FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

KLSM: Ein Relaxed Lock-Free Priority Queue

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jakob Gruber, BA BSc
Matrikelnummer 0203440

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Prof. Dr. Scient. Jesper Larsson Träff

Wien, 18. Dezember 2015

Jakob Gruber	Jesper Larsson Träff

Technische Universität Wien
A-1040 Wien • Karlsplatz 13 • Tel. +43-1-58801-0 • www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

KLSM: A Relaxed Lock-Free Priority Queue

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jakob Gruber, BA BSc
Registration Number 0203440

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Prof. Dr. Scient. Jesper Larsson Träff

Vienna, 18th December, 2015

Jakob Gruber

Jesper Larsson Träff

Erklärung zur Verfassung der Arbeit

Jakob Gruber, BA BSc
Kirschenallee 6/1, 2120 Wolkersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Dezember 2015

Jakob Gruber

Acknowledgements

With thanks to

. . . Martin and Jesper, for your enthusiasm, ideas, and support.

. . . Moni, for being my longstanding partner in everything.

. . . My family, for the financial support and encouragement.

Without you, this thesis would not have been possible.

Kurzfassung

Priority queues sind abstrakte Datenstrukturen welche eine Menge von Key/Value Paaren speichern und effizienten Zugriff auf das minimale (maximale) Element erlauben. Sie sind ein wichtiger Teil in vielen Bereichen der Informatik, wie zum Beispiel Algorithmen (e.g., Dijkstra's kürzester Pfad Algorithmus) und Betriebssystemen (e.g., Priority Schedulers).

Der aktuelle Trend hin zu Multiprozessor Systemen benötigt neue Implementationen von Basisdatenstrukturen, die auch bis auf viele Threads skalieren. Sogenannte Lock-free Strukturen sind in der Hinsicht vielversprechend nachdem sie auf blockierende Synchronisationsmechanismen verzichten.

An parallelen Priority queues ist im vergangenen Jahrzehnt intensiv geforscht worden. In den letzten Jahren hatte sich die Forschung hauptsächlich auf SkipList-basierte Strukturen konzentriert, nachdem sie hervorragende disjoint-access Parallelismus Eigenschaften vorweisen, i.e., Schreibzugriffe an verschiedenen Elementen betreffen separate Teile der Datenstruktur. Contention zwischen verschiedenen Threads und das Datenvolumen durch das Cache-coherency Protokoll kann daher reduziert reduziert werden.

Die Skalierbarkeit hat sich jedoch bisher nur begrenzt verbessert. Auf der einen Seite haben strikte Priority queues einen inhärenten Engpass nachdem jede `delete_min` Operation auf das minimale (maximale) Element zugreift. Und auf der anderen haben SkipLists wegen der dynamischen Allokation jeder Node relativ schlechte Cache Locality, und daher eher niedrigen Throughput bei SkipList-basierten Designs.

Relaxed Datenstrukturen sind ein neuer und vielversprechender Zugang in dem höhere Skalierbarkeit durch schwächere Qualitätsgarantien erlangt wird. Die k-LSM ist ein paralleler, lock-free, und relaxed Priority queue Design, welcher hohe Skalierbarkeit verspricht: 1) einerseits durch die Benutzung von Arrays and der Merge Operation für hohe Cache Locality, und 2) andererseits derart geschwächte semantische Garantien, sodass `delete_min` eines der kP minimale (maximale) Elemente zurückgeben darf, wobei P die Anzahl der Threads und k ein Konfigurationsparameter ist.

Im Laufe dieser Arbeit haben wir eine erweiterte standalone Version der k-LSM Priority queue implementiert und beschreiben ihr Design sowie ihre Implementation bis ins Detail. Letztlich evaluieren wir die k-LSM gegen andere State-of-the-art parallele Priority queues wie die SprayList, den Lindén und Jonsson queue, und Rihani, Sanders und Dementiev's Multiqueues.

In manchen Benchmark Szenarios, inklusive der gängigen generic throughput Benchmark, zeigt die k-LSM hervorragende Skalierbarkeit und hat bis zu einem 10-fachen Speedup gegenüber den anderen Datenstrukturen. In anderen zeigt sich jedoch, dass das Verhalten der k-LSM stark Situationsabhängig ist. Von den anderen Designs scheinen sich die Multiqueues am besten zu verhalten, mit konsistenter Performance auf allen Benchmarks.

Abstract

Priority queues are abstract data structures which store a set of key/value pairs and allow efficient access to the item with the minimal (maximal) key. Such queues are an important element in various areas of computer science such as algorithmics (e.g. Dijkstra's shortest path algorithm) and operating system (e.g. priority schedulers).

The recent trend towards multiprocessor computing requires new implementations of basic data structures which are able to be used concurrently and scale well to large numbers of threads. Lock-free structures promise such scalability by avoiding the use of blocking synchronization primitives.

Concurrent priority queues have been extensively researched over the past decades. In recent years, most research within the field has focused on SkipList-based structures, based mainly on the fact that they exhibit very high disjoint access parallelism, i.e., modifications on different elements access disjoint parts of the data structure. Contention between threads and traffic through the cache-coherency protocol can therefore be reduced.

However, so far scalability improvements have been very limited. On the one hand, strict priority queues have an inherent sequential bottleneck since each `delete_min` operation attempts to access a single minimal (maximal) element. On the other, SkipLists have less than optimal cache locality since each node is usually allocated dynamically, which in turn results in fairly low throughput for SkipList-based designs.

Relaxed data structures are a new and promising approach in which quality guarantees are weakened in return for improved scalability. The k-LSM is a concurrent, lock-free, and relaxed priority queue design which aims to improve scalability by 1) using arrays as backing data structures and the standard merge algorithm as its central operation (for high cache locality); and 2) by relaxing semantic guarantees to allow the `delete_min` operation to return one of the kP minimal (maximal) elements, where P is the number of threads and k is a configuration parameter.

During the course of this thesis, we have implemented an improved standalone version of the k-LSM priority queue and explain its design and implementation in detail. We finally evaluate the the k-LSM against other state-of-the-art concurrent priority queues including the Spraylist, the Lindén and Jonsson queue, and Rihani, Sanders, and Dementiev's Multiqueues.

In some benchmarking scenarios, including the popular generic throughput benchmark, the k-LSM priority queue shows exceptional scalability, outperforming all other queues by up to a factor of ten; in others however, its throughput drops dramatically. Out of the other designs, Multiqueues are the clear winner, performing consistently across all benchmarks.

Contents

CHAPTER 1

Introduction

In the past decade, advancements in computer performance have been made mostly through an increasing number of processors instead of higher clock speeds. This development necessitates new approaches to data structures and algorithms that take advantage of concurrent execution by multiple threads and processors.

Priority queues are essential data structures and are used as building blocks in many situations such as shortest path algorithms and scheduling. In their most basic form, priority queues support two operations traditionally called `insert` and `delete_min`. The `insert` operation places an item into the queue together with its priority, while `delete_min` removes and returns the highest priority item. Both of these operations are expected to have a complexity of at most $O(\log n)$.

Concurrent priority queues have been the subject of research since the 1980s [4, 6, 13, 14, 26, 37, 38, 42, 44]. While early efforts have focused mostly on parallelizing Heap structures by using multiple locks [28], more recently priority queues based on Pugh's SkipLists [46] seem to show more potential due to their excellent disjoint access parallelism [24, 35, 50, 54].

Current research has also begun to examine relaxed data structures which trade strictness of provided guarantees for improved scalability. For instance, the SprayList [3] is an extension of SkipList-based priority queue designs which allows `delete_min` to randomly return one of the $O(P \log^3 P)$ minimal elements, where P is the number of threads. Multiqueues [48] have a simple and elegant design, and use a number of sequential priority queues — items are inserted into a random queue, and deletions return the minimal element of two random queues. Finally, the k-LSM [62] is a composition of two complementing priority queues: a relaxed queue called the Shared LSM (SLSM) which can offer global guarantees; and the Distributed LSM (DLSM), which has extremely high throughput but only observes local guarantees.

The k-LSM in particular has been shown to have very high scalability; but since it has only been implemented as part of the task-scheduling framework Pheet (`www.pheet.org`), direct comparisons against other state-of-the-art priority queues have been difficult so far. In order to remedy this point, we have implemented a standalone k-LSM variant in order to verify the findings of [62] outside the context of the Pheet framework.

In order to evaluate the performance of concurrent priority queue designs, recent literature [3, 8, 28, 35, 50, 54, 62] has overwhelmingly relied on a uniform workload, uniform key generation throughput benchmark, in which all threads perform a roughly equal mix of insertions and deletions, and item keys are generated uniformly at random within the key domain. It is, however, important to realize that this style of benchmarking induces a near-Last In First Out (LIFO)-like behavior in a priority queue: over time as minimal elements are removed from the queue, the queue becomes biased towards higher keys. Newly inserted items have a high probability of being one of the minimal items, thus soon becoming candidates for pending deletion.

We do not believe that the uniform workload, uniform key generation benchmark provides sufficient information about the properties of a priority queue. Consequently, our benchmarks have been extended with two parameters: key generation may be either uniformly at random, descending (for LIFO-like behavior) or ascending (First In First Out (FIFO)-like). Workload may be either uniform (in which all threads perform a roughly equal mixture of insertions and deletions), or split (half of all threads are dedicated inserters while the other half are deleters).

Our measurements show that while some data structures such as Multiqueues perform roughly equally in all situations, others seem to be more specialized and behave very well in some cases, and worse in others. The k-LSM in particular does extremely well in the standard uniform/uniform benchmark, outperforming the best other queue by almost a factor of ten. Unfortunately, its throughput drops in other cases.

Finally, quality of returned results are yet another facet in evaluation of relaxed data structures. Neither the Multiqueues, nor the SprayList offer fixed quality bounds; and while the k-LSM does guarantee that each returned item is one of the kP minimal elements, it would also be useful to determine how well the data structure performs on average.

In addition to the throughput benchmarks, we have also performed such quality measurements on the k-LSM and other comparable priority queues. Quality is measured through the rank of returned items, where the rank is the position of an item within the sorted contents of the queue. The k-LSM seems to return much better results than guaranteed, averaging at a rank of around $\frac{1}{20}$ of the upper quality bound.

The thesis is structured as follows: Chapter 2 outlines basic concepts and definitions. Chapter 3 provides an outline of sequential priority queues, with a focus on designs which have also been relevant to concurrent algorithms. In Chapter 4, we cover important concurrent priority queues with strict semantics, e.g.: the Hunt et al. queue as a representative of early heap-based queues using fine-grained locking to avoid the bottleneck of a single global lock; lock-free SkipList-based structures offering better disjoint-access parallelism; and a very recent design called the CBPQ. Chapter 5 presents three novel relaxed priority queues: the SprayList, Multiqueues, and the *k*-LSM. The implementation of the standalone *k*-LSM is examined in-depth in Chapter 6. Finally, experimental results are shown and discussed in Chapter 7, and the thesis is concluded in Chapter 8.

CHAPTER 2

Definitions

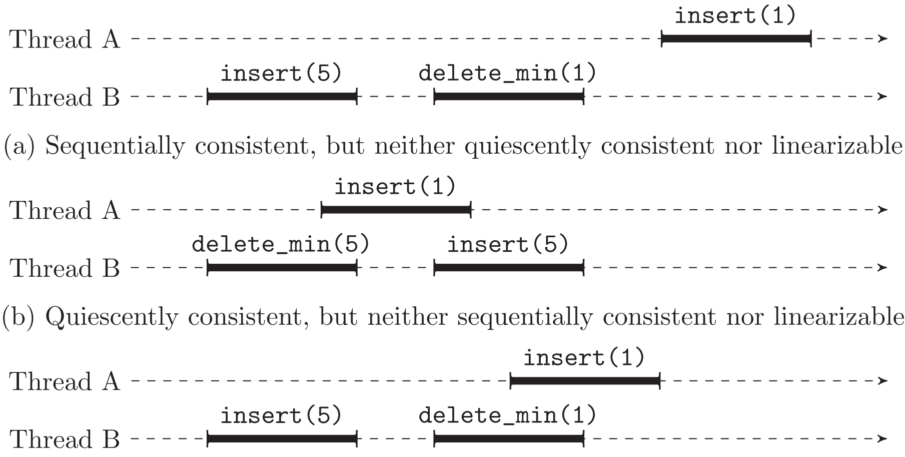
This chapter introduces basic terms and concepts required by the remaining thesis.

2.1 Lock-freedom

Concurrent data structures are intended to be accessed simultaneously by several processes at at the same time. *Lock-based* structures limit the number of processes that may enter a critical section at once through the use of locks. *Lock-free* data structures eschew the use of locks, and guarantee that at least one process makes progress at all times. Since lock-free structures are non-blocking, they are not susceptible to priority inversion (in which a high priority process is prevented from running by a low priority process) and deadlock (two processes wait for a resource held by the other). *Wait-freedom* further guarantees that every process finishes each operation in a bounded number of steps. In practice, wait-freedom often introduces an unacceptable overhead; lock-freedom however has turned out to be both efficient and to scale well to large numbers of processes. Recently, Kogan and Petrank have also developed a methodology for implementing efficient wait-free data structures from lock-free cases [33].

2.2 Correctness Conditions

There are several different criteria which allow reasoning about the correctness of concurrent data structures. *Linearizable* [25] operations appear to take effect at a single instant in time between the operation invocation and response at so-called linearization points. A sequence of concurrent linearizable operations must have an equivalent effect to some legal sequential sequence of the same operations. *Quiescently consistent* [51] data structures guarantee that the result of a set of parallel operations is equal to the result of a sequential ordering after a period of quiescence, i.e. an interval without active operations, has passed; however, no guarantees are given while one or more operations



(a) Sequentially consistent, but neither quiescently consistent nor linearizable.

(b) Quiescently consistent, but neither sequentially consistent nor linearizable.

(c) Sequentially consistent, quiescently consistent and linearizable.

Figure 2.1: A history of concurrent operations on an initially empty min-priority queue.

are in progress. Linearizability as well as quiescent consistency are composable — any data structure composed of linearizable (quiescently consistent) objects is also linearizable (quiescently consistent). *Sequential consistency* [34] requires the result of a set of operations executed in parallel to be equivalent to the result of some sequential ordering of the same operations. Contrary to linearizability and quiescent consistency, sequential consistency is not composable.

Figure 2.1 shows two examples of thread histories. The first is sequentially consistent since intra-thread reordering (delaying Thread B such that key 1 is inserted before it is deleted) results in a valid sequential history. It is is neither quiescently consistent (no valid history exists during the period of quiescence after `delete_min`), nor linearizable (there are no linearization points resulting in a valid history). Example 2.1c on the other hand is sequentially consistent, quiescently consistent, and linearizable (if `delete_min` has its linearization point after that of `insert(1)`).

2.3 Relaxation

In recent years, weaker versions of these criteria have been investigated as promising approaches towards higher scalability. Correctness criteria such as linearizability are usually applied to all operations and all threads (so-called *global ordering semantics*). On the other hand, in *local ordering semantics*, threads maintain thread-local copies of a central data structure, and modification to distinct local copies are not linearized between threads. *Quantitative relaxation* is recent model, which relaxes semantic guarantees such that operations may occur out of order in relation to a valid sequential operation sequence.

Quasi-linearizability [2], proposed in 2010, was the first relaxed correctness condition

$$\begin{tikzcd}
\text{RAM} \arrow[d, leftrightarrow] & \text{L3 Cache} \arrow[l, "\text{accesses}"'] \arrow[d, leftrightarrow] & \text{L1, L2 Caches} \arrow[l, "\text{accesses}"'] \arrow[d, leftrightarrow] & \\
\text{Machine} \arrow[r, "\text{contains}"] & \text{Processor} \arrow[r, "\text{contains}"] & \text{Processing Core} \arrow[r, "\text{contains}"] & \text{Processing Unit}
\end{tikzcd}$$

Figure 2.2: A typical system topology.

and sets a fixed upper bound to the distance to a valid sequential operation sequence. Quantitative relaxation [22] is a closely related concept but is based on the semantics of a data structure, e.g., allowing a priority queue to return the second-minimal item. ρ-relaxation [60, 61] is similar to quantitative relaxation, and defines correctness guarantees in terms of how many items may be skipped, or ignored, by an operation. Wimmer distinguishes between temporal ρ-relaxation, based on the recency of items, and structural ρ-relaxation, which relies on the position of an item within the data structure. Local linearizability [20] is a recent, relatively weak guarantee, and simply requires each thread-induced history (containing only operations on items inserted by that thread) to be linearizable.

2.4 Synchronization Primitives

Lock-free algorithms and data structures are commonly constructed using synchronization primitives such as Compare-And-Swap (CAS), Fetch-And-Add (FAA), Fetch-And-Or (FAO), and Test-And-Set (TAS). The CAS instruction, which atomically compares a memory location to an expected value and sets it to a new value if they are equal, is implemented on most modern architectures and can be considered a basic building block of lock-free programming since it is the most powerful of these operations [23]. More exotic primitives such as Double-Compare-And-Swap (DCAS) and Double-Compare-Single-Swap (DCSS) exist as well, but are not yet widely available in hardware and require software emulations (with associated overhead) to be used [21].

2.5 Hardware Topologies

Multiprocessor machines (Figure 2.2) are often built by combining several physical processors, each containing a collection of processing cores (and their associated cache hierarchy), which themselves may contain one or more processing units each. Every processing unit is capable of running one independent hardware thread.

Shared memory multiprocessor machines usually have a so-called non-uniform memory access (NUMA) architecture, in which the efficiency of memory accesses is determined by both the physical location of the memory and the active processing core. Often, each processing core has one or more dedicated levels of memory cache (typically L1 and L2 caches), and a level of shared cache per processor (typically L3). Access to these cache levels is rises by about an order of magnitude with each level; L1 caches being the fastest, followed by accesses to L2 and L3 caches, Random Access Memory (RAM), and finally the hard disk. Many concurrent algorithms are only efficient while all participating threads are located on a single processor and may share an L3 cache.

Cache coherency is a problem which arises in programs executed concurrently whenever multiple threads have an identical memory location (variable) cached in their local caches. Whenever a thread updates the variable, other threads' caches must be notified that a the variable has changed, and its value must be updated if required. This type of traffic is called the *cache coherency protocol*.

Locality is another critical aspect of effective cache use. Cache contents are loaded by blocks containing not just the requested location, but also its neighborhood. Thus algorithms with sequential data access patterns incur fewer cache loads than those with random accesses, adding up to significant performance gains.

An area in memory accessed frequently by a large number of processes is said to be *contended*. Contention is a limiting factor regarding scalability: concurrent reads and writes to the same location must be serialized by the cache coherence protocol, and only a single concurrent CAS can succeed while all others must retry. *Disjoint-access parallelism* is the concept of spreading such accesses in order to reduce contention as much as possible.

2.6 C++11 Memory Model

While thread support has previously only been added on to the C++ language through the POSIX Threads (pthreads) library (and others) [**boehm2005threads**], the C++11 standard finally specifies a fully multi-threaded abstract state machine with a clearly defined memory model. A memory model restricts the order in which changes to memory locations by one thread may become visible to other threads; for instance, usage of the the `std::atomic` type together with its `load()` and `store()` operations ensures portable multithreaded behavior across different architectures. It is possible to vary the strictness of provided guarantees between sequential consistency (on the strict end) and relaxed behavior (guaranteeing only atomicity).

CHAPTER 3

Sequential Priority Queues

Priority queues have a long history, having been extensively studied since the early days of the field of computer science. Some of the oldest designs, such as the heap, have remained popular in practical use up today. This chapter will detail syntax and semantics of general shared memory priority queues, and provide an overview of important sequential priority queue types, going into further detail for those designs which are also relevant for use in a concurrent environment.

3.1 Syntax and Semantics

Before going further, we define syntax and semantics for strict, sequential priority queues as used in this thesis.

```
template <class K, class V>
class priority_queue {
public:
    void insert(const K &key, const V &val);
    bool delete_min(V &val);
};
```

Figure 3.1: A generic priority queue class.

A priority queue is a data structure which holds a collection of key-value pairs, and provides two methods (see Figure 3.1): `insert`, which takes a key and a value argument, possibly of different types, and inserts them into the queue; and `delete_min`, which writes the value of the least item within the queue into the given parameter, and returns true if the operation succeeded. `delete_min` may fail if, for instance, the priority queue is empty.

Popular libraries such as *boost*[1], *LEDA*[2], and the C++ Standard Library (STL) use a slightly different interface (Figure 3.2) in which deletions are split into a read-only `top` method, which returns the minimal element without removing it; and `pop`, which deletes the minimal item without returning it. However, the split-deletion interface style causes issues in a concurrent setting in which it is essential that both reads and modifications must operate on the same element. For instance, imagine a case in which two threads concurrently each call `top` followed by `pop`; it is easy to construct a situation in which both `top` operations return a reference to the same element, while the two `pop` operations delete two distinct elements (one of which is lost).

Note that there is also another difference in that our priority queue syntax separates the key and value types, while the STL queue does not. Furthermore, `top` returns the entire item, while `delete_min` returns only the value — however, these points are cosmetic in nature and our interface could be adapted with relative ease.

```
template <class T>
class priority_queue {
public:
    void push(const T &value);
    const T &top() const;
    void pop();
};
```

Figure 3.2: The STL priority queue class.

There are various further possible extensions to the given interface:

- `empty` states whether the queue is empty.
- `peek_min` (or `top`) returns the minimal item's value without removing it from the queue.
- `decrease_key` decreases the key of a given item. This operation is vital e.g. for an efficient implementation of Dijkstra's shortest paths algorithm[15].
- `meld` merges two priority queues.

