\text{Remove}(100)$ operations return false). One of the remaining updates takes effect by performing a CAS first, causing the other 9 to restart, all of which will (most likely) be ineffective, despite other concurrent operations. So using only one CAS success and 10 CAS attempts, 20 operations have been applied.

We can confirm this explanation by restricting the data to look only at the number of *effective* updates per second, rather than the number of overall operations per second. Figure 5.4 shows the same benchmarks as Figure 5.3, but the data shown is the number of effective updates per second. We can see that the number of effective updates the universal list is able to perform is much lower than the other lists. This highlights an interesting point: with concurrent data structures, the scheduling of operations can greatly affect the amount of work that needs to be done by an algorithm.

The performance of Harris' linked list drops significantly as more threads are added, evidence that the list is exhibiting its $\Omega(nc)$ worst-case behaviour. Interestingly, the Harris list still seems to be "optimal" in terms of the number of effective updates it can apply: looking at Figure 5.4 we see that all lists apart from the lazy and versioned lists apply the same number of effective updates, despite their varying performance overall.

Fomitchev and Ruppert's list shows a performance decrease coming from the uncontended workload in Figure 5.2, but retains its good scaling properties. As the number of threads is increased, its performance remains relatively constant, consistent with its worst-case amortised complexity of $O(n + c)$ per operation.

The versioned list behaves similarly to the Fomitchev and Ruppert list. I did not expect this, since if a pre-validation fails (which it does if the targets predecessor has since been deleted) the operation has to perform a full abort and rescan from the start of the list. It seems that this behaviour is sufficiently rare for it to not emerge even in this artificially contended case.

5.4 The modification of Fomitchev & Ruppert's list

My modification of Fomitchev and Ruppert's list has been included in all of the experiments so far. The standard workloads shown in Figure 5.1 and Figure 5.2 show the differences between the two lists nicely. I only changed the contains operation, so as the update ratio u is raised closer to 1, the performance of the two lists is almost identical. However, in cases where the update ratio u is lower, the performance of the list is increased substantially.

Figure 5.5 shows the performance of the original list against the modificaiton, on a standard workload with an update ratio of $u = 10\%$ for both the Intel and AMD machines. The modification is strictly faster than the original in all the workloads tested, with a 25% performance increase observed on the 64 core AMD machine at size 128 and update ratio 10%.

5.5 Summary of experimental results

The first experiment used a uniform workload at update ratios of 20% and 90%. We observed that lock-free linked lists have good scaling properties overall, and are immune to some inherent drawbacks of locks, such as performance decreasing when the number of threads exceeds the number of physical hyperthreads or cores on the machine. However, there may be locking data structures, such as the versioned linked list, which perform even faster than a lock-free data structure.

The second experiment used a highly contended workload where all updates targeted two values. It showed that the effects of contention are observable in practice, also demonstrated that algorithms which appear almost identical under a uniform workload may have very different behaviours under high contention. Lists such as the lazy linked list and the Harris linked list which perform full restarts suffer large performance penalties.

The third experiment, comparing Fomitchev and Ruppert's original algorithm to my modification, shows that the modified algorithm performs much better in practice, while having the same asymptotic time complexity. This, along with comparisons of these algorithms from the previous two experiments, shows that modifying a lock-free operation to be wait-free and perform no helping can have a practical performance benefit, even in situations of high contention.

