Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.

http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Effective Memory Reclamation for Lock-Free Data Structures in C++

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Manuel Pöter
Matrikelnummer 0226003

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.-Prof. Dr. Scient. Jesper Larsson Träff

Wien, 22. Jänner 2018

Manuel Pöter — Jesper Larsson Träff

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Effective Memory Reclamation for Lock-Free Data Structures in C++

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Manuel Pöter
Registration Number 0226003

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dr. Scient. Jesper Larsson Träff

Vienna, 22nd January, 2018

Manuel Pöter
Jesper Larsson Träff

Erklärung zur Verfassung der Arbeit

Manuel Pöter
Utendorfgasse 29/2/2, 1140 Wien

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, 22. Jänner 2018

Manuel Pöter

Acknowledgements

First and foremost I would like to thank my advisor Prof. Dr. Jesper Larsson Träff for his support, guidance, and endless patience.

During the course of this project, I solicited advice from many friends and colleagues, all of whom have been very generous in their help. In particular, I would like to thank Martin Wimmer, Martin Toeltsch, Sebastian Redl and Christian Schauer for all the fruitful discussions.

Last, but not least, I would like to thank my beloved wife and my parents. Without your support I would not be where I am today.

Kurzfassung

Diese Arbeit befasst sich mit der Problematik der Wiederverwendung von alloziertem Speicher im Kontext von nebenläufigen Datenstrukturen. Speicherverwaltung ist ein wesentlicher Aspekt in fast allen shared-memory concurrent Datenstrukturen und Algorithmen, bestehend aus Allokation und Freigabe bzw. Wiederverwendung von Resourcen. Speziell die Freigabe bzw. Wiederverwendung nicht mehr benötigter Objekte stellt eine große Herausforderung dar und ist daher nach wie vor ein aktives Forschungsgebiet. Diese Arbeit bietet einen ausführlichen Überblick über den aktuellen Forschungsstand und beschreibt einen Großteil der aktuellen Reclamation Schemata. Desweiteren wird ein neues Schema namens *Stamp-it* vorgestellt.

Einige der beschriebenen Reclamation Schemata wurden in C++ implementiert. Die Implementierung basiert auf einem generalisierten, abstrakten Interface, welches für den C++ Standard vorgeschlagen wurde. Es wurden die folgenden Schemata implementiert: Lock-free Reference Counting, Hazard Pointers, Quiescent State-based Reclamation, Epoch-based Reclamation, New Epoch-based Reclamation und Stamp-it. Ausführliche Erklärungen der Implementierungen werden ebenso präsentiert wie Argumente zu ihrer Korrektheit, basierend auf dem C++11 Speichermodell.

Die implementierten Schemata wurden in einer umfangreichen, experimentellen Analyse untersucht. Es wurden sowohl neue, als auch häufig genutzte Benchmarks auf vier verschiedenen shared-memory Systemen eingesetzt. Die Ergebnisse zeigen, dass Stamp-it in den meisten Fällen vergleichbare, in einigen Fällen auch bessere Performance bietet als andere Schemata.

Abstract

This thesis deals with the aspect of memory reclamation in concurrent data structures. Memory management is a critical component in almost all shared-memory, concurrent data structures and algorithms, consisting in the efficient allocation and the subsequent reclamation of shared memory resources. Especially the reclamation of no longer used memory becomes a real challenge in the face of concurrent lock-free data structures, and therefore this is still a very active research topic. This work provides an extensive overview over the current state of the art, and also presents yet another reclamation scheme called *Stamp-it*.

Some of the discussed reclamation schemes have been implemented in C++, based on a generalized, abstract interface that has been proposed for the C++ standard. The implemented schemes are: Lock-free Reference Counting, Hazard Pointers, Quiescent State-based Reclamation, Epoch-based Reclamation, New Epoch-based Reclamation and Stamp-it. A detailed discussion of these implementations is provided, including correctness arguments based on the the C++11 memory model semantics.

The implemented schemes have been analyzed in an extensive, experimental evaluation, presenting results for both new and commonly used benchmarks, on four different shared-memory systems with hardware supported thread counts ranging from 48 to 512. The results show Stamp-it to be competitive with and in many cases and aspects outperforming other schemes.

Contents

CHAPTER 1

Introduction[1]

The clock speed of processors has stagnated for several years now. Since it turns out to be difficult to further increase the performance of a single-core CPU, the tendency is to provide more and more cores per CPU to gain performance by parallel execution.

Traditionally, multi-threaded programs use mutual-exclusive locks for synchronization, but with the growing number of cores the efficiency of this approach becomes worse since the lock operations get serialized and the threads tend to spend a lot of their time waiting for some lock. Although recent results showed that in many cases blocking data structures can be practically wait-free [DG16], the need for alternative data structures that are actually lock-free or even wait-free (or at least hybrids, i.e., partially lock-/wait-free) becomes more and more important.

1.1 Progress guarantees

Mutual exclusion via locks may be easy and intuitive, however it has several drawbacks. Acquiring a lock usually requires a considerable overhead even if it is not contended. But in case of contention performance is even worse as the whole execution serializes and threads have to wait for the lock to be released. If a thread dies, stalls, blocks, or enters an infinite loop while holding a lock, the whole process will deadlock when another thread tries to acquire the lock.

A better alternative are so called *non-blocking* data structures that provide *progress guarantees* so that eventually every thread will be able to finish its operation. Non-blocking data structures are further divided into *lock-free* and *wait-free* data structures and the respective guarantees are defined as follows:

Lock-freedom: An algorithm is *lock-free* if it guarantees that infinitely often some thread finishes in a finite number of steps. Therefore, lock-freedom permits individual threads to starve but still guarantees system-wide progress.

[1]The results of this thesis have been the basis of a technical report [PT17], as well as a poster [PT18]

Wait-freedom: An algorithm is *wait-free* if it guarantees that every thread finishes its execution in a finite number of steps. Wait-freedom implies lock-freedom, so all wait-free algorithms are also lock-free.

Developing lock-free and wait-free data structures is an active research topic and a growing number of various lock-free and wait-free data structures has been published, like lists [Har01], queues [MS96], dequeues [ST05], hash maps [Mic02], binary search trees [EFRvB10], and many more.

There are also a number of data structures that cover the "middle ground", i.e., they have operations that require a lock (usually operations that change the data structure like, e.g., insertion/removal of elements in a hash map) as well as operations that are lock/wait-free (e.g., looking up a key in a hash map).

1.2 Memory reclamation

Memory management is an essential part of every data structure that supports dynamic insertion and removal of entries. It involves two different tasks: memory allocation (i.e., reserving memory for exclusive use for the caller) and memory reclamation (usually this means returning previously allocated memory to the memory manager). Efficient memory allocation is a complex topic of its own—especially when concurrency comes into play—and a large number of memory allocators has been designed and published, like Hoard [BMBW00], Google's TCMalloc [GM11] or FreeBSD's [Eva06]. But this is not the topic of this thesis. Instead, the focus is on memory reclamation, in particular on the problem of deciding when it is safe to reclaim the memory of a particular allocation, i.e., when is it guaranteed that no thread can possibly hold a reference to this allocation and how can this be determined. While this is relatively simple for single-threaded applications, it becomes a real challenge in the face of concurrent lock-free data structures.

This is due to the fact that it is usually unknown which threads may still hold a reference to a node that has already been removed from the data structure. Freeing the node while other threads still hold a reference to it could lead to a crash or undefined behavior if one of the threads later tries to access the node's memory. Therefore a reclamation scheme for lock-free data structures has to ensure that whenever a thread removes a node from the data structure, the memory occupied by this node will eventually be reclaimed (e.g., returned to the memory manager) and no other concurrently running thread will access the deallocated memory.

A number of schemes have been proposed for identifying allocations that can safely be reclaimed, i.e., how to determine whether there are any threads still holding references to these allocations. Many of these schemes are described in the course of this thesis.

With the release of the Intel Haswell architecture in 2013, hardware transactional memory (HTM) has finally found its way into modern mainstream processors [R.12]. While the current implementation still has some limitations, the rise of transactional memory opens up a lot of new possibilities—not only in the design of non-blocking data structures, but also for memory reclamation schemes like, e.g., StackTrack [AEH$^+$14] (see Section 2.11). Dragojević et al. dedicated a whole paper to the topic of how HTM could help to simplify memory management [DHLM11].

1.3 Garbage collection

Many modern languages like Java or C# provide *automatic garbage collection*[2]. In these languages the memory reclamation problem can simply be delegated to the garbage collector, which makes development of lock/wait-free data structures much easier.

There has been much work on parallel garbage collectors [ABFR11, FDSZ01] as well as garbage collectors that obtain some partial guarantee for progress [ABC$^+$08, HM92, HM01, PFPS07, PPS08, PZM$^+$10]. However, to my knowledge current literature does not offer garbage collection that can perform a full-scale lock-free garbage collection over the entire heap [Pet12].

While state-of-the-art garbage collectors can ensure the progress of the program itself, they all fail to guarantee the progress of the collector itself, causing an eventual failure of allocations and the entire program.

1.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 provides an extensive overview over the current state of the art, describing many of the currently known reclamation schemes. It also introduces my new reclamation scheme called *Stamp-it*. Chapter 3 discusses the topic of memory models, explaining why they are important in a concurrent context, and providing a brief introduction into the memory models of the x86 and ARM/POWER CPU architectures. However, the focus of this chapter lies on Section 3.3, which provides a more detailed overview over the C++11 memory model. Chapter 4 discusses an adapted version of the generalized C++ interface proposed by Robison [Rob13], and provides a detailed discussion of the implemented reclamation schemes and data structures. It also includes correctness arguments for all the implementations based on the C++11 memory models' semantics. Based on these implementations, Chapter 5 presents a large scale experimental study, comparing the performance of the implemented reclamation schemes on four different architectures in various scenarios. Finally, Chapter 6 summarizes the results of this work and draws conclusions.

1.5 Prerequisites

Aside from memory reclamation, this thesis also touches the topic of memory models and provides a detailed discussion of the C++ implementations of some reclamation schemes and data structures. Due to the broad nature of these areas, it is not possible to

[2]Strictly speaking this is not a feature of the language itself, but of the underlying runtime environment such as the Java VM and the .NET runtime. Therefore, all the other languages that are built on top of these systems also benefit from automatic garbage collection.

explain all the details that are touched in the course of this work. I provide references to important concepts that are outside the scope of this work, but in order to be able to easily follow the all explanations, the reader should have some prior knowledge in the following areas.

For Chapter 2, the reader should have a basic understanding of the concept of POSIX signals, as they are used by some of the described reclamation schemes.

To follow the explanations in Chapter 3, a basic understanding of modern CPU architectures including caches and out-of-order execution is required.

The implementations described in Chapter 4 are all done in C++. A thorough understanding of C++, including features introduced in C++11 like "move semantics" and concepts such as "*Resource Acquisition Is Initialization*" (RAII), is recommended. In order to fully understand all the details of the implementations, the reader should also have a good understanding of C++ templates, including more advanced techniques used in template-metaprogramming like "template specialization", "template aliases", "template template parameters" or the "*Curiously Recurring Template Pattern*" (CRTP) [Cop95].

Chapter 4 also provides correctness arguments for the implementations, based on the C++11 memory model. To follow these arguments, a good understanding of this memory model is essential; Section 3.3 should provide the basis for that.

CHAPTER 2

Memory Reclamation Schemes

The memory reclamation problem has been around for decades. So over the years a large number of solutions have been proposed. The following sections provide an overview over most of them, starting with lock-free reference counting (Section 2.1) from the early 90's to very recent proposals like QSense (Section 2.16) from 2016. And with Stamp-it I bring yet another solution to the table that is discussed in more detail in Section 2.17.

2.1 Lock-Free Reference Counting

Reference counting is a well known concept that has been used for decades. The first reclamation scheme for lock-free data structures based on reference counting was presented by John D. Valois in [Val95]. The original proposal contained race conditions that were discovered and corrected by Maged M. Michael and Michael L. Scott [MS95].

For this reclamation scheme each node is equipped with an integer field that is used to track the number of references to the node. Each thread is responsible for updating this reference counter accordingly—incrementing it for each new reference, decrementing it for every dropped reference. The increment is implemented using a simple atomic fetch-and-add operation. The decrement, however, is more complicated and has to be implemented using an atomic *compare-and-swap* (CAS); the reason for this is explained later. When the reference counter drops to zero there are no more references to this node and it can therefore be reclaimed. Like most reference count systems, this method is usable only with acyclic structures, since circular structures are vulnerable to memory leakage. For example two nodes that build a cycle (i.e., that reference each other) prevent the reference counter from dropping to zero. When the last thread releases its references to any of the two nodes the reference counter of that node is still one (due to the reference held by the other node). Therefore, the node is not reclaimed and the nodes are effectively leaked.

Although lock-free reference counting (LFRC) avoids locks it cannot guarantee an upper bound on the amount of memory consumed by removed nodes, since every thread can hold an arbitrary number of references to nodes. It has been shown by Michael [Mic04a] and Hart et al. [HMBW07] that reference counting produces a large overhead that often makes lock-free data structures perform worse than lock-based versions.

There are several other proposals for systems based on this reference counting scheme. Detlefs et al. [DMMJ01] allow changing the node's type upon reclamation but require a *double-compare-and-swap operation* (DCAS)[1], which is usually not supported by current CPUs. Another scheme proposed by Sundell [Sun05] is wait-free. Both schemes are described in more detail in the following sections.

In order to move from one node to the next (e.g., in a linked list) a thread has to perform the following steps:

1. Read the reference to the next node.
2. Increase the reference counter of the next node.
3. Reread the reference to the next node and check whether it has changed in the meantime.
    a) If it has changed, decrease the reference counter of the next node, drop the reference and start again at Step 1.
    b) Otherwise the thread has a safe reference to the next node.

In this sequence there is a race condition between Step 1 (reading the reference) and Step 2 (incrementing the reference counter). It could happen that between these two steps the node's reference counter drops to zero (due to another thread releasing its reference) and therefore the item becomes reclaimable. In Step 2 the thread would increment the reference counter of a potentially reclaimed node.

To overcome this race condition Step 3 rereads the reference to the next node to ensure that it has not changed in the meantime. In case it has changed the thread has to restart the whole procedure.

Due to the described race condition, LFRC cannot be used as a general reclamation scheme where reclaimed memory can be reused arbitrarily. However, LFRC can be used in situations where reclaimed nodes are reused in the same data structure. This is possible because the scheme expects a node's reference counter to be available indefinitely. Therefore, it is possible to update the reference count on a potentially reclaimed node without corrupting the data structure. This would not be the case if the memory was reused otherwise.

In order to reuse reclaimed nodes in the same data structure, a special free-list is used. When a thread wants to decrement the reference count it checks whether it is currently

[1]A double-compare-and-swap (DCAS) operation is a CAS operation that atomically compares and updates two not necessarily contiguous memory locations. This should not be confused with the double-width CAS (DWCAS), that atomically compares and updates two adjacent pointer-sized memory locations. In contrast to DCAS, such DWCAS operations are actually provided by some processor architectures like, e.g., `CMPXCHG16B` on Intel x86.

dropping the last reference and if that is the case tries to set the "claim bit" (usually the LSB of the reference counter) in a single atomic compare-and-swap operation. The thread that successfully sets the claim bit can then safely push the node on the free-list.

The free-list is usually implemented as a LIFO singly-linked list. The reference counting schema is also used to avoid the ABA problem[2] when trying to pop a node from the list.

Unfortunately, reference counting is prone to false sharing[3] as the reference counter is part of the node. Usually other members of a node follow immediately in memory so it is very likely that they share the same cache line. Acquiring or releasing a reference to a node always requires updates of the reference counter and therefore invalidates the whole cache line. This could be avoided by introducing the necessary padding, but at the cost of higher memory overhead.

Another reason for the high overhead is the global free-list that is shared by all threads. The fact that all threads operate on the same list (pushing reclaimed nodes, trying to pop nodes for reuse) can lead to high contention. A very simple way to reduce contention on the shared list would be to use fixed-size thread local free-lists as buffer. Both improvements, the padding as well as the thread-local free-lists, have been implemented for this work. The performance analysis is presented in Section 5.1.

Problems with circular references mark another drawback that is shared by all reference counting based schemes.

2.2 Wait-Free Reference Counting

This schema, proposed by Sundell [Sun05] in 2005, is an adaption of the Lock-Free Reference Counting scheme from Valois to support wait-free execution.

There are essentially two reasons why LFRC is not wait-free:

- LFRC allows increments of the reference count field of possibly reclaimed nodes and verifies afterwards that the pointer still points to the same node. If that is not the case the reference count is decremented and the de-reference scheme is repeated. However, the number of repeats is unbounded.
- Since LFRC does not qualify as a generic reclamation scheme reclaimed nodes are kept in a special free-list which is usually implemented as a LIFO single linked

[2]The ABA problem [IBM83] can occur when a CAS operation is used to update a data structure. Suppose a lock-free stack implemented as a singly-linked list with the values `[A,B,C]`, where `A` is on top of the stack. In order to pop an entry a thread reads the pointer to the head followed by the entry's next pointer and then uses a CAS operation to update the head with the next pointer's value. Suppose a thread loads the head (pointing to `A`) and its next pointer (pointing to `B`). Then some other thread pops `A` and `B` from the stack and then again pushes `A` resulting in `[A,C]`. When the first thread now continues, the CAS operation would expect the head to point to `A`, which is the case, and therefore happily update the head with a pointer to `B`.

[3]False sharing occurs when processors in a shared-memory system make references to different objects within the same coherence block (e.g., a cache line or page) and one of the objects gets altered. This may force the other processors to reload the whole block although it is not logically necessary as the other objects remain unchanged [BS93].

list. The *push* and *pop* operations of such a list can be implemented using a simple compare-and-swap, but care has to be taken about the ABA problem. However, the number of repeats that are required to successfully complete either operation are unbounded.

In order to make the reference counting scheme wait-free, Sundell proposed a solution in which threads announce certain operations together with a helping scheme that guarantees an upper bound on the number of steps to successfully complete the operation.

Before a thread attempts a de-reference operation it announces the location of the link it is about to read.

A concurrent operation that has changed a link is obliged to check for possible announcements before if decrements the reference count of the node the link previously pointed to. If an announcement matches the changed link, the concurrent operation should then provide the de-referencing operation with the address of a node that has a positive reference count and was recently pointed to from the link. This can be done by using the same shared variable that was used for announcing the pending de-referencing operation.

However, using the same shared variable for announcing several subsequent de-reference operations can potentially cause the ABA problem. In order to avoid this, a pool of shared variables is used to announce pending de-referencing operations. The pool continuously keeps track of which shared variables have a pending compare-and-swap operation from a concurrent helping operation, and only allows use of shared variables for new announcements that have no pending compare-and-swap attempts for answering.

In the original LFRC proposal the global free-list is implemented as a singly linked-list, therefore all threads operate on the same head node. However, this is problematic for wait-free algorithms since a successful CAS for one operation means that all the other concurrent CAS attempts will possibly fail. Thus, some operations might need to perform a potentially unbounded number of retries.

The free-list has to provide two methods: *AllocNode* to fetch a new node from the free-list and *FreeNode* to add a reclaimed node to the free-list. The key idea of this solution is to have several free-lists and to force the operations to work on different parts of them. In addition helping mechanisms are used to guarantee that each thread is making progress in the *AllocNode* operation. The *FreeNode* operation does not require any help from other threads, but also participates in the helping scheme to guarantee progress of concurrent *AllocNode* operations.

Each thread has a shared variable in which it announces the need for a free node. A global variable *helpCurrent* is used to track the current thread that requires help in allocating a node. This variable is updated in a round-robin manner once it is guaranteed that this thread has gotten help. This ensures that eventually every thread will have gotten help. For the first successful CAS attempt to remove a node from the free-list, each *AllocNode* operation has to possibly help the thread currently identified by *helpCurrent*. The *FreeNode* operation first tries to provide the to-be-freed node to the thread that currently needs help, and only if this fails the node is added to a free-list. This not only helps threads trying to allocate new nodes, but also reduces the chance of conflicting CAS attempts on the free-list.

In order to avoid conflicts with concurrent *FreeNode* operations, each thread operates on two separate free-lists. Moreover, in order to avoid conflicts with concurrent *AllocNode* operations, all concurrent *AllocNode* operations operate on the same free-list, thus always leaving one of the two free-lists free of conflicts for the corresponding *FreeNode* operation. The free-list that is used for *AllocNode* operations is also updated in a round-robin manner once it is empty.

The performance analysis presented in [Sun05] shows that the wait-free implementation performs slightly worse than the lock-free version. However, the main strength of wait-free algorithms is not in high average performance, but rather in reliable execution guarantees that could be exploited for example in real-time systems. Unfortunately, the paper does not go into any details about the quality of the bounds of this scheme.

One drawback of this scheme is that the number of threads is fixed and has to be known in advance. This can be a real limitation for applications that are designed to run on a variety of different systems. However, the property of reliable execution guarantees is mainly interesting for real-time systems, which usually have a fixed number of threads that is known in advance anyway.

2.3 Generic Lock-Free Reference Counting

The solution proposed by Detlefs et al. [DMMJ01] is another variation of reference counting that enhances on the original scheme by Valois [Val95] in that it allows arbitrary reuse of reclaimed nodes. To achieve this they assume the availability of a *double compare-and-swap* (DCAS) instruction that can atomically access two independent memory locations.

LFRC does not permit arbitrary reuse of reclaimed nodes because there is a race condition between reading the pointer to the node and incrementing the nodes's reference counter; it could happen that the node gets reclaimed before the thread can increment the counter. This is solved by expecting that reference counters are available infinitely and allowing access and modification of reference counters even for reclaimed nodes.

The proposed solution by Detlefs et al. overcomes this limitation by using the DCAS instruction to increment the reference count while atomically ensuring that the pointer to the node still exists. This allows the node to be arbitrary reused since the DCAS operation will fail and not update the reference counter in case the pointer has been changed due to reclamation of the node. However, during execution of a DCAS operation the CPU will probably still access the memory location where the reference counter used to be. So it has to be ensured that a read access to that location is still a valid operation.

The main drawback of this scheme is that to my knowledge none of the current processor architectures provide a DCAS instruction. There are, however, a few proposals on how such an operation can be emulated on systems that only support simple CAS operations like, e.g., [HFP02]. While this scheme could be implemented with such an emulation, the performance would most likely suffer.

2.4 Hazard Pointers

Hazard Pointer based reclamation (HPBR) was introduced by M. Michael in [Mic04a] in 2004. In some publications this scheme is also referred to as *safe memory reclamation* (SMR) [Fra04, Mic02, ST05].

It is primarily based on the observation that, in most algorithms for lock-free dynamic objects, a thread holds only a small number of references that may later be used without further validation. For example queues and linked lists need $K = 2$ references, while stacks require only $K = 1$. However, to the best of my knowledge there are no algorithms for general tree or graph traversal that can provide an upper bound on K.

The core idea is associating a number of single-writer multi-reader shared pointers, called *hazard pointers*, with each thread that may operate on the associated objects. Each thread has K hazard pointers (the number of required hazard pointers depends on the actual algorithm and data structure), hence, if there are p threads, we have $H = pK$ hazard pointers in total.

When a thread wants to access a shared object, it stores the object's reference in one of its unused hazard pointers. This is the way to signal to the other threads that this thread is using this particular object and it is therefore currently not safe to reclaim it. When the thread no longer needs the object it simply resets the according hazard pointer to null.

Nodes that have been removed from the data structure and need to be reclaimed (in [Mic04a] they are referred to as *retired nodes*) are maintained in a thread local list. Whenever the size of a thread's list reaches a threshold R (which can be chosen arbitrarily) the thread tries to reclaim the nodes from the list. Increasing R amortizes reclamation overhead across more elements, but increases memory usage; if R is larger than H by some amount proportional to H, written formally as $R = H + \Omega(H)$, the amortized per-element processing time is constant. But when the threshold for the number of retired nodes is H, there are potentially $pH = p^2K$ unreclaimed nodes, i.e., the number of unreclaimed nodes is quadratic in the number of threads. This can become a huge problem for large numbers of threads as will be shown in Section 5.7.

In order to determine whether it is safe to reclaim a certain node a thread has to scan the hazard pointers of all the other threads to check if one of them is currently using it.

Since each thread has K hazard pointers and can hold R removed elements in its private list, a crashed thread can prevent only $K + R$ removed elements from being reclaimed. This reclamation scheme thus bounds the amount of memory which can be occupied by removed elements, even in the presence of thread failures.

In real life scenarios there are often situations where it is not possible to provide a fixed upper bound on the number of hazard pointers. For these cases HPBR can be extended to support an arbitrary number of hazard pointers per thread. Unfortunately, this change also destroys the two important properties that set HPBR apart from other reclamation schemes: constant processing time per element as well as the upper bound on unreclaimable nodes.

Hazard pointers are patented at the US Patent & Trademark Office under the publication number US20040107227 (A1) [Mic04b]; the patent is held by International Business Machines (IBM). So before using this scheme in a project one has to consider the legal implications of using a patented technology.

Nonetheless, Michael and Wong submitted a proposal to add a hazard pointers implementation to the C++ standard library [MW16].

Aghazadeh et al. [AGW14] introduced an improved version of HPBR by reducing the number of comparisons per scan to one, at the cost of increasing the amount of time between node removal and node reclamation.

2.5 Pass The Buck

In [HLM02] Herlihy et al. presented a formalized problem description called the *Repeat Offender Problem* (ROP). It is defined with respect to a set of *values*, a set of *clients*, and a set of *guards*. Each value can be `free`, `injail`, or `escaping` where initially, all values are `free`. An application dependent `Arrest` action can cause a `free` value to become `injail` at any time. A client can help `injail` values to start `escaping` so they can eventually become `free` again.