However, these are outside the scope of this thesis.

Each priority queue is associated with a priority function, which maps the key domain K to a priority domain P, over which there exists a reflective, antisymmetric, and transitive ordering relation $\leq: P \times P$.

[1] www.boost.org

[2] http://www.algorithmic-solutions.com/leda/index.htm

When K and P are the set of integers $\mathbb{Z}$, a priority queue is termed a "max-priority queue" if the priority function is the identity function $f(x) = x$, i.e., if higher keys have higher priority. It is called a "min-priority queue" if the priority function is the additive inverse $f(x) = -x$.

We are interested mainly in general priority queues within the context of shared memory systems. General priority queues are data structures which allow arbitrary keys of a given, possibly infinite set, and furthermore can hold multiple distinct items with identical keys. Such priority queues are congruent the sorting problem [55], and thus have a lower bound of $O(\log n)$ complexity for either `insert` or `delete_min` (or both). Note that these bounds do not hold for more specialized priority queues; for instance, queues which are limited to a small, previously known set of keys may use a bucket for each key and implement both insertions and deletions in constant time.

The semantics of general sequential priority priority queues are as follows:

- The contents of a priority queue PQ are a set of key-value pairs called items: $PQ \subseteq K \times V$, where K and V are, respectively, the key and value domains.
- Upon creation, at time $t = 0$, a priority queue is empty: $PQ_{t=0} = \emptyset$.
- Let PQ_t^+ be the set of all key-value pairs inserted into the priority queue up to time t, and let PQ_t^- be the set of all items removed up to time t. At each instant in time t, the set of items within the priority queue is then $PQ_t = PQ_t^+ \setminus PQ_t^-$.
- An insertion of key k and value v simply adds the pair to the set of inserted items.
- A deletion at time t finds the item $(k, v) \in PQ_t$ such that $\forall (x, y) \in PQ_t : f(x) \leq f(k)$ according to the priority function defined above. If such an item does not exist, i.e. the queue is empty, false is returned. Otherwise, (k, v) is added to the set of deleted items, and (true, v) is returned to the caller.

Note that we do not define an order within items of identical keys, and thus the priority queue is allowed to choose freely among items of equal priority.

In the following, we assume a min-priority queue and do not distinguish precisely between keys and their associated priority function. For instance, we simply use "the minimal item" to refer to the item within the queue of highest priority.

3.2 Overview

A naive implementation of a priority queue could be realized by using an cyclical array sorted according to the priority function. Access to the minimal element is possible in constant time since it is located at the head of the array. However, insertions need to insert the new item into its correct position within the array, and move all larger items back by one position, resulting in a worst-case complexity of $\Theta(n)$.

More efficient implementation techniques have been known for over half a century. Binary heaps [59] were invented as part of the sorting algorithm Heapsort in 1964 and are still arguable the most popular design for general purpose priority queues. For example, the priority queue container class contained in the C++ STL implements a binary heap on top of any compatible backing structure. When using a heap, both deletions and insertions require reorganization of the data structure and have logarithmic worst-case complexity.

While heaps use an implicit tree structure, it is also possible to use explicit balanced Binary Search Trees (BSTs) as priority queues with identical worst case bounds as heaps. Popular BST variants are, e.g., Red-black trees [5] and AVL trees [1]. Insertions use the standard BST insert algorithm, while deletions remove the tree's leftmost item. Alternatively, SkipLists [46] also provide logarithmic complexity but do not require periodic maintenance operations since they rely on randomization to preserve a balanced data structure.

Fibonacci heaps [18] were invented by Fredman and Tarjan and are based on a collection of heap-sorted trees. In addition to the standard insertion and deletion operations (in amortized logarithmic time), they also support efficient `merge` and `decrease_key` operations in amortized constant time, thus decreasing the complexity of Dijkstra's shortest path algorithm from $O(|E| \log |V|)$ to $O(|E| + |V| \log |V|)$. However, due to their programming complexity and involved constant factors, they are not commonly used in practice. Subsequent publications have proposed various alternatives to Fibonacci heaps such as relaxed heaps [16] and strict Fibonacci heaps [11].

Other well-known priority queue designs are the Skew Heap [53], based on a heap-ordered binary tree and the `meld` operation; the Splay Tree [52], a heuristically balanced BST which attempts to move frequently used sections towards the root of the tree; pairing heaps [19], a self-adjusting version of the binomial heap [58]; and many more. For further information, we refer the reader to reviews such as [31, 49].

3.3 Binary Heaps

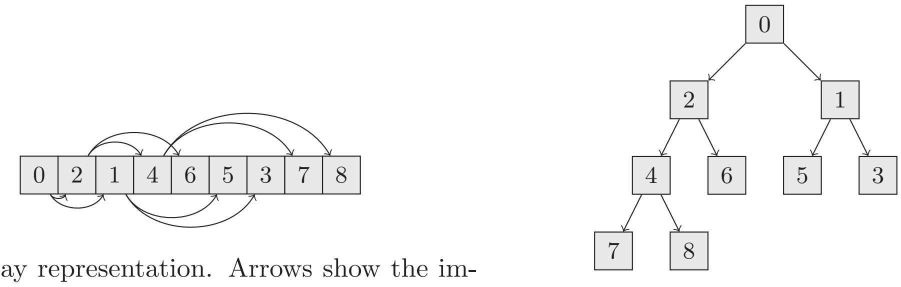
Binary heaps [59] are one of the most common (and also one of the simplest) methods of implementing priority queues. They are based on two concepts; first, that of complete balanced binary trees, composed of inner nodes with exactly two children each. And second, that of heap ordering, the notion that the priority of each child node is less or equal to that of its parent node. Furthermore, instead of using an explicit representation of a binary tree, heaps implicitly encode the position of their contents in a so-called heap-ordered array.

As shown in Figures 3.3 and 3.4, the backing data structure of a binary heap is a flat array of `items` together with a variable `n` tracking the number of items within the heap. The root item, i.e. the item with the highest priority, is located at the head of the array

```
template <class K, class V>
class heap {
public:
    void insert(const K &key, const V &val);
    bool delete_min(V &val);

private:
    std::pair<K, V> items[CAPACITY];
    size_t n;
};
```

Figure 3.3: A binary heap class.



(a) Array representation. Arrows show the implicit parent-child connections between items.

(b) Tree representation.

Figure 3.4: The same heap displayed once in its native array representation, and once as its corresponding tree.

`items[1]`[3]. For each item at position `items[i]`, its left child is located at `items[2i]` and its right child at `items[2i + 1]`. Vice versa, the parent item of position `items[i]` can be found at `items[i / 2]`.

Insertions initially append the new item at the tail of array `items[n + 1]`. Note that heap order might be violated at this point if the inserted item has a higher priority than its parent. If that is the case, the item and its parent are swapped. This operation is repeated until the entire heap again preserves heap order. It may be repeated once for each level in the implicit tree, i.e. at most $\lceil \log n \rceil$ times, and takes constant time in each iteration.

Deletions first remove the root item, located at `items[1]`, and store it as the eventual return value. The heap structure is then preserved by moving the item at the tail of the array (`items[n + 1]`) into the root position, and then swapping it with its highest priority child until the item is again in a position in which it is of higher priority than both children. Like insertions, this operation takes constant effort in each iteration, and is repeated at most once for each level, resulting in a worst case of $O(\log n)$.

[3] For ease of presentation, we use 1-based indexing within this section.

Binary heaps are efficient in practice and conceptually simple as well as easy to implement. They are space-efficient since they do not store any overhead per item such as pointers (in BST-based implementations). One of their main drawbacks is that both deletes and inserts are of logarithmic complexity. Others become apparent only in a concurrent setting, in which their tendency to modify large parts of the data structure during both insertions and deletions causes contention, high cache-coherence protocol traffic, and synchronization issues.

3.4 SkipList

As previously discussed, BSTs are a viable approach to implementing priority queues with logarithmic worst-case bounds for insertions as well as deletions. SkipLists [46] are very similar conceptually, but use a different approach to physically represent the data structure, and use randomization in order to avoid having to perform regular maintenance operations while preserving expected logarithmic time bounds.

```
template <class K, class V>
class skiplist {
public:
    void insert(const K &key, const V &val);
    bool delete_min(V &val);

private:
    skiplist_node<K, V> *head[MAX_HEIGHT];
};

template <class K, class V>
struct skiplist_node {
    K key;
    V val;

    size_t level;
    skiplist_node<K, V> *next[MAX_HEIGHT];
};
```

Figure 3.5: A SkipList priority queue class.

A SkipList is essentially a linked list of specialized nodes, each of which contains a key-value pair, an associated node level, and a list of pointers to the next node (Figures 3.5 and 3.6). The node's level determines how many pointers to the next node exist, and a pointer at level l links to the next node of level $\geq l$ within the data structure. Higher level pointers thus act as shortcuts into the list, avoiding the usual linear cost search operation of linked lists.

Insertions create a new node, assigning the level according to a geometric distribution. The correct position for the new node is determined by searching for the given key within

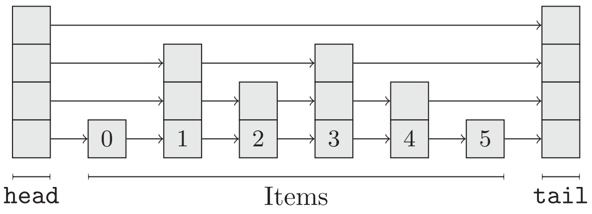


Figure 3.6: A SkipList with 6 elements and a maximal level of 3. Each column represents a single node. The first and last nodes are, respectively, artificial head and tail nodes. Note that the minimal element is always accessible through the first pointer on level 0.

the list. Finally, the new node is physically inserted by modifying pointers of previous nodes and the new node.

A search proceeds by starting at the maximal level, and and repeatedly walking along its associated pointer list until a node is found with a higher key than the desired one. The inspected level is then decremented, and the operation is repeated (if $level > 0$), or the current location returned as the correct position (otherwise).

`delete_min` simply removes the head node of the list by setting its successor at level 0 as the new head node, and then returns the previous head's value.

Note that since node levels are distributed geometrically, we expect each level l to contain twice as many nodes as level $l + 1$. Thus both the expected maximal level of all nodes, as well as the number of steps required to find any given key within the SkipList is $O(\log n)$.

The SkipList has gained popularity in recent research into parallel priority queues, since it does not require the frequent rebalancing operations of BSTs, and operations on nodes of different keys usually access spacially disjoint parts of the data structure. In the context of concurrent operations, both of these properties are very desirable: if several concurrent insertions modify disjoint parts of the SkipList, contention is reduced. And since the SkipList does not require balancing, both insertions and deletions are simpler and more efficient to implement.

Unfortunately, and as we will see in our results in Chapter 7, SkipLists have turned out to be rather slow in practice. This is most likely due to the fact that each node is usually dynamically allocated in non-contiguous parts of memory, resulting in very bad cache locality.

CHAPTER 4

Strict Concurrent Priority Queues

This chapter deals with priority queues with strict semantics which guarantee that deletions return the minimal element within the queue. We provide an overview of the development of such priority queues, from early locking heap-based designs, through more recent lock-free queues backed by SkipList, to one of the latest designs called the CBPQ.

4.1 Fine-grained Locking Heaps

In the following two chapters, we will discuss implementations of several concurrent priority queue implementations. Early designs have mostly been based on search trees [7, 30] and heaps [4, 6, 13, 14, 26, 37, 38, 42, 44]. We chose the priority queue by Hunt et al. [28] as a representative of early concurrent priority queues since it has been shown to perform well [50] in comparison to other efforts of the time such as [4, 41, 63]. It is based on a Heap structure and attempts to minimize lock contention between threads by a) adding per-node locks, b) spreading subsequent insertions through a bit-reversal technique, and c) letting insertions traverse bottom-up in order to minimize conflicts with top-down deletions.

However, significant limitations to scalability remain. A global lock is required to protect accesses to a variable storing the Heap's size which all operations must obtain for a short time. Disjoint-access through bit-reversal breaks down once a certain amount of traffic is reached, since only subsequent insertions are guaranteed to take disjoint paths towards the root node. Note also that the root node is a severe serial bottleneck, since it is potentially part of every insertion path, and necessarily of every `delete_min` operation. Finally, in contrast to later dynamic SkipList-based designs, the capacity of Hunt Heaps is fixed upon creation.

Benchmarking results in the literature have been mixed; a sequential priority queue protected by a single global lock outperforms the Hunt et al. Heap in most cases [28, 54]. Speed-up only occurs once the size of the Heap reaches a certain threshold such that concurrency can be properly exploited instead of being dominated by global locking overhead.

4.2 Lock-free Priority Queues

Traditional data structures such as heaps have fallen out of favor; instead, SkipLists [45, 46] have become the focus of modern concurrent priority queue research [3, 24, 35, 50, 54]. SkipLists are both conceptually simple as well as easy to implement; they also exhibit excellent disjoint-access parallelism properties, and do not require rebalancing due to their reliance on randomization.

A state of the art lock-free SkipList implementation based on the CAS instruction by Fraser [17] is freely available[1] under a BSD license. Fraser exploits unused pointer bits to mark nodes as logically deleted, with physical deletion following as a second step.

SkipLists are dynamic data structures in the sense that they grow and shrink at runtime. In consequence, careful handling of memory accesses and (de)allocations are required. As an additional requirement, these memory management schemes must themselves be both scalable and lock-free to avoid limiting the SkipList itself. Fraser in particular employs lock-free epoch-based garbage-collection, which frees a memory segment only once all threads that could have seen a pointer to it have exited the data structure.

4.3 The Shavit and Lotan Priority Queue

Shavit and Lotan were the first to propose the use of SkipLists for priority queues [35]. Their initial locking implementation [50] builds on Pugh's concurrent SkipList [45], which uses one lock per node per level.

A crucial observation is that nodes which are only partially connected (in the sense that while the lowest level has been connected but not necessarily all other levels) do not affect correctness of the data structure. As soon as the first level (i.e. `node.next[0]`) has been successfully connected, a node is considered to be in the SkipList. Therefore, both insertions and deletions can be split into steps — insertions proceed bottom-up while deletions proceed top-down. Locks are held only for the current level which helps to reduce contention between threads.

Likewise, deletions are split into a logical phase (atomically setting the `node.deleted` flag) and a physical phase which performs the actual pointer manipulations and can be seen as a simple call to the underlying SkipList's `delete_min` function.

[1] `http://www.cl.cam.ac.uk/research/srg/netos/lock-free/`, last visited on December 9th, 2015.

```
struct node_t {
  [...] /**< Standard node members as in Figure 3.5. */
  atomic<bool> deleted; /**< Initially false. */
  time_t timestamp;
  lock_t locks[level + 1];
};
```

Figure 4.1: Shavit and Lotan structure.

A `delete_min` call starts at the list head, and attempts to atomically set the deletion flag using a `CAS(node.deleted, false, true)` call (or equivalent constructs). If it succeeds, the current node is physically deleted and returned to the caller. Otherwise, `node.next[0]` is set as the new current node and the procedure is repeated. If the end of the list is reached, `delete_min` returns false to indicate an empty list.

Note that so far this implementation is not linearizable: consider the case in which a slow thread A is in the middle of a `delete_min` call. Within this context, we refer to the node with key i as node i, or simply i. Several `CAS` operations have failed, and A is currently at node j. A fast thread B then first inserts a node i, followed by a node k such that $i < j < k$, i.e. the former and latter nodes are inserted, respectively, before and after thread A's current node. Assuming further that all nodes between j and k have already been deleted, then thread A will return node k. This execution is not linearizable; however, it is quiescently consistent since operations can be reordered to correspond to some sequential execution at quiescent periods.

Shavit and Lotan counteract this by introducing a `timestamp` for each node which is set upon successful insertion. In this variant, each `delete_min` operation simply ignores all nodes it sees that have not been fully inserted at the time it was called.

Explicit memory management is required to avoid dereferencing pointers to freed memory areas by other threads after physical deletion. This implementation uses a dedicated garbage collector thread in combination with a timestamping mechanism which frees `node`'s memory only when all threads that might have seen a pointer to `node` have exited the data structure.

Herlihy and Shavit [24] recently described and implemented a lock-free, quiescently consistent version of this idea in Java. While mostly identical, notable differences are that a) the new variant is based on a lock-free skiplist, b) the timestamping mechanism was not employed and thus linearizability was lost, and c) explicit memory management is not required because the Java virtual machine provides a garbage collector.

4.4 Sundell and Tsigas

Sundell and Tsigas proposed the first lock-free concurrent priority queue in 2003 [54]. The data structure is linearizable and is implemented using commonly available atomic primitives CAS, TAS, and FAA. In contrast to other structures covered in this thesis, this priority queue is restricted to contain items with distinct priorities. Inserting a new item with a priority already contained in the list simply performs an update of the associated value.

```
struct node_t {
  [...] /**< Standard node members as in Figure 3.5. */
  value_t *value; /**< Values are stored by pointer. */
  size_t valid_level;
  node_t *prev;
};
```

Figure 4.2: Sundell and Tsigas structure.

The structure of each node is basically identical to Figure 3.5. However, Sundell and Tsigas exploit the fact that the two least significant bits of pointers on 32- and 64-bit systems are unused (due to alignment of allocated memory) and reuse these as deletion marks. A simple CAS operation can then be used to atomically update the pointer and the deletion marks: `CAS(node.next[i], ptr, ptr | 1)`. A set least significant bit on a pointer signifies that the current node is about to be deleted. This packed use of `node.next[i]` pointers prevents situations in which a new node is inserted while its predecessor is being removed, effectively deleting both nodes from the list. Likewise, the reuse of the `node.value` pointer ensures that updates of pointer values (which occur when a node with the inserted priority already exists) handle concurrent node removals correctly.

As in the Shavit and Lotan priority queue, insertions proceed bottom-up while deletions proceed top-down — on the one hand, the choice of opposite directions reduces collisions between concurrent insert and delete operations, while on the other hand removing nodes from top levels first allows many other concurrent operations to simply skip these nodes, further improving performance. The `node.valid_level` variable is updated during inserts to always equal the highest level of the SkipList at which pointers in this node have already been set (as opposed to `node.level`, which equals the final level of the node).

A helping mechanism is employed whenever a node is encountered that has its deletion bit set, which attempts to set the deletion bits on all next pointers and then removes the node from the current level. The `node.prev` pointer is used as a shortcut to the previous node, avoiding a complete retraversal of the list.

This implementation uses the lock-free memory management invented by Valois [56, 57] and corrected by Michael and Scott [40]. It was chosen in particular because this scheme can guarantee validity of `prev` as well as all `next` pointers. Additionally, it does not require a separate garbage collector thread.

A rigorous linearizability proof is provided in the original paper [54] which shows linearization points for all possible outcomes of all operations.

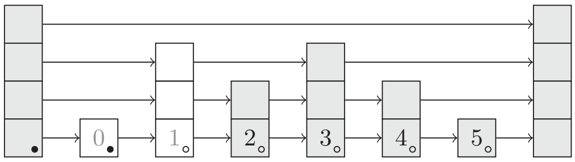


Figure 4.3: A Lindén and Jonsson priority queue with two deleted elements. Deletion flags are shown as a filled (if set) or clear (unset) circle at the lowest node level. Note that deletion flags are located in their predecessors of deleted nodes 0 and 1.

Benchmarks performed by Sundell and Tsigas show their queue performing noticeably better than both locking queues from Sections 4.3 and 4.1, and slightly better than a priority queue consisting of a SkipList protected by a single global lock.

4.5 Lindén and Jonsson

A recent priority queue implementations was published by Lindén and Jonsson in 2013 [35]. They present a linearizable, lock-free concurrent priority queue which achieves a speed-up of $30 - 80\%$ compared to other SkipList-based priority queues by minimizing the number of CAS operations within most `delete_min` operations.

A priority queue implementation is called strict when `delete_min` is guaranteed to return the minimal element currently within the queue (in contrast to relaxed data structures which are discussed further in the next chapter). All such priority queues have an inherent bottleneck, since all threads calling `delete_min` compete for the minimal element, causing both contention through concurrent CAS operations on the same variable as well as serialization effort by the cache coherence protocol for all other processor accessing the same cache line.

```
struct node_t {
  [...] /**< Standard node members as in Figure 3.5. */
  atomic<bool> inserting;
};
```

Figure 4.4: Lindén and Jonsson structure.

In this implementation, most `delete_min` operations only perform logical deletion by setting the deletion flag with a single FAO call; nodes are only deleted physically once a certain threshold of logically deleted nodes is reached.

This mechanism guarantees that the set of all logically deleted nodes must always form a prefix of the SkipList. Recall that in the Sundell and Tsigas queue, deletion flags for node `n` were packed into `n.next` pointers, preventing insertion of new nodes *after* deleted nodes. This implementation instead places the deletion flag into the lowest level `next` pointer of the previous node, preventing insertions *before* logically deleted nodes.

Once the prefix of logically deleted nodes reaches a specified length (represented by `BoundOffset`), the first thread to observe this fact within `delete_min` performs the actual physical deletions by updating `slist.head[0]`, the lowest level pointer to the head of the list, to point at the last logically deleted node with a single CAS operation. The remaining `slist.head` pointers are then updated, and all physically deleted nodes are marked for recycling.