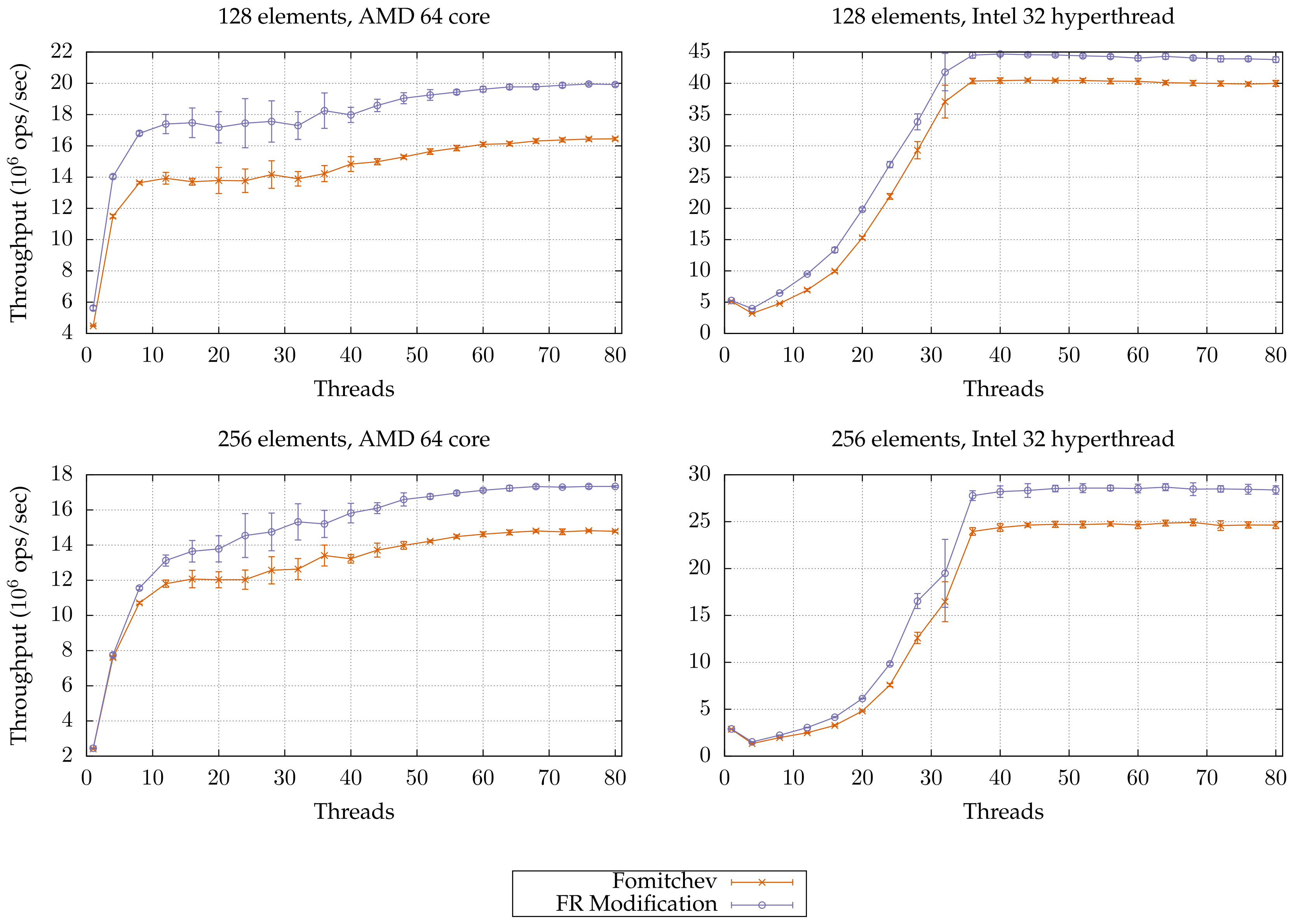


Figure 5.5: Standard workloads at an update ratio of 10%, showing the difference between the original algorithm by Fomitchev and Ruppert and my modification. The performance improvement is as great as 25%.

Chapter 6

Conclusion

6.1 Future Work

A more refined notion of contention

It was briefly discussed in Chapter 2 that recently there has been a view that by introducing more helping into an algorithm, the amortised step complexity can be "tightened" from an additive term of $O(c_I)$ to $O(c_P)$. By Theorem 1, we know these two quantities to be amortised equivalent, so clearly this cannot be distinguished by the point or interval contention. However, I suspect there could be a more refined measure of contention which does separate these cases.

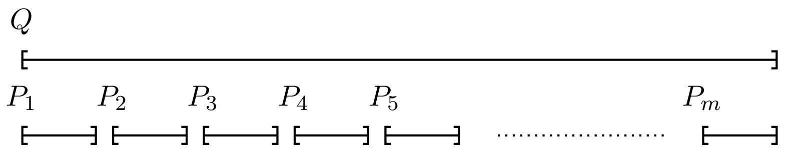


Figure 6.1: Suppose Q causes an inconsistency in the data structure and then gets suspended for a long time. If every operation performs eager helping, an inconsistency caused by Q will only be observed once. If no operations perform helping, it will be observed m times.

Consider an execution of one long-running process Q and m short-running processes $P_1, \ldots, P_m$. Suppose we have a structure featuring logical deletions, and the first step in a removal is to mark a node logically deleted. The long-running process Q marks an element as logically deleted before being suspended for a long time, while the short-running processes repeatedly access the data structure. This is illustrated in Figure 6.1. If the short running processes P_i perform no helping to try to physically remove the node, each will spend extra time traversing a node not present in the set, and incur a constant cost. The total cost of these extra steps is $\Theta(m)$. On the other hand, if every operation eagerly tries to help finish any partially completed delete operation it comes across, the first short operation P_1 will suffer this (constant) cost, and the rest will traverse with no extra cost. So without helping, there are $\Theta(m)$ extra steps that need to be carried out, and with helping there is only $O(1)$.