Clients can only use values that are not `free`. In order to prevent a value v from escaping while it is being used by a client, a guard can be posted on v. However, the guard may fail to prevent v from escaping if it is posted too late. Thus, to safely use v, a client must ensure that v is `injail` at some time *after* it posted a guard on v. Clients can hire and fire guards dynamically, according to their need.

ROP solutions can be used by threads (clients) to avoid dereferencing (using) a pointer (value) to an object that has been freed. In this context, an `injail` pointer is one that has been allocated (arrested) since it was last freed and can therefore be used. Therefore, any solution to the ROP also solves the memory reclamation problem for dynamic lock-free data structures.

Together with the formal problem definition Herlihy et al. also presented a solution to the ROP called “Pass The Buck” (PTB). PTB is quite similar to HPBR—the *guards* concept is essentially the same as the previously described *hazard pointers*. The main difference between these two is the way it is determined which retired nodes can safely be reclaimed. PTB lacks the amortized bound on the memory reclamation overhead per physically deleted node, but uses the concept of *hand offs* to provide value progress, which guarantees that logically deleted nodes will eventually be freed—even in the face of thread failures. On the downside, PTB requires the more expensive compare-and-swap operation while HPBR requires only atomic reads and writes. Also, the algorithm of PTB as it is described in the paper is likely to suffer from false sharing since all the threads operate on different elements from some globally shared arrays. However, this can certainly be improved in a real-world implementation.

2.6 Beware & Cleanup

Beware & Cleanup is a reclamation method presented by Gidenstam et al. in [GPST09]. It is based on a combination of hazard pointers and reference counting and tries to keep the respective strengths while avoiding the drawbacks.

The basic idea is to use hazard pointers to protect thread-local references and reference counters to protect internal links in the data structure. Thus, the reference counter of each node indicates the number of globally accessible links that reference that node.

This scheme provides an upper bound on the number of nodes that are withheld from reclamation by bounding the size of the threads' deletion lists. To achieve this, Beware & Cleanup relies on a *cleanup* procedure that can update links in deleted nodes, provided that the following property is satisfied: Each link in a deleted node that references another deleted node can be replaced with a reference to an active node, with retained semantics for all of the involved threads.

Nodes in the deletion lists might not be available for reclamation due to being referenced by a hazard pointer, or by being referenced from other deleted nodes (i.e., by having a non-zero reference counter). Since those other deleted nodes can be in the same deletion list, as well as in the deletion list of some other thread, the deletion lists of all threads are accessible by any thread.

When the number of nodes in a thread's deletion list reaches a certain threshold, it performs a cleanup of all nodes in its own deletion list. If none of these nodes becomes reclaimable after the cleanup, this must be due to references from nodes in the deletion lists of other threads. Then the thread tries to perform a cleanup on the deletion lists of all other threads as well. This procedure is repeated until the length of the deletion list is below the threshold, effectively bounding the total number of nodes that are not yet reclaimed.

Unfortunately, the cleanup function is specific to each data-structure, so this scheme is not as generic as the other schemes described so far.

2.7 Quiescent state based reclamation

Quiescent state based reclamation (QSBR) is typically used to implement read-copy-update (RCU) schemes [MS98, DMS$^+$12]. It is based on the concept of a *grace period*. A grace period is a time interval $[a, b]$ such that, after time b, all nodes removed before time a can safely be reclaimed. QSBR uses *quiescent states* to detect grace periods. A *quiescent state* for some thread T is a state in which T holds no references to shared nodes. In particular, T holds no references to any shared nodes which have been removed from a lock-free data structure.

A *grace period* is therefore any time interval in which every thread of the system has passed through at least one *quiescent state*. Figure 2.1 illustrates this relationship.

Thread $T1$ goes through quiescent states at times t_1 and t_5, thread $T2$ at times t_2 and t_4, and thread $T3$ at time t_3. Hence, any time interval that contains either $[t_1, t_3]$ or $[t_3, t_5]$ is a valid grace period.

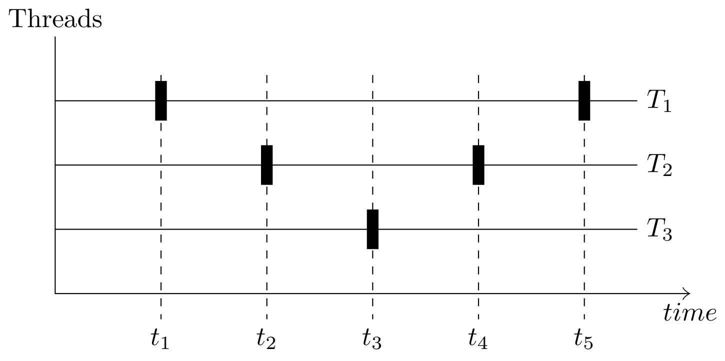

Figure 2.1: Black boxes represent quiescent states. Any time interval that contains $[t_1, t_3]$ or $[t_3, t_5]$ is thus a grace period.

A typical way to implement QSBR is by using a fuzzy barrier [Gup89]. In contrast to a non-fuzzy barrier, which blocks the thread upon entry until all other threads have also reached the barrier, a fuzzy barrier simply skips the protected code block and continues execution if some other thread has not yet entered the barrier. The thread will attempt to enter the fuzzy barrier again during subsequent executions.

For implementing QSBR the fuzzy-barrier is used to protect the code that performs the reclamation. The threads try to enter the barrier and reclaim retired nodes when they pass through a quiescent state.

In order to determine whether all threads have reached the barrier (i.e., whether they went through at least on quiescent state) all threads have to be checked. This incurs a performance overhead linear in the number of threads.

2.8 Epoch based reclamation

Epoch based reclamation (EBR), presented by Fraser in 2004 [Fra04], is also based on the concept of grace periods. Nodes that have been removed from data structures are kept in *limbo lists* that hold the references to the nodes until it is safe to reclaim them (i.e., until no stale references can possibly exist).

In EBR the programmer has to define a *critical region*[4] in which a thread is allowed to access shared objects. These regions have to be entered and left explicitly. A global *epoch count* is used to determine when no stale references exist to any object in a limbo list.

Every thread has a flag that indicates whether this thread is currently in a critical region as well as a local epoch count that identifies the epoch it currently executes in (in case it currently is inside a critical region). The thread's local epoch count may lag at most one epoch behind the global epoch. Each time a thread enters a critical region it

[4]This *critical region* has nothing to do with the term *critical section* that is often used in the context of mutual exclusion.

sets the flag and *observes* the current epoch, i.e., it updates its local epoch to match the global epoch. A thread that removes a node from a data structure places this node on the current limbo list (i.e., the limbo list that is associated with the current epoch).

After some predetermined number of critical region entries, a thread will attempt to update the global epoch. This succeeds only if all threads in a critical region have already observed the current epoch—this can again be achieved by using a fuzzy barrier. In that case the limbo list that was populated two epochs ago can safely be reclaimed and the list itself can immediately be recycled and reused for the next epoch. Thus only three epochs (and limbo lists) are required in total.

In order to determine whether all threads have observed the new global epoch all thread local epochs have to be checked. This produces a performance overhead linear in the number of threads.

2.9 New epoch based reclamation

New epoch based reclamation (NEBR) is an extension to EBR that was proposed by Hart et al. in [HMBW07]. The original description of EBR defines a critical region around every operation. However, entering a critical region requires a sequentially consistent operation and such operations can be quite expensive. This is necessary to guarantee that another thread that tries to update the global epoch actually sees the new value and therefore recognizes that this thread is inside a critical region. Without this guarantee, a race condition can occur, where the global epoch gets updated which in turn allows a node to be freed even though the node is still in use by some thread, just because the update of this thread's critical region flag was not noticed by the thread that updated the global epoch. In EBR every single operation on some lock-free data structure is encapsulated in its own critical region, thus every operation requires such a fence.

During their performance analysis Hart et al. identified this overhead for every single operation to be very significant. Therefore, the proposed solution in NEBR is that a critical region can be expanded over several operations. For example, when a group of operations on some data structure has to be performed together, the critical region is entered before the first operation and left after the last one, effectively expanding the region over all operations and thus distributing the overhead for the region entry over the whole group of operations.

2.10 Drop the Anchor

Drop the anchor (DTA), presented by Braginsky et al. in [BKP13], is a reclamation scheme that combines ideas from hazard pointers and epoch based reclamation. The resulting scheme should reduce overhead in comparison to HPBR while keeping the advantages of bounded memory and failure tolerance.

They presented a specialized technique that is tailored to singly-linked lists. The basic idea is that instead of acquiring a *hazard pointer* each time a pointer to a node is read, a *hazard pointer* is acquired only once for every c pointers read where c can be chosen arbitrarily—higher values improve performance while lower values improve the bounds on the number of unreclaimable nodes. If a thread T_1 suspects that another thread T_2 has crashed, a complex technique called *freezing* [BP11] is used, in which T_1 co-operates with other running threads to replace the part of the data structure that T_2 might potentially access, in order to restore their ability to free memory. Basically all the nodes of the list that T_2 might access will be cut out and replaced with new copies. The cut-out nodes are marked so that T_2 can tell what happened in case it has not crashed but wakes up eventually. This allows memory reclamation to continue even in the face of thread failures and provides an upper bound on the number of nodes that a failed thread can prevent from being reclaimed.

DTA has been analyzed in [BKP13] and [AEH+14] and has been shown to be very efficient; the key to performance being that the expensive anchor and freezing operations are used rarely, and do not affect the "fast path". However, DTA does not qualify as a generic reclamation scheme as it requires the programmer to implement the complex and data structure specific wait-free freezing concept.

The original paper only describes the implementation for a linked list, but the authors believe that their technique can be applied for other non-blocking data structures as well.

2.11 StackTrack

This scheme, presented by Alistarh et al. in [AEH+14], makes use of modern *hardware transactional memory* (HTM) functionality that was introduced with the Intel Haswell architecture [R.12].

The basic idea is that each operation of the lock-free data structure is broken down into a series of transactions in a way such that a successfully committed transaction cannot interfere with any memory reclamation. The scheme relies on the HTM implementation to automatically monitor all pointers stored in the private memory and therefore it does not require the explicit announcement of pointers before they are accessed. The HTM system will automatically abort any transaction that tries to accesses a node which is freed during the transaction. Since lock-free algorithms do not depend on transactions for correctness, each operation can be split into a number of smaller transactions to reduce the probability of transaction aborts.

Currently available implementations of HTM do not offer progress guarantees, i.e., there is no guarantee that a transaction can be ever successfully committed. So to make the algorithm lock-free a fallback path has to be provided that does not use HTM; StackTrack falls back to *hazard pointers*.

This scheme requires the programmer to insert code before and after each operation. In addition, breaking down the operation into smaller transactions requires to insert code after every few lines of the operation implementation. However, the proposal also states that it should be possible to write a compiler to automate much of this.

2.12 ThreadScan

This is another scheme proposed by Alistarh et al. [ALMS15]. It is designed as a memory reclamation library where the programmer provides a data structure implementation with correct free calls and ThreadScan will implement it automatically ensuring efficient memory reclamation.

At a high level it works as follows: deleted nodes are collected in a shared *delete buffer* with a fixed size. When the buffer becomes full, the thread which inserted the last node becomes the *reclaimer* and initiates a *Collect* operation. For that the *reclaimer* signals to all other threads that they need to help examine references to nodes in the *delete buffer*, i.e., each thread has to scan its own stack and registers for such references. If a node is still referenced by some thread it gets marked. By having all threads help marking referenced nodes, the cost of the memory scan is divided among all threads as each thread is scanning only its own stack and registers. The scan is performed word-by-word, checking each chunk against pointers in the delete buffer. In order to guarantee that this scan works correctly, the programmer must not actively “hide” pointers to live nodes. At the end each thread replies with an acknowledgment and resumes its normal execution. Once the *reclaimer* has received all acknowledgments it can safely reclaim all unmarked nodes and return.

The scheme offers strong progress guarantees. This is achieved by relying on the signaling system available on POSIX systems. The helping procedure that gets executed by each thread as part of the *Collect* operation is part of their signal handler. Since the operating system signal handler code always has precedence over the application, this shields the scheme from possible errors in the data structure like, e.g., infinite loops. As a result, strong progress is guaranteed as long as the operating system does not starve threads, which is highly unlikely on todays modern operating systems. In addition, all other data structures preserve their progress guarantees as well since ThreadScan only adds a bounded number of steps to their execution.

2.13 Optimistic Access

This scheme by Cohen and Petrank [CP15b] is designed to provide fast read operations, as reads are the most common. In particular, the goal is to allow reads to be executed without writing to shared memory. This is achieved by allowing a thread to sometimes read a node even after it was reclaimed. The scheme maintains correctness in spite of reading reclaimed nodes by using the following three key properties:

1. A read must not fail, even when accessing reclaimed memory.
2. A read that accesses reclaimed memory will be identified immediately after the read.
3. When a read of such a stale value is detected, the scheme allows a rollback of the optimistic read.

The first requirement can be easily satisfied by using user-level allocators that allocate and de-allocate memory without returning pages to the system.

For the second requirement Optimistic Access divides the memory reclamation into phases and poses the following restrictions. First, an object is never reclaimed at the same phase in which it is unlinked from the data structure; it can only be reclaimed at the next phase or later. Second, a thread that acknowledges a new phase will not access objects that were unlinked in previous phases. Therefore, if a thread is not aware that a phase has changed, its read operation may potentially access a reclaimed object. Otherwise, i.e., if the thread is aware of the current reclamation phase, its read operation is safe.

In order to effectively manage phase changes, each thread is equipped with a *warning-flag*. This flag is set if a new phase had started without the thread noticing, and clear otherwise. During a phase change the warning-flags of all threads are set. When a thread acknowledges a phase change it resets its flag. This way, checking whether a read might have accessed an already reclaimed object is as simple as checking whether the flag is non-zero.

The third requirement presupposes that the lock-free data structure supports to roll back an operation. This is possible in most lock-free data structures that handle races by simply restarting the operation from scratch. However, to formally define a roll-back mechanism that covers a wide range of data structures, Optimistic Access adopts the normalized form for lock-free data structures described in [TP14].

A read operation with Optimistic Access is therefore executed as follows: first the shared memory is read, then the thread's warning-flag is checked and if set, a restart mechanism is used to roll back the execution to a safe point.

In order to prevent write operations from accessing reclaimed memory, the scheme adopts a simplified version of hazard pointers. Before a thread performs a write operation it has to declare the location in a hazard pointer, thus preventing the object from being reclaimed.

A thread that increments the phase number could now reclaim all objects that were unlinked in previous phases and that are not referenced by a hazard pointer. To reduce the overhead caused by too frequent phase changes and the associated restarts, retired objects are gathered in a global buffer and a single reclaiming thread processes all objects unlinked by all threads.

However, it is not possible to implement this scheme in C++ in a way that complies with the C++ standard as the potential read of already reclaimed memory poses a classic data race (see Section 3.3).

2.14 Automatic Optimistic Access

This scheme from Cohen and Petrank [CP15a] is inspired by mark-sweep garbage collectors and builds on the previously described *Optimistic Access*. Just as Optimistic Access, it relies on the normalized form of a data-structure. Unlike all other garbage collectors presented in the literature it strictly satisfies the lock-free property. However, instead of over the entire heap it is applied on a single data-structure with a structured algorithm. Since this scheme works like a garbage collector, the developer does not have to manually insert retire statements, instead the algorithm automatically determines the set of unreachable data-structure nodes and reclaims them for future use.

Since this algorithm is supposed to be lock-free, it cannot rely on handshakes and therefore has to deal with three major difficulties. First, it has to get all threads' roots in a lock-free manner, including those of threads that might not respond. Second, it has to maintain correctness in cases where threads may be executing under the assumption that a previous collection is still active. Finally, it has to ensure proper completion of a collection phase even if a thread fails in the middle of the scan operation.

In order to be able to read the roots of unresponsive threads, each thread records its roots at known intervals, therefore making them available to other threads. Similar to Optimistic Access a thread that detects the read of a stale value has to restart execution from a location where its roots are known.

Threads that are executing code in an outdated reclamation phase could corrupt the reclamation execution of the current phase. In order to avoid this, all of the scheme's shared data is protected using a versioning scheme. Variables that could be corrupted are modified via a CAS operation that fails if the current phase number does not match the local phase. Before a new phase is started, all such phase-protected variables are updated to a new phase. Thus, any update attempt by a late thread will fail.

It has to be guaranteed that all nodes are properly scanned and marked, even if a thread stops responding while performing its part of the scan operation. To this end each thread has a mark-stack containing all the nodes that are not yet processed by this thread. These mark-stacks are readable by other threads and the mark procedure is implemented in a way that satisfies the invariant that children of a marked node are either marked or are visible to other threads. Thus, even if a thread becomes unresponsive, other threads can continue to work on nodes that reside on its stack and no node is lost.

2.15 DEBRA / DEBRA+

Brown [Bro15] proposed DEBRA (Distributed Epoch Based Reclamation) which is an adaptation of the EBR scheme (see Section 2.8). The main difference is that all its operations perform in only $O(1)$ steps.

Just as in EBR, each thread has three limbo lists, a flag that signals whether it is currently inside a critical region or not and a value that represents the latest epoch that this thread has observed. However, in contrast to EBR the thread that enters a critical region incrementally scans the flags of all the other threads, amortizing the cost over n enter operations. A local variable keeps track of the number of threads that have already been verified not to be in a critical region or that have already observed the current epoch. Once all threads have been checked (i.e., the variable is equal to n), the thread can perform a CAS to increment the current epoch.

Brown also proposed yet another extension; the resulting schema is called DEBRA+ which also brings fault tolerance to the table. To achieve this it uses signals, an interprocess communication mechanism supported by POSIX-compliant operating systems, and non-local gotos, which allow threads to begin executing from a different instruction, outside of the current function.

When a thread is preventing the update to the next epoch for too long (e.g., because it has crashed, entered an infinite loop, etc.), the slow thread is *neutralized* using an OS signal. Upon resuming execution, the neutralized thread runs special recovery code to clean up any inconsistencies it might have left before it was neutralized.

This allows DEBRA+ to limit the number of non-reclaimable objects to $O(mn^2)$, where n is the number of threads and m is the largest number of objects removed from the data structure by a single operation.

2.16 QSense

Balmau et al. proposed QSense [BGHZ16], a hybrid approach that combines QSBR and hazard pointers. The idea is to use QSBR for the fast path, as it has little overhead, and to fall back to a variant of hazard pointers in case of *prolonged delays* due to one or more threads not passing through a quiescent period. A prolonged delay is a delay that is long enough such that a number of nodes larger than a given configurable threshold has been removed but cannot safely be reclaimed. If prolonged delays are detected, QSense automatically falls back to a robust reclamation scheme, i.e., a scheme that provides an upper bound on the number of steps for all actions related to memory reclamation, regardless of the progress of other threads.

The proposed fall-back path is called *Cadence* and is based on an amortized hazard pointer's variant. It achieves significantly better performance by avoiding the necessity for memory barriers during data structure traversal. To this end, QSense introduces background threads (called *rooster processes*) that periodically wake up and generate context switches, which act as memory barriers. This ensures that any hazard pointer becomes visible to other threads within a bounded time T—the time between two context switches. By deferring reclamation such that a thread may only reclaim a retired node n after it has been awaiting reclamation for longer than T, it is guaranteed that any hazard pointer potentially protecting n must be visible and thus n can safely be reclaimed—provided that none of the visible hazard pointers is protecting n.

Even though Cadence was proposed and used only in the context of QSense, it would also be possible to use it as a stand-alone memory reclamation scheme.

2.17 Stamp-it

I came up with the first version of this scheme several years ago before I knew about QSBR, EBR, or the timestamp based scheme described by Harris in [Har01]. However, the resulting scheme has several similarities.

As in EBR, the programmer has to define a *critical region* that is entered and left explicitly. A thread is only allowed to access shared objects inside such regions. Reclamation of nodes is deferred to a time when it is guaranteed that no thread can

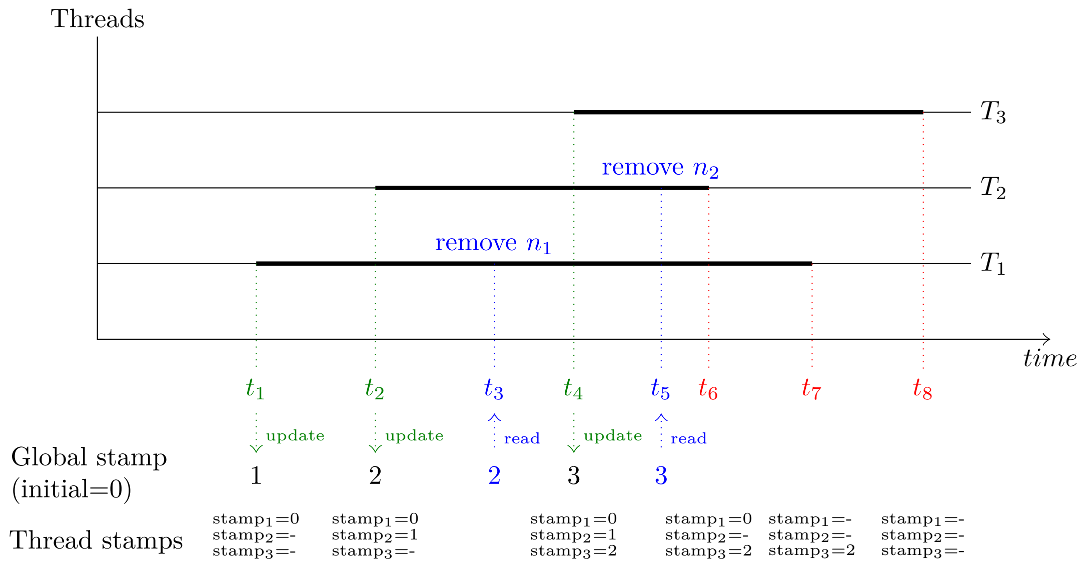
Threads
T_3
remove n_2
T_2
remove n_1
T_1
time
t_1
t_2
t_3
t_4
t_5
t_6
t_7
t_8
update
update
read
update
read
Global stamp (initial=0)
1
2
2
3
3
Thread stamps
stamp$_1$=0 stamp$_2$=- stamp$_3$=-
stamp$_1$=0 stamp$_2$=1 stamp$_3$=-
stamp$_1$=0 stamp$_2$=1 stamp$_3$=2
stamp$_1$=0 stamp$_2$=- stamp$_3$=2
stamp$_1$=- stamp$_2$=- stamp$_3$=2
stamp$_1$=- stamp$_2$=- stamp$_3$=-

Figure 2.2: Example visualizing the idea of Stamp-it. The thick lines mark critical regions.

possibly hold a reference to the node, i.e., when all the threads that happened to be inside a critical region at the time the node was retired have left the critical region.

When a thread enters a *critical region* it increments a *global stamp* using an atomic *fetch-and-add* and stores the returned stamp in a thread-local data structure visible to the other threads. By setting the stamp in the data structure, the thread also signals to other threads that it is now inside a *critical region*. When a thread retires a node for reclamation it simply takes the current value of the *global stamp*, stores it in a special field of the node and appends the node to the end of a thread-local retire list. The node can be reclaimed as soon as all the threads that were inside a critical region at the time the node was added to the retire list have left their respective critical region.

When a thread leaves the critical region, it resets its stamp and tries to reclaim retired nodes from the local retire list in case it contains any. For that, it must determine the *lowest* stamp value of threads that are inside a critical region, i.e., the stamp value of the thread that has entered a critical region at the earliest. Any node in the retire list that has a stamp value that is less or equal to the gathered stamp can then safely be reclaimed. Since retired nodes are appended to the end of the retire list they are strictly ordered by their stamp value. Reclamation starts with the node with the lowest stamp and can stop as soon as the first node with a stamp higher than the current lowest stamp is found. No time is wasted on nodes that cannot yet be reclaimed. Figure 2.2 illustrates this.

The initial value of the global stamp is zero. When thread T_1 enters its critical region at time t_1 it increments the global stamp and stores the old value in its local stamp. The same happens when T_2 enters its critical region at t_2 and T_3 at t_4. At t_3, thread T_1

removes the node n_1 from some data structure and marks it for reclamation. To that end, it reads the current value of the global stamp, which is two since time instant t_2, stores this value in the node and adds it to the local retire list. The node can be reclaimed once all threads that were in a critical region at the time the node was marked (t_3) have left their respective critical region. This can be determined by checking if any thread in a critical region has a local stamp value that is less than the node's stamp. Or more formally, a node with stamp s can be reclaimed if $\forall x(T_x$ is not in critical region $\lor$ $\text{stamp}_x \geqslant s)$. For node n_1 this would be t_7 and for node n_2 it would be t_8. Optionally each thread could exclude itself from this check (if it is guaranteed that it does not hold any references to the nodes in question), in which case the node n_1 could already be reclaimed at t_6.

A straight forward implementation of this scheme is quite simple, but will have runtime complexity linear in the number of threads since all threads have to be scanned in order to determine the lowest stamp. To improve this, the algorithm was redefined based on a data structure that efficiently supports the following operations:

1. Add an element and assign a stamp to it (`push`). Stamps have to be strictly increasing, but not necessarily consecutive.

2. Remove a specific element, return `true` if this element was the one with the lowest stamp (`remove`).

3. Get the highest stamp ever assigned to an element.

4. Get the lowest stamp of all elements.

In addition, a global retire-list is introduced. It is used to collect nodes that could not be reclaimed when their owning thread left its critical region. The responsibility to reclaim these nodes is deferred to the "last" thread as explained below.