Since at any time, the data structure contains a prefix of logically deleted nodes, all `delete_min` operations must traverse this sequence before reaching a non-deleted node. In general, reads of nonmodified memory locations are very cheap; however, benchmarks in [35] have shown that after a certain point, the effort spent in long read sequences significantly outweighs the reduced number of CAS calls. It is therefore crucial to choose `BoundOffset` carefully, with the authors recommending a prefix length bound of 128 for 32 threads.

The actual `delete_min` and `insert` implementations are surprisingly simple. Deletions simply traverse the list until the first node for which `(ptr, d)= FAO(node.next[0], 1)` returns an unset deletion flag (`d = 0`), and then return `ptr`. Insertions occur bottom-up and follow the basic Fraser algorithm [17], taking the separation of deletion flags and nodes into account. The `node.inserting` flag is set until the node has been fully inserted, and prevents moving the list head past a partially inserted node. Fraser's epoch-based reclamation scheme [17] is used for memory management.

The authors provide high level correctness and linearizability proofs as well as a model for the SPIN model checker. Performance has been shown to compare favorably to both Sundell and Tsigas and Shavit and Lotan queues, with throughput improved by up to 80%.

4.6 Chunk-Based Priority Queue

The latest strict priority queue of interest, called the Chunk-Based Priority Queue (CBPQ), was published very recently in a dissertation by Braginsky [8]. It is primarily based on two main ideas: the chunk linked list [10] replaces SkipLists and heaps as the backing data structure, and use of the more efficient FAA instruction is preferred over the more powerful CAS instruction.

The chunked linked list is a lock-free implementation of a linked list of arrays (chunks), each of which is sized to approximately fit into a memory page and thus provide high locality upon traversal. Within the CBPQ, each chunk is assigned to a particular keyrange, and a skiplist is used to efficiently index into the available set of chunks during insertions.

A chunk itself consists of an element array, the upper bound for keys within the chunk, the index of the first unused field within the array, and several further fields related to memory management (which we ignore in this context — for further details about the utilized lock-free memory management scheme called 'Drop the Anchor', refer to [9, 10]).

One of the key elements of the CBPQ is that in general, items within chunks are not ordered and can thus be inserted in a lock-free manner by simply fetching and incrementing a chunk's index field using an FAA instruction, and then physically inserting the item in the appropriate spot. If a chunk exceeds its maximum size after insertion, it is split into two new chunks, each owning half of the original keyrange.

Deletions take items exclusively from the first chunk since it contains the lowest range of keys. This initial chunk is a special case within the design; its item array is immutable and may thus be efficiently read by all threads with reduced cache coherence traffic. Again, a shared index is used in conjuction with the FAA instruction to determine an item to remove in an efficient manner.

Whenever a new initial chunk must be constructed for any reason (e.g., a new item within its keyrange is inserted, or the current initial chunk is empty), the respective items are first inserted into an buffer. The inserting thread waits for a short amount of time in order to possibly batch together insertions from several threads. All participating threads then cooperate to create a new initial chunk.

Benchmarks compare the CBPQ against the Lindén and Jonsson queue and lock-free as well as lock-based versions of the Mound priority queue [36] for different workloads. The CBPQ clearly outperforms the other queues in mixed workloads (50% insertions, 50% insertions) and deletion workloads, and exhibits similar behavior as the Lindén and Jonsson queue in insertion workloads, where Mounds are dominant.

CHAPTER 5

Relaxed Concurrent Priority Queues

The body of work discussed in previous sections creates the impression that the upper limits of strict priority queues have been reached. In particular, Lindén and Jonsson conclude that scalability is solely limited by `delete_min`, and that less than one modified memory location per thread and operation would have to be read in order to achieve improved performance [35] through SkipLists. Through a novel concept, the CBPQ [8] manages to improve on the Linden queue by up to a factor of two, but only scales while executed on a single processor.

Recently, relaxation of provided guarantees have been investigated as another method of reducing contention and improving disjoint-access parallelism. For instance, k-FIFO queues [32] have achieved considerable speed-ups compared to strict FIFO queues by allowing `dequeue` to return elements up to k positions out of order.

Relaxation has also been applied to concurrent priority queues with some success, and in the following sections we discuss three such approaches.

5.1 SprayList

The SprayList is a recent relaxed priority queue design by Alistarh et al. [3]. Instead of taking a distributed approach, the SprayList is based on a central data structure, and uses a random walk in `delete_min` in order to spread accesses over the $O(P \log^3 P)$ first elements with high probability, where P is the number of participating threads.

Fraser's lock-free SkipList [17] again serves as the basis for the priority queue implementation. The full source code is available online at `https://github.com/jkopinsky/SprayList` (last visited on December 9th, 2015). In the SprayList, `insert` calls are simply forwarded to the underlying SkipList.

The delete_min operation executes a random walk, also called a *spray*, the purpose of which is to spread accesses over a certain section of the SkipList uniformly such that collisions between multiple concurrent delete_min calls become unlikely. This is achieved by starting at some initial level, walking a randomized number of steps, descending a randomized number of levels, and repeating this procedure until a node n is reached on the lowest level. If n is not deleted, it is logically deleted and returned. Otherwise, a *spray* is either reattempted, or the thread becomes a cleaner, traversing the lowest level of the SkipList and physically removing logically deleted nodes it comes across. A number of dummy nodes are added to the beginning of the list in order to counteract the algorithm's bias against initial items.

The *spray* parameters are chosen such that with high probability, one of the $O(P \log^3 P)$ first elements is returned, and that each of these elements is chosen roughly uniformly at random. The final effect is that accesses to the data structure are spread out, reducing contention and resulting in a noticeably lower number of CAS failures in comparison to strict priority queues described in Chapter 4.

The authors do not provide any statement as to the linearizability (or other concurrent correctness criteria) of the SprayList, and it is not completely clear how to define it since no sequential semantics are given.

Their benchmarks show promising results: the SprayList scales well at least up to 80 threads, and performs close (within a constant factor) to an implementation using a random remove instead of delete_min, which the authors consider as the performance ideal since contention is minimized (although it might be more fair to narrow that statement down to SkipList-based structures).

5.2 Multiqueue

Multiqueues, published in 2014 by Rihani, Sanders, and Dementiev [48], are a simple and elegant relaxed priority queues design using probabilistic techniques. The published design uses a lock-based approach in combination with any sequential priority queue to construct a scalable concurrent priority queue, but unfortunately without being able to offer any defined quality guarantees (probabilistic or otherwise). In their benchmarks, Multiqueues outperform the SprayList in both throughput (by around a factor of two) and quality.

Similarly to the k-LSM (see Section 5.3 and Chapter 6), the Multiqueue is a configurable data structure. It takes a parameter c, which controls the number of created sequential priority queues. Each thread creates c queues, adding up to cP queues in total.

In addition, each sequential queue is associated with its own lock and a cached copy of its smallest key. Insertions choose a random queue $q_i, i \in [0, cP[$, obtain the associated lock, and insert the given item, updating the cached smallest key if necessary.

Deletions choose *two* queues q_i, q_j at random and peek at their respective cached minimal keys. The queue with the smaller cached key is then locked and popped, again updating the cached key. This technique is somewhat similar to load balancing through random selection [47], but no definite quality analysis has yet been given due to the complications arising through priority queue semantics.

While the Multiqueue is inherently lock-based, it could be made lock-free by simply using lock-free queues as its backing structures.

5.3 *k*-LSM

We now finally come to the main topic of this thesis, the relaxed, linearizable, and lock-free *k*-LSM priority queue. Wimmer first presented this data structure in 2013 [61] and have improved on it continuously since, with the most recent results being published in [62]. The original *k*-LSM queue is integrated as a priority scheduler into Wimmer's *Pheet* task-scheduling system, and an open-source implementation is available at `http://pheet.org`. In this section we describe its design in detail, while Chapter 6 covers our implementation and improvements.

Taken as a black box, the *k*-LSM conforms to the interface as shown in Figure 5.1. It is a template data structure in the sense that it has a configuration parameter k, which determines how far quality guarantees may be relaxed. Given k, the `delete_min` operation may return one of the kP minimal items, where P is the number of threads.

```
template <class K, class V>
class k_lsm {
public:
    void insert(const K &key, const V &value);
    bool delete_min(V &val);
};
```

Figure 5.1: The *k*-LSM interface.

The semantics are as follows: when `insert` completes, the given key-value pair has been inserted into the queue. `delete_min` can either complete successfully, in which case it returns `true`, one of the kP minimal items is removed from the queue and its value is written into the given argument; or it can fail, in which case no item is removed from the queue and `false` is returned. Failures of `delete_min` may occur spuriously even if the queue is non-empty, but the number of such failures are low in practice.

The *k*-LSM is a composite data structure consisting of a global component called the SLSM, and a thread-local component called the DLSM. As implied by their names, both the SLSM and DLSM are based on the Log-structured Merge Tree (LSM) [43] data structure which will be elaborated upon below. Both structures may be used as standalone priority queues, but have complementing advantages and disadvantages which motivates their composition.

Throughout this section, we ignore the complexities of lock-free memory management and simply assume the existence of a garbage collector. The implementation by Wimmer uses their own lock-free memory allocator to handle items, while blocks and block arrays are allocated in a pool and reused. Memory management will be discussed in further detail in Chapter 6.

5.3.1 Log-structured Merge Trees

Log-structured Merge Trees were introduced to the database community in 1996 [43] and reinvented indepently by Wimmer based on the requirements of concurrent priority queues. LSMs are the basic building blocks of the SLSM and DLSM and hence also the k-LSM.

By itself, the LSM is not a parallel data structure, and special care must be taken when accessing shared (i.e. not exclusively owned, thread-local) LSMs. For our purposes, we imagine it to have an interface as shown in Figure 5.2. `insert` simply inserts the given item into the set, while `peek_min` returns the minimal item and `true`, or `false` if the set is empty.

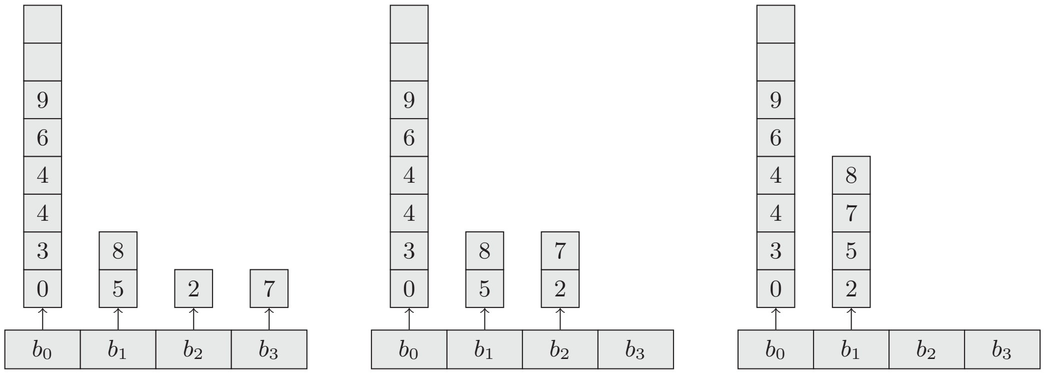
```
template <class Item>
class lsm {
public:
    void insert(const Item &item);
    bool peek_min(Item &item);
};
```

Figure 5.2: The LSM interface.

In more detail, the LSM consists of an ordered set of sorted arrays (called blocks) fulfilling a set of invariants after the completion of each operation:

- Each block B is associated with a level i such that B has a capacity of 2^i items.
- Consider two blocks, B (with associated level i and position ix within the LSM) and B' (with level i' and position ix'); if $ix < ix$ then $i > i'$.
- For each block B with level i, $2^{i-1} \leq |B| \leq 2^i$, where $|B|$ is the number of items stored in B.

Insertions initially create a singleton block B containing the given item. B is then inserted at the tail end of the LSM, possibly violating invariants temporarily. While invariants remain unsatisfied, i.e. while the LSM contains multiple blocks of the same level, B is merged with its predecessor block, replacing both B and the successor block with a new block of doubled capacity. Figure 5.3 displays an insertion example in which the new element with key 7 is inserted, triggering two block merges before completion.



(a) The initial state with item 7 inserted as a new block.

(b) After the first merge.

(c) After the second merge.

Figure 5.3: Insertion of a new element into the LSM. Note that while invariants are violated during the operation (since block capacities are not unique), they are reinstated after the final merge.

Peek operations iterate through each block in the LSM and return the minimal encountered item. Note that this operation is $O(1)$ for each block since the minimal item is located at the head of each block, and thus of logarithmic complexity for the entire LSM.

Both insertions and deletions are of amortized complexity $O(\log n)$ and are usually extremely cache-efficient since the central merge operation accesses items in sequence, and items are stored in contiguous chunks of memory.

5.3.2 Distributed LSM

The DLSM is a concurrent priority queue with purely thread-local guarantees. It adheres to the same interface as the k-LSM as given in Figure 5.1, but with subtly altered semantics in that there is no relaxation parameter k — the DLSM's `delete_min` simply removes the thread-locally minimal item. If the local queue is empty an operation called `spy` is performed, which attempts to copy items from a randomly chosen thread's queue.

The implementation is essentially a thin wrapper on top of a thread-local LSM with some complicating factors. Items store both the key-value pair as well as an atomic flag indicating whether an item has been taken (removed) from the DLSM or not. The LSM itself stores pointers to these items. `delete_min` operations perform a `peek_min` on the underlying local LSM and mark the returned item as taken using an atomic CAS instruction. Special care is taken in order to eventually physically remove taken items from the LSM: merges skip taken items, and whenever a block is noted to be less than half-full the block is shrunk and LSM invariants are reasserted.

So far, all operations have been exclusively thread-local, and thus no attention had to be given to concurrency complications. Unfortunately, `spy` does access other thread's LSMs and we cannot ignore these issues entirely. The set of blocks in the LSM is implemented as a doubly-linked list of blocks. The set of pointers to the previous element may be accessed exclusively by the owning thread, while the set of pointers to the next element can by read by all threads (but written only by the owning thread). When a maintenance operation triggers changes to the block set, merges are performed locally and the new tail of the linked list is spliced in atomically using a CAS instruction. While other threads currently reading the local list may encounter pointers to the same item multiple items within a single `spy` operation if the linked list has been updated concurrently, this is not an issue since the item may only be taken successfully by a single thread when its flag is set atomically.

The DLSM is essentially embarassingly parallel as long as all local queues remain nonempty, and scales exceedingly well even at very high thread counts. This is to be expected, since the LSM is very efficient, and most, if not all operations are thread-local. Insertions and deletions are again of amortized logarithmic complexity and benefit from high cache locality as well as mostly thread-local operations.

However, the DLSM has one major weakness in the lack of any global quality guarantees. The next section discusses the SLSM, which is utilized in the k-LSM in order to reintroduce global guarantees.

5.3.3 Shared LSM

The Shared LSM could be considered the antipole of the previous section's DLSM: it consists of a single, global LSM whereas the DLSM has a local LSM for each thread; it is relaxed, returning one of the k minimal items from `delete_min` whereas the DLSM is strict on a local basis; and unfortunately, it has fairly limited scalability while the DLSM is embarassingly parallel.

The SLSM's interface is once again identical to Figure 5.1. Insertion semantics are as expected, and deletions may either succeed, removing and returning one of the k minimal items within the queue; or fail without altering the queue's item set.

At a high level, the concept of its implementation is simple: it consists of a single global LSM with an associated item range containing the (approximately) k minimal items within the queue. Insertions first add the new item to a local copy of the queue and then atomically replace the global copy with the modified local version. Deletions randomly select one of the $k + 1$ minimal items and attempt to delete it. If it has not yet been taken by another thread, it is removed and the call returns successfully. Otherwise, the same procedure is repeated (possibly updating the range of minimal items) until the queue is either empty or an untaken item is found and returned.

From the previous description, it almost seems as if the SLSM were a simple data structure; however, rest assured that this is not the case. Chapter 6 covers its implementation in more detail.

5.3.4 k-LSM

Taken by themselves, both the DLSM and SLSM are interesting but not particularly practical data structures. The DLSM is fast and scalable, but cannot offer any guarantees as to the quality of returned items, while the SLSM has a significant global bottleneck.

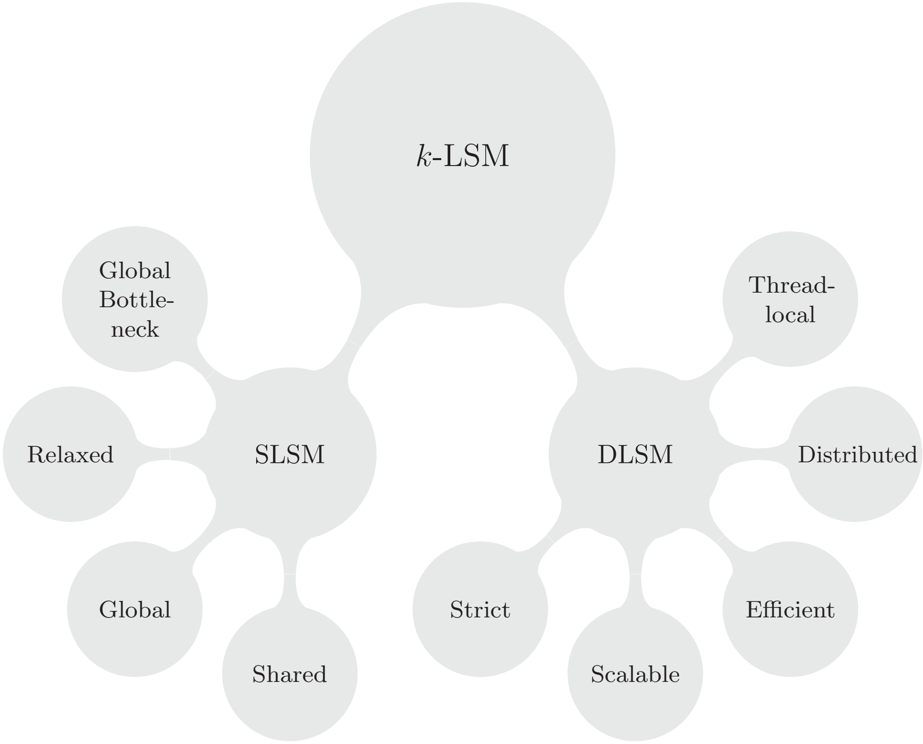


Figure 5.4: A high-level view of the k-LSM concept.

The k-LSM combines both of these designs to reduce these drawbacks. In order to be able to offer global quality guarantees, the thread-local DLSMs are limited to a capacity of k per thread, and `delete_min` returns the minimal item returned from the DLSM and SLSM. Note that we skip up to $k(P-1)$ potentially smaller items located within other thread's local DLSM and up to k items through relaxation of the SLSM. We therefore satisfy the claimed guarantees of returning one of the kP minimal items.

The remaining issue of the SLSM's scalability is ultimately caused by updates to the global LSM, which in turn mostly occur through item insertions. The k-LSM reduces the occurrence of these insertions by a factor of $O(k)$ by only inserting blocks of size $> \frac{k}{2}$ into the SLSM. This harmonizes well with the quality guarantee solution from the previous paragraph - when a local DLSM exceeds its maximal size, the largest block is simply inserted into the SLSM.

The k-LSM as presented in [62] exhibits varying behavior according to the parameter k; at low values ($k \in \{0, 4\}$) scalability and throughput are somewhat comparable the the

Lindén and Jonsson queue. At higher relaxation values of $k \in \{256, 4096\}$, the k-LSM approaches linear speedup up to high thread counts.

CHAPTER 6

Implementation of the k-LSM Priority Queue

Contrary to the previous chapter in which the concept of the k-LSM was briefly described, this chapter will cover the implementation details of the new standalone k-LSM.

But first, a rationale — why is a standalone k-LSM implementation desirable? There are several reasons:

- Comparability. The original k-LSM implementation within the task-scheduling framework Pheet [61, 62] (simply referred to as the *Pheet k-LSM* within this thesis) has been demonstrated to perform very well; but how well exactly when compared to other state of the art queues is not easy to determine since the Pheet k-LSM implementation is tied tightly to the Pheet framework. A standalone k-LSM will allow other for easy comparability against other queues and answer how its performance compares to the state of the art.

- Use in practice. A standalone k-LSM can easily be integrated into applications for use in practice.

- Maintainability. The Pheet k-LSM implementation is focused on maximal performance and makes tradeoffs that pose challenges to maintainability. While the standalone k-LSM also aims to achieve high performance, the reimplementation also explicitly values classical software engineering concepts such as composability, readability, and separation of concerns [27].

- Scientific reproducability. The standalone k-LSM verifies performance claims by Wimmer.

- Further insight and improvements. Finally, a reimplementation by a fresh set of eyes can create new insight into the design and might result in improvements to further increase performance over the original Pheet k-LSM.

The standalone k-LSM (referred to as the *k-LSM* in the remaining chapter) is implemented in C++11 using a minimal set of dependencies on other libraries. The included benchmark application requires the *hwloc* library for hardware-independent thread pinning, while other priority queues included for benchmarking purposes require the GNU Scientific Library (GSL). The k-LSM implementation itself has no external dependencies and uses standard C++11 atomics, and is therefore highly portable.

Unlike the Pheet k-LSM, our implementation does not guarantee locality, i.e., it may occur that a thread skips items it has previously inserted itself. While locality may in some cases be desirable in practise, it was irrelevant for our experiments and omitted in our implementation to lower code complexity. All benchmarks of the Pheet k-LSM also run with the locality option disabled to ensure comparability.