The current measures we have of contention, the point and interval contention, will not separate these cases. Some finer measure of contention is needed to capture this case, and show when helping can really benefit an algorithm in an asymptotic sense. I believe that some of the theory developed in Chapter 3 could be useful in determining and analysing a new measure of contention that separates these two cases.

Another point to note is that the performance of the lock-free and lock-based structures changes as the number of threads exceeds the number of hyperthreads or physical cores – lock-free structures appear to hold a constant level of performance while the lock-based structures deteriorate rapidly. Currently we have no good way of analysing this, even though intuitively it is something to do with contention: any context switch away from an operation holding a lock will have a huge impact on performance.

Contention in other data structures

Throughout this work I have focused on linked lists, so that I could conduct an in-depth survey and analyse the effects of contention thoroughly. However, linked lists are rarely used on their own as search structures. Many search structures are built on linked lists, or factor linked lists into their design. Perhaps the most obvious of these is a hash table using separate chaining, where linked lists are used as the "buckets" which store elements with colliding hashes. The algorithms described here can also be applied fairly directly to building concurrent skip lists, ordered data structures with expected $O(\log n)$ operation times.

To the best of my knowledge, Fomitchev and Ruppert's list has never been implemented before. Based on what I have observed, their algorithm performs a little slower than the Harris list, but with the wait-free contains modification it comes very close to the Harris list performance. It also has much better behaviour during highly contended workloads, as observed in practice. This could make the Fomitchev and Ruppert list the ideal lock-free list of choice for very low-latency applications.

In the same paper where they describe their list, Fomitchev and Ruppert also describe how to apply the list algorithms to building a concurrent skip list. Based on what has been experimentally observed in Chapter 5, I would expect their skip list to have better asymptotic performance under high contention than Fraser's, a skip list built using the algorithms from the Harris linked list [Fra04]. Contention in a skip list may be more realistic than contention in a linked list, since it's possible to have a number of processes on the order of $\log n$ even for a very large data structure. Furthermore, a similar wait-free modification for the contains operation is applicable to the skip list, which could improve performance even more.

Memory reclamation

In all of the linked lists I have considered here, I have not included any memory reclamation system. Memory reclamation is not straightforward in a highly concurrent environment, since it is hard to tell when no other thread has a reference to an object, even when that object has been removed from the data structure. Managed languages such as Java have a built-in garbage collector which makes sure unreachable objects are cleaned up, but there are settings where a garbage collector is either not available, or inappropriate (due, for example, to long pauses while collecting). Furthermore, with every linked list in Chapter 4, when a thread removes a node, it knows that no operations which start accessing the data structure after that time will ever reach that node. It would be nice to be able to use this information.

One strategy for reclamation I used successfully on another lock-free data structure (not mentioned in this thesis) was based on a paradigm called read-copy-update (RCU) which is used in the Linux kernel. The name is somewhat misleading: it's mostly a strategy for concurrent memory reclamation (although there are efforts to build concurrent data structures using it, for example the tree in [AA14]). In the Linux kernel, there are some data structures which allow any number of concurrent readers and at most one concurrent writer. The writer will acquire a lock, *read* relevant portions of the structure, make a local *copy*, and then publish this copy by performing an *update* on shared data. No synchronisation is required on the part of the readers. Once this update is complete, the writer may be left with parts of the data structure which are now unreachable. They cannot yet be discarded, as other operations may still be within the data structure. So the update puts the unreachable objects on a list of items to be freed later.

There are userspace implementations of RCU, and while they can't use all of the tricks that the kernel uses to make RCU fast, they provide a regular interface to reclaiming memory. Anywhere a sequential program would have called $\mathrm{Free}(v)$, a drop-in replacement can be used which saves v to a per-thread reclamation list. Once an RCU grace period has passed, the reclamation list can be considered safe, and nodes within it used again. This interface is in contrast to methods of reclamation like hazard pointers, which are difficult to implement and complicate the algorithm.