The algorithm uses this data structure as follows. Upon entering a critical region the thread adds itself to the data structure, and gets a new stamp value. Stamp values are strictly increasing, therefore, defining a total order in which all threads have entered their respective critical region.

When a thread retires a node it requests the highest stamp from the data structure, increments that value, stores it in the node and appends the node to the end of its local retire-list. If this pushes the number of entries in the local retire-list over a certain threshold it immediately performs a reclaim operation. The increment is used to create a stamp that is larger than the stamps of all threads that were in a critical region at the time the node was retired. This is essentially equivalent to using the global stamp as previously described.

The reclaim operation requests the lowest stamp from the data structure and reclaims all entries from the local retire-list with a stamp value less or equal to the requested one. Since new nodes are appended to the end, the elements in the local retire-list are ordered by their stamp values. This makes the reclaim operation very efficient as it always has a runtime linear in the number of nodes that can currently be reclaimed; no time is wasted on nodes that cannot yet be reclaimed.

Upon leaving a critical region the thread removes itself from the data structure and performs a reclaim operation on the local retire-list. If the remove operation returns `false` and the number of nodes in the local retire-list exceeds some threshold the thread pushes all remaining entries to the global retire-list as an ordered sublist (internally called *chunk*). If the remove operation returns `true`, i.e., the thread had the smallest stamp and was therefore "lagging behind" the most and blocking reclamation, it will perform a reclaim operation on the global retire-list. In contrast to the local retire-list the global retire-list is not totally ordered and therefore does not provide the same runtime guarantees. However, since it is organized as a list of sorted sublists, each sublist needs to be scanned only up to the node which has a stamp that is larger than or equal to the lowest stamp returned. Therefore, if we maintain additional links from sublist to sublist, the resulting total runtime is $O(n+m)$ where n is the total number of reclaimable nodes and m is the number of ordered sublists in the global retire-list.

I decided to implement the data structure as a lock-free doubly-linked list based on the proposal by Sundell and Tsigas [ST05]. This data structure maintains sentinel *head* and *tail* nodes which are used to store the highest and lowest stamp values, respectively. The `push` operation first increments the head's stamp using an atomic fetch-and-add, stores the returned value in the node it is currently inserting and then tries to insert the node into the linked list, right after the head, using an atomic compare-and-swap operation. The `remove` operation unlinks the node from both directions, and returns `true` if the node was last, i.e., the tail's predecessor.

Every thread holds a thread-local control block that is used as a node in this list. A thread that enters a critical region simply calls `push` with its node. Thus, the linked list in direction from tail (smallest stamp) to head (largest stamp) defines the order in which the threads have entered their respective critical regions. When a thread leaves its critical region it calls `remove`. If the return value is `true`, it first updates tail's stamp to match the value of the new predecessor, and then it performs a reclaim operation on its local retire-list as well as the global retire-list. Otherwise, the thread performs a reclaim operation on its local retire-list, and if the number of remaining nodes exceeds some threshold, it moves the remaining local list to the global retire-list.

The algorithm is lock-free. In the absence of contention, entering and leaving critical regions and takes constant time. The reclamation operation takes time proportional to the number of reclaimable nodes; the time per node is therefore amortized constant. The implementation is described in detail in Section 4.4.6. Section 5.2 shows experimental results that even under load, the number of retry iterations is small (constant).

CHAPTER 3

Memory Models

This chapter describes the memory models of the two most common micro-architectures, x86 and ARM, as well as the memory model that was introduced with the C++11 standard.

The memory model is at the heart of the concurrency semantics of any shared-memory system. It defines the set of values that a read operation in a program is allowed to return, therefore defining the basic semantics of shared variables. It is impossible to meaningfully reason about a program or any part of the language implementation (including hardware) without an unambiguous memory model. Reclamation schemes are no exception to this rule; since their purpose is to solve the reclamation problem of concurrent data structures, they are themselves inherently concurrent. The C++11 memory model described in this chapter will be used extensively in Chapter 4 when the implementations of the various schemes are discussed.

Recent books by Michael L. Scott [Sco13] and Sorin et al. [SHW11] provide good introductions to memory models in both hardware and software. Overviews of the complex issues can be found in numerous papers, for instance those by Adve and Gharachorloo [AG96], Adve and Boehm [AB10, AB11], and McKenney [McK05] to mention a few.

The memory model can be refined to differentiate between the *programming language memory model* and the *hardware memory model.*

- Language memory model: Defines the optimizations, instruction re-writes, and reorderings a compiler is allowed to perform.

- Hardware memory model: Defines the optimizations and instruction reorderings a specific architecture implementation is allowed to perform.

These optimizations can cause instructions to be executed or perceived in an order that differs from what is defined in the source code, resulting in the definitions of the following four orderings:

Source code order: Defines the order in which the memory operations are specified in the source code by the programmer.

Program order: Defines the order in which the memory operations are specified in the machine code, i.e., the code that is executed by the CPU. Note that this can differ from the source code order, because depending on the definition of the language memory model, compilers are allowed to reorder instructions as part of the optimization process.

Execution order: Defines the order in which the individual memory-reference instructions are executed on a given CPU. The execution order can differ from the program order due to optimizations based on the hardware memory model of the specific CPU-implementation.

Perceived order: Defines the order in which a CPU perceives its and other CPUs' memory operations. The perceived order can differ from the execution order due to caching, interconnect and memory-system optimizations. Different CPUs can perceive the same set of memory operations as occurring in different orders. This is also defined by the hardware memory model.

The reason why these orders can be different stems from the fact that increases in memory performance have not kept up with the rate at which CPU instruction performance has increased. Trying to hide the fact that memory operations are increasingly expensive compared to simple register-to-register instructions, modern CPUs receive increasingly large caches in order to reduce the overhead of these memory accesses.

However, CPUs have become so fast that even these caches cannot keep up with them. Therefore, caches are often partitioned into *banks* that can operate nearly independently from each other. This allows each of the banks to run in parallel in order to better keep up with the CPU. Memory is usually divided evenly among the banks by address, e.g., even-numbered cache lines are processed by bank 0 while odd-numbered cache lines are processed by bank 1. However, this type of hardware parallelism now allows memory operations to complete out of order.

Suppose two memory write operations where the first one is processed by bank 0 and the second one is processed by bank 1. Now if bank 0 is already busy processing an earlier request and bank 1 is idle, the second write would be visible to another CPU *before* the first write—the writes would be perceived *out of order* by other CPUs. However, this kind of reordering is not limited to write operations; read operations can be reordered in a similar manner.

3.1 Sequential consistency

Sequential consistency is the most intuitive memory model and also the easiest for reasoning about the correctness of an algorithm or data structure. That is why most publications on concurrent algorithms or data structures either explicitly or implicitly assume a sequentially consistent memory model.

A natural view of the execution of a multi-threaded program is as follows. For each step one of the threads is randomly chosen and the next instruction in that thread's program gets executed. This process is repeated until the program as a whole terminates. This is effectively equivalent to taking all the steps of all threads and interleaving them in some way, resulting in a single total order of all steps. Therefore, whenever an object is accessed, the last value stored to the object in this order is retrieved. Following Lamport [Lam79], an execution that can be understood as such an interleaving is referred to as sequentially consistent.

For an example see Listing 3.1 which shows an implementation of Dekker's mutual exclusion algorithm [Dij65]. The steps of the two threads can be interleaved in many ways, but since the program order is preserved it is ensured that at least one of the load operations sees the value of the prior store operation, i.e., the program order, execution order and perceived order are all identical. Therefore is is impossible that both, *r1* and *r2*, are zero.

Listing 3.1: Dekker's mutual exclusion algorithm

```
Initially: X = 0, Y = 0

Thread 1:
  X = 1
  r1 = Y

Thread 2:
  Y = 1
  r2 = X
```

Unfortunately, ensuring sequential consistency is quite expensive and none of todays processor architectures provide a fully sequentially consistent memory model. While they allow to enforce sequential consistency at certain points, normal execution is not sequentially consistent, but highly dependent on the implementation of the specific architecture.

3.2 Weaker memory models

3.2.1 x86-TSO

Even though the Intel x86 memory model is somewhat weaker than the sequentially consistent model, it is still one of the strongest models amongst todays modern CPU implementations. However, as Sewell et al. [SSO+10] point out, for a long time the information provided by Intel as well as AMD on their respective architecture implementations were partly purely informal, missing concrete examples and sometimes even inconsistent with the actual implementation.

Based on these results they formally described a new memory model called “x86-TSO” (Total Store Order) which is consistent with the concrete examples in Intel's and AMD's latest documentation available at that time. This model is illustrated in Figure 3.1.

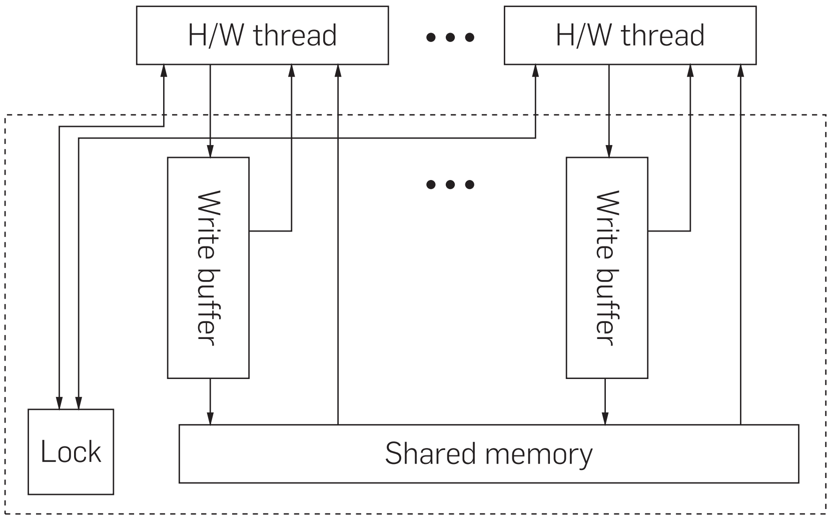

Figure 3.1: x86-TSO block diagram—taken from [SSO+10].

As can be seen in the figure, the hardware threads interact with a storage subsystem, which is represented by the dotted box. The state of this storage subsystem comprises a shared memory that maps addresses to values, a global lock to indicate when a particular hardware thread has exclusive access to memory, and one store buffer per hardware thread. A formal definition of the behavior of the storage subsystem can be found in [SSO+10], but the main points are:

- The store buffers are FIFO and a reading thread must read its own most recent buffered write, if there is one, to that address. Otherwise reads are satisfied from shared memory.
- An `mfence` instruction flushes the store buffer of that thread.
- To execute a `lock`'d instruction[1], a thread must first acquire the global lock. At the end of the instruction, it flushes its store buffer and releases the lock. While the lock is held by one thread, no other thread can read. This essentially means that `lock`'d instructions enforce sequential consistency.
- A buffered write from a thread can propagate to the shared memory at any time except when some other thread holds the lock.

x86-TSO does not permit local reordering except of reads after writes to different addresses.

[1]These are read-modify-write instructions with a `lock` prefix for atomicity like, e.g., `lock xadd` (atomic fetch-and-add) or `lock cmpxchg` (atomic compare-and-swap). A complete list of instructions that support the `lock` prefix can be found in [Int16, 8.1.2.2].

Since writes are buffered, the new value is not visible to other threads until it has propagated to the shared memory. Therefore, Dekker's algorithm from Listing 3.1 no longer guarantees mutual exclusion under the x86-TSO model, as it is perfectly possible that *r1* as well as *r2* are both zero. This could be resolved by either introducing an `mfence` instruction after a first store operation, or by performing the store operation using a `lock xchg` instruction.

Another memory model that is very similar to x86-TSO is the SPARC v8 TSO model [SPA92].

3.2.2 ARM and POWER

ARM as well as POWER architectures have considerably more relaxed memory models, allowing a wider range of hardware optimizations. Maranget et al. [MSS12] provide a very detailed and extensive description of both architectures and their observable behaviors.

While this relaxation can improve performance, power efficiency and hardware complexity, it makes the life of a programmer, who is implementing concurrent data structures, significantly harder. In contrast to TSO models the following behaviors are possible on these architectures:

1. Hardware threads can perform reads and writes out-of-order, or even speculatively, i.e., before preceding conditional branches have been resolved. Any local reordering is allowed unless specified otherwise.

2. The memory system does not guarantee that a write becomes visible to all other hardware threads at the same time.

Since a certain ordering of instructions is crucial already for the simplest non-blocking data structures, these architectures provide various memory barriers and dependency guarantees that the programmer has to use correctly in order to enforce the desired ordering.

To understand the behavior of such a machine it is sometimes helpful to think of each hardware thread as effectively having its own copy of memory, which is illustrated in Figure 3.2. The collection of all the memories and their interconnection (i.e., everything except the threads) is usually referred to as the *storage subsystem*. A write by one thread may propagate to other threads in any order, and the propagations of writes to different addresses can be interleaved arbitrarily, unless they are constrained by barriers or coherence. One can also think of barriers as propagating from the hardware thread that executed them to each of the other threads. The ARM `dbm` and POWER `sync` barrier instructions can be used to enforce the following orderings between two instructions:

Read/Read: The barrier ensures that they are satisfied and committed in program order.

Read/Write: The barrier ensures that the read is satisfied and committed before the write can be committed (and thus propagated and become visible to others).

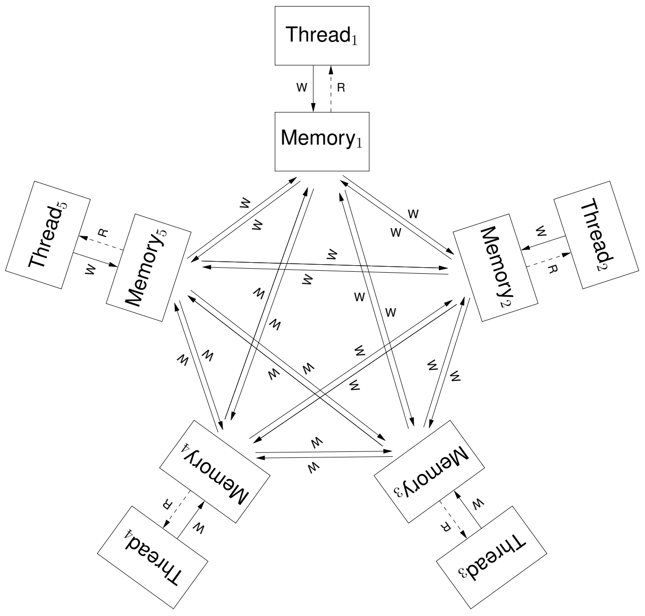

Figure 3.2: Storage subsystem—taken from [MSS12].

Write/Write: The barrier ensures that the first write is committed and has propagated to all other threads before the second write is committed.

Write/Read: The barrier ensures that the write is committed and has propagated to all other threads before the read is satisfied.

The POWER architecture provides with the `lwsync` instruction an additional "lightweight sync", which is weaker and potentially faster than `sync`. It mainly differs in the way a write before the barrier is handled relative to the second instruction:

Write/Write: The barrier ensures that for any particular thread, the first write propagates to that thread before the second.

Write/Read: The barrier ensures that the write is committed before the read is satisfied, but the read can be satisfied before the write is propagated to any other thread.

In addition to barriers, these architectures provide the following dependencies to enforce orderings:

Address Dependency: There is an address dependency from a read to a program-order-later read or write when the value read by the first instruction is used to compute the address of the second instruction.

Control Dependency: There is a control dependency from a read to a program-order-later read/write where the value read by the first instruction is used to compute the condition of a conditional branch that is program-order-before the second instruction.

Data Dependency: There is a data dependency from a read to a program-order-later write where the value read by the first instruction is used to compute the value that is written by the second instruction.

3.3 The C++11 memory model

On August 12th 2011 the new C++ standard, now commonly referred to as C++11, was approved and ratified by ISO, replacing the previous version C++03. Since this official ISO C++ standard is not freely available I will instead refer to the "Working Draft, Standard for Programming Language C++" from January 2012 [C++12], which only differs from the the C++11 standard in some minor editorial changes.

This new C++ standard is the first version to define the notion of *multi-threaded executions*. The C++ standard prior to C++11 specified program execution in terms of observable behavior, which in turn described sequential execution on an implicitly single-threaded abstract machine. Therefore multi-threaded C++ programs relied on libraries for threading support like POSIX threads, Win32, or Boost. Unfortunately a pure library approach, in which the compiler is designed independently of threading issues, includes all sorts of problems [Boe05]. Without a clearly defined memory model as a common ground between the compiler, the hardware, the threading library, and the programmer, multi-threaded C++ code is fundamentally at odds with compiler and processor-level optimizations [MA04]. That is why with the introduction of multi-threaded executions also a new memory model had to be defined.

The memory model defines when multiple threads may access the same memory location, and specifies when updates by one thread become visible to other threads. It is largely based on the work by Boehm, Alexandrescu et al. [BA08, ABH+04].

One of the most important aspects is the definition of a data race [C++12, 1.10.21, p. 14]:

> The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens-before the other. Any such data race results in undefined behavior.

Conflicting actions are defined as follows [C++12, 1.10.4, p. 11]:

> Two expression evaluations conflict if one of them modifies a memory location and the other one accesses or modifies the same memory location.

This definition implies that any program written according to the old standard that uses some other threading libraries and shares any data between those threads exhibits undefined behavior. The memory operations are ordered by means of the *happens-before* relationship that can be roughly described as follows:

> Let A and B represent operations performed by a multi-threaded process. If A *happens-before* B, then the memory effects of A effectively become visible to the thread performing B before B is performed.

The *happens-before* relation (denote: $\rightarrow$) is a strict partial order and as such transitive, irreflexive, and antisymmetric.

Transitivity: $\forall a, b, c, \text{if } a \rightarrow b \text{ and } b \rightarrow c, \text{ then } a \rightarrow c$

Irreflexivity: $\forall a, a \not\rightarrow a$

Antisymmetry: $\forall a, b, \text{if } a \rightarrow b \text{ then } b \not\rightarrow a$

The complete formal definition specifically for C++ can be found in [C++12, 1.10, p. 11-14].

Sequenced-before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations [C++12, 1.9.13, p. 10]. Given any two evaluations A and B, if A is *sequenced-before* B, then the execution of A shall precede the execution of B.

A *happens-before* order between two operations from the same thread (source code order) is implicitly given by the *sequenced-before* order [C++12, 1.10.12, p. 13]. A *happens-before* order between two operations from different threads (in the standard this is referred to as *inter-thread-happens-before*) must be established using atomic operations.

3.3.1 Atomic operations

The C++11 standard library introduces a new generic class `std::atomic<T>` that provides the following atomic operations to work with instances of `T`:

- `load`
- `store`
- `exchange`
- `compare_exchange_weak`
- `compare_exchange_strong`

For integral and pointer types it also provides the following operations:

- `fetch_add`
- `fetch_sub`

And for integral types only it provides the following additional operations.

- `fetch_and`

- `fetch_or`
- `fetch_xor`

For many of these operations the class also provides operators like the assignment operator for `store` or post-fix increment for `fetch_add`. These operators are a nice syntactic sugar, but they rely on the standard memory order for all operations and do not allow to customize it. By default, all operations provide sequential consistency, but the methods all take a parameter `memory_order` that allows to customize and relax the used memory order. The available memory orders and their effects are discussed in Section 3.3.2.

The `atomic` class can work with any type `T`, regardless of its size. For types with a size less or equal to the size of a pointer all the operations are usually lock-free. For other types the implementation falls back to a lock-based version to achieve atomicity. The class provides the `is_lock_free` method to determine whether the operations on the given type can be performed in a lock-free manner.

3.3.2 Memory orders

Each atomic operation takes a parameter of the type `memory_order` which is an enum type with the following values (from strong to relaxed):

- `memory_order_seq_cst`
- `memory_order_acq_rel`
- `memory_order_release`
- `memory_order_acquire`
- `memory_order_consume`
- `memory_order_relaxed`

As explained in Section 3.1, there is a single total order S of all sequentially consistent operations. An operation B that performs a load on an object M will observe the result of the last modification A of M that precedes B in S [C++12, 29.3.3, p. 1104]. From this follows that there is always a *happens-before* relation between two `memory_order_seq_cst` operations operating on the same object.

`memory_order_consume` and `memory_order_acquire` can only be used for operations that perform a *read*, `memory_order_release` can only be used for operations that perform a *write* and `memory_order_acq_rel` can only be used for operations that perform a *read-modify-write* operation. Although the language does not enforce these constraints some implementations do check them at runtime[2].

[2]For example when the `DEBUG` macro is defined the Microsoft STL implementation inserts code to verify these constraints at runtime.

A *happens-before* relationship can be established by using the following combinations of memory orders[3]:

- `memory_order_seq_cst + memory_order_seq_cst`
- `memory_order_acquire + memory_order_release`
- `memory_order_consume + memory_order_release`

An atomic operation A that performs a store-release operation on an atomic object M *synchronizes-with* an atomic operation B that performs a load-acquire operation on M and takes its value from any side effect in the release sequence (defined below) headed by A. This *synchronize-with* order is compatible with the *inter-thread-happens-before* order.

An example can be seen in Listing 3.2: Thread A writes two values to the two variables x and y. In order to guarantee that when thread B sees the new value of y it also sees the new value of x, a *happens-before* relation has to be established. In line 5 thread A uses *release* semantics to store the new value of y while in line 8 thread B uses *acquire* semantics to load the value of y. If this *acquire* load returns the value stored by the *release* store the two operations *synchronize-with* each other, therefore establishing a *happens-before* relation. Since the store to x is *sequenced-before* the store to y and the load of y is *sequenced-before* the load of x it follows that the store to x *happens-before* to load of x.

Listing 3.2: Example of *synchronize-with* relation with release/acquire operations.

```
1 std:.atomic<int> x, y;
2
3 // thread A
4 x.store(1, std::memory_order_relaxed);
5 y.store(2, std::memory_order_release);
6
7 // thread B
8 y.load(std::memory_order_acquire);
9 x.load(std::memory_order_relaxed);
```

`memory_order_consume` is based on the address dependency concept described in Section 3.2.2. It is not only more complicated but also weaker than `memory_order_acquire`. According to Hans Boehm the current definition of `memory_order_consume` in the standard is not useful [Boe16]. He proposed to temporarily deprecate `memory_order_consume` in C++17 and the proposal was accepted in the Oulu meeting in July 2016. Therefore I will not go into more detail about this memory order in the course of this work.

`memory_order_relaxed` can never be used to create a *happens-before* order.

All modifications to a particular atomic object occur in some particular total order, called the *modification order*. If A and B are modifications of an atomic object M and A *happens-before* B, then A precedes B in the modification order of M. There are separate

[3]`memory_order_acq_rel` is the combination of `memory_order_release` and `memory_order_acquire`. So wherever either one is used it is also possible to use `memory_order_acq_rel`.

modification orders for each atomic object and there is no requirement that these can be combined into a single total order for all objects.

Atomic read-modify-write operations shall always read the last value in the modification order written before the write associated with the read-modify-write operation [C++12, 29.3.12, p. 1105].

A *release sequence* is a subsequence of the modification order of an atomic object. It is headed by a release operation A and followed by an arbitrary number of

- atomic operations performed by the same thread that performed A or
- atomic read-modify-write operations.

For operations performed by the same thread that performed A, it is not relevant which memory order is used—it can even use `memory_order_relaxed`. If a thread is reading a value that is part of a *release sequence* using acquire semantics, this read synchronizes-with the release operation that is heading the sequence. Note that there can exist several release sequences on the same object at the same time. Suppose there are two release-CAS operation on some atomic object A. Since both use release semantics, they both act as head of their own release sequence. And since a CAS is an atomic read-modify-write operation, the second CAS is also part of the release sequence headed by the first CAS. So an acquire-load on A that returns the value stored by the second CAS will actually synchronize-with *both* release-CAS operations.

The C++ standard describes two different CAS operations for atomic objects: `compare_exchange_strong` and `compare_exchange_weak`. The difference between these operations is that `compare_exchange_weak` is allowed to fail spuriously, that is, act as if `*obj != *expected` even if they are equal, but it can result in better performance on some platforms. Both operations take two `memory_order` parameters: The first one describes the semantics of the read and write operations in case of success, and the second one describes the semantics of the reload operation in the failure case. In addition, the standard defines overloads for both operations taking only a single `memory_order` parameter. They forward to the two parameter version, passing the given `memory_order` as the first argument. The second argument is also derived from the given `memory_order` by removing any semantics that are only relevant for write operations, i.e., `memory_order_release` is replaced with `memory_order_relaxed` and `memory_order_acq_rel` is replaced with `memory_order_acquire`.

The C++11 standard states that “the failure argument shall be no stronger than the success argument” [C++12, 29.6.5.20, p. 1113]. But Bastien and Boehm noted that the standard does not define the term “stronger” in this context, and also questioned whether there is even a point in restricting success/failure orderings [BB16]. Based on their proposal this requirement was therefore removed in C++17.

3.3.3 Fences

Another synchronization operation that can be used to establish a *happens-before* relation is a *fence* [C++12, 29.8, pp. 1116]. Just like the operations on atomics the `atomic_thread_fence` operation also takes a `memory_order` parameter and, depending on the order, has the following effects:

- has no effects,
  `if order == memory_order_relaxed`
- is an *acquire-fence*,
  `if order == memory_order_acquire || order == memory_order_consume`
- is a *release-fence*,
  `if order == memory_order_release`
- is both an *acquire-fence* and a *release-fence*,
  `if order == memory_order_acq_rel`
- is a sequentially consistent acquire- and release-fence,
  `if order == memory_order_seq_cst`

A release-fence A synchronizes with an acquire-fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would be heading if it were a release operation. Alternatively a release-fence can synchronize with a load-acquire on object M and an acquire-fence can synchronize with a store-release on object M, given that the same sequenced before relations between the fences and the corresponding operation are in place.