Full code for the k-LSM implementation is available at `https://github.com/schuay/kpqueue`. The directory structure is as follows: `doc/` contains autogenerated documentation in the Doxygen[1] format, `lib/` contains external source code for other priority queues used for benchmarking, `misc/` contains various utility scripts which handle conversion of raw benchmarking output from various formats into plots using R[2], `src/` contains code for the standalone k-LSM and various benchmarking tools, and `test/` contains the test suite.

The following sections describe the k-LSM implementation in top-down order — Section 6.1 begins with the k-LSM itself, Sections 6.2 and 6.3 descend to the component queues SLSM and DLSM, while several other components are covered in Section 6.4. The various applied methods of lock-free memory management are covered in Section 6.5.

6.1 k-LSM Internals

Figure 6.1 shows an overview of the class inter-class connections involved in the k-LSM design. The `k_lsm` itself simply holds a DLSM and SLSM instance and delegates most work to these lower-level designs. Details of the `k_lsm` are covered in this section. The `dist_lsm` (Section 6.3) consists of a linked list of `block` instances, which in turn contain several `item` instances (Section 6.4). The `shared_lsm` organizes blocks using a global `block_array`, which also contains an `block_pivot` instance (Section 6.2). Memory management is handled by several classes, covered further in Section 6.5.

While the general k-LSM interface has been covered previously in Section 5.3, several details have been omitted for clarity. Figure 6.2 displays the full header file containing the

[1] `http://www.stack.nl/~dimitri/doxygen/`, last visited on December 9th, 2015.

[2] `https://www.r-project.org/`, last visited on December 9th, 2015.

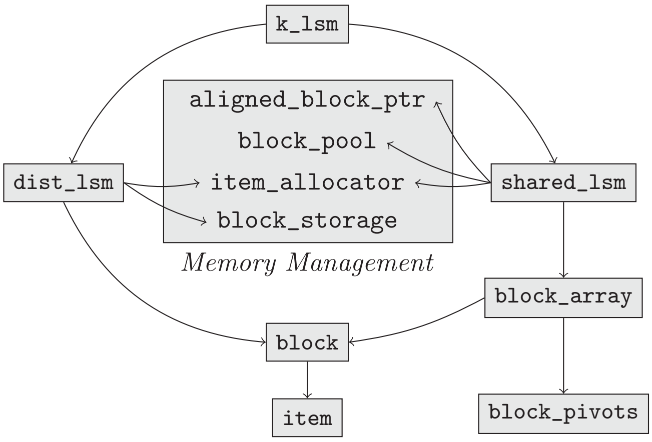


Figure 6.1: Overview of the `k_lsm` class structure.

`k_lsm` class. In this instance, implementation details such constructors and destructors, as well as the `init_thread` function, are included in order to present a complete picture of what a class implementation looks like. Future code listings will skip these details.

```
template <class K, class V, int Rlx>
class k_lsm {
public:
    k_lsm();
    virtual ~k_lsm() { }

    void insert(const K &key);
    void insert(const K &key,
                const V &val);

    bool delete_min(V &val);

    void init_thread(const size_t) const { }
    constexpr static bool supports_concurrency() { return true; }

private:
    dist_lsm<K, V, Rlx>   m_dist;
    shared_lsm<K, V, Rlx> m_shared;
};

#include "k_lsm_inl.h"
```

Figure 6.2: The *k*-LSM header.

The `k_lsm` class is a template class parameterized by `K` and `V`, respectively the key- and value types, as well as the relaxation parameter `Rlx`, which corresponds to the k in k-LSM.

Code for template class must be exclusively located in header files, but it is possibly to preserve readability using a pattern of inlined headers: the main header includes a class declaration, while an inline header contains the class definition and is itself included in the standard header. The `init_thread` function is provided for queues which require per-thread initialization.

Trivialities aside, implementation of the `k_lsm` class is extremely simple since it relies on a composition of the SLSM and DLSM queues. Insertion simply triggers insertion into the local DLSM, passing a pointer to the global SLSM queue in case the maximal size of local blocks is exceeded.

```
template <class K, class V, int Rlx>
bool
k_lsm<K, V, Rlx>::delete_min(V &val)
{
    typename block<K, V>::peek_t
            best_dist = block<K, V>::peek_t::EMPTY(),
            best_shared = block<K, V>::peek_t::EMPTY();

    do {
        m_dist.find_min(best_dist);
        m_shared.find_min(best_shared);

        if (!best_dist.empty() && !best_shared.empty()) {
            if (best_dist.m_key < best_shared.m_key) {
                return best_dist.take(val);
            } else {
                return best_shared.take(val);
            }
        }

        if (!best_dist.empty() /* and best_shared is empty */) {
            return best_dist.take(val);
        }

        if (!best_shared.empty() /* and best_dist is empty */) {
            return best_shared.take(val);
        }
    } while (m_dist.spy() > 0);

    return false;
}
```

Figure 6.3: The `k_lsm::delete_min` implementation.

Deletions (Figure 6.3) call `find_min` on both the SLSM and DLSM queues and return the lesser of both items in the standard case. In case both queues are empty, the local

DLSM attempts to copy items from another thread by triggering a call to `spy`, and finally retrying deletion.

These implementations seem simple; but that is because details are delegated to responsible classes. The best example in this case is the act of item removal — how can we ensure that an item is taken only by a single thread and that no items are lost? And since memory for items is reused (see Section 6.5), how can we prevent the ABA problem [39] in which memory is reused after being retrieved (i.e. the thread reads data different than had been intended)?

The answer lies within the `peek_t` class, which contains both a pointer to the item and an associated expected item version. Within `peek_t::take`, a CAS instruction is used to atomically increment the item's version, if, and only if it previously matched the given expected version. In the presence of two identical CAS calls by different threads, only one will succeed while the other must fail. The thread which has seen a CAS failure will return from from `delete_min` with a so-called spurious failure, returning `false` even though the queue might be non-empty; while the other thread will successfully return the item.

6.1.1 Linearizability, Lock-freedom & Complexity

The *k*-LSM is lock-free and linearizable, and both insertions as well as deletions have an amortized complexity of $O(\log n)$ (complexities are analyzed by ignoring artifacts such as retries caused by concurrent operations). We do not rigorously prove these statements but attempt to give a rough sketch for each:

- *Linearizability.* A successful `delete_min` is linearizable at the point when the SLSM has verified its local copy of the global array for the last time. Unsuccessful deletes have their linearization point upon failure of the CAS call on an item's version within `item::take`. `insert` is linearizable at the linearization points of `shared_lsm::insert` and `dist_lsm::insert`, which will be discussed in the following sections.

- *Lock-freedom.* Assuming the corresponding operations on the SLSM and DLSM are lock-free, `k_lsm::insert` is obviously lock-free since it merely calls `dist_lsm::insert`. To show that `delete_min` is lock-free, consider Figure 6.3. As long as either the SLSM or DLSM are non-empty, the loop body is executed exactly once. If, on the other hand, both are empty, then `spy` attempts to copy items from another local DLSM. If no items are copied, the `k_lsm::delete_min` method returns; otherwise, the loop body is re-executed. If it is, note that the local DLSM must be nonempty, as we have just copied items from another thread. There are now two possibilities: either the second loop iteration successfully finds an item and returns; or we again find both the DLSM and SLSM empty — but if that is the case, then other threads must have made progress in the meantime and deleted all items which we have spied at the end of the previous iteration. The `delete_min` operation is thus lock-free.

- *Complexity.* Assuming the corresponding operations on the SLSM and DLSM have amortized logarithmic complexity, then `k_lsm::insert` and `k_lsm::delete_min` are also in amortized $O(\log n)$. Insertions consist simply of calls to `dist_lsm::insert`. Deletions find a minimal element in both the SLSM and DLSM and take it in constant time, repeating the operation at most once (ignoring retries caused by concurrent deletes) when both components are empty.

6.2 Shared LSM Internals

The SLSM is the central, global component of the k-LSM and contains all items that have overflowed the local DLSM capacities. The SLSM batches insertions by inserting blocks rather than items, and relaxes deletions by keeping track of a so-called pivot range, which contains a subset of the k smallest items.

```
template <class K, class V, int Rlx>
class shared_lsm {
public:
    void insert(const K &key,
                const V &val);
    void insert(block<K, V> *b);

    bool delete_min(V &val);
    void find_min(typename block<K, V>::peek_t &best);

private:
    versioned_array_ptr<K, V, Rlx> m_global_array;
    thread_local_ptr<shared_lsm_local<K, V, Rlx>>
       m_local_component;
};
```

Figure 6.4: The SLSM header.

Figure 6.4 shows the header file for the `shared_lsm` class. The interface contains the standard insertion and deletion functions since the SLSM may be used as a standalone priority queue. However, it also has a batched insertion function which takes a block argument, and a `find_min` function used by the k-LSM to retrieve one of the smallest items without removing it.

The SLSM itself contains a versioned pointer (see Figure 6.5) to the global block array, which is used by all threads to determine the contents of the SLSM. Versioned pointers (also known as tagged pointers) reserve a subset of the available bits for a version number, incremented each time the global block array is modified. The pointer is versioned since block arrays are heavily reused — each thread allocates two block arrays and uses them in an alternating fashion when setting the global array pointer (see Section 6.5).

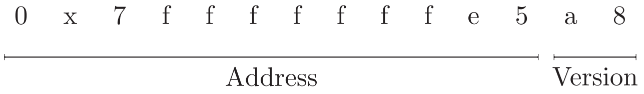


Figure 6.5: Structure of a versioned array pointer, in this case containing an 8 bit version. This particular versioned pointer contains an address to the array located at `0x7fffffffe500`, with an associated version of `0xa8`, or 168 in decimal notation.

Without versioning, this might cause issues: consider a case in which thread A retrieves the global array pointer *ptr* (which has been allocated by thread B) and then stalls. Thread B then reuses the array located at *ptr* and fills it with new contents. When thread A continues execution and successfully verifies that the global pointer appears to be unchanged, *ptr* contains different content than A expects — this is an instance of the ABA problem.

Versioning is used to solve several aspects of this issue. First, the block array itself contains an integral version number which may be compared with a local copy the check whether the array has remained unchanged. Second, a portion of the version number (more precisely, the least 11 bits) is packed into the global array pointer itself to ensure valid results of CAS operations when setting the global pointer. The class `versioned_array_ptr` handles operations on such packed pointers, and itself contains an `aligned_block_array`, which is simply a block array allocated at a 2048 byte boundary (i.e., the least 11 bits are zeroed).

The thread-local SLSM component is wrapped within the `thread_local_ptr` class (see Section 6.4), which allows lock-free retrieval of a thread-local instance of the given class. Contrary to the standard thread-local mechanisms provided by e.g. pthreads (`pthread_getspecific`) or C++ (the `thread_local` keyword), `thread_local_ptr` allows usage as a class member, and access to other threads' local instances.

The local SLSM component, `shared_lsm_local`, contains most of the vital logic, and its interface is presented in Figure 6.6.

Members include pools for memory management of several classes, which are covered in further depth in Section 6.5. Furthermore, a cached copy of the item previously returned from `peek` (the local equivalent to `shared_lsm::find_min`) is kept in order to avoid useless repeated lookups. If, for instance, the *k*-LSM chooses the item returned by the DLSM to return from `k_lsm::delete_min`, then a subsequent call to `shared_lsm::find_min` (and thus `shared_lsm_local::peek`) may simply return the cached item if the global block array has remained unchanged.

Insertions are ultimately handled by `insert_block`, shown in Figure 6.7. We skip over single-item insertions since we are mostly concerned with the SLSM in the context of the *k*-LSM (but essentially, they simply create a singleton block and insert it).

Block insertions initially ensure that the local array copy (a shallow copy of the global array, containing only pointers to the original blocks) is up to date by retrieving the

```
template <class K, class V, int Rlx>
class shared_lsm_local {
public:
    void insert(block<K, V> *b,
                versioned_array_ptr<K, V, Rlx> &global_array);

    void peek(typename block<K, V>::peek_t &best,
              versioned_array_ptr<K, V, Rlx> &global_array);

private:
    typename block<K, V>::peek_t m_cached_best;

    /** A copy of the global block array, updated regularly. */
    block_array<K, V, Rlx> m_local_array_copy;

    /** Memory management. */
    item_allocator<item<K, V>, typename item<K, V>::reuse>
       m_item_pool;
    block_pool<K, V> m_block_pool;
    aligned_block_array<K, V, Rlx> m_array_pool_odds;
    aligned_block_array<K, V, Rlx> m_array_pool_evens;
};
```

Figure 6.6: The header of the local SLSM component.

current global block array and comparing versions, then updating the copy if necessary. It is then possible to access the local copy without fear of concurrent modifications (as long as one is careful when reading blocks themselves).

A new block array copy is then constructed from the local array copy, its version is incremented, and the given block is inserted into it (details in Section 6.2.1).

Finally, the global array pointer is updated using a CAS operation. The update fails if the global array has changed since it has been last read by this thread, in which case the insertion is retried. Otherwise, the insertion is successful. The linearization point for insertions into the SLSM is upon a successful CAS in line 24.

It is important to realize that any work done within this function prior to a failed CAS is wasted work. Insertions into the block array copy trigger expensive merges and other maintenance, and we must attempt to minimize such failures as much as possible. Using higher values of k, i.e. batching more items together during insertions is one way of reducing CAS failures.

Experiments using performance counters show that on average, there are 0.75 retries per insertion when running a simple throughput benchmark with $k = 256$ on 35 threads, and 2.95 retries per insertion when running on 80 threads. As expected, absolute performance is higher with 35 threads, resulting in around 15% more completed insertions.

```
template <class K, class V, int Rlx>
void
shared_lsm_local<K, V, Rlx>::insert_block(
        block<K, V> *b,
        versioned_array_ptr<K, V, Rlx> &global_array)
{
    while (true) {
        /* Fetch a consistent copy of the global array. */

        block_array<K, V, Rlx> *observed_packed;
        version_t observed_version;
        refresh_local_array_copy(observed_packed, observed_version
           , global_array);

        /* Create a new version which we will attempt to push
           globally. */

        auto &new_blocks = /* A local pooled instance. */
        auto new_blocks_ptr = new_blocks.ptr();
        new_blocks_ptr->copy_from(&m_local_array_copy);
        new_blocks_ptr->increment_version();
        new_blocks_ptr->insert(b, &m_block_pool);

        /* Try to update the global array. */

        if (global_array.compare_exchange_strong(observed_packed,
                                                 new_blocks)) {
            break;
        }
    }
}
```

Figure 6.7: The `shared_lsm_local::insert_block` implementation.

Deletions (Figure 6.8) follow a similar pattern: the local copy of the global block array is refreshed, and a peek operation is then performed on the array copy. The process is repeated if the global array has changed since the local copy has last been updated.

As previously mentioned, if there is a cached item from a previous call to `peek` that has not yet been taken, and the global array has not changed in the meantime, then the cached item can simply be returned without any additional work.

Again, performance counters indicate that in the standard uniform throughput benchmark (see also Chapter 7), the SLSM item cache is hit very frequently. On 35 threads, 95% of all `peek` calls simply hit the cache and return without doing further work. Of the remaining calls that do real work, only 2% result in a retry.

These numbers show that the SLSM item cache is very effective; however, the high

```
template <class K, class V, int Rlx>
void
shared_lsm_local<K, V, Rlx>::peek(
        typename block<K, V>::peek_t &best,
        versioned_array_ptr<K, V, Rlx> &global_array)
{
    if (local_array_copy_is_fresh(global_array)
            && !m_cached_best.empty()
            && !m_cached_best.taken()) {
        best = m_cached_best;
        return;
    }

    block_array<K, V, Rlx> *observed_packed;
    version_t observed_version;

    do {
        refresh_local_array_copy(observed_packed,
                                 observed_version,
                                 global_array);
        best = m_cached_best = m_local_array_copy.peek();
    } while (global_array.load()->version() != observed_version);
}
```

Figure 6.8: The `shared_lsm_local::peek` implementation.

number of cache hits also imply that almost all items are actually taken from the DLSM during the uniform throughput benchmark — this might be a symptom of a problem with the benchmark itself, as we will see in the next chapter. The low frequency of retries show that peeks are relatively inexpensive, since they are rarely interrupted by a concurrent insertion.

6.2.1 Block Arrays

Block arrays are the lowest level of the SLSM and represent the actual LSM. The `block_array` class (Figure 6.10) contains an array of blocks conforming to the LSM invariants at the boundaries of functions (see Section 5.3.1).

The LSM is represented through member variables holding the block pointer array together with its size. Additionally, block arrays contain pivot ranges (storing ranges within the blocks which contain a subset of the k smallest items), a version (to avoid the ABA problem), and an efficient random number generator.

An example block array is shown in Figure 6.9. It contains four blocks ($b_{0\ldots3}$), each with an associated pivot range ($p_{0\ldots3}$) containing the 7 minimal items within the block array. Some items have been deleted, and as shown it is possible for these items to be either

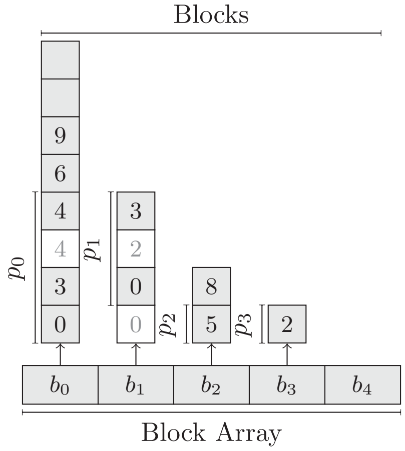


Figure 6.9: Schematic block array structure containing blocks b_i and pivot ranges p_i, $i \in [1, 4]$. Items shown in white have been deleted.

within or before a pivot range (never after). Deleted items within a pivot range cause so-called pivot fragmentation.

```
template <class K, class V, int Rlx>
class block_array {
public:
    /** May only be called when this block is not visible to other
        threads. */
    void insert(block<K, V> *block,
                block_pool<K, V> *pool);

    /** Callable from other threads. */
    typename block<K, V>::peek_t peek();

private:
    block<K, V> *m_blocks[MAX_BLOCKS];
    size_t m_size;

    block_pivots<K, V, Rlx, MAX_BLOCKS> m_pivots;

    std::atomic<version_t> m_version;

    xorshf96 m_gen;
};
```

Figure 6.10: The block array header.

During insertions, a block is first inserted into the block array such that it remains ordered by capacity. Then, while the block array contains multiple blocks of the same capacity, it is merged with its predecessor block until all LSM invariants are satisfied. The block array is then compacted: blocks which are less than half-filled are shrunk (possibly triggering other merges), and empty blocks are removed.

Finally, pivot ranges are updated (if possible) or recalculated (if necessary). Most pivot range modifications are incremental updates — when a range is less than half-filled (i.e., range size $< \frac{k}{2}$), the range is grown from its current state. On the other hand, if new range exceeds its upper bounds (range size $> k$), a completely new pivot range is determined.

Peeks, shown in Figure 6.11, use the pivot range to determine a random item within the k least items in the block array. This is done as follows: for each block within the block array, pivots store a range of items guaranteed to be within the k least items of the array. An item within this range is picked at random, and returned if it has not yet been taken; otherwise, pivot range maintenance is performed and the operation is repeated.

As it turns out, representing pivots as contiguous ranges within each block might not be sufficient to maintain maximal performance. Experiments show that for every successful peek, 3.5 failed peeks must be performed, i.e. every successful peek iterates through the loop almost 5 times on average.

This effect is caused by fragmentation of the pivot range, which has detrimental effects both on item selection (through random selection of a taken item) and relaxation (since the pivot range actually contains fewer elements than its size, calculated through pivot range boundaries, implies). In Figure 6.9, both p_0 and p_1 are fragmented, i.e., they contain deleted items. Note that the initial deleted item in b_1 is handled correctly and not included in the pivot range.

We experimented with an alternative data structure for pivot maintainance based on a balanced binary interval tree to store taken items for each block. Each time a thread encountered a taken item, it would be marked in the interval tree and excluded from further random selections. Even though operations on the tree were in $O(\log n)$ where n is the number of marked taken elements within the tree, maintenance overhead turned out to exceed any gains we made by more accurate pivot management.

6.2.2 Block Pivots

In more detail, block pivots consist of upper and lower index boundaries for each block (see Figure 6.12). The `shrink` and `grow` methods maintain the pivot range.

Care is taken to recalculate the pivot range as infrequently as possible. Recall that the pivot range is a member variable of the block array (Figure 6.10); the pivot range is thus calculated once when the new block is created upon insertion in `block_array::insert`, and then propagated to other threads when local block array copies are updated. From that point on, threads maintain their own pivot ranges — however, the next insertion will create a new global pivot range which all threads then acquire. In the ideal case, pivot ranges will thus never need to be recalculated by local threads if insertions are frequent.

```
template <class K, class V, int Rlx>
typename block<K, V>::peek_t
block_array<K, V, Rlx>::peek()
{
    while (true) {
        /* Not shown: pivot range maintenance. */
        int selected_element = m_gen() % m_pivots.count(m_size);

        size_t block_ix;
        block<K, V> *b = nullptr;
        const typename block<K,  V>::block_item *best = nullptr;
        for (block_ix = 0; block_ix < m_size; block_ix++) {
            const int elems_in_range = m_pivots.count_in(block_ix)
               ;

            if (selected_element >= elems_in_range) {
                /* Element not in this block. */
                selected_element -= elems_in_range;
                continue;
            }

            b = m_blocks[block_ix];
            best = b->peek_nth(m_pivots.nth_ix_in(selected_element
               , block_ix));

            break;
        }

        if (!best->taken()) {
            ret = *best;
            return ret;
        } else {
            /* Not shown: pivot range maintenance. */
        }
    }
}
```

Figure 6.11: The partial `block_array::peek` implementation.