6.2 Conclusions

The major contributions presented in this thesis are new theoretical results about measures of contention, and an in-depth survey of high performance concurrent linked lists and their behaviour under contention. The theory of interval graphs was used to show that three currently used measures of contention are equivalent, which was conjectured to be false in recent work. This led to a simplified proof of the time complexity of a lock-free linked list, and the design of a new list-based set algorithm that has the lowest amortised time complexity known for lock-free list-based sets, and that outperforms all of the concurrent linked lists studied in the majority of my experiments.

Six different concurrent linked lists were implemented in C, and their performance was evaluated under a wide range of conditions, showing that contention has an observable impact on performance. The scaling properties of the lock-based and lock-free lists were explored, showing that lock-free algorithms perform well regardless of the number of logical threads and physical cores on the target machine, while lock-based algorithms are highly sensitive to this. Finally, avenues to further this area of research have been given, including the investigation of new measures of contention, and building lock-free skip lists with memory reclamation and good amortised time guarantees.

Bibliography

[AA14] Maya Arbel and Hagit Attiya. “Concurrent Updates with RCU: Search Tree as an Example”. In: (2014).

[AF03] Hagit Attiya and Arie Fouren. “Algorithms adapting to point contention”. In: *Journal of the ACM (JACM)* 50.4 (2003), pp. 444–468.

[AST02] Yehuda Afek, Gideon Stupp, and Dan Touitou. “Long lived adaptive splitter and applications”. In: *Distributed Computing* 15.2 (2002), pp. 67–86.

[CNT14] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. “Efficient lock-free binary search trees”. In: *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM. 2014, pp. 322–331.

[Ell+10] Faith Ellen et al. “Non-blocking binary search trees”. In: *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM. 2010, pp. 131–140.

[Ell+14] Faith Ellen et al. “The amortized complexity of non-blocking binary search trees”. In: *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM. 2014, pp. 332–340.

[FR04] 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*. ACM. 2004, pp. 50–59.

[Fra04] Keir Fraser. “Practical lock-freedom”. PhD thesis. University of Cambridge, 2004.

[GC96] Michael Greenwald and David Cheriton. “The synergy between non-blocking synchronization and operating system structure”. In: *ACM SIGOPS Operating Systems Review* 30.si (1996), pp. 123–136.

[GKR12] Vincent Gramoli, Petr Kuznetsov, and Srivatsan Ravi. “Brief announcement: From sequential to concurrent: correctness and relative efficiency”. In: *Proceedings of the 2012 ACM symposium on Principles of distributed computing*. ACM. 2012, pp. 241–242.

[Gra+15] Vincent Gramoli et al. *A Concurrency-Optimal List-Based Set*. Tech. rep. abs/1502.01633. arXiv, 2015.

[Gra15] Vincent Gramoli. “More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of Synchronisation on Concurrent Algorithms”. In: *PPoPP*. 2015, pp. 1–10.

[Har01] Timothy L Harris. “A pragmatic implementation of non-blocking linked-lists”. In: *Distributed Computing*. Springer, 2001, pp. 300–314.

[Hel+06] Steve Heller et al. “A lazy concurrent list-based set algorithm”. In: *Principles of Distributed Systems*. Springer, 2006, pp. 3–16.

[Her91] Maurice Herlihy. “Wait-free synchronization”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (1991), pp. 124–149.

[Her93] Maurice Herlihy. “A methodology for implementing highly concurrent data objects”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.5 (1993), pp. 745–770.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. “Obstruction-free synchronization: Double-ended queues as an example”. In: *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*. IEEE. 2003, pp. 522–529.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[HW13] Philip W Howard and Jonathan Walpole. “Relativistic red-black trees”. In: *Concurrency and Computation: Practice and Experience* (2013).

[HW90] Maurice P Herlihy and Jeannette M Wing. “Linearizability: A correctness condition for concurrent objects”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[OS13] Rotem Oshman and Nir Shavit. “The SkipTrie: low-depth concurrent search without rebalancing”. In: *Proceedings of the 2013 ACM symposium on Principles of distributed computing*. ACM. 2013, pp. 23–32.

[Pug90] William Pugh. “Skip lists: a probabilistic alternative to balanced trees”. In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[Sco13] Michael L. Scott. *Shared-Memory Synchronization*. Ed. by Mark D. Hill. Synthesis Lectures on Computer Architecture 23. Morgan & Claypool Publishers, 2013.

[TSP92] John Turek, Dennis Shasha, and Sundeep Prakash. “Locking without blocking: making lock based concurrent data structure algorithms nonblocking”. In: *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM. 1992, pp. 212–222.

[Val95] John D Valois. “Lock-free linked lists using compare-and-swap”. In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. ACM. 1995, pp. 214–222.