An adapted version of the previous example can be seen in Listing 3.3. The release-fence in line 5 is sequenced before the store operation in line 6 and the load operation in line 9 is sequenced before the acquire-fence in line 10. Therefore, when the load operation in line 9 returns the value written by the store operation in line 6, the acquire-fence *synchronizes with* the release-fence. From here the happens-before relation for the operations on x follows as already described in the previous example for the release/acquire operations in Listing 3.2.

Listing 3.3: Example of *synchronize-with* relation with release/acquire fences.

```
 1 std::atomic<int> x, y;
 2 
 3 // thread A
 4 x.store(1, std::memory_order_relaxed);
 5 std::atomic_thread_fence(std::memory_order_release);
 6 y.store(2, std::memory_order_relaxed);
 7 
 8 // thread B
 9 y.load(std::memory_order_relaxed);
10 std::atomic_thread_fence(std::memory_order_acquire);
11 x.load(std::memory_order_relaxed);
```

A `memory_order_seq_cst`-fence is not only both a *release* and an *acquire*-fence, but also provides some additional properties [C++12, 29.3.4-29.3.8]. They are also part of the single total order of all sequentially consistent operations, enforcing the following observations:

- For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.

- For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

- For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.

- For atomic operations A and B on an atomic object M, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B occurs later than A in the modification order of M.

CHAPTER 4

Implementation

Almost all the proposals for reclamation schemes come with a proof-of-concept implementation, but they can usually be described as follows: An implementation that is supposed to prove that the concept works, but not designed or implemented in a way that it would be suitable for use in a real life application. In contrast to that, I implemented some of the previously described schemes, using an abstract interface that allows data structures to be designed in a reclamation scheme agnostic way. The implementations are designed to work with an arbitrary number of threads and allow threads to be started and stopped arbitrarily and should be suitable for use in real world applications out of the box.

The following reclamation schemes were implemented:

- Lock-free reference counting (Section 4.4.1)
- Hazard pointers (Section 4.4.2)
- Epoch based reclamation (Section 4.4.3)
- New epoch based reclamation (Section 4.4.4)
- Quiescent state based reclamation (Section 4.4.5)
- Stamp-it (Section 4.4.6)

The reason why these schemes were chosen lies in the fact that they are generic—i.e., they do not have to be tailored to a specific data structure like, e.g., Drop the Anchor or Beware & Cleanup—and they are portable, i.e., they do not rely on OS or platform specific features such as POSIX signals or Hardware Transactional Memory like, e.g., ThreadScan, DEBRA+ or StackTrack.

The implementation is in C++ and makes use of the C++11 memory model as well as several other new language features that were introduced in C++11 and C++14. The complete source code including the scripts to run the benchmarks and analyze the results is available on GitHub `https://github.com/mpoeter/emr`.

The implementation of the reclamation schemes is based on a proposal for the C++ standard by Robison [Rob13]. I made a number of changes and adaptations compared to the original proposal; they are described in more detail in Section 4.1. Unfortunately, this proposal does not seem to have received much attention. Even though McKenney et al. pointed out the importance of this topic in [MWM16], they did not reference Robison's proposal. Instead they brought in a separate proposal to add hazard pointers to the C++ standard library [MW16]. I chose to implement Robison's proposal since it defines only an abstract interface and therefore allows a large number of different reclamation schemes to be implemented and used.

It defines the following fundamental abstractions:

- A *marked_ptr* allows one or more low-order bits to be borrowed. Many lock-free algorithms rely on such mark tricks, e.g., [Har01, Boe04, ST05].

- A *concurrent_ptr* acts like an atomic *marked_ptr*, i.e., it supports atomic operations.

- A *guard_ptr* is an object that can atomically take a snapshot of the value of a *concurrent_ptr* and if the target has not yet been deleted, guarantees that the target will not be deleted as long as the *guard_ptr* holds a pointer to it.

It is important to note that only *guard_ptr* references protect against deletion. In effect, a *concurrent_ptr* is a "weak" pointer and a *guard_ptr* is a "shared ownership" pointer, conceptually similar to `std::weak_ptr` and `std::shared_ptr` with the following key differences:

- *concurrent_ptr* and *guard_ptr* are abstract interfaces (a.k.a. "concepts"), not concrete interfaces.

- They support a snapshot operation that is conceptually similar to the `std::weak_ptr::lock()` method.

- A `std::weak_ptr` can indicate whether it has "expired", i.e., its target was deleted. A *concurrent_ptr* gives no such indication even if, as it can in some implementations, point to freed memory.

Similar to the *allocator* in existing containers of the standard library, the *reclaimer* is passed as a *policy* (i.e., as a template argument) to a concurrent container that requires usage of a reclamation scheme. A reclaimer type `R` has to define the following abstractions necessary for safe destruction and deletion:

- `R::concurrent_ptr<T>`: acts like an atomic markable pointer to objects of type `T`. It supports atomic operations such as `load`, `store`, and `compare_exchange_weak`. The class `T` must be derived from `enable_concurrent_ptr`.

- `R::enable_concurrent_ptr<T, N, D>`: defines a mandatory base class for targets of `concurrent_ptr`; `T` is the derived class, `N` is the number of mark bits supported, which defaults to zero, and `D` is the deleter type that should be used for all objects of type `T`.

- `R::region_guard`: allows some reclamation schemes to amortize the overhead; this is explained in more detail in Section 4.1.6.

The intent of `enable_concurrent_ptr<T, N, D>` is to provide implementers of reclaimers with two things:

- A way to force the alignment of targets, which is a common way to provide mark bits in the pointers.

- A place to embed reclaimer state, such as reference counts, in the user's objects.

The class `concurrent_ptr<T>` provides two auxiliary types:

- `concurrent_ptr<T>::marked_ptr` : Acts like a pointer, but has `N` mark bits, where `N` is specified by the base class `enable_concurrent_ptr<T, N>` of `T`.

- `concurrent_ptr<T>::guard_ptr` : Similar to a `marked_ptr`, but has shared ownership of its target *if* the target has not been deleted.

To obtain a snapshot from `concurrent_ptr` and populate a `guard_ptr` the `acquire` and `acquire_if_equal` methods can be used. In wait-free algorithms, `acquire` may be problematic with some schemes like hazard pointers, Pass The Buck, or even LFRC, because it may have to loop indefinitely. For these cases `acquire_if_equal` can be used as it simply stops trying if the value in variable `p` does not match the provided value in variable `m` and reports whether it was successful or not.

Releasing a `guard_ptr` follows the standard smart pointer interface. For a `guard_ptr` instance `g`, the operation `g.reset` releases ownership and sets `g` to `nullptr`; the destructor of `guard_ptr` implicitly calls `reset`.

In order to release a node, the `reclaim` method on a `guard_ptr` has to be called. This operation also resets the `guard_ptr`.

An example of these types and how they are used is shown in Listing 4.1.

Listing 4.1: Example how this interface is used.

```cpp
// Let's assume we have a type "Reclaimer" that implements this interface.

// Forward declaration of our node struct so we can use it in the following aliases.
struct node;

// Define a number of aliases for simpler code.
using concurrent_ptr = typename Reclaimer::template concurrent_ptr<node, 0>;
using marked_ptr = typename concurrent_ptr::marked_ptr;
using guard_ptr = typename concurrent_ptr::guard_ptr;

```

```
// We want to use our node with concurrent_ptr, so we have to derive
// it from enable_concurrent_ptr.
struct node : Reclaimer::template enable_concurrent_ptr<node>
{};

// Let's create a new node and store it in some publicly available concurrent_ptr.
marked_ptr new_node = new node();
concurrent_ptr cp;
cp.store(new_node);

// Acquire a guard to the node referenced by cp.
// This will protect the node from getting reclaimed as long as the guard_ptr exists.
guard_ptr guard;
guard.acquire(cp);

// Mark the node for reclamation. This will reset the guard_ptr and ensure that the
// node gets reclaimed once it is safe.
guard.reclaim();
```

More concrete examples can be found in the `list` and `queue` implementations in Section 4.5.1 and Section 4.5.2.

This interface does impose a few limitations. It is not possible to implement schemes that require data structure specific functions like, e.g., “Drop the anchor” (see Section 2.10) or “Beware & Cleanup” (see Section 2.6).

But there are also some data structures that cannot be implemented using this interface. One example is the doubly linked-list by Sundell [ST05], which uses a special reference counting scheme that also considers the internal links between the nodes. Such a scheme is not compatible with this interface.

4.1 Interface

My implementation follows the proposed interface from Robison [Rob13], but as it is stated in the paper it is only a “sketch, not a complete proposal”, so I made the following adaptations and corrections:

- I added move constructors and move assignment operators for `marked_ptr` and `guard_ptr` pointer types.
- I marked move constructor and move assignment operator for `concurrent_ptr` as deleted.
- I added the concept of `region_guard`; this is used by schemes like NEBR, QSBR and Stamp-it to amortize the overhead.
- The comments for `operator bool` for `marked_ptr` and `guard_ptr` state that it should return true if both the pointer and the mark bits are zero, which is the complete opposite of the definition for native pointers. Therefore I implemented it consistently with the behavior of native pointers and changed the documentation accordingly.

- The `is_lock_free` method for `marked_ptr` and `guard_ptr` are not necessary, since these classes are not designed to be used by multiple threads concurrently. So I removed them.
- I added a conversion operator to `guard_ptr` to allow implicit conversion of `guard_ptr` instances to `marked_ptr`.
- I adapted the interface of `concurrent_ptr` to be consistent with that of `std::atomic`:
  - I added `std::memory_order` parameter to the `load` method that defaults to `std::memory_order_seq_cst`.
  - I added `std::memory_order` parameter to both `store` methods that defaults to `std::memory_order_seq_cst`.
  - I adapted the `compare_exchange_weak` methods; the interface as originally proposed defined different overloads with all combinations of `guard_ptr` `/marked_ptr` instances for the two parameters `expected` and `desired`. However, the `compare_exchange_weak` method of `std::atomic` loads the actual value into `expected` when the comparison fails, but this would not be possible for `guard_ptr` instances. The new version therefore only uses `marked_ptr` for both parameters. The second parameter, which is passed by value, still accepts `guard_ptr` instances since they are now implicitly convertible to `marked_ptr`.
  - I added `compare_exchange_strong` methods.
- I removed the `Deleter` template parameter from `concurrent_ptr` and instead added it to `enable_concurrent_ptr`; the reason for this is explained in more detail in Section 4.1.1.
- I added a non-member function `acquire_guard` for easy inline initialization of `guard_ptr` variables.

4.1.1 The **`enable_concurrent_ptr`** class

Every reclaimer must define a class `enable_concurrent_pointer` that is used as mandatory base class for targets of `concurrent_ptr`. This base class does not only define the `number_of_mark_bits` and an alias for the `Deleter` for internal use in the reclaimer, but also allows to enforce alignment of instances or to store additional information like a reference counter. The minimal definition of such a class can be seen in Listing 4.2.

Listing 4.2: enable_concurrent_ptr

```
template <
  class T,
  std::size_t N = 0,
  class DeleterT = std::default_delete<T>>
struct enable_concurrent_ptr
{
  static constexpr std::size_t number_of_mark_bits = N;
  using Deleter = DeleterT;
};
```

`enable_concurrent_pointer` is a class template with the following template parameters:

`T` is the derived class; this is an application of the *curiously recurring template pattern* (CRTP) [Cop95].

`N` is the number of mark bits that a `marked_ptr` must reserve when used with this class; this parameter defaults to zero.

`DeleterT` is the deleter functor that shall be applied once an object can safely be reclaimed.

The class must define a member `number_of_mark_bits` that is set to `N` and a type alias `Deleter`.

In the original proposal the `Deleter` was not a parameter of `enable_concurrent_ptr` but of `concurrent_ptr`. However, all implemented reclamation schemes except LFRC collect the *to-be-reclaimed* nodes in some list in order to defer reclamation until a later time when it is safe to do so. Such a list can contain arbitrary nodes from different data structures, potentially using different deleters. The information which deleter shall be used must therefore be stored together with the node. But if this information is not already part of the node itself, it would require an additional memory allocation to store this information, even in cases where the deleter itself has no data members like `std::default_deleter`. In order to avoid this additional memory allocation I decided to move the `Deleter` parameter to the `enabled_concurrent_ptr` class, which allows to embed `Deleter` instances directly in the node.

The lists are implemented as simple singly linked lists. Since they can contain arbitrary nodes of different types, they have to derive from a common base class that contains the next pointer and also a pure virtual function that allows deletion of the node in consideration of custom deleters.

4.1.2 The `deletable_object` class

`deletable_object` (see Listing 4.3) is an internal helper class that is used by most of the reclamation schemes as the common base class for `enable_concurrent_ptr`.

Listing 4.3: deletable_object

```
struct deletable_object
{
  virtual void delete_self() = 0;
  deletable_object* next = nullptr;
protected:
  virtual ~deletable_object() = default;
};
```

The `next` pointer is used to build the single-linked list of *to-be-reclaimed* nodes. The pure virtual `delete_self` method is required because such lists contain only pointers to `deletable_object` instances, but the `Deleter` expects an instance of the derived type. Therefore, a derived class has to override `delete_self` and call the deleter with the appropriate parameter. To avoid duplication of this code for each reclamation scheme, I moved it to two different base classes and again made use of the *curiously recurring template pattern* to down-cast to the correct derived type which was handed down as template parameter. The two different base classes are `deletable_object_with_empty_deleter` and `deletable_object_with_non_empty_deleter`. I made this distinction because the size of an empty class is not zero, but one. Unconditionally storing a deleter instance as member in the object would therefore produce an unnecessary memory overhead that can be avoided this way. Instead of using these classes directly, I defined an alias `deletable_object_impl` that detects whether the given deleter is an empty class and resolves to the correct base class. The full listing of these classes can be found in Listing 4.4.

Listing 4.4: deletable_object implementations with empty and non-empty deleter

```
template <class Derived, class Deleter, class Base>
struct deletable_object_with_empty_deleter : Base
{
  virtual void delete_self() override
  {
    Deleter deleter{};
    deleter(static_cast<Derived*>(this));
  }

  void set_deleter(Deleter deleter) {}
};

template <class Derived, class Deleter, class Base>
struct deletable_object_with_non_empty_deleter : Base
{
  virtual void delete_self() override
  {
    Deleter& my_deleter = reinterpret_cast<Deleter&>(deleter_buffer);
    Deleter deleter(std::move(my_deleter));
    my_deleter.~Deleter();
    deleter(static_cast<Derived*>(this));
  }

  void set_deleter(Deleter deleter)
  {
    reinterpret_cast<Deleter&>(deleter_buffer) = std::move(deleter);
  }
private:
  char deleter_buffer[sizeof(Deleter)];
};

template <
  class Derived,
  class Deleter,
  class Base = deletable_object
>
using deletable_object_impl = std::conditional_t<
    std::is_empty<Deleter>::value,
    deletable_object_with_empty_deleter<Derived, Deleter, Base>,
    deletable_object_with_non_empty_deleter<Derived, Deleter, Base>
  >;
```

`deletable_object_with_non_empty_deleter` holds a `char` array as buffer for a `Deleter` instance. When a deleter is set using the `set_deleter` method the deleter instance is *moved* into the buffer. This allows the `Deleter` type to be non-default-constructible.

The template parameter `Base` defaults to `deletable_object` which should be fine for most of the cases. In case there are special requirements this parameter can be used to define a custom base class as long as it defines the same interface as `deletable_object`. This is used for example in the implementation of Stamp-it because this scheme requires that the base class has an additional field to store the stamp value from the time it was marked for deletion.

4.1.3 The **`marked_ptr`** class

Many lock-free algorithms rely on the ability to store special flags in a pointer (e.g., [Har01, Boe04, ST05]). The `marked_ptr` class defines a high-level interface to a pointer of which a number of low-order bits can be borrowed to store additional information. The number of bits that shall be used for marking can be defined via a template parameter. A complete listing of the implementation of `marked_ptr` can be found in Listing 4.5.

Runtime assertions ensure that the specified mark value does not use more bits than reserved, as well as that the pointer value does not occupy bits that are reserved for marking.

Listing 4.5: Interface of marked_ptr

```
template <class T, std::size_t N>
class marked_ptr {
public:
  // Construct a marked ptr
  marked_ptr(T* p = nullptr, uintptr_t mark = 0) noexcept {
    assert(mark <= MarkMask && "mark exceeds the number of bits reserved");
    assert((reinterpret_cast<uintptr_t>(p) & MarkMask) == 0 &&
      "bits reserved for masking are occupied by the pointer");
    ptr = reinterpret_cast<T*>(reinterpret_cast<uintptr_t>(p) | mark);
  }

  // Set to nullptr
  void reset() noexcept { ptr = nullptr; }

  // Get mark bits
  uintptr_t mark() const noexcept {
    return reinterpret_cast<uintptr_t>(ptr) & MarkMask;
  }

  // Get underlying pointer (with mark bits stripped off).
  T* get() const noexcept {
    return reinterpret_cast<T*>(reinterpret_cast<uintptr_t>(ptr) & ~MarkMask);
  }

  // True if get() != nullptr || mark() != 0
  explicit operator bool() const noexcept { return ptr != nullptr; }

  // Get pointer with mark bits stripped off.
  T* operator->() const noexcept { return get(); }

  // Get reference to target of pointer.
  T& operator*() const noexcept { return *get(); }

  inline friend bool operator==(const marked_ptr& l, const marked_ptr& r) {
    return l.ptr == r.ptr; }
  inline friend bool operator!=(const marked_ptr& l, const marked_ptr& r) {
    return l.ptr != r.ptr; }

  static constexpr std::size_t number_of_mark_bits = N;
private:
  static constexpr uintptr_t MarkMask = (1 << N) - 1;
  T* ptr;
};
```

4.1.4 The `concurrent_ptr` class

A `concurrent_ptr` is basically an atomic `marked_ptr`. For consistence, I adapted it so that it defines the same interface as `std::atomic`. In addition it defines aliases for the reclaimer's `marked_ptr` and `guard_ptr` types. A complete listing of the interface can be found in Listing 4.6.

Listing 4.6: Interface of concurrent_ptr

```
//! T must be derived from enable_concurrent_ptr<T>. D is a deleter.
template <
  class T,
  std::size_t N,
  template <class, std::size_t> class MarkedPtr,
  template <class T2, class MarkedPtrT, class Deleter> class GuardPtr,
  class DefaultDelete = std::default_delete<T>
>
class concurrent_ptr {
public:
  struct marked_ptr : MarkedPtr<T, N> {};

  template <class D = DefaultDelete>
  using guard_ptr = GuardPtr<T, marked_ptr, D>;

  concurrent_ptr(const marked_ptr& p = marked_ptr()) noexcept : ptr(p) {}
  concurrent_ptr(const concurrent_ptr&) = delete;
  concurrent_ptr(concurrent_ptr&&) = delete;
  concurrent_ptr& operator=(const concurrent_ptr&) = delete;
  concurrent_ptr& operator=(concurrent_ptr&&) = delete;

  // Atomic load that does not guard target from being reclaimed.
  marked_ptr load(std::memory_order order = std::memory_order_seq_cst) const;

  // Atomic store.
  void store(const marked_ptr& src,
             std::memory_order order = std::memory_order_seq_cst);

  // Shorthand for store (src.get())
  template <class D>
  void store(const guard_ptr<D>& src,
             std::memory_order order = std::memory_order_seq_cst);

  bool compare_exchange_weak(marked_ptr& expected, marked_ptr desired,
    std::memory_order order = std::memory_order_seq_cst);
  bool compare_exchange_weak(marked_ptr& expected, marked_ptr desired,
    std::memory_order order = std::memory_order_seq_cst) volatile;
  bool compare_exchange_weak(marked_ptr& expected, marked_ptr desired,
    std::memory_order success, std::memory_order failure);
  bool compare_exchange_weak(marked_ptr& expected, marked_ptr desired,
    std::memory_order success, std::memory_order failure) volatile;

  bool compare_exchange_strong(marked_ptr& expected, marked_ptr desired,
    std::memory_order order = std::memory_order_seq_cst);
  bool compare_exchange_strong(marked_ptr& expected, marked_ptr desired,
    std::memory_order order = std::memory_order_seq_cst) volatile;
  bool compare_exchange_strong(marked_ptr& expected, marked_ptr desired,
    std::memory_order success, std::memory_order failure);
  bool compare_exchange_strong(marked_ptr& expected, marked_ptr desired,
    std::memory_order success, std::memory_order failure) volatile;
};
```

4.1.5 The **guard_ptr** class

A `guard_ptr` is basically a `marked_ptr` that protects the object that it points to from being reclaimed. In that sense it is conceptually similar to a `std::shared_ptr`. In contrast to the other two pointer types, `guard_ptr` implementations are very specific to the concrete reclamation scheme. A complete listing of the interface can be found in Listing 4.7.

Listing 4.7: Interface of guard_ptr

```
template <class T>
class guard_ptr
{
  using Deleter = typename T::Deleter;
public:
  guard_ptr() noexcept;
  ~guard_ptr();

  // Guard a marked ptr.
  guard_ptr(const marked_ptr& p);

  explicit guard_ptr(const guard_ptr& p);
  guard_ptr(guard_ptr&& p) noexcept;

  guard_ptr& operator=(const guard_ptr& p) noexcept;
  guard_ptr& operator=(guard_ptr&& p) noexcept;

  // Get underlying pointer
  T* get() const noexcept;

  // Get mark bits
  uintptr_t mark() const noexcept;

  // Support implicit conversion from guard_ptr to marked_ptr.
  operator marked_ptr() const noexcept;

  // True if get() != nullptr || mark() != 0
  explicit operator bool() const noexcept;

  // Get pointer with mark bits stripped off. Undefined if target has been reclaimed.
  T* operator->() const noexcept;

  // Get reference to target of pointer. Undefined if target has been reclaimed.
  T& operator*() const noexcept;

  // Swap two guards.
  void swap(guard_ptr& g) noexcept;

  // Atomically take snapshot of p, and *if* it points to unreclaimed object,
  // acquire shared ownership of it.
  void acquire(concurrent_ptr<T>& p,
               std::memory_order order = std::memory_order_seq_cst);

  // Like acquire, but quit early if p != expected.
  bool acquire_if_equal(concurrent_ptr<T>& p,
                        const marked_ptr& expected
                        std::memory_order order = std::memory_order_seq_cst);

  // Release ownership. Postcondition: get() == nullptr.
  void reset() noexcept;

  // Reset. Deleter d will be applied some time after all owners release their
    ownership.
  void reclaim(Deleter d = Deleter()) noexcept;
};
```

A guard_ptr has to be acquired by the methods acquire or acquire_if_equal to ensure that the guard_ptr holds a safe reference that protects the object from being reclaimed. These methods take a snapshot of the value of a concurrent_ptr and store a safe reference to the object in the guard_ptr if the target has not yet been deleted.

In wait-free algorithms `acquire` may be problematic when implemented with hazard pointers or Pass The Buck, because it may have to loop indefinitely in order to acquire a safe reference. In these cases `acquire_if_equal` can be used as it simply stops trying if the value in `p` does not match a provided value `m` and reports whether it was successful or not.

Both methods, `acquire` and `acquire_if_equal` take an optional `memory_order` parameter that defines the order of the read operation on the `concurrent_ptr` object `p`. The default value is `memory_order_seq_cst`.

Releasing a `guard_ptr` follows the standard smart pointer interface; the operation `g.reset` releases ownership and sets `g` to `nullptr`, the destructor of `guard_ptr` implicitly calls reset.

The `reclaim` method resets the `guard_ptr` and marks the node for deletion.

One important limitation of `guard_ptrs` is that they must not be moved between threads, i.e., move construction or move assignment operations must not be used to transfer ownership of a `guard_ptr` from one thread to another. The reason for this simply is that these move operations are optimized based on the assumption that both, the target and the source operands, belong to the same thread. However, copy construction and copy assignment do not suffer from this limitation.

Reimplementing the whole interface for every reclamation scheme would result in a lot of code duplication as many methods like `get`, `mark` and `swap` as well as all the operators would be identical. For that reason I introduced a base class using the *curiously recurring template pattern* (CRTP) [Cop95][1] that already provides implementations for all these methods and operators (see Listing 4.8). The CRTP is used for forwarding to scheme specific implementations of methods like `reset`, which is required in the destructor and in `swap`. The `swap` method is implemented by swapping the two underlying `marked_ptr` values and then calling `do_swap` on the derived class to allow implementation of scheme specific behavior. For schemes where this is not needed `do_swap` is already implemented as an empty dummy function in the base class.

Listing 4.8: guard_ptr base class

```cpp
template <class T, class MarkedPtr, class Derived>
class guard_ptr {
public:
  ~guard_ptr() { self().reset(); }

  T* get() const noexcept { return ptr.get(); }
  uintptr_t mark() const noexcept { return ptr.mark(); }

  operator MarkedPtr() const noexcept { return ptr; }
  explicit operator bool() const noexcept { return static_cast<bool>(ptr); }

```

[1]The “curiously recurring template pattern” is a C++ idiom in which a class `X` derives from a class template instantiation using itself as template argument. The purpose of doing this is to use the derived class in the base class. From the perspective of the base object, the derived object is itself, but downcasted. Therefore, the base class can access the derived class by performing a `static_cast` on itself using the given template parameter.

```
  T* operator->() const noexcept { return ptr.get(); }
  T& operator*() const noexcept { return *ptr; }

  // Swap two guards
  void swap(Derived& g) noexcept
  {
    std::swap(ptr, g.ptr);
    self().do_swap(g);
  }

protected:
  guard_ptr(const MarkedPtr& p = MarkedPtr{}) noexcept : ptr(p) {}
  MarkedPtr ptr;

  void do_swap(Derived& g) noexcept {} // empty dummy

private:
  Derived& self() { return static_cast<Derived&>(*this); }
};
```

4.1.6 The **`region_guard`** class

A `region_guard` is an additional concept that I introduce, because it is required for reclamation schemes like NEBR, QSBR and Stamp-it. In these schemes a `guard_ptr` can only exist inside a *critical region*, so unless the thread is already inside a critical region the `guard_ptr` automatically enters a new one. Entering and leaving critical regions are usually rather expensive operations, but `region_guard`s allow to amortize this overhead. The constructor of a `region_guard` enters a new region (unless the thread is already inside one) and leaves the region in the destructor. Any `guard_ptr` instances that are created inside the scope of the `region_guard` can simply use the current critical region and save the overhead of entering a new one.