The original Pheet k-LSM grows the pivot range iteratively by starting at the first block, tentatively picking the next key beyond the current pivot range as the new upper bound for keys within the range, and then recalculating the pivot range for each block. If the new pivot range size is too small, the process is repeated; if it is too large, the pivot range is reverted to its previous state, and the process is repeated on the next block.

This algorithm can become expensive and cause a large amount of repeated and wasted work. Instead of iteratively using existing keys, the standalone k-LSM uses binary search

```
template <class K, class V, int Rlx, int MaxBlocks>
class block_pivots {
public:
    size_t shrink(block<K, V> **blocks,
                  const size_t size);
    size_t grow(const int initial_range_size,
                block<K, V> **blocks,
                const size_t size);

private:
    int m_upper[MaxBlocks];
    int m_lower[MaxBlocks];
    K m_maximal_pivot;
};
```

Figure 6.12: The block pivot header.

to arrive at the desired value of the upper key bound in logarithmic time. As an additional optimization, the previously used key upper bound is stored and reused as a lower bound for the binary search in `grow` operations (in which the new upper bound must be higher than its previous value) and as upper bounds in `shrink` calls.

6.2.3 Linearizability, Lock-freedom & Complexity

The SLSM is linearizable, lock-free, and `insert` as well as `peek` have an amortized complexity of $O(\log n)$.

- *Linearizability.* Insertions into the SLSM have a single linearization point at Line 24 in Figure 6.7. Once the CAS operation has successfully set the block, it is visible globally.

  Successful `peek` operations are linearized when the local copy of the global array is verified for the last time in Line 22 of Figure 6.8, while unsuccesful peeks are linearized when `take` is called on the returned item.

- *Lock-freedom.* Insertions are lock free: they exclusively use non-blocking primitives, and retries are triggered only if another thread has made progress (Figure 6.7, Lines 12 and 24).

  It is slightly more difficult to argue lock-freedom for peeks. As in the case of insertions, non-blocking primitives are used and retries are only triggered by progress of other threads (Figure 6.8, Lines 18 and 22). But what happens if if `block_array::peek` picks a taken item? Due to readability, code handling that case has been omitted from Figure 6.11; but for each picked taken item, we iterate from the beginning of the pivot range of the current block, mark encountered taken items as taken (the pivot range size therefore decreases by one), and return the first nontaken item within the range. If such an item does not exist, we backtrack and select another random item within the random range. This process is repeated until the SLSM is determined to be empty (at most $\log n$ times), or an item is found.

- *Complexity.* Insertions essentially copy the given block ($O(1)$ per item), refresh the local array and create a copy of it in logarithmic time; finally, the given block is inserted, possibly triggering a number of merge operations. These merges are of amortized logarithmic complexity: intuitively, note that any item in an array of size 2^l has been merged at most l times, and each merge incurs constant overhead per item. Insertions are thus in amortized $O(\log n)$.

  Peeks create a copy of the global array in logarithmic time and then call `block_array::peek` on the local copy. In the simplest case, `block_array::peek` randomly selects an item, finds it in logarithmic time (since it iterates through blocks), and returns it. If the selected item is taken, it scans the block pivot range, removes observed taken items from the pivot range, and returns the first untaken item. Since each item is removed from the pivot range at most once, this overhead can be charged to its deletion cost. The pivot range also needs to be updated when its size sinks below $\frac{k}{2}$, i.e. once every $\frac{k}{2}$ deletions from the SLSM. Pivot maintenance is done through binary search over a subset of the key domain. In the case of 32-bit integers, it therefore has at most 32 iterations, and each iteration considers at most k items, for a complexity of $O(k)$. Since pivot maintenance is executed once every $O(k)$ deletions, it has amortized constant complexity per peek.

6.3 Distributed LSM Internals

The DLSM is a completely distributed priority queue without any global quality guarantees. It is, however, extremely scalable, and mostly responsible for the high performance of the k-LSM under benign situations.

Its interface, shown in Figure 6.13, consists of the standard `insert` (not shown) and `find_min`, a `spy` function, as well as a special insertion function used from the k-LSM which includes an SLSM argument. Again, a thread-local pointer contains the local part of the data structure.

The local component, called `dist_lsm_local` and shown in Figure 6.14, contains the actual LSM representation, memory management pools for blocks and items, a cached item (as in the SLSM), and a random number generator used to determine a victim during `spy` calls.

Contrary to the SLSM, the DLSM uses a linked list of blocks to represent the local LSM; this is due to historical reasons and is not expected to make any difference in achieved performance. Blocks (covered further in Section 6.4) contain `next` pointers accessible by all threads, and `prev` pointers which may only be used by the owning thread. Insertions proceed similar to the SLSM: a singleton block is created which is then inserted at the

```
template <class K, class V, int Rlx>
class dist_lsm {
public:
    void insert(const K &key,
                const V &val,
                shared_lsm<K, V, Rlx> *slsm);

    void find_min(typename block<K, V>::peek_t &best);

    int spy();

private:
    thread_local_ptr<dist_lsm_local<K, V, Rlx>> m_local;
};
```

Figure 6.13: The DLSM header.

tail of the list. Trailing blocks are then merged until there is again only a single block of each capacity within the LSM.

Block merges are performed locally, and are invisible to other threads until the newly formed LSM tail is appended atomically to the linked list of blocks by a single CAS instruction on a `next` pointer (or the list head).

In order to accomodate the k-LSM design in which DLSMs may never exceed the relaxation parameter k (`Rlx` within the code) in size, the DLSM insertion method has an `slsm` argument containing the global component of the k-LSM. Whenever the largest block of the DLSM exceeds $\frac{k}{2}$ in size at the end of an insertion, the block is removed from the DLSM and added to the SLSM instead.

An interesting facet of the current implementation is that since the DLSM and SLSM use different memory management methods for blocks, the DLSM block must be copied into an SLSM block upon insertion. However, since this copy is only performed at most once for each item, it only adds an overhead of $O(1)$.

Code for the `peek` function is shown in Figure 6.15. Similar to the SLSM, the DLSM keeps a local cache of the previously determined item with the least key, which is updated both during calls to `peek` and during insertions (in case the newly inserted item is smaller than the previously known least item). If a valid cached least item is present when `peek` is called, it is simply returned and no further work is done.

Otherwise, we iterate through each block in the LSM and inspect its least item. Since that item is located at the head of the block, access to it can be done in constant time. If the LSM is nonempty, the minimal such item is returned; alternatively, we fall back to the minimal item of the spied block. In total, there are a logarithmic number of blocks, and thus the complexity of `peek` is $O(\log n)$.

```
template <class K, class V, int Rlx>
class dist_lsm_local {
public:
    void insert(const K &key,
                const V &val,
                shared_lsm<K, V, Rlx> *slsm);

    void peek(typename block<K, V>::peek_t &best);

    int spy(class dist_lsm<K, V, Rlx> *parent);

private:
    std::atomic<block<K, V> *> m_head; /**< The largest  block. */
    block<K, V>                *m_tail; /**< The smallest block. */
    block<K, V>                *m_spied;

    block_storage<K, V, 4> m_block_storage;
    item_allocator<item<K, V>, typename item<K, V>::reuse>
       m_item_allocator;

    typename block<K, V>::peek_t m_cached_best;

    xorshf96 m_gen;
};
```

Figure 6.14: The local DLSM header.

The implementation of spy is another area in which the standalone k-LSM deviates from the Pheet k-LSM. In Pheet, spying attempts to copy the entire contents of another thread's local LSM, and adds the copied items to its own linked list of blocks.

However, this has the disadvantage of raising the frequency of DLSM overflows coupled with resulting block insertions into the SLSM, as well as causing the insertion of duplicate items into the SLSM. Semantics of the SLSM remain valid even in the presence of duplicates — recall that the LSMs store as pointers to items, and each item may only be taken atomically by a single thread; however, such duplicate items waste both space and time.

The standalone k-LSM simplifies spy to only copy the largest block from another thread's local LSM, and storing that block separately from the main local block linked list. This change makes spy more efficient since only a single block needs to be copied and no merges must be performed, and additionally prevents insertion of duplicates into the global SLSM because the spied block is never merged.

```
 1 template <class K, class V, int Rlx>
 2 void
 3 dist_lsm_local<K, V, Rlx>::peek(typename block<K, V>::peek_t &best
       )
 4 {
 5     /* Short-circuit. */
 6     if (!m_cached_best.empty() && !m_cached_best.taken()) {
 7         best = m_cached_best;
 8         return;
 9     }
10
11     for (auto i = m_head.load(std::memory_order_relaxed);
12             i != nullptr;
13             i = i->m_next.load(std::memory_order_relaxed)) {
14
15         /* Not shown: block maintenance. Empty blocks are removed,
16          * sparsely filled blocks are shrunk. */
17
18         auto candidate = i->peek();
19         if (best.empty() ||
20                 (!candidate.empty() && candidate.m_key < best.
                       m_key)) {
21             best = candidate;
22         }
23     }
24
25     if (best.empty() && m_spied != nullptr) {
26         best = m_spied->peek();
27     }
28
29     m_cached_best = best;
30 }
```

Figure 6.15: The `dist_lsm_local::peek` implementation.

6.3.1 Linearizability, Lock-freedom & Complexity

The DLSM is linearizable, lock-free, and `insert` as well as `peek` have an amortized complexity of $O(\log n)$.

- *Linearizability.* Insertions into the DLSM are linearizable when the `m_next` pointer, accessible from other threads, is set atomically. Within the context of the k-LSM, insertions into the DLSM may also overflow the data structure, in which case the operation is linearized at the linearization point of `shared_lsm::insert`.

  Peeks are linearized when `best` is set for the last time (this could occur in Figure 6.15, Lines 7, 21, or 26).

- *Lock-freedom.* Both insertions and deletions use only non-blocking primitives and execute at most $\lceil \log n \rceil$ loop iterations for block merges and are thus lock-free.

- *Complexity.* An insertion executes at most a logarithmic number of merges. As in the SLSM, this incurs an amortized complexity of $O(\log n)$ per item, since in a size n LSM each item has been merged at most $\log n$ times and each merge uses constant overhead per item.

  In the standard case, a peek operation inspects each of the $\log n$ blocks once, and determines its minimal item in constant time for a total complexity of $O(\log n)$. Additionally, whenever a block is found to be less than half-full, it is shrunk and possibly merged with its successor block. Assume this occurs for a block of capacity c; then there must have been at least $\frac{c}{2}$ delete operations from the current block previously. The shrink-merge would incur an overhead of at most $\frac{c}{2}$ (shrinking) plus c (merging), for an amortized total of $O(1)$ per item.

6.4 Component Internals

This section covers the implementation of so-called components, basic building blocks which are used throughout the k-LSM. The most essential of these are blocks and items, the base containers used to store all items within the k-LSM; and thread-local pointers, which implement local storage while still granting access to other threads' data on demand.

6.4.1 Thread-Local Pointers

Thread-local memory is a core requirement of the k-LSM: the DLSM is inherently distributed and has one separate LSM per thread, and the SLSM also caches data such as a copy of the global block array on a per-thread basis.

There are several mechanisms and libraries which offer thread-local storage, the most popular being the C++11 `thread_local` keyword [29] and pthreads `pthread_getspecific`, `pthread_setspecific` functions [12].

We have evaluated both of these for use in the k-LSM; unfortunately, neither are suitable. `thread_local` is only usable at namespace and block scopes or for static members, but the k-LSM uses nonstatic thread-local class members. And while POSIX threads could be used to create thread-local class members (`pthread_getspecific` uses a key-value mechanism), neither pthreads nor `thread_local` support access to other threads' storage, which is required by the DLSM's `spy` function.

It therefore seemed necessary to implement a custom mechanism to handle our thread-local storage needs. The interface of the resulting class is shown in Figure 6.16, and contains functions to get the current thread's storage, other threads' storage, and determine the current thread ID as well as the current number of threads.

```
template <class T>
class thread_local_ptr {
public:
    T *get();
    T *get(const int32_t tid);

    static size_t current_thread();
    static size_t num_threads();

private:
    lockfree_vector<T> m_items;
};
```

Figure 6.16: The thread-local pointer header.

The class itself is backed on a combination of a lock-free vector which provides the actual storage space and grows as required in a lock-free manner, as well as an artificial, strictly ascending `thread_local` thread id which is set by each thread when it first accesses the `thread_local_ptr`. The `get` methods then simply resolve to vector accesses.

The lock-free vector itself is based on the simple idea of an array of exponentially growing arrays with capacities of $2^i, i \in \mathbb{N}$. The full implementation is shown in Figure 6.17. If the array containing the desired item does not yet exist, a new array is created and added to the array of arrays using a CAS instruction.

Micro-benchmarks comparing the different thread-local storage mechanisms have shown that the `thread_local` outperforms both pthreads and our custom implementation by roughly a factor of 2. However, the increased cost is dominated by actual work within the k-LSM's `insert` and `delete_min` functions, and does not contribute meaningfully to k-LSM benchmark results.

6.4.2 Blocks and Items

Blocks and items are the base containers used within the k-LSM: items store key-value pairs, blocks store items, and blocks in turn are stored by the different LSM variants — linked-list LSMs in the DLSM, and array-based LSMs in the SLSM.

Items are conceptually simple, consisting only of the key-value pair and a version number (see Figure 6.18). Even numbered versions are considered reusable and may be reclaimed at any time by the memory manager, while odd versions are in use. Both `initialize` and `take` increment the version by one and thereby both invalidate other concurrent CAS instructions and set the item's taken status.

In their most basic form (Figure 6.19), blocks are likewise simple constructs. A block contains an array of items of a certain capacity which is always equal to 2^n for some

```
template <class T>
class lockfree_vector {
public:
    T *get(const int n)
    {
        // Determines index of block containing n'th item.
        const int i = index_of(n);

        T *bucket = m_buckets[i].load(std::memory_order_relaxed);
        if (bucket == nullptr) {
            bucket = new T[1 << i];
            T *expected = nullptr;
            if (!m_buckets[i].compare_exchange_strong(expected,
                bucket)) {
                delete[] bucket;
                bucket = expected;
            }
        }

        return &bucket[n + 1 - (1 << i)];
    }

private:
    std::atomic<T *> m_buckets[bucket_count];
};
```

Figure 6.17: The lockfree vector header.

$n \in \mathbb{N}$. The array stores `block_item` instances, which consists of an expected version and a cached key together with a pointer to the actual item.

It is crucial to understand how a `block_item` interacts with the remaining data structure in order to guarantee atomic access to each item. Upon insertion, while the item and its version are only visible to a single thread, the current item version (together with the item's key) is copied into the `block_item`. From this point on, the `block_item` may be copied multiple times, but remains read-only and immutable. Thus when a thread attempts to take an item using a CAS instruction with the `block_item`'s version, it is guaranteed that `block_item.m_version` has remained unchanged from the time the item has been inserted. If any thread has taken that particular item, then any further attempts to take it must fail.

Insertions into a block simply copy the given item's key and version into a `block_item` instance which is then appended to the block item array by incrementing the `m_last` field. `peek` iterates through the block beginning at index `m_first`, and returns the first untaken item (any taken item encountered causes an increment of `m_first`).

Block merges are one of the central operation of the LSM, occurring in both the SLSM and

```
template <class K, class V>
class item {
public:
    void initialize(const K &key,
                    const V &val) {
        m_version.fetch_add(1, std::memory_order_relaxed);
        m_key = key;
        m_val = val;
    }

    bool take(const version_t version,
              V &val) {
        val = m_val;

        version_t expected = version;
        return m_version.compare_exchange_strong(
                expected,
                expected + 1,
                std::memory_order_relaxed);
    }

private:
    std::atomic<version_t> m_version;
    K m_key;
    V m_val;
};
```

Figure 6.18: The item header.

DLSM as part of most insertions (during the actual insertion and also during concluding maintenance operations) and of many deletions (through maintenance when blocks are less than half full). A block merge consists of both the actual merge of both blocks' items as well as removal of any encountered taken items, called prune.

We have evaluated several different implementations of `merge` — a merge with simultaneous pruning of taken items, lazy multi-way merging at the end of an `insert` operation, and finally a simple two-phase merge-prune. Each of these methods has its own strenghts and weaknesses; the simultaneous merge & prune iterates through each item only once per `merge` call, but produces lengthier code. The lazy multi-way merge bundles several `merge` calls into one, avoiding repeated useless work, but is even more complex to code and understand. Finally, the simple two-phase merge needs to iterate through each item twice, and needlessly merges taken items before pruning them; however, it has turned out to be the most efficient method of operation.

Code for `block::merge` is shown in Figure 6.20. The first stage consists of a standard pointer-based merge algorithm, combining concise code with excellent spacial locality

```
template <class K, class V>
class block {
public:
    struct block_item {
        K m_key;
        item<K, V> *m_item;
        version_t m_version;
    };

public:
    void insert(item<K, V> *it,
                const version_t version);

    void merge(const block<K, V> *lhs,
               const block<K, V> *rhs);

    peek_t peek();

private:
    const size_t m_capacity;
    block_item *m_block_items;
    size_t m_first; /* The first untaken item index. */
    size_t m_last;  /* The first index beyond the last item. */

    static constexpr size_t MAX_SKIPPED_PRUNES = 16;
    size_t m_skipped_prunes;
};
```

Figure 6.19: The block header.

properties. When compared to a version of the algorithm using array indexing instead of pointer walks, the pointer walks provide noticeably superior performance. An important point to note is that this stage does not read the atomic version variable of each item; this is possible since item keys are cached in the `block_item`, and we can thus completely ignore any issues concerning reused items (with changed keys) and taken items.

We have also settled on a strategy that does not perform a prune on each merge. Instead, prunes are only performed once in a while, saving further expensive reads of the atomic item version and a second iteration of the entire block in which almost every item may be moved in the worst case.

6.5 Memory Management

Any lock-free data structure such as the k-LSM also requires lock-free memory management, and unlike standard sequential memory management, its lock-free counterpart is not a solved problem.

```
template <class K, class V>
void
block<K, V>::merge(const block<K, V> *lhs,
                   const size_t lhs_first,
                   const block<K, V> *rhs,
                   const size_t rhs_first)
{
    const size_t lhs_last = lhs->m_last;
    const size_t rhs_last = rhs->m_last;

    auto l = lhs->m_block_items + lhs_first;
    auto r = rhs->m_block_items + rhs_first;
    const auto lend = lhs->m_block_items + lhs_last;
    const auto rend = rhs->m_block_items + rhs_last;

    auto dst = m_block_items;
    while (l < lend && r < rend) {
        *dst++ = (l->m_key < r->m_key) ? *l++ : *r++;
    }

    while (l < lend) *dst++ = *l++;
    while (r < rend) *dst++ = *r++;

    /* Prune (code omitted). */
}
```

Figure 6.20: The `block::merge` implementation.

To illustrate, the lock-free priority queues briefly discussed in Chapters 4 and 5 use a wide variety of memory management methods. The Shavit and Lotan queue uses a dedicated garbage collection thread in combination with timestamps in order to free memory only once it cannot be visible to any threads. The Sundell and Tsigas priority queue uses the memory management method by Valois [56, 57] which does not require a separate garbage collector thread. Lindén and Jonsson use Fraser's epoch-based reclamation scheme [17], and the CBPQ makes use of the author's 'Drop the Anchor' lock-free memory management scheme [9, 10]. The SprayList does not have any robust memory management at the time of writing; it simply preallocates a set amount of memory in advance and fails in case it runs out of space.

Memory management in both the standalone k-LSM and the Pheet k-LSM are based on reuse. Once allocated, a memory chunk is never freed but reused extensively. Lock-free memory management is required for components accessed across threads, i.e. items, blocks, and block arrays. These components have differing requirements and thus each has its own memory management variant, described further in the following sections.

6.5.1 Items

The standalone k-LSM (as well as the Pheet k-LSM) use Wimmer's wait-free memory management [60, 61] for items. A detailed presentation of the scheme is omitted and is presented in detail in [60].

A general overview is as follows: each thread owns a thread-local copy of Wimmer's block-based memory manager. The memory manager has a single function called `acquire`, which returns a pointer to an available item. Internally, `acquire` operates on a list of arrays of items. An amortized complexity of $O(1)$ per allocation is guaranteed by iterating through items and returning the first unused one found as long as amortized guarantees are not violated, and simply creating a new block if they are.

As explained previously, the ABA problem is handled by linking each item with a specific integral version, with even versions representing unused items and odd versions being in use. Marking items used can thus use an FAA instruction (since it can only be allocated by its owner thread), while taking an item (i.e. marking it unused) can be done with an atomic CAS.

6.5.2 Blocks

Blocks (and block arrays in the next section) are managed based on the idea of a fixed-size pool. It is possible to bound the number of blocks of any given capacity which may be allocated at once on a single thread, and thus we may simply pre-allocate the required number of blocks.

Handling of block allocations has subtly different requirements in the SLSM and DLSM: in the DLSM, blocks have a simple lifecycle in which they are either used or unused. In the SLSM, blocks may be in short-term use locally; in long-term global use; or unused. We thus use two separate mechanisms for managing block memory.

The DLSM uses the more primitive memory manager, called `block_storage` (Figure 6.21). It consists of an array of block N-tuples, which are dynamically allocated by `get_block` whenever a block of an as-of-yet unused capacity is requested. The first unused block within the requested capacity N-tuple is then returned. In the case of the DLSM, at most four blocks of the same capacity may be used during a single operation.

Block management within the SLSM is complicated by the fact that blocks go through an additional lifecycle stage. During `shared_lsm::insert`, the new merged blocks are first created locally, during which allocated blocks are marked as in local use. If the newly created block array is published successfully, all blocks it contains are then marked as in global use. If, on the other hand, publishing fails, then all locally used blocks are marked as unused and the operation is restarted.

The SLSM block manager, the `block_pool`, is structured similarly to `block_storage`, except that it additionally stores a block status and version for each block (Figure 6.22). The version represents the version of the most recently published global array containing

```
template <class K, class V, int N>
class block_storage {
private:
    struct block_tuple {
        block<K, V> *xs[N];
    };

public:
    block<K, V> *get_block(const size_t i);

private:
    block_tuple m_blocks[MAX_BLOCKS];
    size_t m_size;
};
```

Figure 6.21: The `block_storage` header.

the corresponding block and is set when a global array is published successfully. A block may be reusable in two circumstances: either it is marked as unused; or it is set as globally used, but there is a more recently published block of the same capacity, in which case the older block is safe to use since there may only be a single block of each capacity within an LSM. The states `BLOCK_FREE` and `BLOCK_LOCAL` may transition freely between each other, but `BLOCK_GLOBAL` is never explicitly marked unused and may thus only transition to `BLOCK_LOCAL` if a newer published block of the same capacity exists.

Blocks do not have their own versions, and are instead associated with a versioned block array in order to avoid the ABA problem.

6.5.3 Block Arrays

Block arrays are used within the SLSM to represent array-based LSMs. Memory management is simple as there may only be a single published block array at a time. Each thread has two preallocated block arrays. Which one of these is chosen for allocation depends on the version of the to-be-constructed array — one for odd versions, the other for even versions.

Like items, block arrays are associated with an integral version as previously described. The full version is included in the block array itself, while a truncated version is packed into the published block array pointer to prevent the ABA problem.

```
template <class K, class V>
class block_pool {
private:
    enum block_status {
        BLOCK_FREE,
        BLOCK_LOCAL,
        BLOCK_GLOBAL,
    };

public:
    block<K, V> *get_block(const size_t i)
    {
        int max_global_version = /* maximal version of level i */;
        for (int j = ix(i); j < ix(i + 1); j++) {
            if (m_status[j] == BLOCK_FREE
                    || (m_status[j] == BLOCK_GLOBAL
                        && m_version[j] != max_global_version)) {
                m_status[j] = BLOCK_LOCAL;
                if (m_pool[j] == nullptr) {
                    m_pool[j] = new block<K, V>(i);
                }
                return m_pool[j];
            }
        }
        return nullptr;
    }

private:
    block<K, V> *m_pool[BLOCKS_IN_POOL];
    block_status m_status[BLOCKS_IN_POOL];
    version_t    m_version[BLOCKS_IN_POOL];
};
```

Figure 6.22: The `block_pool` header.

CHAPTER 7

Results & Discussion

This chapter presents and discusses results for a variety of benchmarks which have been executed on a number of different platforms. We evaluate both the standalone and Pheet k-LSM's in various degrees of relaxation together with a selection of other representative parallel priority queues.

Section 7.1 covers the different benchmark types we used, while Section 7.2 presents the various priority queue algorithms and Section 7.3 lists our systems and methodology. Results are finally shown and discussed in Section 7.4.

7.1 Benchmarks

Initially when we started work on the k-LSM, we relied almost exlusively on a benchmark which we call the uniformly random throughput benchmark, in which the priority queue is prefilled with a certain number of elements, and each thread then performs an equal mixture of insertion/deletion operations. The operation type is chosen uniformly at random with a probability of 50%/50% insertions/deletions, and key values for insertions are likewise chosen uniformly at random within the range of 32-bit integers. The benchmark is run for a certain amount of time, and the number of performed operations per second is then reported as the resulting throughput.

This type of benchmark is widely popular within priority queue research and has been used as the basis for performance evaluations in most publications, including [3, 8, 28, 35, 50, 54, 62]. Its popularity is understandable: the uniform random throughput benchmark is easy to understand as well as to implement, and it allows for some form of consistency and comparability between different publications.

However, within the course of this thesis, it has become clear that this benchmark has a significant drawback, since it causes priority queues to degrade to quasi-LIFO queue performance over time and may severely distort obtained results. This is caused by the fact that inserted keys are chosen uniformly at random — over time, as lower keys are removed from the priority queue its contents become biased towards higher keys. Newly inserted keys have a high chance of being within the lowest keys of the queue and quickly becoming a candidate for removal by `delete_min`. Especially with queues having relaxed semantics such as the SprayList and the k-LSM, this can lead to a situation in which the behavior of a priority queue very closely approximates a relaxed LIFO queue.

In order to get a better picture of a queue's overall performance, we have introduced parameters to obtain variations on the random throughput benchmark. The goal of these parameters was to preserve the aspect of the random throughput benchmark which caused the data structure to remain at a more or less constant size, allowing a benchmark to run without resulting in either an empty or an ever-growing queue.

The first introduced parameter is called the *workload*; a balanced workload performs a roughly equal amount of insertions and deletions on each thread, while a split workload performs only insertions on half of all threads, and only deletions on the other half. The second parameter concerns *key generation*, and we have experimented both with uniform key generation (uniformly at random within the range of 32-bit integers), and ascending key generation (uniformly at random within a smaller range that grows over time, i.e. selection from $[0, U] + t$), as well as descending generation.

Finally, the quality of the produced output is evaluated by using a variant of the throughput benchmark. Quality in this sense compares the output sequence of the measured priority queue against a strict sequential queue. Each insertion and deletion is tagged with a timestamp, which is then used to reconstruct a global, approximate linear sequence of insertions and deletions. This sequence is then replayed using a strict sequential queue, and ranks of each deleted item within the queue at the time of deletion are then recorded (with the rank of the least item being 1). Theoretically, strict queues should result in an average rank of 1, but in practice we can expect deviations of this value due to 'simultaneous' operations between threads and imprecise timestamps. However, even though we note that the average rank is not an exact value, it still provides a good feel for the quality of a queue's returned results.

The strict priority queue used within the quality benchmark is required to support an operation which deletes a specific element and return its rank. Contrary to similar benchmarks by Rihani, Sanders, and Dementiev [48] who rely on the C++ standard library's `multiset` class (resulting in $O(r)$ deletions, where r is the rank of the deleted item), we use a custom data structure based on a BST which supports the required operation in logarithmic time. This allows us to efficiently evaluate results with higher ranks than would otherwise be possible. Additionally, we minimize interference with the actual benchmark by storing the thread-local sequences temporarily and processing them offline, while Rihani, Sanders, and Dementiev perform rank determination interleaved with the actual benchmark operations.

7.2 Algorithms

In our experiments, we compare the standalone k-LSM against a selection of other priority queue algorithms, as well as the original Pheet implementation. The used algorithms are as follows:

- *GlobalLock* (`globallock`) An instance of the `std::priority_queue<T>` class provided by the C++ standard library, protected by a single global lock. This naive algorithm is included as a baseline, and serves to show the minimal acceptable performance of a concurrent priority queue.

- *Linden* (`linden`) Code for the Lindén and Jonsson priority queue [35] is provided by the authors under an open source license[1]. It is lock-free and uses Fraser's lock-free SkipList design. The aim of this implementation is to minimize contention in calls to `delete_min`. We chose 32 as the `BoundOffset` in order to optimize performance on a single processor. A `BoundOffset` of 128 performed only marginally better at high thread counts. The Lindén and Jonsson queue represents strict, lock-free and SkipList-based priority queues.

- *SprayList* (`spray`) A relaxed, lock-free concurrent priority queue based on Fraser's SkipList using random walks to spread data accesses incurred by `delete_min` calls. Code provided by Alistarh et al. is available on Github[2].

- *Multiqueues* (`multiq`) A simple, elegant recent relaxed concurrent priority queue design by Rihani, Sanders, and Dementiev. Contrary to the SprayList, the Multiqueue implementation is lock-based. Since code for the SprayList is not publicly available, we use our own reimplementation for benchmarks.

- *Pheet k-LSM* (`pheet16, pheet128, ...`) Wimmer's original implementation of the k-LSM within the Pheet task scheduling framework is used mostly to verify the behavior of the standalone reimplementation. We used a range of values of k to vary between fairly strict ($k = 16$) to fairly relaxed ($k = 4096$) behavior. Unlike the other priority queues, the Pheet k-LSM was measured using the benchmarks integrated into Pheet. The code was retrieved from their Launchpad site[3].

- *Standalone k-LSM* (`klsm16, klsm128, ...`) The standalone k-LSM reimplementation was measured by using a wide range of values for the relaxation parameter k. Both the standalone and Pheet k-LSM implementations are linearizable, lock-free, and relaxed priority queues. Code for the standalone k-LSM, as well as the entire benchmarking suite, is available on Github at `https://github.com/schuay/kpqueue`.

[1] `http://user.it.uu.se/~jonli208/priorityqueue`, last visited on December 9th, 2015.

[2] `https://github.com/jkopinsky/SprayList`, last visited on December 9th, 2015.

[3] `http://www.pheet.org`, last visited on December 9th, 2015.

Unfortunately, we were not able to benchmark all algorithms on all machines and with all parameters. The SprayList implementation has been very unstable throughout our tests and crashes when using either split workload or ascending key generation. Both the Linden queue and the SprayList also require libraries which were not available on our Solaris machine, and thus could not be compiled there. Finally, neither the Linden queue nor the SprayList support insertion and deletion of key-value pairs, therefore it was not possible to evaluate them using our quality benchmark. In these cases, results are simply omitted for the affected data structures.

7.3 Environment and Methodology

Three different machines are used to run benchmarks:

- `mars` An 80-core system consisting of 8 Intel Xeon E7-8850 processors with 10 cores each and 1 TB of RAM. The processors are clocked at 2 GHz and have 32 KB L1, 256 KB L2, and 24 MB of L3 cache per core.
- `saturn` A 48-core machine with 4 AMD Opteron 6168 processors with 12 cores each, clocked at 1.9 GHz. `saturn` has 64 KB of L1, 512 KB of L2, and 5 MB of L3 cache per core and 125 GB of RAM.
- `ceres` A 64-core SPARCv9-based machine with 4 processors of 16 cores each. Cores are clocked at 3.6 GHz and have 8-way hardware hyperthreading. `ceres` has 16 KB of L1, 128 KB of L2, and 8 MB of L3 cache per core and 1 TB of RAM.

Figure 7.1 shows the topology of one of eight cores on `mars`, generated with *hwloc's* `lstopo` tool. Topology graphs for all machines can be found in Appendix A.

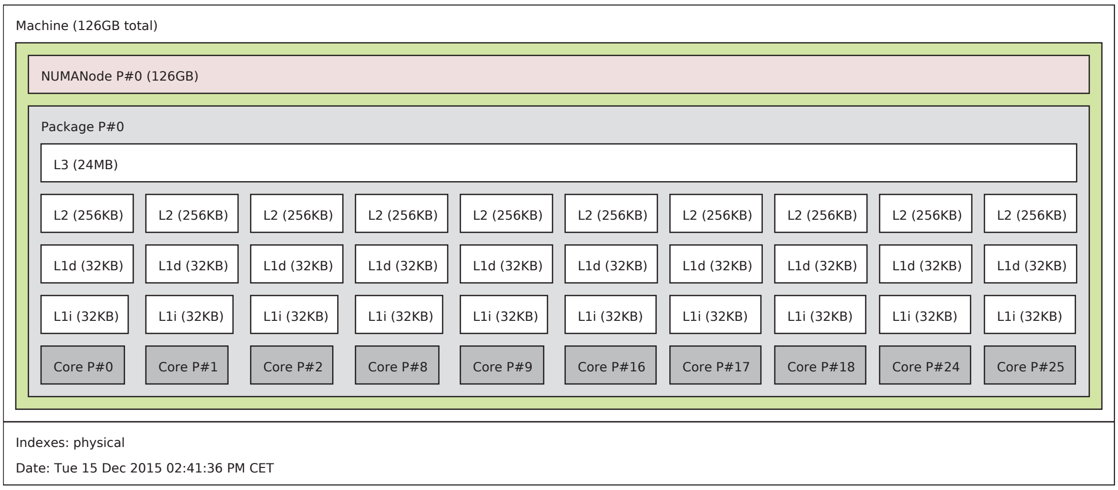


Figure 7.1: Topology of one of eight nodes on `mars`.

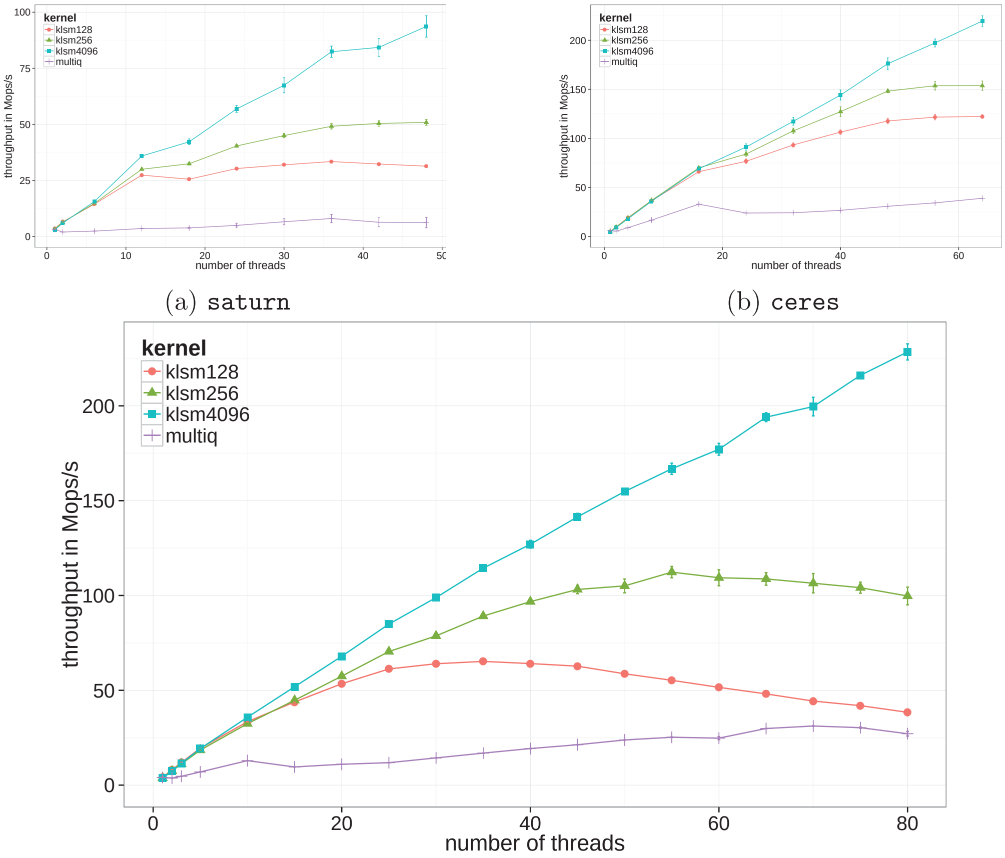
All applications are compiled using `gcc` — version `5.2.1` on `mars` and `saturn`, and version `4.8.2` on `ceres`. We use an optimization level of `-O3` and enable link-time optimizations using `-flto`.

Each benchmark is executed 10 times, and we report on the mean values and confidence intervals. Priority queues are pre-filled with 10^6 items before the benchmark is started.

7.4 Results

7.4.1 Generic throughput, uniform workload, uniform key generation

We first examine results for the random throughput benchmark with uniform workload (each thread does a roughly equal amount of insertions and deletions) and uniform key generation. Unless stated otherwise, performance measurements are reported for benchmarks on `mars`.



(a) `saturn`

(b) `ceres`

(c) `mars`

Figure 7.2: Uniform workload, uniform keys.

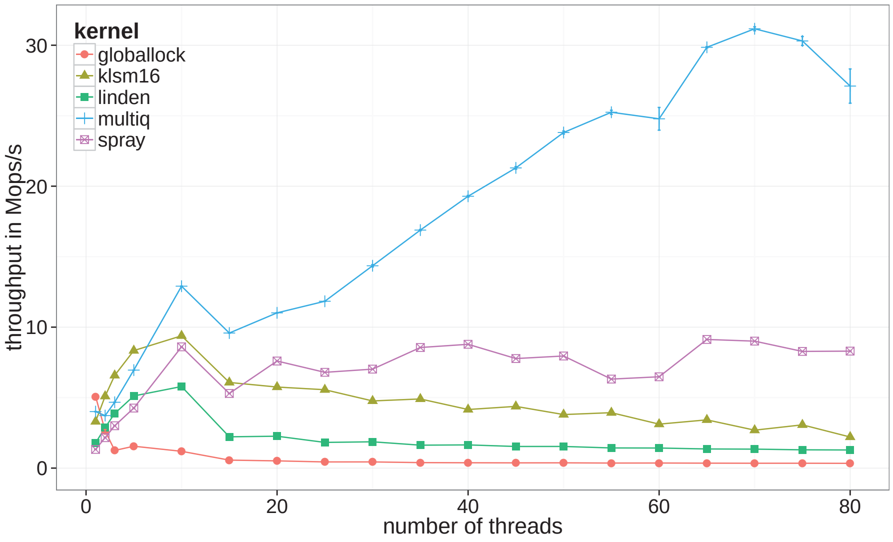


Figure 7.3: Uniform workload, uniform keys on `mars` (detail view of slower queues).

Figures 7.2 and 7.3 displays the throughput in operations per second over a range of threads. The k-LSM dominates all other priority queues at medium to high relaxation ($k \in \{128, 256, 4096\}$), reaching over 200 million operations (MOps) and scaling up to 80 threads. The k-LSM's scalability heavily depends on the value of k; at low values the k-LSM behaves similar to other concurrent priority queues (see Figure 7.3 for a detail view of slower queues). At $k = 128$, the k-LSM scales well until around 25 threads, and higher values of k improve scalability up to 50 ($k = 256$) and 80 threads ($k = 4096$).

Multiqueues (`multiq`) perform the best out of all other data structures, reaching around 30 MOps at 75 threads. Although their performance suffers once cores on more than one processor are used (this occurs at 15 threads in our figure), they eventually recover and keep scaling until almost the maximal thread count.

Likewise, the SprayList initially peaks at 10 threads, loses performance at 15 threads, but then never manages to scale beyond the performance of a single processor. As expected, the Lindén and Jonsson queue performs well while executed on a single processor, but throughput stays constant at just over 2 MOps above 10 threads. The `globallock` is the highest performer when executed sequentially, but throughput stays low at around 1 MOps at higher thread counts.

While the general trends are consistent across machines, each of the measured machines clearly has different potential for scalability. For instance, on `saturn` the k-LSM with $k = 128$ only makes minor gains when utilizing more than one processor, while the Multiqueues barely scale at all. On the other hand, on `ceres` even $k = 128$ seems to be able to scale further beyond the maximum number of cores.

Figure 7.4 compares throughput of our new standalone k-LSM to that of the Pheet

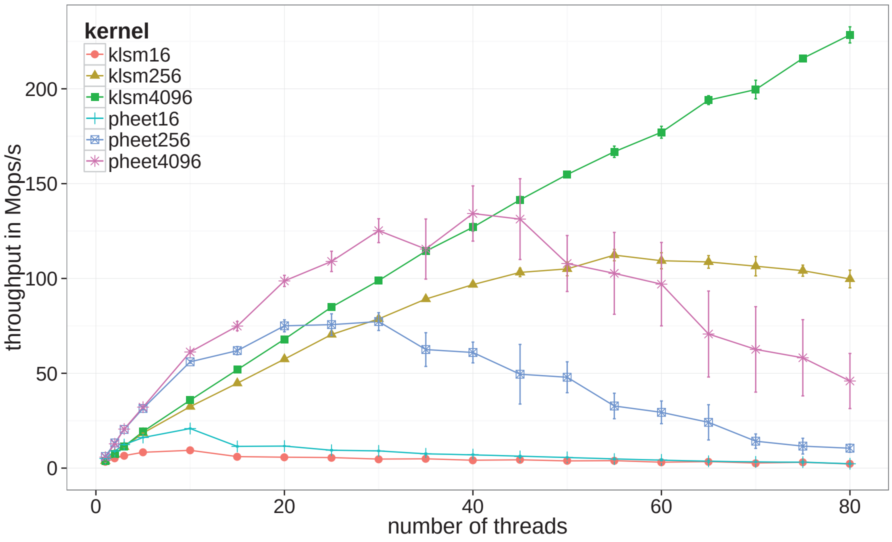


Figure 7.4: Uniform workload, uniform keys on `mars`. Comparison of the standalone k-LSM against Pheet's k-LSM.

k-LSM implementation. Interestingly, their behavior differs significantly, with the Pheet k-LSM performing stronger at lower thread counts but the standalone k-LSM scaling better at high concurrency, reaching higher absolute performance, and having more predictable throughput (i.e., less variance). It is not necessarily surprising that the two implementations behave differently, since the standalone k-LSM has various significant differences in implementation details (e.g., pivot calculation based on binary-search, completely segregated DLSM and SLSM, . . .).

	20 threads		40 threads		80 threads	
	Mean	Std. Dev.	Mean	Std. Dev	Mean	Std. Dev
`globallock`	1.9	1.0	2.9	1.7	4.8	2.9
`klsm16`	20	15	23	20	15	9
`klsm128`	33	31	55	46	430	294
`klsm256`	42	42	71	61	750	828
`klsm4096`	297	496	625	1014	10353	12667
`multiq`	984	2899	2252	7433	3787	12549

Table 7.1: Rank error for the uniform workload, uniform key generation benchmark on `mars`.