The `region_guard` class does not define any member functions, it only uses the RAII[2] concept to leave the region upon destruction.

In order to provide a consistent interface every reclamation scheme has to define a `region_guard` class regardless of whether the scheme actually supports this concept. For reclamation schemes that do not support it, it is sufficient to define an empty `region_guard` class.

4.2 Correctness

The main task of a reclamation scheme is to ensure that a node only gets reclaimed when it is guaranteed that no thread is holding a reference to it. The respective publications of the various schemes usually contain correctness proofs. Like most publications, however, they usually assume a sequentially consistent memory model. In my implementation

[2]Resource Acquisition Is Initialization: `http://en.cppreference.com/w/cpp/language/raii`

I tried to relax all memory operations as much as possible, so based on the semantics of the C++11 memory model it has to be shown that the required properties are still fulfilled.

Without sequential consistency, there is another important aspect that has to be considered. It must be ensured that all changes made to a node are visible to the thread that reclaims that node, i.e., there must be a happens-before relation between the reclaim operation of the node and any changes made to that node. Since a thread can only access a node while it is holding a `guard_ptr`, there is a happens-before relation between any changes made to the node and the subsequent `reset` of the `guard_ptr`. What remains to be shown is that there is also a happens-before relation between the `reset` and the node's reclamation.

I will not provide complete formal proofs for the implementations of all the schemes and data structures, but I provide arguments about the correctness of key functions, taking into account possible thread interactions. In particular I will show that the implementations fulfill the just described key properties for all reclamation schemes.

In the source code all the relevant atomic operations have a comment with a number and a short explanation, an example is shown in Listing 4.9. The acquire-load operation in this example has the number 1 and it synchronizes-with one of the release-CAS operations with the numbers 3, 5 or 8, depending on the value read.

Listing 4.9: Example for atomic operation with comment.

```
// (1) - this acquire-load synchronizes-with the release-CAS (3, 5, 8).
t.load(tail, std::memory_order_acquire);
```

In the correctness arguments the atomic operations are presented like this:

$$t_1\texttt{: p.load}^{(1)}_{acq}$$

This denotes that thread t_1 performs a `load` operation with `memory_order_acquire` on object `p`. The superscript "(1)" denotes that this operation has the number 1 in the source code. This allows an easy mapping of the operations in the arguments to the actual source code and vice versa. When a sequence of operations is executed by a single thread t_x, the prefix t_x is only shown for the first operation, for the remaining operations in the sequence it is omitted to reduce clutter. For CAS operations the second memory order (if used) is also listed in the subscript.

For the operations' memory orderings the following abbreviations are used in the subscripts:

rlx	— `memory_order_relaxed`
rel	— `memory_order_release`
acq	— `memory_order_acquire`
ar	— `memory_order_acq_rel`
sc	— `memory_order_seq_cst`

For the relations between two operations the following nomenclature is used:

$a \xrightarrow{\text{sb}} b$	– denotes that a is *sequenced-before* b.
$a_{rel} \xrightarrow{\text{sw}} b_{acq}$	– denotes that acquire operation b *synchronizes-with* the release operation a.
$a \xrightarrow{\text{hb}} b$	– denotes that a *happens-before* b.
$a \xrightarrow{\text{rf}} b$	– denotes that b *reads-from* a, i.e., the value load by operation b is the value stored by operation a.
$o.a \xrightarrow{\text{mo}} o.b$	– denotes that a and b are both operations on object o where a is ordered before b in the *modification-order* of object o.
$a \xrightarrow{\text{sco}} b$	– denotes that a and b are both sequentially consistent operations and a is ordered before b in the single total order of all sequentially consistent operations.
$a_{rel}[b, c, ...]$	– denotes that a is an operation with release semantic acting as head of a release sequence that may contain an arbitrary number of the operations b, c etc. in an arbitrary order. An acquire-load operation l that reads any value written by one of these operations (i.e., $x \xrightarrow{\text{rf}} l$, where $x \in \{a, b, c...\}$) will therefore synchronize-with a.

In C++ CAS operations are called `compare_exchange`, but to save some space they are written as `cmpxchg` in these annotations.

4.3 Helper classes

A number of helper classes have been implemented which are shared by several different reclamation schemes. Some of the more important ones are described in the next sections.

4.3.1 The `thread_block_list` class

This class is used by all schemes except LFRC to manage a number of entries, where each thread that participates in the scheme owns exactly one such entry. Each scheme defines a class `thread_control_block` that is passed to `thread_block_list` as template argument and defines the type of the list's entries. The only requirement for these types is that they have to derive from `thread_block_list::entry`. Each scheme thus defines a global `thread_block_list` instance that manages all the `thread_control_block` instances.

Unfortunately, the memory reclamation problem also applies to the entries of this list, and since the list is part of the reclamation scheme itself, we cannot rely on the scheme to solve it. For that reason an entry that has been added to the list cannot be freed and has to exist indefinitely. However, when a terminating thread releases its entry it gets marked as unused, allowing newly started threads to reuse this entry. The total number of entries is therefore bounded by the maximum number of concurrently running threads.

In addition the data structure also contains a linked list of *abandoned retired nodes*. This is used to solve the problem that occurs when a thread terminates but still holds some retired nodes that cannot yet be reclaimed. The idea is that the terminating thread *abandons* these nodes by adding them to a linked list in the global `thread_block_list`. Other threads regularly check whether this list contains any entries and if that is the case try to *adopt* them, thus making sure that all nodes will be reclaimed eventually.

The `thread_block_list` class defines two nested classes:

- `iterator` – an implementation of the standard C++ forward iterator concept that allows traversing the list.
- `entry` – this is the base class for all list entries. It contains the `in_use` flag and the `next_entry` pointer.

In addition it provides the following methods:

- `acquire_entry` – tries to adopt an existing but abandoned entry from the list or, if this fails, creates a new entry and adds it to the list. The return value is a reference to the adopted/created instance.
- `release_entry` – simply marks the entry as abandoned so that other threads can pick it up.
- `abandon_retired_nodes` – adds the given nodes to the linked list of abandoned retired nodes.
- `adopt_abandoned_retired_nodes` – checks if there are abandoned nodes and tries to adopt them.
- `begin` and `end` – return iterators that allow traversing the list entries.

The respective reclamation schemes are responsible to ensure that newly started threads are handled correctly when iterating the `thread_block_list` as the list's implementation uses only acquire/release semantic and thus does not guarantee that newly added threads are already visible when loading the `begin` iterator.

4.3.2 The **`aligned_object`** class

The default `operator new` uses an allocator that only guarantees fundamental alignment. Improved versions that also allow allocation of over-aligned data, i.e., data with alignment greater than the default alignment which usually corresponds to the system's native pointer size, are in discussion for C++17 [Nel16]. The `aligned_object` class (shown in Listing 4.10) defines `operator new` and `operator delete` to allow dynamic allocation of objects with custom alignment. A class with special alignment requirements can simply inherit from `aligned_object`, passing itself as template argument. The appropriate alignment is either obtained via `std::alignment_of` or it can be explicitly passed as template argument.

Listing 4.10: aligned_object

```cpp
template <typename Derived, std::size_t Alignment = 0>
struct aligned_object
{
  static void* operator new(size_t sz)
  {
    return boost::alignment::aligned_alloc(
      Alignment == 0 ? std::alignment_of<Derived>() : Alignment, sz);
  }

  static void operator delete(void* p)
  {
    boost::alignment::aligned_free(p);
  }
};
```

The `Alignment` template parameter defaults to zero instead of `std::alignment_of<Derived>` because when a class derives from `aligned_object` and passes itself as template argument, the class is not yet fully defined and trying to get the alignment of an incomplete type results in a compiler error.

4.4 Reclamation schemes

The implementation of the reclamation schemes is not tailored to the benchmarks, but is kept as generic as possible so that they can actually be used in real-world application. All schemes (except LFRC) support the use of custom deleters and they are all designed to work with an arbitrary number of threads that can be spawned and stopped arbitrarily. The number of threads can be dynamic and does not have to be known at compile time.

The description of the reclamation schemes focuses on the implementation details of `enable_concurrent_ptr`, `guard_ptr`, `region_guard` (if appropriate) and other classes that are specific to the respective scheme.

4.4.1 Lock-Free Reference Counting

The implementation of *lock-free reference counting* takes the following template parameters:

- `InsertPadding` – if set to true, additional padding bytes are inserted after the members of `enable_concurrent_ptr` to ensure that the members of any inherited class are in a different cache line and therefore no false sharing can appear. The default value is false.

- `ThreadLocalFreeListSize` – defines the number of elements per thread used in the local free list buffer to reduce contention on the global free list. The default value is zero, i.e., local free lists are completely deactivated.

Michael [Mic04a] and Hart et al. [HMBW07] have evaluated the performance of LFRC in comparison to other reclamation schemes and all the results showed rather bad performance for LFRC. However, I have not found any publications on possible performance improvements for LFRC. I have introduced these two parameters to evaluate their impact on the performance; a detailed analysis of the results can be found in Section 5.1.

4.4.1.1 The `enable_concurrent_ptr` class

In a first attempt I tried to embed the `ref_count` as a normal member directly in the class. Unfortunately, this approach had several problems that were all related to the fact that the `ref_count` of an object has to remain indefinitely and must not be altered during construction or destruction of an object. In this first attempt the `ref_count` was initialized once when the memory was allocated in `operator new`. Subsequent calls to `operator new` that pop the node from the free list must maintain the integrity of `ref_count` and not change its value. However, *value-initialization* of an object[3] can cause *zero-initialization* under certain circumstances[4], i.e., the whole memory of the object gets zeroed out before the default constructor is called, therefore invalidating the `ref_count`.

While this behavior is highly unintuitive, there are ways to avoid it that would have to be carefully documented in detail. But unfortunately it gets even worse—according to the C++ standard memory returned by `operator new` has "indeterminate value"[C++12][8.5.12, p. 191]. Therefore, every operation in `operator new` that writes to the newly allocated memory can be considered a *dead store* and can therefore be optimized away. The GCC 6 release contains a more aggressive dead-store elimination specifically targeted for these cases[5]. And not only is the memory *before object construction* defined to be indeterminate, but the same goes for the memory *after destruction*. Relying on the integrity of the `ref_count` after object destruction is therefore undefined behavior.

To keep a long story short: It is simply not possible to implement this scheme in a standard conformable way with `ref_count` being a normal class member. For that reason I switched to a different approach, in which the `ref_count` is moved to a separate header structure that is located in memory directly before the node.

The `enable_concurrent_ptr` class definition is shown in Listing 4.11. Its only data member is the `next_free` pointer that is required by the free list. Aside from that it defines custom `operator new` and `operator delete`. There are also two nested classes: `unpadded_header` contains the `ref_count` and a flag that is required to avoid calling the destructor on an already destroyed object; this is explained later in more detail. `padded_header` inherits from `unpadded_header` and only adds an additional padding member to ensure that the `ref_count` does not share a cacheline with the actual node that follows immediately in memory. Depending on whether `InsertPadding` is set, `header` is defined as an alias for `unpadded_header` or `padded_header`.

[3]An object gets *value-initialized* when the initializer is an empty set of parentheses [C++12][8.5.11, p. 191].

[4]The details on when *value-initialization* actually causes *zero-initialization* are rather complicated. For those who are really interested in the gory details see [C++12][8.5.8, p. 191].

[5]`https://gcc.gnu.org/gcc-6/porting_to.html#flifetime-dse`

The global free-list is implemented as a nested class of `enable_concurrent_ptr` (see Section 4.4.1.2).

Listing 4.11: LFRC's enable_concurrent_ptr

```cpp
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, std::size_t N, class DeleterT>
class lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
    enable_concurrent_ptr
{
protected:
  enable_concurrent_ptr() noexcept {
    destroyed().store(false, std::memory_order_relaxed); }
  enable_concurrent_ptr(const enable_concurrent_ptr&) noexcept = delete;
  enable_concurrent_ptr(enable_concurrent_ptr&&) noexcept = delete;
  enable_concurrent_ptr& operator=(const enable_concurrent_ptr&) noexcept = delete;
  enable_concurrent_ptr& operator=(enable_concurrent_ptr&&) noexcept = delete;
  virtual ~enable_concurrent_ptr() noexcept {
    destroyed().store(true, std::memory_order_relaxed); }
public:
  using Deleter = DeleterT;
  static_assert(std::is_same<Deleter, std::default_delete<T>>::value,
                "lock_free_ref_count reclamation can only be used with "
                "std::default_delete as Deleter.");

  static constexpr std::size_t number_of_mark_bits = N;
  unsigned refs() const {
    return getHeader()->ref_count.load(std::memory_order_relaxed) >> 1; }

  void* operator new(size_t sz);
  void operator delete(void* p);

private:
  bool decrement_refcnt();
  bool is_destroyed() const {
    return getHeader()->destroyed.load(std::memory_order_relaxed); }
  void push_to_free_list() { global_free_list.push(static_cast<T*>(this)); }

  struct unpadded_header {
    std::atomic<unsigned> ref_count;
    std::atomic<bool> destroyed;
  };
  struct padded_header : unpadded_header {
    char padding[64 - sizeof(unpadded_header)];
  };
  using header = std::conditional_t<InsertPadding, padded_header, unpadded_header>;
  header* getHeader() { return static_cast<header*>(static_cast<void*>(this)) - 1; }
  const header* getHeader() const {
    return static_cast<const header*>(static_cast<const void*>(this)) - 1; }

  std::atomic<unsigned>& ref_count() { return getHeader()->ref_count; }
  std::atomic<bool>& destroyed() { return getHeader()->destroyed; }
  concurrent_ptr<T, N> next_free;

  friend class lock_free_ref_count;

  using guard_ptr = typename concurrent_ptr<T, N>::guard_ptr;
  using marked_ptr = typename concurrent_ptr<T, N>::marked_ptr;

  class free_list;
  static free_list global_free_list;
};
```

The constructor sets the `destroyed` flag to `false` to signal that this is a freshly initialized object that has not yet been destroyed, while the destructor sets `destroyed` to true to signal the opposite.

The LSB of `ref_count` is used as *claim-bit* to ensure that an object is only added to the free-list by a single thread. Therefore two constants `RefCountInc` and `RefCountClaimBit` are defined for updating the `ref_count`. For newly created objects `ref_count` is always initialized to `RefCountInc`, i.e., the reference count is 1.

For objects that are taken from the free-list the `ref_count` is incremented and the *claim-bit* is reset to zero.

The `operator new` (shown in Listing 4.12) first tries to pop an element from the global free-list and if that fails calls `::operator new` to allocate memory for a new node plus the leading `header`. For this newly allocated memory the `ref_count` gets initialized to `RefCountInc` and the pointer gets increased by `sizeof(header)` to get the actual node pointer.

Listing 4.12: LFRC's operator new

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, std::size_t N, class Deleter>
void* lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  enable_concurrent_ptr<T, N, Deleter>::operator new(size_t sz)
{
  T* result = global_free_list.pop();
  if (result == nullptr)
  {
    auto h = static_cast<header*>(::operator new(sz + sizeof(header)));
    h->ref_count.store(RefCountInc, std::memory_order_relaxed);
    result = static_cast<T*>(static_cast<void*>(h + 1));
  }

  return result;
}
```

The `operator delete` (shown in Listing 4.13) calls `decrement_refcnt` to reduce the `ref_count` and only if it returns `true`, i.e., the *claim-bit* was set successfully, it can continue and push the object on the global free-list.

Listing 4.13: LFRC's operator delete

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, std::size_t N, class Deleter>
void lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  enable_concurrent_ptr<T, N, Deleter>::operator delete(void* p)
{
  auto node = static_cast<T*>(p);
  if (node->decrement_refcnt())
    node->push_to_free_list();
}
```

When an `operator delete` is called on an object, the compiler first calls the destructor on that object before the actual `operator delete` is called. Suppose a new node gets allocated and immediately destroyed by manually calling delete on it. This would first call the destructor and subsequently `operator delete`, which in turn calls `decrement_refcnt`. However, it is not guaranteed that this call to `decrement_refcnt` brings the `ref_count` down to zero, since it could be that another thread has also gathered a reference to this node when they were competing on getting a node from the free-list. Eventually the second thread will also release its reference, at which point `ref_count` might drop to zero, so the second thread would now be responsible for reclaiming the node. However, it must not call delete on the node since this would lead to the execution of the destructor on an already destroyed object—a direct path to the land of undefined behavior. That is why we need a flag to determine whether the node already got destroyed or not. Listing 4.18 shows how this is handled in `guard_ptr::reset`.

4.4.1.2 The `free_list` class

The `free_list` (see Listing 4.14) consists of a singly-linked list as well as a thread-local free-list that can be activated optionally. It contains only two public methods that allow to `push` or `pop` nodes.

Listing 4.14: LFRC's free_list

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, std::size_t N, class Deleter>
class lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  enable_concurrent_ptr<T, N, Deleter>::free_list
{
public:
  T* pop();
  void push(T* node);

private:
  void add_nodes(T* first, T* last);

  // the free list is implemented as a FILO single linked list
  // the LSB of a node's ref_count acts as claim bit, so for all nodes on the free
    list the bit has to be set
  concurrent_ptr<T, N> head;

  class thread_local_free_list
  {
  public:
    ~thread_local_free_list() noexcept;
    bool push(T* node);
    T* pop();
  private:
    size_t number_of_elements = 0;
    T* head = nullptr;
  };

  static constexpr size_t max_local_elements = ThreadLocalFreeListSize;
  static thread_local_free_list& local_free_list()
  {
    // workaround for gcc issue causing redefinition of __tls_guard when
    // defining this as static thread_local member of free_list.
    alignas(64) static thread_local thread_local_free_list local_free_list;
    return local_free_list;
  }
};
```

The `thread_local_free_list` is a simple singly-linked list with a bounded number of entries. Since its `push` and `pop` methods operate on thread-local objects, they are straight forward and are therefore omitted here. When a thread terminates it calls the destructor of its `thread_local_free_list` which adds all remaining nodes (if any) to the global free list, so that they are not leaked. Unfortunately defining a destructor for `thread_local_free_list` causes an issue with gcc complaining about a redefinition of `__tls_guard` (the other definition stems from one of the other implemented schemes) when `local_free_list` is defined as a simple `static thread_local` member. As workaround for this issue a static function with a `static thread_local` variable is used instead.

When a thread terminates, any remaining nodes from the local free-list are added to the global free-list so that they are not leaked. This is done using the `add_nodes` method (shown in Listing 4.15). It takes two pointers to nodes that must build a singly-linked list in which `first` points to the first node and `last` to the last node of the list, i.e., the `next_free` links must form a path from `first` to `last`. In order to add the given nodes to the global free-list, `add_nodes` first loads the `head` pointer, stores its value in the `last`'s `next_free` pointer and then performs a CAS operation in an attempt to update `head` to `first`. The load operation uses acquire semantics to synchronize it with the release-CAS operation performed by previous `add_nodes` operations. The same goes for the CAS operation if the comparison fails and it has to perform a reload.

Listing 4.15: LFRC's free_list::add_nodes

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, std::size_t N, class Deleter>
void lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  enable_concurrent_ptr<T, N, Deleter>::free_list::add_nodes(T* first, T* last)
{
  // (2) - this acquire-load synchronizes-with the release-CAS (3)
  auto old = head.load(std::memory_order_acquire);
  do {
    last->next_free.store(old, std::memory_order_relaxed);
    // (3) - if this release-CAS succeeds, it synchronizes-with acquire-loads (1, 2)
    //       if it failes, the reload synchronizes-with itself (3)
  } while (!head.compare_exchange_weak(old, first,
                                       std::memory_order_release,
                                       std::memory_order_acquire));
}
```

The `free_list::push` operation is shown in Listing 4.16. It first checks whether local free-lists are enabled, i.e., if `max_local_elements` is greater zero, and if so tries to push the node onto the local free-list. Since `max_local_elements` is a compile time constant this whole if-statement will be optimized away when local free-lists are disabled by setting this value to zero. Otherwise, the node is added to the global free-list. For this we simply reuse the previously described `add_nodes` method and treat our node as a singly-linked list with a single entry.

Listing 4.16: LFRC's free_list::push

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, std::size_t N, class Deleter>
void lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  enable_concurrent_ptr<T, N, Deleter>::free_list::push(T* node)
{
  if (max_local_elements > 0 && local_free_list().push(node))
    return;

  add_nodes(node, node);
}
```

The `free_list::pop` operation (shown in Listing 4.17) is more complicated. First it also checks whether local free-lists are enabled, and if that is the case it tries to pop a node from the local free-list, otherwise it tries to take a node from the global free-list. Therefore it acquires a `guard_ptr` for the `head`. This is necessary to prevent the node from getting re-added to the list as long as the operation has not finished, effectively avoiding the ABA problem. The `acquire_guard` operation uses acquire semantics in order to synchronize with the release-CAS operation in `push`. Then it tries to remove the first entry by updating `head` with the `next_free` value. This operation uses relaxed semantics since it is a read-modify-write operation that, in case of success, always comes after a successful CAS operation in `push` in the modification order of `head`, and as such it is part of a release sequence. Any operation that performs an acquire-load on `head` that "sees" the value written by this CAS therefore synchronizes with the CAS that heads the release sequence (i.e., the CAS in `push`). This ensures the necessary happens-before relation between a `push` and a subsequent `pop` of the same node.

After the entry was removed successfully, the `ref_count` is updated to clear the `ClaimBit` and increase the reference count.

Listing 4.17: LFRC's free_list::pop

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, std::size_t N, class Deleter>
T* lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  enable_concurrent_ptr<T, N, Deleter>::free_list::pop()
{
  if (max_local_elements > 0)
    if (auto result = local_free_list().pop())
      return result;

  guard_ptr guard;
  while (true)
  {
    // (1) - this acquire-load synchronizes-with the release-CAS (3)
    guard = acquire_guard(head, std::memory_order_acquire);
    if (guard.get() == nullptr)
      return nullptr;

    // Note: ref_count can be anything here since multiple threads
    // could have gotten a reference to the node on the freelist.
    marked_ptr expected(guard);
    // since head is only changed via CAS operations it is sufficient to use
    // relaxed order for this operation as it is always part of a release-sequence
    // headed by (3)
    if (head.compare_exchange_weak(
      expected,
      guard->next_free.load(std::memory_order_relaxed),
      std::memory_order_relaxed))
    {
      auto ptr = guard.get();
      // clear claim bit
      ptr->ref_count().fetch_sub(RefCountClaimBit, std::memory_order_relaxed);
      ptr->next_free.store(nullptr, std::memory_order_relaxed);
      guard.ptr.reset(); // reset guard_ptr to prevent decrement of ref_count
      return ptr;
    }
  }
}
```

4.4.1.3 The `guard_ptr` class

The `guard_ptr` protects a shared object by ensuring that, as long as the `guard_ptr` holds a reference to that object, its `ref_count` cannot drop to zero. To achieve this the `guard_ptr` increments the `ref_count` of the object when:

- the `guard_ptr` is constructed with a `marked_ptr`
- the `acquire` or `acquire_if_equal` method is called to acquire a safe reference from a `concurrent_ptr`
- the `guard_ptr` gets copy constructed or copy assigned

Move construction and move assignment does not alter the `ref_count` as it simply moves ownership and clears the source `guard_ptr`.

The `ref_count` gets decremented when a `guard_ptr` gets `reset`; the destructor implicitly calls `reset` in case it holds a valid reference. The implementation of `reset` (shown in

Listing 4.18) first calls `decrement_refcnt`. If it returns true, i.e., if the claim bit was set successfully, it checks if the object has already been destroyed and if not explicitly calls the destructor. Only then the object is pushed onto the free-list.

Listing 4.18: LFRC's guard_ptr::reset

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, class MarkedPtr>
void lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  guard_ptr<T, MarkedPtr>::reset() noexcept
{
  auto p = this->ptr.get();
  this->ptr.reset();
  if (p == nullptr)
    return;

  if (p->decrement_refcnt())
  {
    if (!p->is_destroyed())
      p->~T();

    p->push_to_free_list();
  }
}
```

An object can be reclaimed once `ref_count` drops to zero. Since the `ref_count` gets initialized with `RefCountInc`, an additional decrement has to be performed in `reclaim` before resetting the `guard_ptr`. The `decrement_refcnt` method of `enable_concurrent_ptr` (shown in Listing 4.19) returns `true` when the `ref_count` has reached zero and this thread has successfully set the *claim-bit*. It contains a single CAS operation that is used to decrement the `ref_count` and, if the counter drops to zero, implicitly set the claim-bit. When the CAS performs a simple decrement it uses release-semantics, but when a thread tries to set the claim-bit it uses acquire-semantic. The reason for this is explained in more detail in Section 4.4.1.5.

Listing 4.19: LFRC's enable_concurrent_ptr::decrement_refcnt

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, std::size_t N, class Deleter>
bool lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  enable_concurrent_ptr<T, N, Deleter>::decrement_refcnt()
{
  unsigned old_refcnt, new_refcnt;
  do {
    old_refcnt = ref_count().load(std::memory_order_relaxed);
    new_refcnt = old_refcnt - RefCountInc;
    if (new_refcnt == 0)
      new_refcnt = RefCountClaimBit;
    // (4) - this release/acquire CAS synchronizes with itself
  } while (!ref_count().compare_exchange_weak(old_refcnt, new_refcnt,
                                              new_refcnt == RefCountClaimBit
                                                ? std::memory_order_acquire
                                                : std::memory_order_release,
                                              std::memory_order_relaxed));

  // free node iff ref_count is zero AND we're the first thread to "claim" this node for reclamation.
  return (old_refcnt - new_refcnt) & RefCountClaimBit;
}
```

The `reclaim` method (shown in Listing 4.20) performs the additional decrement of `ref_count` as already described. The according `fetch_sub` operation uses release semantics to synchronize-with the `fetch_add` operation in `acquire` (see Listing 4.21). All other operations that alter `ref_count` are read-modify-write operations and are therefore part of a release sequence headed by `reclaim`. The described synchronize-with relation is therefore also established even in cases where other operations interfere and update `ref_count`.