We also evaluated the quality of the results using a rank error metric, shown in Table 7.1. For each item that is removed from a priority queue, a rank of r signifies that it is the r-lowest item within the queue. Therefore, by definition, a strict priority queue such as `globallock` should have a mean rank of 1.0 and a standard deviation of 0. Unfortunately, our quality benchmark only reports approximate results — measured timestamps may differ slightly between threads, and the timestamps cannot be read exactly at the linearization point of insertions and deletions.

Rank errors for the strict `globallock` queue are reported in order to show that the magnitude of the resulting inaccuracies is very low. Results returned by the k-LSM usually far exceed the provided guarantees. Recall that the k-LSM may return any of the kP minimal elements, where P is the number of threads. While the `klsm128` may, at 20 threads, return up to a rank error of 2580, the actual mean rank error is merely 33. Compared to the k-LSM, Multiqueues have a high rank error at lower thread counts, but the error increases less rapidly at higher concurrency.

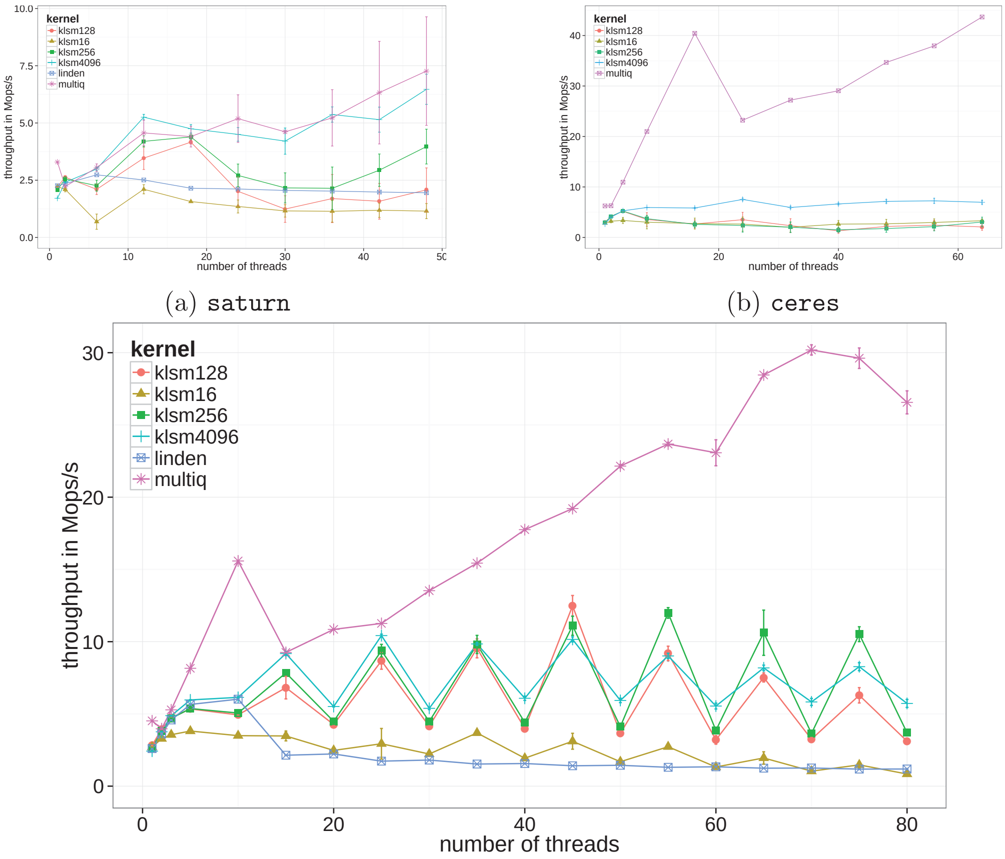
7.4.2 Generic throughput, uniform workload, ascending key generation

Figure 7.5 shows results for the throughput benchmark with a uniform workload and ascending keys. The k-LSM performs radically different to the previously discussed benchmark with uniform key generation in this scenario; instead of exceptional performance and scalability proportional to the degree of relaxation, the k-LSM's throughput never manages to exceed that of the Multiqueues. instead, throughput fluctuates between a minimum of roughly 5 MOps at processor boundaries and a maximum of around 12 MOps otherwise. While not shown here, we verified that the Pheet k-LSM behaves similarly to the standalone k-LSM.

Multiqueues, the Lindén and Jonsson queue, and the `globallock` baseline perform similarly to uniform key generation, while the SprayList crashes and could not be measured.

This result is both surprising and disappointing. Performance counters indicate that when using uniform key generation, 97% of all deleted items are taken from the DLSM — and this is obviously the reason for the k-LSM's high performance. But how is this kind of skewed balance between the SLSM and the DLSM possible? Since `k_lsm::delete_min` peeks at one item from the SLSM and one from the DLSM, wouldn't it be reasonable to expect a 50/50 proportion of items taken from the either component?

It turns out that the answer to this question is a definite "no" in the case of uniform key generation: initially, after prefill is completed, the k-LSM contains a selection of keys randomly distributed over the range of integers. However, as time goes on, lower keys are removed and the k-LSM's key range becomes skewed towards higher keys. At some point, when most low keys have been deleted from the SLSM, a stable state is reached in which the global SLSM contains mostly old, high keys while the local DLSMs contain new, low keys. Items are usually removed from the local DLSM, and thus updates to the global SLSM are infrequent; furthermore, since the best known item within the SLSM is



(a) `saturn`

(b) `ceres`

(c) `mars`

Figure 7.5: Uniform workload, ascending keys.

cached, and items are usually taken from the DLSM, the item cache is highly efficient and most calls to `shared_lsm::peek()` do no actual work.

The ascending key generation benchmark is designed to move outside of the comfort zone of the k-LSM in which the DLSM is highly utilized. Since the values of inserted keys rise over time and a min-priority queue always removes the least key, a relaxed FIFO-like behavior results. In the case of the k-LSM, items will thus usually be initially inserted into the local DLSM; then moved to the SLSM over time, and finally deleted from the SLSM once most lower items have been removed from the priority queue. Contrary to uniform key generation, emphasis is placed on the SLSM and the efficient DLSM cannot be exploited for its full potential.

Again, behavior across the different machines differs significantly. On `saturn`, no algorithm appears to scale well, with Multiqueues reaching maximal absolute performance at 48 threads with less then 10 MOps. On `ceres`, Multiqueues dominate, reaching a peak of around 40 MOps, while the k-LSM never exceeds 10 MOps. Finally, behavior on `mars` is similar, but with Multiqueues reaching a lower initial peak at one processor, and a more pronounced oscillating behavior of the k-LSM.

	20 threads		40 threads		80 threads	
	Mean	Std. Dev.	Mean	Std. Dev	Mean	Std. Dev
`globallock`	4.5	4.2	10.3	8.8	19.6	16.9
`klsm16`	7	5	11	9	20	18
`klsm128`	21	18	22	19	25	22
`klsm256`	38	33	38	33	49	37
`klsm4096`	505	474	465	436	483	457
`multiq`	101	120	202	239	419	500

Table 7.2: Rank error for the uniform workload, ascending key generation benchmark on `mars`.

With ascending key generation, the rank error of all measured k-LSM variants as well as the Multiqueues seems to be much more stable than when using uniform key generation (see Table 7.2). On the other hand, the measured rank error of the strict `globallock` queue is significantly higher. This can be explained by the nature of our ascending keygeneration, which results in more frequent key duplicates between items; and in the case of the `globallock`, duplicate keys lead to higher rank errors because of the way rank error is measured within our benchmark.

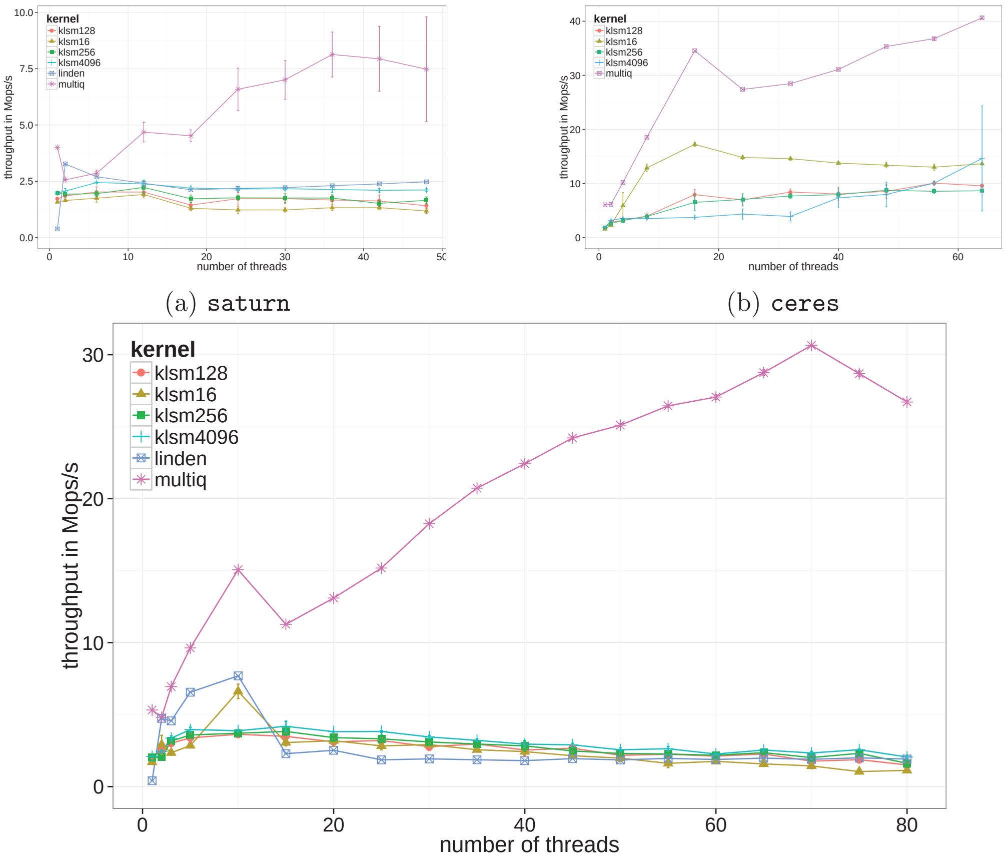
Keys are also generated in a smaller, more predictable range; for the Multiqueues, this results in queues on all threads with very similar contents, and thus rank errors are smaller.

As explained above, using ascending key generation results in a bias in the k-LSM such that more items are removed from the SLSM than the DLSM. And since the SLSM has a lower rank guarantee than the k-LSM (k instead of kP), the resulting rank errors are low.

7.4.3 Generic throughput, split workload, uniform key generation

In the case of a split workload, in which each thread is either a dedicated inserter or deleter, und uniform key generation, the benchmarked priority queues performed roughly as with uniform workload and ascending key generation (see Figure 7.6). Multiqueues perform consistently as in other scenarios, while the k-LSM does not scale at all. Surprisingly, the `klsm16` performs at least as well as more relaxed k-LSM variants; on `ceres`, it even outperforms the other variants by more than a factor of two at low thread counts.

Rank errors (Table 7.3) are generally higher for the k-LSM than both the uniform workload/uniform key generation and uniform/ascending cases, possibly because the



(a) saturn

(b) ceres

(c) mars

Figure 7.6: Split workload, uniform keys.

DLSM component is often empty (or filled with only with outdated, spied elements from another thread); thus instead of comparing two items and returning the lesser, it simply returns the SLSM item. Keys are generated uniformly, and the SLSM is therefore biased towards larger keys.

7.4.4 Generic throughput, split workload, ascending key generation

Finally, results for split workload and ascending key generation are shown in Figure 7.7. This setting combines the two variants which are especially detrimental for the k-LSM; and as expected no k-LSM variants perform well, and never exceed 5 MOps on any machine and with any thread count.

Contrary to the k-LSM, Multiqueues seem to perform best in this benchmark scenario, reaching up to 50 MOps on `ceres` and 17 MOps on `saturn`.

The biggest surprise, however, was that the strict Linden queue becomes quite competitive,

	20 threads		40 threads		80 threads	
	Mean	Std. Dev.	Mean	Std. Dev	Mean	Std. Dev
`globallock`	1.9	1.0	2.7	1.6	4.5	2.7
`klsm16`	23	10	52	20	120	55
`klsm128`	134	74	258	158	700	416
`klsm256`	235	140	821	448	1981	1112
`klsm4096`	3814	2176	11652	8924	35001	21352
`multiq`	653	2559	478	1422	2111	6002

Table 7.3: Rank error for the split workload, uniform key generation benchmark on `mars`.

outperforming all k-LSM variants and scaling up to 80 threads on `mars` and 48 threads on `saturn`. We suspect that this is because of improved spacial cache locality: inserter threads insert at the back of the skiplist, and can keep the tail of the list as well as high-level shortcut nodes within their cache, while deleter threads only access the front of the skiplist.

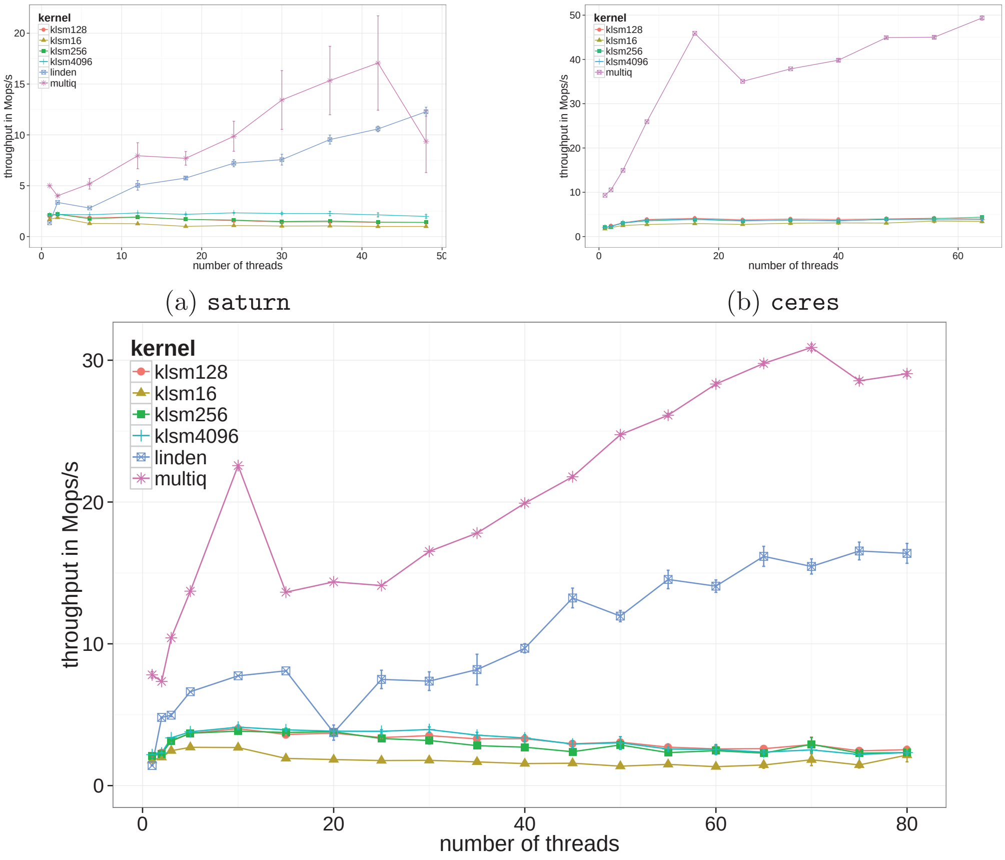
	20 threads		40 threads		80 threads	
	Mean	Std. Dev.	Mean	Std. Dev	Mean	Std. Dev
`globallock`	2.6	2.1	9.1	8.3	10.2	17.0
`klsm16`	4	3	11	9	22	24
`klsm128`	18	15	21	17	38	126
`klsm256`	33	28	50	130	92	365
`klsm4096`	428	388	488	435	1578	4186
`multiq`	133	417	317	726	1015	1946

Table 7.4: Rank error for the split workload, ascending key generation benchmark on `mars`.

As in the uniform workload, ascending key generation benchmark, rank errors (Table 7.4 are fairly low. However, the Multiqueues show very unpredictable behavior while within a single processor (≤ 10 threads), with high mean rank errors and a wide spread (2452 mean rank error and 8790 standard deviation on 10 threads).

7.4.5 Generic throughput, uniform workload, descending key generation

We have attempted to be unbiased in these benchmarks, examining both situations beneficial to the k-LSM (e.g., the uniform/uniform benchmark) as well as less well suited scenarios (split workloads and ascending key generation). Furthermore, we have



(a) saturn

(b) ceres

(c) mars

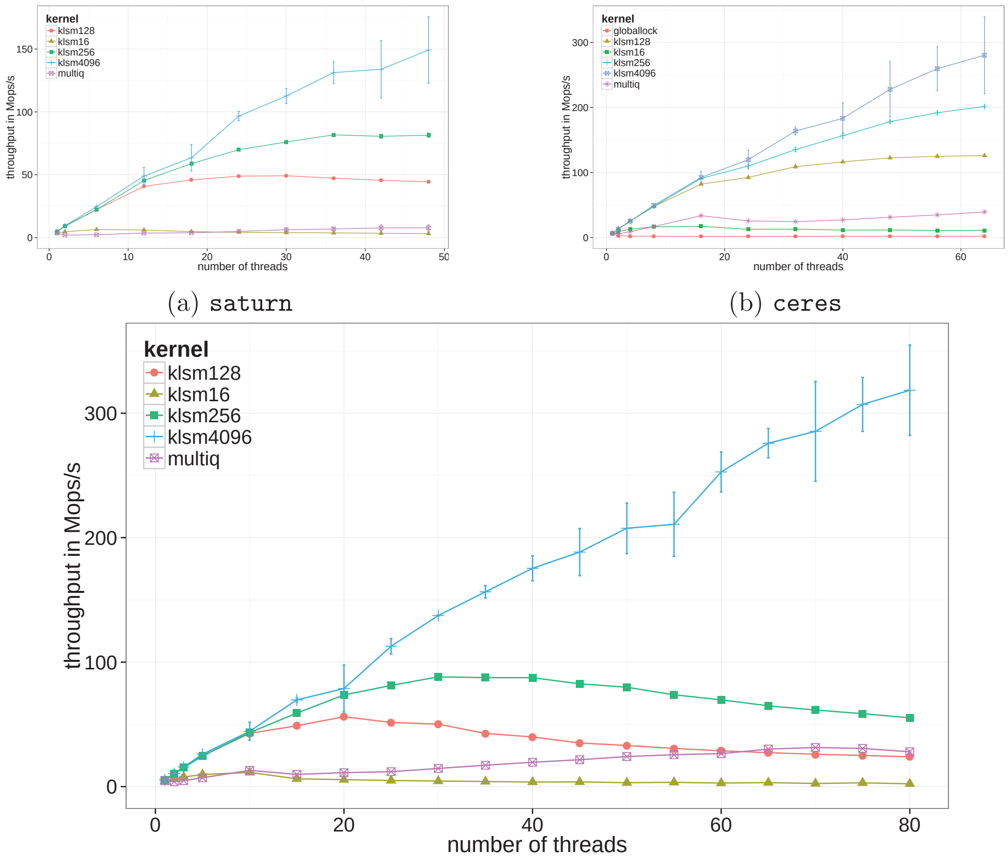
Figure 7.7: Split workload, ascending keys.

determined throughput of the k-LSM is proportional to the degree that the distributed component is utilized.

Ascending key generation (which induces FIFO-like behavior) and split workloads are on one end of the spectrum, and place high stress on the SLSM. The uniform workload, uniform key generation benchmark places high emphasis on the DLSM and results in greatly increased throughput.

In this final section, we would like to examine the most beneficial of these situations, and induce LIFO-like behavior by using descending key generation. As new keys are very likely to be within the least keys of the queue, the DLSM is very highly utilized, and contents of the SLSM are fairly static (with sufficient relaxation).

Results are displayed in Figure 7.8. As expected, peak throughput of the k-LSM reaches new heights of around 150 MOps/s on `saturn` and around 300 MOps/s on `ceres` and `mars`. Multiqueue performance remains stable and is comparable to the uniform/uniform



(a) `saturn`

(b) `ceres`

(c) `mars`

Figure 7.8: Uniform workload, descending keys.

case.

Surprisingly, lower relaxation k-LSM's actually have deteriorated scalability on `mars`, possibly since a higher amount of low-key items enter the SLSM when local DLSMs overflow.

CHAPTER 8

Conclusion

Priority queues are one of the most important abstract data structures in computer science, and much effort has been put into parallelizing them efficiently. In this thesis, we have outlined the evolution of concurrent priority queues from initial heap-based designs, through a period of increasingly efficient SkipList queues, to current research in relaxed data structures.

We also developed a standalone version of the k-LSM relaxed, lock-free priority queue and described its implementation in detail. The standalone k-LSM follows the principles described in [62], but does not rely on any external framework or non-standard library. It is engineered to be easily integrated into any application, allowing direct comparability against other relaxed priority queue designs for the first time.