Listing 4.20: LFRC's guard_ptr::reclaim

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, class MarkedPtr>
void lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  guard_ptr<T, MarkedPtr>::reclaim(Deleter d) noexcept
{
  if (this->ptr.get() != nullptr)
  {
    // ref_count was initialized with "1", so we need an additional
    // decrement to ensure that the node gets reclaimed.
    // ref_count cannot drop to zero here -> no check required.
    // (7) - this release-fetch-sub synchronizes-with the acquire-fetch-add (5, 6)
    this->ptr->ref_count().fetch_sub(RefCountInc, std::memory_order_release);
  }
  reset();
}
```

Listing 4.21: LFRC's guard_ptr::acquire

```
template <bool InsertPadding, size_t ThreadLocalFreeListSize>
template <class T, class MarkedPtr>
void lock_free_ref_count<InsertPadding, ThreadLocalFreeListSize>::
  guard_ptr<T, MarkedPtr>::acquire(concurrent_ptr<T>& p,
                                   std::memory_order order) noexcept
{
  for (;;)
  {
    reset();
    auto q = p.load(std::memory_order_relaxed);
    this->ptr = q;
    if (q.get() == nullptr)
      return;

    // (5) - this acquire-fetch_add synchronizes-with the release-fetch_sub (7)
    // this ensures that a change to p becomes visible
    q->ref_count().fetch_add(RefCountInc, std::memory_order_acquire);

    if (q == p.load(order))
      return;
  }
}
```

4.4.1.4 The **region_guard** class

LFRC does not use the concept of `region_guard`'s and therefore only defines an empty dummy class.

4.4.1.5 Correctness

Suppose a pointer p in a data structure references the node n. Assume, without loss of generality, that thread t_1 removes n from the data structure. After the node has been removed it calls `reclaim` to mark the node for reclamation. For LFRC this means that the `ref_count` gets decremented by a `fetch_sub` operation with `memory_order_release` (4.1). `reclaim` internally also resets the `guard_ptr`, causing another decrement of the reference counter.

A thread that tries to acquire a safe reference from a pointer p first reads the pointer's value, increments the `ref_count` of the returned node and then re-checks that p's value is unchanged (4.2).

$$t_1 : \underbrace{\texttt{p.store}_{rel}}_{\text{remove}(n)} \xrightarrow{\text{sb}} \underbrace{\texttt{n.ref\_count.fetch\_sub}^{(7)}_{rel}}_{\text{guard\_ptr.reclaim}} \tag{4.1}$$

$$t_2 : \underbrace{\texttt{n.ref\_count.fetch\_add}^{(5)}_{acq} \xrightarrow{\text{sb}} \texttt{p.load}_{acq}}_{\text{guard\_ptr.acquire}} \tag{4.2}$$

Suppose that another thread t_2 got a reference to n by reading p before it was updated, but did not yet increment the `ref_count`. If t_2's increment happens-before t_1's second decrement, i.e., it is first in the modification order of n's `ref_count`, then `ref_count` cannot drop to zero and t_2 therefore has a safe reference to n, regardless of whether it recognizes that p has already been changed (4.3). If, however, the decrement is first in the modification order, then the acquire-increment synchronizes-with the release-decrement, effectively establishing a happens-before order between the update to p by t_1 and the reload of p by t_2 (4.4). Thus, t_2 recognizes that p has changed in the meantime and that it has to retry.

$$\begin{aligned} &t_2\texttt{: n.ref\_count.fetch\_add}^{(5)}_{acq} \xrightarrow{\text{mo}} t_1\texttt{: n.ref\_count.fetch\_sub}^{(7)}_{rel} \implies \\ &\quad t_2 \text{ has a safe reference to } n \end{aligned} \tag{4.3}$$

$$\begin{aligned} &t_1\texttt{: n.ref\_count.fetch\_sub}^{(7)}_{rel} \xrightarrow{\text{mo}} t_2\texttt{: n.ref\_count.fetch\_add}^{(5)}_{acq} \implies \\ &\quad t_1\texttt{: n.ref\_count.fetch\_sub}^{(7)}_{rel} \xrightarrow{\text{sw}} t_2\texttt{: n.ref\_count.fetch\_add}^{(5)}_{acq} \implies \\ &\quad t_1\texttt{: p.store}_{rel} \xrightarrow{\text{hb}} t_2\texttt{: p.load}_{acq} \quad \text{i.e., } t_2 \text{ recognizes that } p \text{ has changed.} \end{aligned} \tag{4.4}$$

The happens-before relation between changes made to a node and reclamation of that node is established via changes to `ref_count`. Suppose that thread t_1 wants to reclaim a node n that was protected by t_2. When t_2 drops its reference to n, it performs a release-CAS to decrement n's `ref_count` (4.6). When t_1 drops its reference to n, it recognizes that it is the last thread, so it performs a CAS operation to decrement the `ref_count` and implicitly set the claim-bit, but in this case it uses acquire-sematics (4.5). Assume that t_1 can successfully set the claim-bit. This implies that the CAS by t_2 precedes the CAS by t_1 in `ref_count`'s modification order.

All modifications of a node's `ref_count` are performed by read-modify-write operations, so every release-CAS is the head of a release sequence. Therefore, a synchronize-with relation is established with any release-CAS that precedes the acquire-CAS in `ref_count`'s modification order. So in this example the acquire-CAS by t_1 synchronizes-with the release-CAS by t_2 (4.7).

$$t_1 : \underbrace{\texttt{n.ref\_count.cmpxchg}^{(4)}_{acq}}_{\text{decrement\_refcnt}} \tag{4.5}$$

$$t_2 : \underbrace{\texttt{n.ref\_count.cmpxchg}^{(4)}_{rel}}_{\text{decrement\_refcnt}} \tag{4.6}$$

$$\begin{aligned} t_2\texttt{: n.ref\_count.cmpxchg}^{(4)}_{rel} \xrightarrow{\text{mo}} t_1\texttt{: n.ref\_count.cmpxchg}^{(4)}_{rel} \implies \\ t_2\texttt{: n.ref\_count.cmpxchg}^{(4)}_{rel} \xrightarrow{\text{sw}} t_1\texttt{: n.ref\_count.cmpxchg}^{(4)}_{acq} \end{aligned} \tag{4.7}$$

4.4.2 Hazard Pointers

The implementation of the *hazard pointers* scheme takes a single template parameter `Policy` that controls how hazard pointers are allocated/deallocated and how the threshold for the local retire list is calculated. The two implemented policies are `static_hazard_pointer_policy` and `dynamic_hazard_pointer_policy`. Both policies are explained in more detail in Section 4.4.2.2.

4.4.2.1 The `enable_concurrent_ptr` class

The `enable_concurrent_ptr` implementation for HPBR (shown in Listing 4.22) inherits from `deletable_object_impl` but does not contain any other members that would set it apart from the minimal definition.

Listing 4.22: hazard pointer's enable_concurrent_ptr

```
template <class T, std::size_t N = 0, class Deleter = std::default_delete<T>>
class enable_concurrent_ptr : private detail::deletable_object_impl<T, Deleter>
{
public:
  static constexpr std::size_t number_of_mark_bits = N;
protected:
  enable_concurrent_ptr() noexcept = default;
  enable_concurrent_ptr(const enable_concurrent_ptr&) noexcept = default;
  enable_concurrent_ptr(enable_concurrent_ptr&&) noexcept = default;
  enable_concurrent_ptr& operator=(const enable_concurrent_ptr&) noexcept = default;
  enable_concurrent_ptr& operator=(enable_concurrent_ptr&&) noexcept = default;
  ~enable_concurrent_ptr() noexcept = default;
private:
  friend detail::deletable_object_impl<T, Deleter>;

  template <class, class>
  friend class guard_ptr;
};
```

The scheme internally uses only references to the `deletable_object` base class, i.e., the retire list is a linked list of `deletable_object` entries and also the hazard pointers store references to the `deletable_object` base class. This is necessary to correctly protect classes that use multiple inheritance. An instance of such a class can be referenced by several pointers with different values, simply because the pointers refer to different types (e.g., different base classes). It is therefore necessary to fall back to a common base class, i.e., `deletable_object`.

4.4.2.2 The `static_hazard_pointer_policy` and `dynamic_hazard_pointer_policy` classes

These policies are used to parameterize this reclamation scheme. They control how many hazard pointers each thread can hold at the same time, how these hazard pointers are managed and how the threshold for the number of retired nodes is calculated. The `static_hazard_pointer_policy` class corresponds to the originally proposed scheme with a fixed number of hazard pointers, while `dynamic_hazard_pointer_policy` corresponds to the proposed extension that allows each thread to hold an arbitrary number of hazard pointers. Both policies take three template parameters:

K the number of hazard pointers—the default number is 2.

A, B two constants that are used in the formula $A * \texttt{numHPs}() + B$, where `numHPs` is a function that returns the number of active hazard pointers, i.e., the total number of hazard pointers that could potentially be used by the currently running threads at this point in time. This formula is used to compute the threshold R for the number of retired nodes in order to achieve amortized constant processing time for reclamation. Formally this is written as $R = H + \Omega(H)$, where R is the threshold and H is the maximum number of unreclaimable nodes (i.e., the number of active hazard pointers). The default number for A is 2 and for B 100.

A policy needs to define:

- a method `retired_nodes_threshold`—this method shall calculate the threshold for the number of retired nodes.

- a method `number_of_active_hazard_pointers`—this method shall return the maximum number of hazard pointers that can be in active use at this time.
- a class `thread_control_block`

The `thread_control_block` class has to define the following methods:

- `initialize`
- `abandon`
- `alloc_hazard_pointer`
- `release_hazard_pointer`
- `gather_protected_pointers`

Every thread holds a local `thread_data` instance which contains a reference to its private `thread_control_block` instance. At the same time all `thread_controls_block` instances are registered in a global `thread_block_list` (see Section 4.3.1) so that they can be traversed.

When a thread terminates, it cannot release its local `thread_control_block` instance since it could be accessed by another thread that is currently traversing the `thread_block_list`. Instead it calls `abandon` to mark the block as *unused*. `abandon` also decrements the global *number of active hazard pointers* since the hazard pointers of this block can no longer be used.

When a new thread is started, it first walks through the `thread_block_list` and tries to *adopt* an abandoned block. Only if this fails, it creates a new `thread_control_block` instance and registers it in the `thread_block_list`. For both, newly created and adopted blocks, the thread then calls `initialize`, which increases the global *number of active hazard pointers* and initializes the *internal linked list* of hazard pointers.

When a `guard_ptr` is created with a valid pointer, it *allocates* a hazard pointer from the thread-local control block (i.e., reserves it for exclusive use by this `guard_ptr`) and *releases* the hazard pointer once it gets reset. Therefore the hazard pointers can be allocated and released in an arbitrary order. To avoid a linear search to find the next free hazard pointer, unused hazard pointers are organized in a *linked list* in which every free hazard pointer holds the address of the next free hazard pointer. In order to differentiate these links to other hazard pointers from used hazard pointers, the pointer's LSB is used as mark bit. The head of this list (i.e., the pointer to the first free hazard pointer) is stored in the thread-local `thread_data` instance. This allows very efficient allocation and release of hazard pointers in an arbitrary order.

When a thread performs a *scan*, it traverses the global `thread_block_list` and calls `gather_protected_pointers` for each active `thread_control_block` (i.e., blocks that are not abandoned). `gather_protected_pointers` walks through all the hazard pointers of the block, checks if the hazard pointer points to an object (i.e., the mark bit is not set) and only then adds it to the list of protected objects.

The `static_hazard_pointer_policy` and `dynamic_hazard_pointer_policy` classes share a lot of code since the base functionality is the same for both. The difference between the two policies is that `static_hazard_pointer_policy` only has a fixed size array with K hazard pointers and it will throw a `bad_hazard_pointer_alloc` exception when `alloc_hazard_pointer` is called and all K hazard pointers are already in use.

The `dynamic_hazard_pointer_policy` class also starts with a fixed size array with K hazard pointers, but it dynamically allocates additional blocks with $\max(K, \text{totalHPs}/2)$ additional hazard pointers where $totalHPs$ is the total number of hazard pointers available to this `thread_control_block` (i.e., K plus the sum of hazard pointers in dynamically allocated blocks). These blocks are organized in a simple linked list so that they can be traversed in `gather_protected_pointers`. This allows to dynamically grow the number of hazard pointers. But since they have to stay available indefinitely, it is not possible to shrink it.

4.4.2.3 The `thread_data` class

The `thread_data` class (see Listing 4.23) is a thread-local data structure that holds:

- the linked list of retired nodes,
- the size of the linked list, i.e., the number of retired nodes currently in the list,
- a reference to a `thread_control_block`; this reference is initialized lazily the first time it is required, so that threads that do not use this scheme do not add to the total number of active hazard pointers,
- a *hint* that is passed to `thread_control_block`'s `allocate_hazard_pointer` and `release_hazard_pointer` methods to improve efficiency. For the two implemented policies the hint is simply a pointer to the head of the linked list of unused hazard pointers.

Listing 4.23: HPBR's thread_data

```
template <typename Policy>
struct alignas(64) hazard_pointer<Policy>::thread_data :
  detail::aligned_object<thread_data>
{
  using HP = typename thread_control_block::hazard_pointer*;

  thread_data();
  ~thread_data();

  HP alloc_hazard_pointer();
  void release_hazard_pointer(HP& hp);

  std::size_t add_retired_node(detail::deletable_object* p);
  void scan():

private:
  void ensure_has_control_block();

  detail::deletable_object* retire_list;
  std::size_t number_of_retired_nodes;
  typename thread_control_block::hint hint;

  thread_control_block* control_block;
};
```

Both methods, `allocate_hazard_pointer` and `release_hazard_pointer`, ensure that a `thread_control_block` has been allocated for this thread and simply forward to the according methods of the `thread_control_block` instance.

The `add_retired_node` method adds the given node to the local `retire_list`, increments `number_of_retired_nodes` and returns the resulting number.

The `scan` method (shown in Listing 4.24) iterates over all entries in the `global_thread_block_list` and gathers all the active hazard pointers of each thread in a vector. The sequentially consistent fence in line 8 is required to establish the necessary happens-before relations; this is described in more detail in Section 4.4.2.4. The `protected_pointers` vector is then sorted and handed to `reclaim_nodes`.

Listing 4.24: HBR's thread_data::scan

```
 1 template <typename Policy>
 2 void hazard_pointer<Policy>::thread_data::scan()
 3 {
 4   std::vector<const detail::deletable_object*> protected_pointers;
 5   protected_pointers.reserve(Policy::number_of_active_hazard_pointers());
 6
 7   // (8) - this seq_cst-fence enforces a total order with the seq_cst-fence (4)
 8   std::atomic_thread_fence(std::memory_order_seq_cst);
 9
10   auto adopted_nodes = global_thread_block_list.adopt_abandoned_retired_nodes();
11
12   std::for_each(global_thread_block_list.begin(), global_thread_block_list.end(),
13     [&protected_pointers](const auto& entry)
14     {
15       if (entry.is_active())
16         entry.gather_protected_pointers(protected_pointers);
17     });
18
19   // (9) - this acquire-fence synchronizes-with the release-store (5)
20   std::atomic_thread_fence(std::memory_order_acquire);
21
22   std::sort(protected_pointers.begin(), protected_pointers.end());
23
24   auto list = retire_list;
25   retire_list = nullptr;
26   number_of_retired_nodes = 0;
27   reclaim_nodes(list, protected_pointers);
28   reclaim_nodes(adopted_nodes, protected_pointers);
29 }
```

The `reclaim_nodes` method simply iterates over the given list and for each retired node performs a binary search in the sorted `protected_pointers` vector to check whether this node is protected by some thread. If that is the case, the node gets re-added to the internal retire list. Otherwise it calls `delete_self` on the node and moves on. The same concept is applied to the nodes that we may have adopted. Note that it is important to call `adopt_abandoned_retired_nodes` *before* we gather the list of protected pointers to avoid a race condition where we would try to reclaim a node that was added to the abandoned retired nodes after we have gathered the protected pointers. The abandoned node could be protected by a hazard pointer that set *after* we gathered the list of protected pointers, but we would not recognize this and would therefore reclaim that node while it is still used by some other thread.

4.4.2.4 The `guard_ptr` class

The `guard_ptr` class protects a shared object by ensuring that a globally visible hazard pointer is referencing that object. Therefore the `guard_ptr` sets a hazard pointer for this object when:

- the `guard_ptr` is constructed with a `marked_ptr`,
- the `acquire` or `acquire_if_equal` method is called to acquire a safe reference from a `concurrent_ptr`,
- the `guard_ptr` gets copy constructed or copy assigned.

Move construction and move assignment simply move ownership of the hazard pointer and clear the source `guard_ptr`.

A `guard_ptr` only allocates a hazard pointer when it is actually protecting an object. Thus, default constructed `guard_ptr`s or instances that have been `reset` do not add to the total number of used hazard pointers.

The `reclaim` method (shown in Listing 4.25) simply stores the deleter, adds the node to the local retire-list and checks if the threshold for the maximum number of local retired nodes is exceeded. If that is the case it calls `scan`, which gathers all active hazard pointers of all threads and reclaims all nodes from the local retire-list that are not protected by any hazard pointer.

Listing 4.25: HPBR's guard_ptr::reclaim

```cpp
template <typename Policy>
template <class T, class MarkedPtr>
void hazard_pointer<Policy>::guard_ptr<T, MarkedPtr>::
  reclaim(Deleter d) noexcept
{
  auto p = this->ptr.get();
  reset();
  p->set_deleter(std::move(d));
  if (local_thread_data.add_retired_node(p) >= Policy::retired_nodes_threshold())
    local_thread_data.scan();
}
```

The acquire method (shown in Listing 4.26) allocates a hazard pointer if necessary, i.e., only if p is not null but actually points to a valid object, and if this guard_ptr instance does not already have an allocated hazard pointer. It then stores the loaded value of p in the hazard pointer and performs a reload of p to check whether it has changed in the meantime.

Listing 4.26: HPBR's guard_ptr::acquire

```
template <typename Policy>
template <class T, class MarkedPtr>
void hazard_pointer<Policy>::guard_ptr<T, MarkedPtr>::
  acquire(concurrent_ptr<T>& p, std::memory_order order)
{
  auto p1 = p.load(std::memory_order_relaxed);
  if (p1 != nullptr && hp == nullptr)
    hp = local_thread_data.alloc_hazard_pointer();

  auto p2 = p1;
  do
  {
    if (p2 == nullptr)
    {
      reset();
      return;
    }

    p1 = p2;
    hp->set_object(p1.get());
    // (1) - this load operation potentially synchronizes-with any
    //       release operation on p.
    p2 = p.load(order);
  } while (p1.get() != p2.get());

  this->ptr = p2;
}
```

When a guard_ptr gets reset, it calls release_hazard_pointer to return the hazard pointer to the internal linked list of available hazard pointers. The new value that resets the hazard pointer and signals that it does not protect a node is written using release semantics. This is important to establish a happens-before relation with a thread that tries to reclaim the previously protected node; the details of how this relation is established and why this is necessary are explained in Section 4.4.2.6.

4.4.2.5 The **region_guard** class

HPBR does not use the concept of region_guards and therefore only defines an empty dummy class.

4.4.2.6 Correctness

Assume, without loss of generality, that thread t_1 removes some node n from a data structure by updating some pointer p. Before n can safely be reclaimed, t_1 has to perform a `scan` of all other hazard pointers to ensure that no other thread holds a reference to n. To ensure that t_1 "sees" all relevant hazard pointers it has to perform a sequentially consistent `atomic_thread_fence` before it scans the threads (4.8).

When t_2 tries to obtain a safe reference to p it calls `guard_ptr.acquire`, which internally sets the hazard pointer and performs a sequentially consistent `atomic_thread_fence` (4.9)[6].

$$t_1 : \underbrace{\texttt{p.store}_{rel}}_{\text{remove}(n)} \xrightarrow{\text{sb}} \underbrace{\texttt{fence}^{(8)}_{sc} \xrightarrow{\text{sb}} \texttt{t2.hp.load}_{rlx} \xrightarrow{\text{sb}} \texttt{fence}^{(9)}_{acq}}_{\text{scan}} \tag{4.8}$$

$$t_2 : \underbrace{\texttt{hp.store}_{rlx} \xrightarrow{\text{sb}} \texttt{fence}^{(4)}_{sc} \xrightarrow{\text{sb}} \texttt{p.load}^{(1)}_{acq}}_{\text{guard\_ptr.acquire}} \tag{4.9}$$

Since there is a single total order of all sequentially consistent operations, including the fences in `acquire` (performed by t_2) and in `scan` (performed by t_1), and sequentially consistent fences provide the additional properties described in Section 3.3, it follows that either t_1 sees the hazard pointer of t_2 (4.10), or that t_2 sees that updated value of p (4.11).

$$\begin{aligned} &t_2\texttt{: fence}^{(4)}_{sc} \xrightarrow{\text{sco}} t_1\texttt{: fence}^{(8)}_{sc} \implies \\ &\quad t_2\texttt{: hp.store}^{(3)}_{rlx} \xrightarrow{\text{rf}} t_1\texttt{: t2.hp.load}_{rlx} \\ &\quad \text{i.e., } t_1 \text{ "sees" the hazard pointer, } t_2 \text{ therefore has a safe reference to } n \end{aligned} \tag{4.10}$$

$$\begin{aligned} &t_1\texttt{: fence}^{(8)}_{sc} \xrightarrow{\text{sco}} t_2\texttt{: fence}^{(4)}_{sc} \implies \\ &\quad t_1\texttt{: p.store}^{(3)}_{rel} \xrightarrow{\text{rf}} t_2\texttt{: p.load}^{(1)}_{acq} \\ &\quad \text{i.e., } t_2 \text{ recognizes that } p \text{ has changed} \end{aligned} \tag{4.11}$$

One might think that it would suffice to use `memory_order_relaxed` for the reload of p, because the first load in `guard_ptr.acquire` is sequenced-before the sequentially consistent fence in `set_object`, and the fence would therefore synchronize-with any release operation on p. However, this is true only if the second load operations reads the exact same value from the same write operation, i.e.:

$$\begin{tikzcd}
t_1\texttt{: p.store(n)}_{rel} \arrow[r, "\text{sb}"] \arrow[d, "\text{rf}"] \arrow[dr, "\text{sw}"] \arrow[drr, "\text{rf}"] & \cdots & \\
t_2\texttt{: p.load}_{rlx} \arrow[r, "\text{sb}"] & \texttt{fence}^{(4)}_{sc} \arrow[r, "\text{sb}"] & \texttt{p.load}^{(1)}_{rlx}
\end{tikzcd}$$

[6] Strictly speaking, `guard_ptr.acquire` calls the internal method `set_object`, which sets the hazard pointer and performs the fence.

But it could happen that the node referenced by p gets reclaimed before the hazard pointer becomes globally visible and thus the node could already be reused, causing a subsequent write to store the same value in p.

$$\begin{tikzcd}
t_1\colon \texttt{p.store(n)}_{rel} \arrow[r, "\text{sb}"] \arrow[d, "\text{rf}"] \arrow[dr, "\text{sw}"] & \text{reclaim and reuse } n \arrow[r, "\text{sb}"] & \texttt{p.store(n)}_{rel} \arrow[d, "\text{rf}"] \\
t_2\colon \texttt{p.load}_{rlx} \arrow[r, "\text{sb}"] & \texttt{fence}^{(4)}_{sc} \arrow[r, "\text{sb}"] & \texttt{p.load}^{(1)}_{rlx}
\end{tikzcd}$$

The thread t_2 would not recognize this since the values read by the two load operations are identical. From the perspective of the reclamation scheme this would not be a problem, as it would still have correctly obtained a safe reference since the hazard pointer contains the correct value. But it might be a problem from the perspective of the data structure, as no synchronize-with relation would have been established—even if this would be required by the data structure and the call to `acquire` would have used the correct ordering.

What remains to be shown is that there exists a happens-before relation between the reclaim operation and any changes made to the reclaimed node. Suppose that thread t_1 wants to reclaim a node n that was protected by t_2. As already mentioned before, the reset operation of a `guard_ptr` uses release semantics to return the hazard pointer to the internal list (4.13). When t_1 performs the `scan` operation we read all the hazard pointers of all threads. If a hazard pointer does not protect a node we can ignore it, but the important point is that we have read its value. Suppose that t_1 "sees" the new value of the hazard pointer, i.e., the node is no longer protected by t_2 and can therefore be reclaimed. After scanning all hazard pointers, t_1 perform an acquire-fence (4.12). Since the load operation during the scan read the value previously stored by t_2, the fence synchronizes-with that release-store (4.14). In case t_2 already reused the hazard pointer, we could read some newer value. But since the hazard pointers can only be changed by the owning thread t_2, the new value would be part of a release-sequence headed by release-store that we are interested in. When t_2 abandons its `thread_control_block` it can be reused by some new thread. In this case the changes made by the new thread would no longer be part of a release sequence headed by t_2's changes. In this situation the required synchronize-with relation gets established via the `thread_control_block`'s `in_use` flag. Thus, we have successfully establish the required happens-before relation and the node can safely be reclaimed.

$$t_1 : \underbrace{\texttt{t2.hp.load}_{rlx} \xrightarrow{\text{sb}} \texttt{fence}^{(9)}_{acq}}_{\text{scan}} \tag{4.12}$$

$$t_2 : \underbrace{\texttt{hp.store}^{(5)}_{rel}}_{\text{guard\_ptr.reset}} \tag{4.13}$$

$$\begin{aligned} &t_2\colon \texttt{hp.store}^{(5)}_{rel}[\texttt{store}^{(3)}_{rlx}, \texttt{store}^{(5)}_{rel}] \xrightarrow{\text{rf}} t_1\colon \texttt{t2.hp.load}_{rlx} \implies \\ &\quad t_2\colon \texttt{hp.store}^{(5)}_{rel} \xrightarrow{\text{sw}} t_1\colon \texttt{fence}^{(9)}_{acq} \end{aligned} \tag{4.14}$$

4.4.3 Epoch based reclamation

The implementation of the *epoch based reclamation* scheme takes a single template parameter `UpdateThreshold`, which defines the number of critical region entries upon which a thread tries to update the global epoch.

4.4.3.1 The `enable_concurrent_ptr` class

The `enable_concurrent_ptr` implementation for EBR is identical to that of HPBR (see Section 4.4.2.1).

4.4.3.2 The `thread_control_block` class

Every thread that uses this scheme holds a reference to a `thread_control_block` instance. The instances are all managed by the `global_thread_block_list`. It holds the following members:

- `is_in_critical_region` – a flag that signals whether this thread is currently inside a critical region.
- `local_epoch` – the epoch that this thread is currently observing (in case it is in a critical region).

The `thread_control_block` class acts purely as container for the `local_epoch` and `is_in_critical_region` members, which have to be accessible by other threads in order to determine whether it is safe to update the *global epoch*.

4.4.3.3 The `thread_data` class

The `thread_data` class (see Listing 4.27) is a thread-local data structure that holds the following data members:

- `enter_count` – is used to keep track of the number of nested entries.
- `entries_since_update` – is used to keep track of the number of entries since this thread has last performed an epoch update.
- `retire_lists[number_epochs]` – the retire lists, one for each epoch.
- `control_block` – pointer to the thread-local `thread_control_block` instance. This member is initialized lazily the first time it is required.

Listing 4.27: EBR's thread_data