The standalone k-LSM implementation also includes an extensive parameterizable benchmark suite with several other state-of-the-art queues such as the Linden queue, the Multiqueue, and the SprayList as comparisons. We not only test priority queues under the standard scenario of uniform workload and uniform key generation, but also examine combinations of split workloads (in which threads are either dedicated inserters or deleters) and ascending/descending key generation (in which the value of generated item keys increases/decreases over time).

As demonstrated previously by Wimmer et al. in [62], the k-LSM scales exceptionally well in the standard uniform workload, uniform throughput benchmark, outperforming other concurrent queues by up to a factor of 10 with sufficient relaxation. Our standalone implementation improves further upon this result, with the k-LSM ($k = 4096$) showing linear speedup up to the maximal number of threads on all test machines. This is a significant improvement over the Pheet k-LSM implementation by Wimmer, which does not scale beyond 40 threads in our measurements.

Unfortunately, this result does not carry over to the other benchmarking scenarios. Ascending key generation causes priority queues to behave in a quasi-FIFO manner, which places more emphasis on the global, slower SLSM component. Split workloads increases stress on the SLSM because the local DLSM does not have a continual flow of new items. In both cases, performance of the k-LSM drops dramatically while the Multiqueues continues to perform reliably, delivering better performance than all k-LSM variants in these altered benchmark environments.

The distribution of item keys over time is critical to the k-LSM; ascending keys result in quasi-FIFO behavior, stressing the SLSM and resulting in low throughput. Descending keys on the other hand cause quasi-LIFO behavior, and causes most deletions to take items from the efficient, local DLSM component and resulting in exceptional performance.

The uniform workload, uniform key generation benchmark, which is used by most literature on concurrent priority queues, also causes priority queues to approximate the behavior of a LIFO queue — over time, lower keys are removed from the queue, and the uniformly generated newly inserted keys have a high chance of being within the minimal keys of the queue. This may distort results, and as we have seen with the k-LSM, sometimes drastically. We would therefore strongly recommend the usage of a variety of different settings for priority queue performance evaluation.

The strict Linden priority queue performs as expected in most cases; it scales decently while executed on a single processor, but performance drops with higher concurrency. However, to our surprise, the Linden queue is actually a relevant contender in the split workload, ascending key generation benchmark, scaling up to the maximal thread count. We expect that this is due to increased locality (inserter threads traverse the list tail while deleter threads travers its head) and decreased contention because of disjoint access between inserters and deleters.

Multiqueues seem to be the most universal of the examined data structures, displaying stable performance throughout all benchmark scenarios. Their main drawback is that, by design, they do not provide any fixed quality guarantees, instead relying on randomization to provide results with acceptable rank errors. Additionally, their average degree of relaxation is fairly high.

To our knowledge, we are also the first to report on quality metrics for the k-LSM. These show that the standalone k-LSM not only satisfies the provided guarantees of returning one of the kP least elements; the provided results are usually of far better quality. At low concurrency, Multiqueues seem to be somewhat comparable to the k-LSM with $k = 4096$, but the rank error of Multiqueues grows slower than the k-LSM's with rising thread counts. However, a distinct advantage of the k-LSM is that it can provide a fixed upper bound to the rank error.

The k-LSM remains an highly interesting data structure. It is highly sensitive to key distribution over time and the mixture of performed operations, performing exceptionally well under ideal circumstances but below average in others. Thus the k-LSM might be an ideal candidate for some applications, while others might be better served by a more universal data structure such as the Multiqueue.

The k-LSM's throughput depends directly on the degree that the global SLSM component is utilized. A crucial observation however, is that the SLSM can easily be replaced by any other lock-free, relaxed priority queue which can handle batch insertions of entire blocks. There may be potential for further performance gains by replacing the SLSM with other, more efficient central data structures. Another promising idea would be to combine the DLSM with Multiqueue concept in order to create a lock-free Multiqueue. We leave these ideas to future work.

APPENDIX A

Machine Topologies

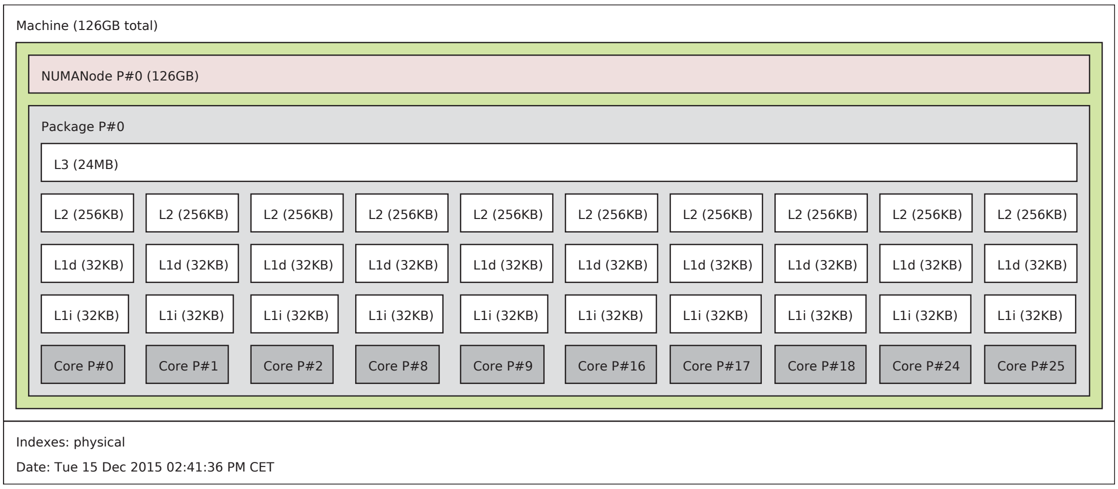


Figure A.1: Topology of one of eight nodes on `mars`. `mars` is an 80-core system consisting of 8 Intel Xeon E7-8850 processors with 10 cores each and 1 TB of RAM. The processors are clocked at 2 GHz and have 32 KB L1, 256 KB L2, and 24 MB of L3 cache per core.

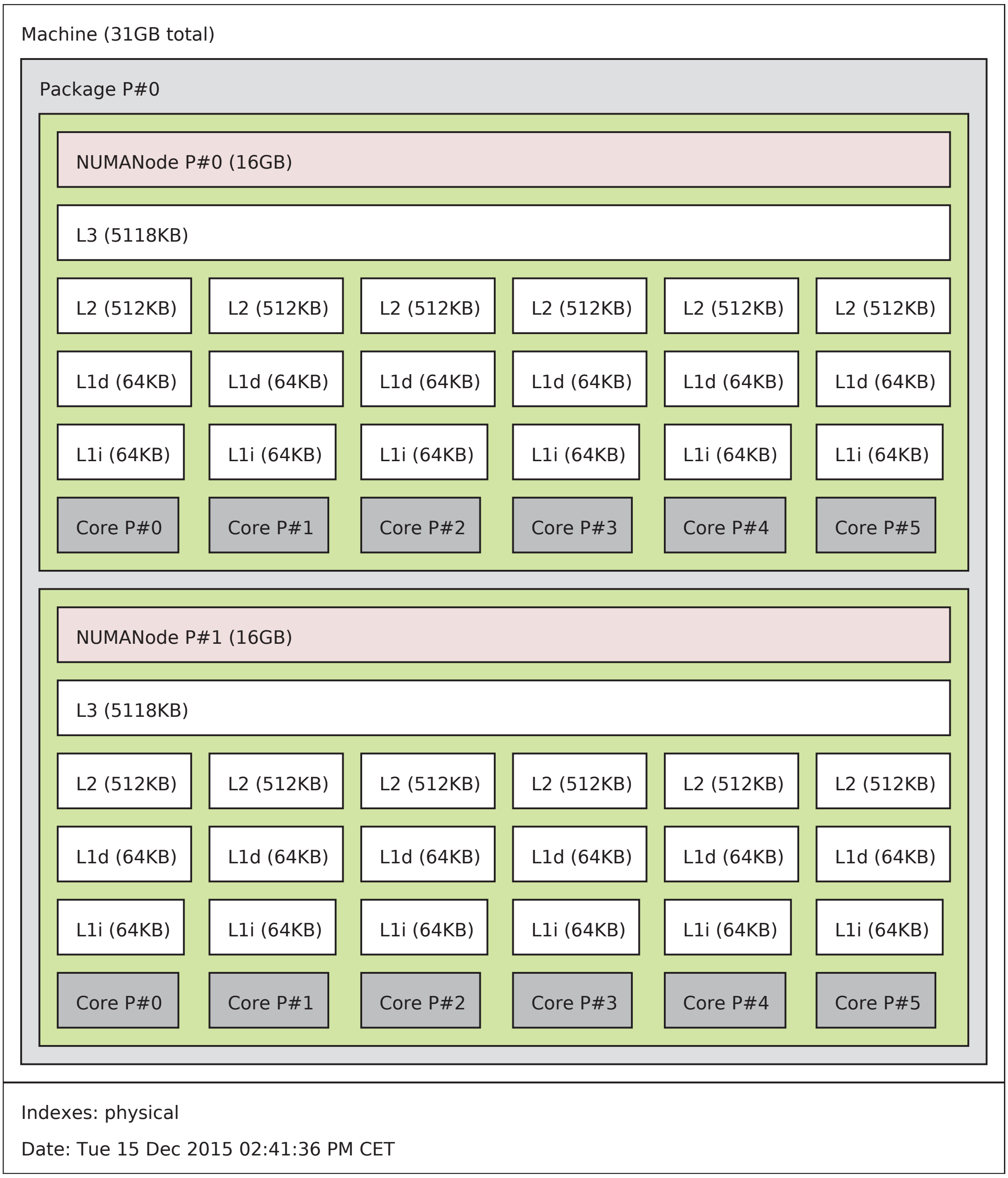


Figure A.2: Topology of one of four nodes on `saturn`. `saturn` is a 48-core machine with 4 AMD Opteron 6168 processors with 12 cores each, clocked at 1.9 GHz. `saturn` has 64 KB of L1, 512 KB of L2, and 5 MB of L3 cache per core and 125 GB of RAM.

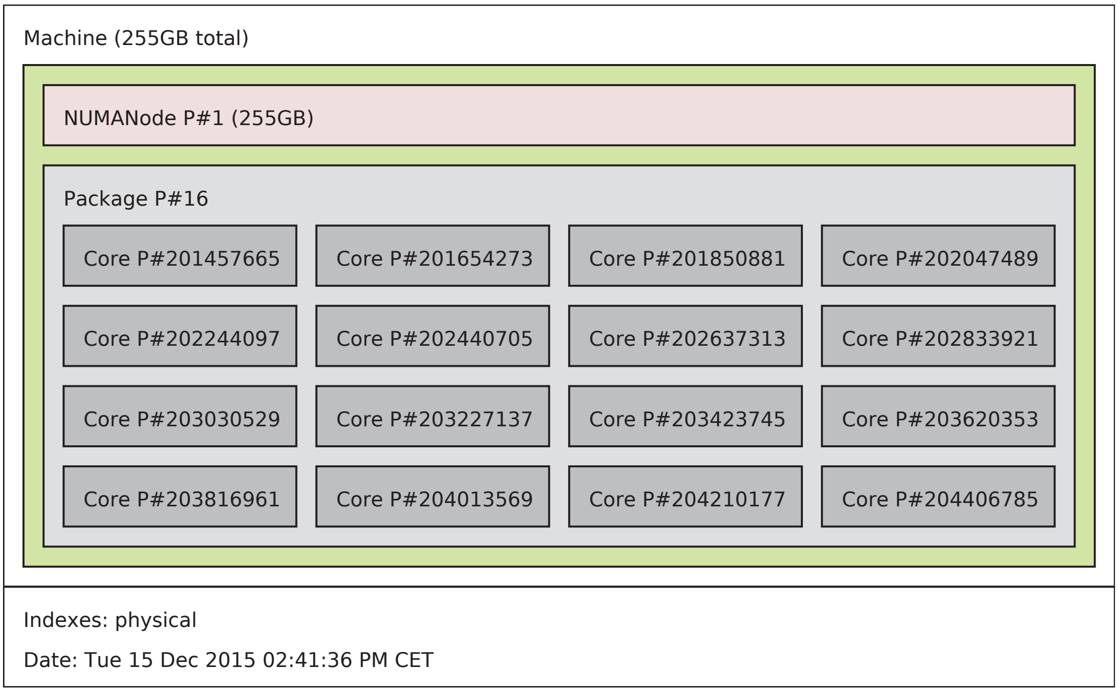


Figure A.3: Topology of one of four nodes on `ceres`. `ceres` is a 64-core SPARCv9-based machine with 4 processors of 16 cores each. Cores are clocked at 3.6 GHz and have 8-way hardware hyperthreading. `ceres` has 16 KB of L1, 128 KB of L2, and 8 MB of L3 cache per core and 1 TB of RAM.

List of Figures

List of Tables

Bibliography

[1] M AdelsonVelskii and Evgenii Mikhailovich Landis. *An algorithm for the organization of information*. Tech. rep. DTIC Document, 1963.

[2] Yehuda Afek, Guy Korland, and Eitan Yanovsky. “Quasi-linearizability: Relaxed consistency for improved concurrency”. In: *Principles of Distributed Systems*. Springer, 2010, pp. 395–410.

[3] Dan Alistarh et al. “The SprayList: A scalable relaxed priority queue”. In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. 2015, pp. 11–20.

[4] Rassul Ayani. “LR-algorithm: concurrent operations on priority queues”. In: *Proceedings of the 2nd International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 1990, pp. 22–25.

[5] Rudolf Bayer. “Symmetric binary B-trees: Data structure and maintenance algorithms”. In: *Acta informatica* 1.4 (1972), pp. 290–306.

[6] Jit Biswas and James C Browne. *Simultaneous update of priority structures*. Tech. rep. Texas Univ., Austin (USA). Dept. of Computer Sciences, 1987.

[7] Joan Boyar, Rolf Fagerberg, and Kim S Larsen. *Chromatic priority queues*. Citeseer, 1994.

[8] Anastasia Braginsky. “Multi-Threaded Coordination Methods for Constructing Non-blocking Data Structures”. PhD thesis. 2015.

[9] Anastasia Braginsky, Alex Kogan, and Erez Petrank. “Drop the anchor: lightweight memory management for non-blocking data structures”. In: *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2013, pp. 33–42.

[10] Anastasia Braginsky and Erez Petrank. “Locality-conscious lock-free linked lists”. In: *Distributed Computing and Networking*. Springer, 2011, pp. 107–118.

[11] Gerth Stølting Brodal, George Lagogiannis, and Robert E Tarjan. “Strict fibonacci heaps”. In: *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. ACM. 2012, pp. 1177–1184.

[12] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[13] Sajal K Das, Falguni Sarkar, and M Cristina Pinotti. “Distributed priority queues on hypercube architectures”. In: *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 1996, pp. 620–627.

[14] Narsingh Deo and Sushil Prasad. “Parallel heap: An optimal parallel priority queue”. In: *The Journal of Supercomputing* 6.1 (1992), pp. 87–98.

[15] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[16] James R Driscoll et al. “Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation”. In: *Communications of the ACM* 31.11 (1988), pp. 1343–1354.

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

[18] Michael L Fredman and Robert Endre Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms”. In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[19] Michael L Fredman et al. “The pairing heap: A new form of self-adjusting heap”. In: *Algorithmica* 1.1-4 (1986), pp. 111–129.

[20] Andreas Haas et al. “Local Linearizability”. In: *arXiv preprint arXiv:1502.07118* (2015).

[21] Timothy L Harris, Keir Fraser, and Ian A Pratt. “A practical multi-word compare-and-swap operation”. In: *Distributed Computing*. Springer, 2002, pp. 265–279.

[22] Thomas A Henzinger et al. “Quantitative relaxation of concurrent data structures”. In: *ACM SIGPLAN Notices*. Vol. 48. 1. ACM. 2013, pp. 317–328.

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

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

[25] 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.

[26] Qin Huang and WW Weihl. “An evaluation of concurrent priority queue algorithms”. In: *Proceedings of the 3rd International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 1991, pp. 518–525.

[27] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000.

[28] Galen C Hunt et al. “An efficient algorithm for concurrent priority queue heaps”. In: *Information Processing Letters* 60.3 (1996), pp. 151–157.

[29] ISO/IEC. *ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++*. Geneva, Switzerland, 2014.

[30] Theodore Johnson. *A Highly Concurrent Priority Queue Based on the B-link Tree.* Tech. rep. MR 1311488, 1991.

[31] Douglas W Jones. "An empirical comparison of priority-queue and event-set implementations". In: *Communications of the ACM* 29.4 (1986), pp. 300–311.

[32] Christoph M Kirsch, Michael Lippautz, and Hannes Payer. *Fast and scalable k-fifo queues.* Tech. rep. 2012-04, Department of Computer Sciences, University of Salzburg, 2012.

[33] Alex Kogan and Erez Petrank. "A methodology for creating fast wait-free data structures". In: *ACM SIGPLAN Notices.* Vol. 47. 8. ACM. 2012, pp. 141–150.

[34] Leslie Lamport. "How to make a multiprocessor computer that correctly executes multiprocess programs". In: *IEEE Transactions on Computers (TC)* 100.9 (1979), pp. 690–691.

[35] Jonatan Lindén and Bengt Jonsson. "A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention". In: *Principles of Distributed Systems.* Springer, 2013, pp. 206–220.

[36] Yujie Liu and Michael Spear. "A lock-free, array-based priority queue". In: *ACM SIGPLAN Notices* 47.8 (2012), pp. 323–324.

[37] Carlo Luchetti and M Cristina Pinotti. "Some comments on building heaps in parallel". In: *Information processing letters* 47.3 (1993), pp. 145–148.

[38] Bernard Mans. "Portable distributed priority queues with MPI". In: *Concurrency: Practice and Experience* 10.3 (1998), pp. 175–198.

[39] Maged M Michael. "ABA prevention using single-word instructions". In: *IBM Research Division, RC23089 (W0401-136), Tech. Rep* (2004).

[40] Maged M Michael and Michael L Scott. *Correction of a Memory Management Method for Lock-Free Data Structures.* Tech. rep. DTIC Document, 1995.

[41] RV Nageshwara and Vipin Kumar. "Concurrent access of priority queues". In: *IEEE Transactions on Computers (TC)* 37.12 (1988), pp. 1657–1665.

[42] Stephan Olariu and Zhaofang Wen. "Optimal parallel initialization algorithms for a class of priority queues". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2.4 (1991), pp. 423–429.

[43] Patrick O'Neil et al. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[44] Sushil K Prasad and Sagar I Sawant. "Parallel heap: A practical priority queue for fine-to-medium-grained applications on small multiprocessors". In: *Proceedings of the 7th International Symposium on Parallel and Distributed Processing (IPDPS).* IEEE. 1995, pp. 328–335.

[45] William Pugh. *Concurrent maintenance of skip lists*. Tech. rep. 1998.

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

[47] Andrea W Richa, M Mitzenmacher, and R Sitaraman. “The power of two random choices: A survey of techniques and results”. In: *Combinatorial Optimization* 9 (2001), pp. 255–304.

[48] Hamza Rihani, Peter Sanders, and Roman Dementiev. “Multiqueues: Simpler, faster, and better relaxed concurrent priority queues”. In: *arXiv preprint arXiv:1411.1209* (2014).

[49] Robert Rönngren and Rassul Ayani. “A comparative study of parallel and sequential priority queue algorithms”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 7.2 (1997), pp. 157–209.

[50] Nir Shavit and Itay Lotan. “Skiplist-based concurrent priority queues”. In: *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 2000, pp. 263–268.

[51] Nir Shavit and Asaph Zemach. “Diffracting trees”. In: *ACM Transactions on Computer Systems (TOCS)* 14.4 (1996), pp. 385–428.

[52] Daniel Dominic Sleator and Robert Endre Tarjan. “Self-adjusting binary search trees”. In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 652–686.

[53] Daniel Dominic Sleator and Robert Endre Tarjan. “Self-adjusting heaps”. In: *SIAM Journal on Computing* 15.1 (1986), pp. 52–69.

[54] Håkan Sundell and Philippas Tsigas. “Fast and lock-free concurrent priority queues for multi-thread systems”. In: *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE. 2003, 11–pp.

[55] Mikkel Thorup. “Equivalence Between Priority Queues and Sorting”. In: *Proceedings of the 43rd Symposium on Foundations of Computer Science*. FOCS ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 125–134. ISBN: 0-7695-1822-2. URL: `http://dl.acm.org/citation.cfm?id=645413.652157`.

[56] John D Valois. “Lock-free linked lists using compare-and-swap”. In: *Proceedings of the 14th annual ACM symposium on Principles of Distributed Computing (PODC)*. ACM. 1995, pp. 214–222.

[57] John David Valois. “Lock-free data structures”. PhD thesis. 1996.

[58] Jean Vuillemin. “A data structure for manipulating priority queues”. In: *Communications of the ACM* 21.4 (1978), pp. 309–315.

[59] John William Joseph Williams. *Algorithm-232-heapsort*. 1964.

[60] Martin Wimmer. “Variations on Task Scheduling for Shared Memory Systems”. PhD thesis. 2014.

[61] Martin Wimmer et al. “Data Structures for Task-based Priority Scheduling”. In: *arXiv preprint arXiv:1312.2501* (2013).

[62] Martin Wimmer et al. “The lock-free k-LSM relaxed priority queue”. In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. 2015, pp. 277–278.

[63] Yong Yan and Xiaodong Zhang. “Lock bypassing: An efficient algorithm for concurrently accessing priority heaps”. In: *Journal of Experimental Algorithmics (JEA)* 3 (1998), p. 3.