```
template <std::size_t UpdateThreshold>
struct epoch_based<UpdateThreshold>::thread_data
{
  ~thread_data();
  void enter_critical();
  void leave_critical();
  void add_retired_node(detail::deletable_object* p);

private:
  void ensure_has_control_block();

  void do_enter_critical();
  void do_leave_critical();

  void add_retired_node(detail::deletable_object* p, size_t epoch);

  bool try_update_epoch(unsigned curr_epoch);
  void adopt_orphans();

  unsigned enter_count = 0;
  unsigned entries_since_update = 0;
  thread_control_block* control_block = nullptr;
  std::array<detail::deletable_object*, number_epochs> retire_lists = {};
};
```

The `enter_critical` method increments `enter_count` and if its value was zero it calls `do_enter_critical`. The `leave_critical` and `do_leave_critical` methods are the respective counterparts. `enter_critical` and `leave_critical` are called by the `guard_ptr` when it acquires a pointer respectively when it gets reset. This makes sure that the critical region is entered when the first `guard_ptr` is created and only left when the last `guard_ptr` has been destroyed, i.e., a critical region starts with the creation of the first `guard_ptr`, and its scope is equivalent to the union of scopes of all `guard_ptr`s created inside of it.

When the thread terminates and the `thread_data` instance gets destroyed, there is a fair chance that the `retire_lists` still contain unreclaimed nodes. Since this thread is already terminating, we have to defer reclamation of these nodes to some other thread. For this we use a special `orphan` node. The `orphan` struct (shown in Listing 4.28) is a helper class that stores the thread's `retire_lists` and a `target_epoch`, which defines the epoch when it is safe to reclaim all the items from the `retire_lists`. The `thread_data` destructor creates such an `orphan`, adds it to the list of abandoned nodes in `global_thread_block_list` and releases the thread's `thread_control_block`. This `orphan` node will later be picked up by some other thread when it adopts all the abandoned nodes.

Listing 4.28: orphan helper class

```
template <unsigned Epochs>
struct orphan : detail::deletable_object_impl<orphan<Epochs>>
{
  orphan(unsigned target_epoch,
         std::array<detail::deletable_object*, Epochs> &retire_lists):
    target_epoch(target_epoch),
    retire_lists(retire_lists)
  {}

  ~orphan()
  {
    for (auto p: retire_lists)
      detail::delete_objects(p);
  }

  unsigned target_epoch;
  std::array<detail::deletable_object*, Epochs> retire_lists;
};
```

The `adopt_orphans` method tries to adopt any orphans from the global abandoned nodes list. Since the `orphan` class inherits from `deletable_object`, any instance can simply be added to the appropriate retire list, which is defined by the instance's `target_epoch`. Thus, when the `orphan` gets reclaimed, all the objects in the contained `retire_lists` get reclaimed as well. A thread calls `adopt_orphans` anytime it successfully updated the global epoch.

The `do_enter_critical` method (shown in Listing 4.29) starts by setting the `is_in_critical_region` flag before it loads the `global_epoch`. If the current `local_epoch` is different from the `global_epoch` the thread is observing a new epoch, so it can reclaim all objects in the according `retire_list`, set `local_epoch` to the new value and reset `entries_since_update` to zero. Otherwise it increases `entries_since_update`, and if the new value reaches `UpdateThreshold`, it resets `entries_since_update` to zero and calls `try_update_epoch`. Getting true from the call means that the `global_epoch` has been updated, so the thread is already observing this new epoch and it can thus also reclaim all objects in the according `retire_list` and set the `local_epoch` to the new value.

Listing 4.29: EBR's do_enter_critical

```
template <std::size_t UpdateThreshold>
void epoch_based<UpdateThreshold>::thread_data::do_enter_critical()
{
  ensure_has_control_block();

  control_block->is_in_critical_region.store(true, std::memory_order_relaxed);
  // (3) - this seq_cst-fence enforces a total order with itself
  std::atomic_thread_fence(std::memory_order_seq_cst);

  // (4) - this acquire-load synchronizes-with the release-CAS (7)
  auto epoch = global_epoch.load(std::memory_order_acquire);
  if (control_block->local_epoch.load(std::memory_order_relaxed) != epoch)
  {
    entries_since_update = 0;
  }
  else if (entries_since_update++ == UpdateThreshold)
  {
    entries_since_update = 0;
    const auto new_epoch = (epoch + 1) % number_epochs;
    if (!try_update_epoch(epoch, new_epoch))
      return;

    epoch = new_epoch;
  }
  else
    return;

  // We either just updated the global_epoch or we are observing a new epoch from
  // some other thread. Either way, we can reclaim all the objects from the old
  // 'incarnation' of this epoch.

  control_block->local_epoch.store(epoch, std::memory_order_relaxed);
  detail::delete_objects(retire_lists[epoch]);
}
```

The `try_update_epoch` method (shown in Listing 4.30) checks whether any of the `thread_control_block` instances in the global `thread_block_list` (excluding its own instance) is currently inside a critical region (i.e., it has the `is_in_critical_region` flag set) and has not yet observed the current `global_epoch`. If that is the case there is a thread lagging behind, so the `global_epoch` cannot be updated and the function therefore returns `false`. Otherwise it performs a CAS operation on `global_epoch`, trying to set it to the next value. If this succeeds, the thread tries to adopt any potential orphans. Otherwise, a failing CAS operation simply means that another thread was faster, but the `global_epoch` was still updated. Either way, we have to return `true` to signal the successful update to `do_enter_critical`. However, we have to use `compare_exchange_strong` instead of `compare_exchange_weak` to avoid spurious fails as they would spoil this guarantee.

Listing 4.30: EBR's try_update_epoch

```
template <std::size_t UpdateThreshold>
bool epoch_based<UpdateThreshold>::thread_data::
  try_update_epoch(unsigned curr_epoch, unsigned new_epoch)
{
  const auto old_epoch = (curr_epoch + number_epochs - 1) % number_epochs;
  auto prevents_update = [old_epoch](const thread_control_block& data)
  {
    return data.is_in_critical_region.load(std::memory_order_relaxed) &&
           data.local_epoch.load(std::memory_order_relaxed) == old_epoch;
  };

  // If any thread hasn't advanced to the current epoch, abort the attempt.
  auto can_update = !std::any_of(global_thread_block_list.begin(),
                                 global_thread_block_list.end(),
                                 prevents_update);
  if (!can_update)
    return false;

  if (global_epoch.load(std::memory_order_relaxed) == curr_epoch)
  {
    // (6) - this acquire-fence synchronizes-with the release-store (5)
    std::atomic_thread_fence(std::memory_order_acquire);

    // (7) - this release-CAS synchronizes-with the acquire-load (4)
    bool success = global_epoch.compare_exchange_strong(curr_epoch, new_epoch,
                                                         std::memory_order_release,
                                                         std::memory_order_relaxed);
    if (success)
      adopt_orphans();
  }

  // Return true regardless of whether the CAS operation was successful or not. It is
  // not import that THIS thread updated the epoch, but it got updated in any case.
  return true;
}
```

4.4.3.4 The `guard_ptr` class

The `guard_ptr` protects a shared object by ensuring that the `is_in_critical_region` is set. For this the `guard_ptr` calls `enter_critical` when:

- the `guard_ptr` is constructed with a `marked_ptr`
- the `acquire` or `acquire_if_equal` method is called to acquire a safe reference from a `concurrent_ptr`
- the `guard_ptr` gets copy constructed or copy assigned

The counterpart `leave_critical` is called when a non-null `guard_ptr` gets `reset`. Move construction and move assignment simply move ownership of the referenced object and clear the internal member of the source `guard_ptr`.

If `acquire` or `acquire_if_equal` have already called `enter_critical`, but the subsequent load of the specified pointer returns `null`, then the `guard_ptr` immediately calls `leave_critical`. This ensures that only a non-null `guard_ptr` adds to the total number of `enter_count`.

The `reclaim` method simply adds the node to the `thread_control_block`'s current `retire_list` and `reset`s the `guard_ptr`.

4.4.3.5 The `region_guard` class

EBR does not use the concept of `region_guard`'s and therefore only defines an empty dummy class.

4.4.3.6 Correctness

Assume, without loss of generality, that thread t_1 is currently observing epoch 0 when it removes the node n from some data structure. The removed node is therefore added to `retire_list[0]` and all the nodes in this retire list can be reclaimed as soon as the thread observes that the global epoch has advanced by three, i.e., due to the wrap-around it is again 0.

$$\begin{tikzcd}
t_1 & \text{modification\_order(global\_epoch)} \\
\texttt{enter\_critical} \arrow[d, "\text{sb}"] & \texttt{global\_epoch.cmpxchg(0)}_{rlx} \arrow[l, "\text{rf}"'] \arrow[d, "\text{mo}"] \\
\texttt{remove(n)} \arrow[d, "\text{sb}"] & \texttt{global\_epoch.cmpxchg(1)}_{rlx} \arrow[d, "\text{mo}"] \arrow[dl, dotted, "\text{rf}"'] \\
\cdots \arrow[d, "\text{sb}"] & \texttt{global\_epoch.cmpxchg(2)}_{rlx} \arrow[l, dotted, "\text{rf}"'] \arrow[d, "\text{mo}"] \\
\texttt{enter\_critical} \arrow[d, "\text{sb}"] & \texttt{global\_epoch.cmpxchg(0)}_{rlx} \arrow[l, "\text{rf}"'] \\
\texttt{reclaim(n)} &
\end{tikzcd}$$

Note that t_1 has to observe at least one of the two intermediate epochs 1 or 2. Otherwise it would not be able to recognize that this is a re-occurrence of epoch 0 and therefore the nodes in `retire_list[0]` could not be reclaimed. Assume, without loss of generality, that t_1 performs the last update of `global_epoch` that ultimately leads to the reclamation of n. It follows that the `acquire` call that leads to the update of `global_epoch` is sequenced-after the remove operation (4.15).

A thread t_2 that tries to acquire a safe reference to n sets its `critical_region` flag and performs a sequentially consistent fence before it loads p's value (4.16).

$$t_1 : \underbrace{\texttt{p.store}_{rel}}_{\text{remove}(n)} \xrightarrow{\text{sb}} \underbrace{\texttt{critical\_region.store}_{rlx} \xrightarrow{\text{sb}} \texttt{fence}_{sc}^{(3)} \xrightarrow{\text{sb}} \underbrace{\texttt{t2.critical\_region.load}_{rlx}}_{try\_update\_epoch}}_{\text{guard\_ptr.acquire}} \tag{4.15}$$

$$t_2 : \underbrace{\texttt{critical\_region.store}_{rlx} \xrightarrow{\text{sb}} \texttt{fence}_{sc}^{(3)} \xrightarrow{\text{sb}} \texttt{p.load}_{acq}^{(1)}}_{\text{guard\_ptr.acquire}} \tag{4.16}$$

And since the sequentially consistent fences enforce a total order, there are only two possible scenarios: either t_1 recognizes that t_2 is in a critical region, therefore the global epoch cannot be updated preventing the removed node from being reclaimed (4.17), or t_2 reads the new value of p and the removed node can therefore safely be reclaimed (4.18).

$$\begin{aligned} & t_2\!: \texttt{fence}_{sc}^{(3)} \xrightarrow{\text{sco}} t_1\!: \texttt{fence}_{sc}^{(3)} \implies \\ & \quad t_2\!: \texttt{critical\_region.store}_{rlx} \xrightarrow{\text{rf}} t_1\!: \texttt{t2.critical\_region.load}_{rlx} \\ & \quad \text{i.e, } t_1 \text{ “sees” that } t_2 \text{ is in a critical region} \end{aligned} \tag{4.17}$$

$$\begin{aligned} & t_1\!: \texttt{fence}_{sc}^{(3)} \xrightarrow{\text{sco}} t_2\!: \texttt{fence}_{sc}^{(3)} \implies \\ & \quad t_1\!: \texttt{p.store}_{rel} \xrightarrow{\text{rf}} t_2\!: \texttt{p.load}_{acq}^{(1)} \\ & \quad \text{i.e., } t_2 \text{ reads the new value of } p \end{aligned} \tag{4.18}$$

What remains to be shown is that there exists a happens-before relation between the reclaim operation and any changes made to the reclaimed node. Suppose that thread t_1 wants to reclaim a node n that was previously protected by t_2. In order for t_1 to be able to reclaim n, it must observe a later epoch than t_2 when it protected n. The corresponding `try_update_epoch` that advances the `global_epoch` is performed by thread t_3.

The `leave_critical` operation uses release semantics to set the `in_critical_region` flag to false (4.20). When t_3 performs the `try_update_epoch` operation, it reads all the `critical_region` flags of all threads. If a thread is not in a critical we can ignore it, but the important point is that we have read the flag. Suppose that t_3 "sees" that t_2's flag is set to false. After scanning all threads, t_3 performs an acquire-fence followed by the release-CAS that updates the global epoch. Since the load operation during the scan read the value previously stored by t_2, the fence synchronizes-with that release-store (4.22). In case the load operation during the scan reads a newer value, that value would be part of a release-sequence headed by the release-store, so we still get the same synchronize-with relation.

When t_1 enters its critical region it performs an acquire-load on `global_epoch` (4.19). If this load returns the new value written by the t_3 (or some newer value that would be part of a release-sequence headed t_3's release-CAS), the acquire-load synchronizes-with the release-CAS (4.23). Therefore it follows that the reset operation by t_2 happens-before the reclaim operation by t_1 (4.24).

$$t_1 : \underbrace{\texttt{global\_epoch.load}^{(4)}_{acq}}_{\text{enter\_critical}} \tag{4.19}$$

$$t_2 : \underbrace{\texttt{critical\_region.store}^{(5)}_{rel}}_{\text{guard\_ptr.reset}} \tag{4.20}$$

$$t_3 : \underbrace{\texttt{t2.critical\_region.load}_{rlx} \xrightarrow{\text{sb}} \texttt{fence}^{(6)}_{acq} \xrightarrow{\text{sb}} \texttt{global\_epoch.cmpxchg}^{(7)}_{rel}}_{\text{try\_update\_epoch}} \tag{4.21}$$

$$\begin{aligned} S_1 = t_2\texttt{: critical\_region.store}^{(5)}_{rel} \xrightarrow{\text{rf}} t_3\texttt{: t2.critical\_region.load}_{rlx} \implies \\ t_2\texttt{: critical\_region.store}^{(5)}_{rel} \xrightarrow{\text{sw}} t_3\texttt{: fence}^{(6)}_{acq} \end{aligned} \tag{4.22}$$

$$\begin{aligned} S_2 = t_3\texttt{: global\_epoch.cmpxchg}^{(7)}_{rel}[\texttt{cmpxchg}^{(7)}_{rel}] \xrightarrow{\text{rf}} t_1\texttt{: global\_epoch.load}^{(4)}_{acq} \implies \\ t_3\texttt{: global\_epoch.cmpxchg}^{(7)}_{rel} \xrightarrow{\text{sw}} t_1\texttt{: global\_epoch.load}^{(4)}_{acq} \end{aligned} \tag{4.23}$$

$$\begin{aligned} S_1 \wedge S_2 \implies \\ t_2\texttt{: critical\_region.store}^{(5)}_{rel} \xrightarrow{\text{hb}} t_1\texttt{: global\_epoch.load}^{(4)}_{acq} \end{aligned} \tag{4.24}$$

4.4.4 New epoch based reclamation

As described in Section 2.8 this scheme is an extension to EBR that allows to expand a critical region over several operations in order to distribute the overhead for the region entry. Therefore, the implementation of this scheme is in large parts identical to the EBR implementation; The `enable_concurrent_ptr` and `guard_ptr` classes are exactly the same, `thread_control_block` has a few adaptations and `region_guard` is no longer an empty dummy class.

4.4.4.1 The `thread_data` class

The `thread_data` class (shown in Listing 4.31) is also very similar to that of EBR, with only a few adaptations.

- `region_entries` – is used to keep track of the number of region entries
- `nested_critical_entries` – is used to keep track of the number of critical entries nested in the current region.
- `critical_entries_since_update` – is used to keep track of the number of critical entries since this thread has last performed an epoch update.
- `retire_lists[number_epochs]` – the retire lists, one for each epoch.
- `control_block` – pointer to the thread-local `thread_control_block` instance. This member is initialized lazily the first time it is required.

Listing 4.31: NEBR's thread_data

```
template <std::size_t UpdateThreshold>
struct new_epoch_based<UpdateThreshold>::thread_data
{
  ~thread_data();

  void enter_region();
  void leave_region();

  void enter_critical();
  void leave_critical();

  void add_retired_node(detail::deletable_object* p);

private:
  void ensure_has_control_block();
  void do_enter_critical();
  void add_retired_node(detail::deletable_object* p, size_t epoch);
  bool try_update_epoch(unsigned curr_epoch);
  void adopt_orphans();

  unsigned critical_entries_since_update = 0;
  unsigned nested_critical_entries = 0;
  unsigned region_entries = 0;
  thread_control_block* control_block = nullptr;
  std::array<detail::deletable_object*, number_epochs> retire_lists = {};
};
```

In contrast to EBR, the NEBR implementation introduces an additional tracking level: Region entries (for both, `region_guard`s and `guar_ptr`s) and critical entries (for `guard_ptr` s only).

- The `enter_region` method increments `region_entries` and, if it was zero, sets the `is_in_critical_region` flag.
- The `leave_region` method decrements `region_entries` and, if the new value is zero, clears the `is_in_critical_region` flag.
- The `enter_critical` method first calls `enter_region` (every `guard_ptr` implicitly creates a new region unless another `guard_ptr` or `region_guard` has already created one), then it increments `nested_critical_entries` and, if it was zero, calls `do_enter_critical`, which is basically identical to that of EBR.
- The `leave_critical` method first decrements `nested_critical_entries` and then calls `leave_region`.

4.4.4.2 The `region_guard` class

This is the first scheme that uses the concept of `region_guard`s. The idea is to amortize the cost of the sequentially consistent fence on setting the `is_in_critical_region` flag over a number of operations.

The `region_guard` implementation is straight forward. Upon construction it calls `enter_region` on the thread-local `thread_data` instance and upon destruction it calls `leave_critical`. Since the implementation tracks the number region/critical entries, `region_guard`s and `guard_ptr`s can be arbitrarily mixed and nested.

A critical region thus starts with the creation of the first `guard_ptr` or `region_guard`, and its scope is equivalent to the union of scopes of all `guard_ptr` and `region_guard` instances created inside of it.

4.4.4.3 Correctness

The correctness argument for this reclamation scheme is essentially the same as for EBR (see Section 4.4.3.6).

4.4.5 Quiescent state based reclamation

The `quiescent_state_based` implementation is very similar to that of `new_epoch_based`. The fuzzy barrier is implemented based on the same epoch concept, i.e., the barrier is entered only after all threads have observed the current epoch.

The main difference to EBR/NEBR is that the concept of critical regions is removed. The global epoch only gets updated when all threads have gone through at least one quiescent state. Conceptually this can be seen as if any thread that holds an entry in the global `thread_block_list` is considered to be inside a critical region the whole time. A thread would go through a quiescent state once the last `guard_ptr` or `region_guard` gets destroyed, i.e., the thread is in a quiescent state while the destructor is executing. Once the destructor returns, the thread is again considered to be inside a critical region, even though no `guard_ptr` or `region_guard` exists. That is why the implementation internally still uses the `enter_region` and `leave_region` methods as in NEBR to track the number of `region_entries` caused by any `guard_ptr` or `region_guard` instances.

4.4.5.1 The `thread_data` class

The `thread_data` class (shown in Listing 4.32) is very similar to that of NEBR, with only a few adaptations. The `enter_critical` and `leave_critical` methods together with their counters are removed and `do_enter_critical` is replaced by the `quiescent_state` method, which is called from `leave_region` once `region_entries` drops to zero.

Listing 4.32: QSBR's thread_data

```
struct quiescent_state_based::thread_data
{
  ~thread_data();
  void enter_region();
  void leave_region();
  void add_retired_node(detail::deletable_object* p);

private:
  void ensure_has_control_block();
  void quiescent_state();
  void add_retired_node(detail::deletable_object* p, size_t epoch);
  bool try_update_epoch(unsigned& curr_epoch);
  void adopt_orphans();

  unsigned region_entries = 0;
  thread_control_block* control_block = nullptr;
  std::array<detail::deletable_object*, number_epochs> retire_lists = {};
};
```

While EBR and NEBR use expensive sequentially consistent fences, the QSBR implementation gets away with only acquire/release semantic. The adapted `try_update_epoch` implementation is shown in Listing 4.33. The most important change is the additional acquire-`atomic_thread_fence` that is required to synchronize-with any release store to `local_epoch`. The details of how this ensures the required happens-before relations are described in Section 4.4.5.6

Listing 4.33: QSBR's try_update_epoch

```
bool quiescent_state_based::thread_data::
  try_update_epoch(unsigned curr_epoch, unsigned new_epoch)
```

```
{
  auto old_epoch = (curr_epoch + number_epochs - 1) % number_epochs;
  auto prevents_update = [old_epoch](const thread_control_block& data)
  {
    return data.is_active() &&
           data.local_epoch.load(std::memory_order_relaxed) == old_epoch;
  };

  // If any thread has not advanced to the current epoch, abort the attempt.
  bool can_update = !std::any_of(global_thread_block_list.begin(),
                                 global_thread_block_list.end(),
                                 prevents_update);
  if (!can_update)
    return false;

  if (global_epoch.load(std::memory_order_relaxed) == curr_epoch)
  {
    // (4) - this acquire-fence synchronizes-with the release-store (3)
    std::atomic_thread_fence(std::memory_order_acquire);

    // (5) - this acq_rel-CAS synchronizes-with the acquire-load (2)
    //       and the acq_rel-CAS (1)
    bool success = global_epoch.compare_exchange_strong(curr_epoch, new_epoch,
                                                        std::memory_order_acq_rel,
                                                        std::memory_order_relaxed);
    if (success)
      adopt_orphans();
  }

  // return true regardless of whether the CAS operation was successful or not
  // it is not import that THIS thread updated the epoch, but it got updated in any
    case
  return true;
}
```

4.4.5.2 The `enable_concurrent_ptr` class

The `enable_concurrent_ptr` implementation for QSBR is identical to that of HPBR (see Section 4.4.2.1).

4.4.5.3 The `thread_control_block` class

The `thread_control_block` for QSBR is trivial as it only contains a single data member for the thread's `local_epoch`:

```
struct quiescent_state_based::thread_control_block :
  detail::thread_block_list<thread_control_block>::entry
{
  std::atomic<unsigned> local_epoch;
};
```

4.4.5.4 The **guard_ptr** class

The `guard_ptr` implementation of QSBR is essentially identical to that of EBR (see Section 4.4.3.4).

4.4.5.5 The **region_guard** class

The `region_guard` implementation of QSBR is essentially identical to that of NEBR (see Section 4.4.4.2).

4.4.5.6 Correctness

Assume, without loss of generality, that thread t_1 is removing a node n from some data structure and putting it on a local `retire_list`. In order to do this, t_1 has to have a `guard_ptr` instance that protects n. The `retire_list` containing n can be reclaimed once the `global_epoch` has advanced three times, but in order to advance the `global_epoch`, every thread must have observed the then current epoch, i.e., every thread, including t_1, must have gone through at least one quiescent state before the `global_epoch` can be advanced. When a thread observes a new epoch in `quiescent_state`, it uses `memory_order_acquire` to load the new value of `global_epoch` and announces this by storing it in `local_epoch` using `memory_order_release` (4.25).

Assume that thread t_2 is trying to advance `global_epoch`. It scans all threads to check whether they have already observed the current epoch by reading `local_epoch` from the threads' respective `thread_control_block`. If this is not the case the attempt fails. Otherwise, the thread performs an acquire-fence and, since all the `local_epoch` reads are sequenced-before the fence (4.26) and since the reads returned the current epoch, the fence synchronizes-with the respective `local_epoch` release-store operations (4.28).

In the next step t_2 updates `global_epoch` using a `compare_exchange_strong` operation[7] with `memory_order_acq_rel` (4.26). So when thread t_3 observes the new epoch and performs an acquire-load on `global_epoch` in `quiescent_state` (4.27), it synchronizes-with t_2's acq_rel-CAS operation (4.29), and due to the transitivity of the happens-before relation this ensures that t_1's store happens-before t_3's load on p (4.30).

$$S_1 = t_1 : \underbrace{\texttt{p.store}_{rel}}_{\text{remove}(n)} \xrightarrow{\text{sb}} \underbrace{\texttt{local\_epoch.store}^{(3)}_{rel}}_{\text{quiescent\_state}} \tag{4.25}$$

$$S_2 = t_2 : \underbrace{\texttt{t}_\texttt{x}\texttt{.local\_epoch.load}_{rlx} \xrightarrow{\text{sb}} \texttt{fence}^{(4)}_{acq} \xrightarrow{\text{sb}} \texttt{global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx}}_{\text{try\_update\_epoch}} \tag{4.26}$$

$$S_3 = t_3 : \underbrace{\texttt{global\_epoch.load}^{(2)}_{acq}}_{\text{quiescent\_state}} \xrightarrow{\text{sb}} \underbrace{\texttt{p.load}^{(6)}_{acq}}_{\text{guard\_ptr.acquire}} \tag{4.27}$$

[7] In this case a `compare_exchange_strong` operation is necessary, since it is not part of a loop and we must therefore avoid spurious failures.

$$S_4 = \underbrace{t_1\texttt{: local\_epoch.store}^{(3)}_{rel}}_{\text{quiescent\_state}} \xrightarrow{\text{rf}} \underbrace{t_2 : \texttt{t}_\texttt{x}\texttt{.local\_epoch.load}_{rlx} \xrightarrow{\text{sb}} t_2\texttt{: fence}^{(4)}_{acq}}_{\text{try\_update\_epoch}} \implies$$
$$t_1\texttt{: local\_epoch.store}^{(3)}_{rel} \xrightarrow{\text{sw}} t_2\texttt{: fence}^{(4)}_{acq} \tag{4.28}$$

$$S_5 = \underbrace{t_2\texttt{: global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx}}_{\text{try\_update\_epoch}} \xrightarrow{\text{rf}} \underbrace{t_3\texttt{: global\_epoch.load}^{(2)}_{acq}}_{\text{quiescent\_state}} \implies \tag{4.29}$$
$$t_2\texttt{: global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx} \xrightarrow{\text{sw}} t_3\texttt{: global\_epoch.load}^{(2)}_{acq}$$

$$S_6 = S_1 \wedge S_2 \wedge S_3 \wedge S_4 \wedge S_5 \implies \tag{4.30}$$
$$t_1\texttt{: p.store}_{rel} \xrightarrow{\text{hb}} t_2\texttt{: p.load}^{(1)}_{acq}$$

It is important to note that this argument implies that there exists a happens-before relation between the reclaim operation and any changes made to the reclaimed node.

However, one problem remains to be solved: In HPBR, EBR and NEBR the sequentially consistent fences implicitly ensure that newly added threads are recognized and therefore also considered when checking the currently observed epochs. As this scheme does not use costly sequentially consistent operations, it still has to be ensured that newly added threads are correctly considered. This means that we have to show that either the update of p happens-before the update of the `thread_block_list` or vice versa.

Since we want to completely avoid sequentially consistent operations, we use the `global_epoch` to synchronize all the threads. The `ensure_has_control_block` method (shown in Listing 4.34) reads the current `global_epoch`, stores it in `local_epoch` and attempts a CAS operation with the same value using `memory_order_acq_rel`.

Listing 4.34: QSBR's ensure_has_control_block

```
void quiescent_state_based::thread_data::ensure_has_control_block()
{
  if (control_block == nullptr)
  {
    control_block = global_thread_block_list.acquire_entry();
    auto epoch = global_epoch.load(std::memory_order_relaxed);
    do {
      control_block->local_epoch.store(epoch, std::memory_order_relaxed);

      // (1) - this acq_rel-CAS synchronizes with the acquire-loads (todo)
      //       and the acq_rel-CAS (todo)
    } while (!global_epoch.compare_exchange_weak(epoch, epoch,
                                                 std::memory_order_acq_rel,
                                                 std::memory_order_relaxed));
  }
}
```

We do not actually want to change the `global_epoch`, thus we are writing the same value we just read, but we want to perform a release-write. As mentioned in Section 3.3, read-modify-write operations always read the last value in the modification order. It thus follows that either the CAS in `ensure_has_control_block` synchronizes-with the CAS in `try_update_epoch` (4.32), or the other way round (4.33).

$$\underbrace{t_1\colon \texttt{local\_epoch.store}_{rlx} \xrightarrow{\text{sb}} \texttt{global\_epoch.cmpxchg\_weak}^{(1)}_{ar,rlx}}_{\text{ensure\_has\_control\_block}} \xrightarrow{\text{sb}} \texttt{p.load}_{acq} \tag{4.31}$$

$$\begin{gathered}\underbrace{t_1\colon \texttt{global\_epoch.cmpxchg\_weak}^{(1)}_{ar,rlx}}_{\text{ensure\_has\_control\_block}} \xrightarrow{\text{mo}} \underbrace{t_2\colon \texttt{global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx}}_{\text{try\_update\_epoch}} \implies \\ t_1\colon \texttt{global\_epoch.cmpxchg\_weak}^{(1)}_{ar,rlx} \xrightarrow{\text{rf}} t_2\colon \texttt{global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx} \implies \\ t_1\colon \texttt{global\_epoch.cmpxchg\_weak}^{(1)}_{ar,rlx} \xrightarrow{\text{sw}} t_2\colon \texttt{global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx}\end{gathered} \tag{4.32}$$

$$\begin{gathered}\underbrace{t_2\colon \texttt{global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx}}_{\text{try\_update\_epoch}} \xrightarrow{\text{mo}} \underbrace{t_1\colon \texttt{global\_epoch.cmpxchg\_weak}^{(1)}_{ar,rlx}}_{\text{ensure\_has\_control\_block}} \implies \\ t_2\colon \texttt{global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx} \xrightarrow{\text{rf}} t_1\colon \texttt{global\_epoch.cmpxchg\_weak}^{(1)}_{ar,rlx} \implies \\ t_2\colon \texttt{global\_epoch.cmpxchg\_strong}^{(5)}_{ar,rlx} \xrightarrow{\text{sw}} t_1\colon \texttt{global\_epoch.cmpxchg\_weak}^{(1)}_{ar,rlx}\end{gathered} \tag{4.33}$$

So there is either a happens-before relation between the update to p and a potential read of p by the new thread, or between the CAS on `global_epoch` by the new thread and the CAS in `try_update_epoch`, which itself happens-before the reclamation of p. It is therefore guaranteed that the new thread is recognized and considered when checking the observed epochs.

4.4.6 Stamp-it

As described in Section 2.17, this scheme requires a data structure that efficiently supports the following operations:

- Add an element and assign a stamp to it; stamps have to be strong monotonically increasing.
- Remove a specific element from its current position and return true if this element was the one with the lowest stamp at that point in time.
- Get the highest stamp assigned to an element so far, i.e., the last stamp that has been assigned to an element.
- Get the lowest stamp of all elements currently in the data structure.

My implementation of this data structure is called `thread_order_queue`[8] and is built on the ideas of the lock-free doubly-linked list by Sundell and Tsigas [ST05]. It requires two static dummy nodes, *head* and *tail*, which never get removed and are also used to manage the highest and lowest stamp values; the highest stamp is stored in *head* and the lowest one in *tail*.

To insert or delete a node from the data structure one has to update the respective set of `prev` and `next` pointers. These have to be changed consistently, but not necessarily all at once. The solution proposed by Sundell and Tsigas is to treat the doubly-linked list as a singly-linked list with auxiliary information in the `prev` pointers, with the `next` pointers being updated before the `prev` pointers. Thus, the `next` pointers always form a consistent singly-linked list, but the `prev` pointers only give hints for where to find the previous node. The implementation is described in more detail in Section 4.4.6.5

A well known problem for non-blocking implementations based on singly-linked lists, are insert operations that conflict with concurrent delete operations, i.e., the thread performing the insert tries to update the `next` pointer of a node that is currently getting deleted by another thread. Harris solved this problem [Har01] by introducing a deletion mark in the pointer's LSB. Before removing a node from the list, the deletion mark has to be set causing all CAS operations to fail that attempt to update the `next` pointer as part of a concurrent insert operation. For the doubly-linked list, the same concept has to be applied for both pointers, `next` and `prev`.

Every thread has a single `thread_control_block` instance acting as a node that can be inserted and removed in the `thread_order_queue`. Since these instances are "reused", special care has to be taken regarding the ABA problem. Since `next` and `prev` are already `marked_ptr`s (for the deletion mark), I decided to spare a few more bits for a *version tag*, i.e., some of the lower bits are used to store a tag that gets incremented with every change to the pointer value. This solution to the ABA problem has already been described in [IBM83]. Instead of using a second variable for the tag and updating both, pointer and tag, atomically with a DWCAS[9] operation, I simply squeeze both values into a single pointer. However, this is not a definite solution since the tag value can wrap around, leaving the theoretical possibility that a pointer could have been updated between a read and a subsequent successful CAS operation. But by using enough bits for the tag value we can reduce the probability on such an incorrect update to a negligible level, so one can consider it to be practically impossible[10]. In order to ensure that the lower bits are actually unused and the tag does not interfere with the pointer value the nodes have to be allocated at properly aligned addresses.

[8] The name derives from the fact that the order of nodes in the queue reflects the order in which the threads have entered their respective critical region.

[9] DWCAS is a double-word CAS operation that atomically compares and updates two consecutive words, while DCAS/2CAS usually refer to an operation that atomically updates two independent words.

[10] Assume that a pointer p has a tag consisting of n bits and that with every update of p the tag value would be incremented. Suppose a thread t performs a load on p followed by an update of p using a CAS operation. For an "incorrect" update it would be necessary that between the load and the CAS operations other threads would have to perform *exactly* a multiple of 2^n operations on p, in order to produce a wrap-around that results in the exact same tag value that p had when t performed the load; otherwise the CAS operation would fail. Obviously a higher value of n reduces this risk considerably.

Originally I used the 16 lowest bits, where the LSB is the mark bit and the remaining 15 bits were used for the version tag—I assumed this to be more than enough to reliably prevent any ABA problem. But during extensive stress tests of the data structure using the GuardPtr benchmark (see Section 4.6.1) occasionally weird errors occurred or some assertions would hit, suggesting that the data structure was in a state that was supposed to be impossible. This happened repeatedly when running the GuardPtr benchmark at full load after a runtime of about 10-15 hours.

In order to verify whether these errors were actually caused by undetected ABA situations I increased the number of bits for the version tag to 21 and added additional assertions after every failed CAS operation, checking whether the CAS would have also failed with a tag with only 15 bits. In several subsequent test runs, the new assertion was hit after a runtime of 10-15 hours, but no errors occurred nor did any other assertions fire, thus confirming that those errors were indeed caused by undetected ABA situations. Increasing the number of bits in the assertion to 16 increased the average runtime until the first assertion hit to about 20-30 hours. After another increase to 17 bits the benchmark ran without any problems for more than 50 hours in several test runs. These experiments show that the probability of an undetected ABA error is roughly reduced by half with every additional bit spent for the version tag. In the end I decided to use 17 bits as the new default value for the version tag. However, one should consider that these problems only occurred in a stress test with the worst case scenario, running for long time. In a more realistic scenario, where changes to the data structure occur much less frequently, it would be highly unlikely to run into this issue, even with less than the 17 bits used now.

In the other chapters the term “node” was used to denote some piece of memory that was managed by the reclamation scheme. However, with the `thread_order_queue` Stamp-it has an internal data structure that also uses some “nodes”. In order to avoid confusion I will refer to those internal nodes as *blocks* (since they are `thread_control_block` instances).

4.4.6.1 The `deletable_object_with_stamp` class

All the other schemes use `deletable_object` as base class for `enable_concurrent_ptr` as they have no special requirements. In case of Stamp-it, each node has to store the current stamp at the time it is marked for reclamation and we have to be able to access this stamp value later when trying to finally reclaim the node. Since the retire-lists only store pointers to the base class (and we do not care about the concrete type), the stamp value therefore has to be a member of the base class.

In addition the `deletable_object_with_stamp` class contains a `next_chunk` pointer that is used to connect chunks of retired nodes in the global retire-list. Such a “chunk” is a sorted list of nodes, linked by the `next` pointer. The global retire-list is therefore a list of chunks, linked by the `next_chunk` pointer of each lists head node. The head of this global retire-list is managed by the `thread_order_queue`.

The complete definition of the class can be seen in Listing 4.35.

Listing 4.35: Stamp-it's deletable_object_with_stamp

```
struct deletable_object_with_stamp
{
  virtual void delete_self() = 0;
  deletable_object_with_stamp* next = nullptr;
  deletable_object_with_stamp* next_chunk = nullptr;
protected:
    virtual ~deletable_object_with_stamp() = default;
private:
  stamp_t stamp;
  friend class stamp_it;
};
```

4.4.6.2 The `enable_concurrent_ptr` class

The `enable_concurrent_ptr` implementation for Stamp-it (see Listing 4.36) is almost identical to that of HPBR and EBR. The only difference is that `deletable_object_with_stamp` is used as base class for `deletable_object_impl`.

Listing 4.36: Stamp-it's enable_concurrent_ptr

```
template <class T, std::size_t N = 0, class Deleter = std::default_delete<T>>
class enable_concurrent_ptr :
  private detail::deletable_object_impl<T, Deleter, deletable_object_with_stamp>
{
public:
  static constexpr std::size_t number_of_mark_bits = N;
protected:
  enable_concurrent_ptr() noexcept = default;
  enable_concurrent_ptr(const enable_concurrent_ptr&) noexcept = default;
  enable_concurrent_ptr(enable_concurrent_ptr&&) noexcept = default;
  enable_concurrent_ptr& operator=(const enable_concurrent_ptr&) noexcept = default;
  enable_concurrent_ptr& operator=(enable_concurrent_ptr&&) noexcept = default;
  ~enable_concurrent_ptr() noexcept = default;
private:
  friend detail::deletable_object_impl<T, Deleter, deletable_object_with_stamp>;

  template <class, class>
  friend class guard_ptr;
};
```

4.4.6.3 The `thread_data` class

The `thread_data` class (see Listing 4.37) is very similar to that of QSBR and NEBR. It defines the usual methods to enter or leave critical regions and to add a retired node to the internal retire-list.

Listing 4.37: Stamp-it's thread_data

```
struct stamp_it::thread_data
{
  ~thread_data();
  void enter_region();
  void leave_region();
  void add_retired_node(deletable_object_with_stamp* p);

private:
  void ensure_has_control_block();

  void process_local_nodes();
  void process_global_nodes();

  // This threshold defines the max. number of nodes a thread may collect
  // in the local retire-list before trying to reclaim them. It is checked
  // every time a new node is added to the local retire-list.
  static const std::size_t try_reclaim_threshold = 40;
  // The max. number of nodes that may remain a threads local retire-list
  // when it leaves it's critical region. If there are more nodes in the
  // list, then the whole list will be added to the global retire-list.
  static const std::size_t max_remaining_retired_nodes = 20;

  thread_control_block* control_block = nullptr;
  unsigned region_entries = 0;
  std::size_t number_of_retired_nodes = 0;

  deletable_object_with_stamp* first_retired_node = nullptr;
  deletable_object_with_stamp** prev_retired_node = &first_retired_node;
}
```

In contrast to other schemes, the internal retire-list is organized as a FIFO queue where new retired nodes are added to the end of the queue; that is the reason for the two pointers `first_retired_node` and `prev_retired_node`. This way the list of retired nodes is ordered by their stamp values, with `first_retired_node` pointing to the node with the lowest stamp.

The class contains definitions of two constants that are relevant for the reclamation process: `try_reclaim_threshold` and `max_remaining_nodes`. When a new retired node is added to the list and the number of retired nodes exceeds the `try_reclaim_threshold`, we immediately try to reclaim as many nodes as possible from our local retire-list. Since the check whether a node can be reclaimed is very cheap, this threshold can be quite low. When a thread leaves its critical region we also reclaim as many nodes as possible, but there is a good chance that some nodes cannot be reclaimed yet and therefore remain in our local retire-list. If the number of remaining nodes exceeds `max_remaining_nodes`, we take the whole local retire-list and add it as a “chunk” to the global retire-list. That way some other thread can handle them.

There are two methods to process and reclaim retired nodes: `process_local_nodes` and `process_global_nodes`. As the name suggests, the first one reclaims all nodes from the local retire-list that have a stamp value less than `tail`’s stamp. It is called from `add_retired_node` when the number of nodes exceeds the `try_reclaim_threshold`, and from `leave_region` when this thread was *not* the last one. If, on the other hand, this thread was the last one, `leave_region` instead calls `process_global_nodes`. This method tries to fetch the global retire-list, adds the local retire-list as another chunk to it and walks the list of chunks, reclaiming as many nodes from each chunk as possible. The remaining list of chunks is then again added to the global retire-list.

4.4.6.4 The `thread_control_block` class

Each thread holds a `thread_control_block` instance in its local `thread_data` instance. This `thread_control_block` is used as a node representing the owning thread in the `thread_order_queue`, but to avoid confusion with other usages of the term "node" it is referred to as "block". The definition of the `thread_control_block` class is shown in Listing 4.38. Similar to the previous schemes it derives from `thread_block_list::entry` so we inherit all the methods for adopting and abandoning blocks. As additional data member it only holds the `next` and `prev` pointers and the `stamp`, which are all used in the `thread_order_queue` as described later.

Listing 4.38: Stamp-it's thread_control_block

```cpp
struct stamp_it::thread_control_block :
  detail::aligned_object<thread_control_block, 1 << MarkBits>,
  detail::thread_block_list<thread_control_block>::entry
{
  using concurrent_ptr =
    std::atomic<detail::marked_ptr<thread_control_block, MarkBits>>;

  concurrent_ptr prev;
  concurrent_ptr next;

  std::atomic<stamp_t> stamp;
};
```

4.4.6.5 The `thread_order_queue` class

This data structure is used to keep track which threads have entered a critical region and in which order. It holds two dummy blocks `head` and `tail` that cannot be removed, and new blocks are inserted right after `head`. The `prev` pointers define the direction from `head` to `tail`; this direction is always kept consistent[11]. The `next` pointers define the direction from `tail` to `head`, but they only act as hints where to find the next block. It is thus possible that in an intermediate state a block, which is already inserted in the `prev` list, does not occur in the `next` list (and the other way round in case of removal).

As just mentioned, the `prev` pointer points towards `tail` and `next` points towards `head`, i.e., the `prev` direction goes from `head` to `tail` and the `next` direction from `tail` to `head`. When looking at a block and its neighbors, i.e., its successor and predecessor, it is important to also consider the direction we are currently looking at. For example, the predecessor of a block b in `prev` direction is some block whose `prev` pointer is pointing

[11]For those who are familiar with the originally proposed doubly-linked list by Sundell and Tsigas [ST05] this is a minor but important difference; in the original version *next* was kept consistent and *prev* acted as hints. However, I reversed the directions for this data structure as it seemed more natural in this context.

to b, while the predecessor of b in `next` direction would be some other block whose `next` pointer is pointing to b. In many cases I will explicitly state the direction we are looking at to avoid any ambiguity. When no direction is mentioned, we are usually talking about the `prev` direction, since this is the one that is kept consistent and is therefore the "single source of truth".

Each block, including `head` and `tail`, holds a *stamp counter*. When a new block gets inserted, it loads `head`'s `prev` pointer and stores it in its local `prev`, then it increases `head`'s `stamp` using an atomic fetch-and-add operation, stores the returned value in its local `stamp` and then performs a CAS on `head->prev` in order to insert itself into the `prev` list. This ensures that `head` always holds the highest stamp and that the stamps in `prev` direction are strong monotonically decreasing, i.e., the stamp of the newly added block is greater than all other blocks (except `head`). The only exception to this is the `tail` block, as its stamp should always reflect the stamp value of its immediate predecessor in `prev` direction; the reason for this is explained in more detail later.

Even though the `next` pointers only act as hints, it is still guaranteed that they only point to blocks with a higher stamp value, i.e., the stamps in `next` direction are strong monotonically increasing (with the `tail` block again being an exception). More specifically, the `next` pointer of some block b can point to:

- `head`: This can be the case when `head` is the actual predecessor of b, or when some other block c has inserted itself between `head` and b, but did not yet update b's `next` pointer. Note, when b has already marked its `prev` pointer, it can no longer be updated by c, so this inconsistency can only be resolved once b is fully removed from both lists.

- a block c which is still in the `prev` list: This is the "normal case", i.e., usually c is the predecessor of b; unless b is `head`, in which case the previously described exception is possible.

- a block c which is removed from the `prev` list: This is the intermediate state when c has been removed from the `prev` list, but not yet from the `next` list. So `prev->next` points to c, but c is no longer the immediate predecessor of b in `prev` direction when starting from `head`. However, by following c's `next` pointer (and potentially those of other removed blocks) one should end up at b's new predecessor. It is, however, possible that the previously described exception occurs and the `next` pointer of some block on this path actually points to `head`. In this case one has start there and follow the `prev` list to find b's real predecessor.

An example is shown in Figure 4.1.

The links of the blocks t_1, t_3 and t_4 are all marked so they cannot be updated. The block t_3 is already fully removed, i.e., it is not referenced by any `prev` nor `next` pointer. The blocks t_1 and t_4 are marked for deletion, but are not yet fully removed; t_1 has been removed from the `prev` list, but is still in the `next` list; t_4 is still fully linked.

t_5 did already finish its `push` operation (since it is not green), but the `next` pointer of its successor, t_4, still points to `head`. This indicates that t_4's `next` pointer was already

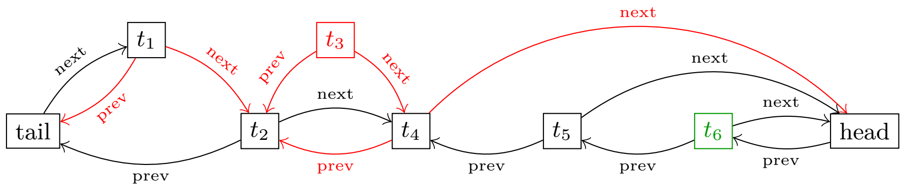

Figure 4.1: Example of blocks with their links; green blocks are not yet fully inserted, red blocks are removed; red links are marked.

marked, so it could not be updated by t_5. On the other hand, t_6 is currently inside the `push` operation; it has successfully inserted itself in the `prev` list, but the update of t_5's `next` pointer is still pending.

The two lowest bits of the stamp counter are used to embed flags to track a block's state:

- `PendingPush`: This flag is used to signal that the block is currently getting inserted into the queue.
- `NotInList`: This flag is used to signal that the block has been completely removed from the queue, i.e., it is no longer part of the `prev` nor the `next` list. This implies that the owning thread is no longer inside a critical region.

The two flags are mutually exclusive, so they cannot both be set at the same time.

The public interface of the class contains the following methods:

- `push`: Inserts the given block right after `head`.
- `remove`: Removes the block from the queue by removing it from the `prev` list as well as the `next` list.

A thread that enters a critical region calls `push` with its own `thread_control_block`.

When a thread leaves a critical region, it removes itself from the queue. If this thread was the last one (i.e., the `prev` pointer points to *tail*), then it tries to update `tail`'s `stamp` to the stamp of the new "last" thread, i.e., the new predecessor of `tail` in `prev` direction.

A block can be in four different states:

- *in the queue*: In this state the delete marks of the `prev` and `next` pointer, as well as the `NotInList` flag are cleared, the `PendingPush` flag is undetermined. Note that this state only determines that the block was correctly inserted in *prev* direction, but it does not state anything about the *next* direction. The reason for this is that we cannot update the `next` pointer of a block when it is already marked for removal.

- *in the process of getting removed from the queue*: In this state the delete mark of the `prev` pointer is set, the mark of `next` is undetermined and the `NotInList` and `PendingPush` flags are cleared.
- *fully removed from the queue*: In this state the `NotInList` flag and `prev`'s delete mark are set, the `PendingPush` flag is cleared and `next`'s delete mark is undetermined (because it gets reset in the `push` operation before the `PendingPush` flag is set).
- *in the process of getting inserted into the queue's next list*: In this state the `PendingPush` flag is set, the `NotInList` flag and `next`'s delete mark are cleared and `prev`'s delete mark is undetermined.

All these states and their transitions are depicted in Figure 4.2.

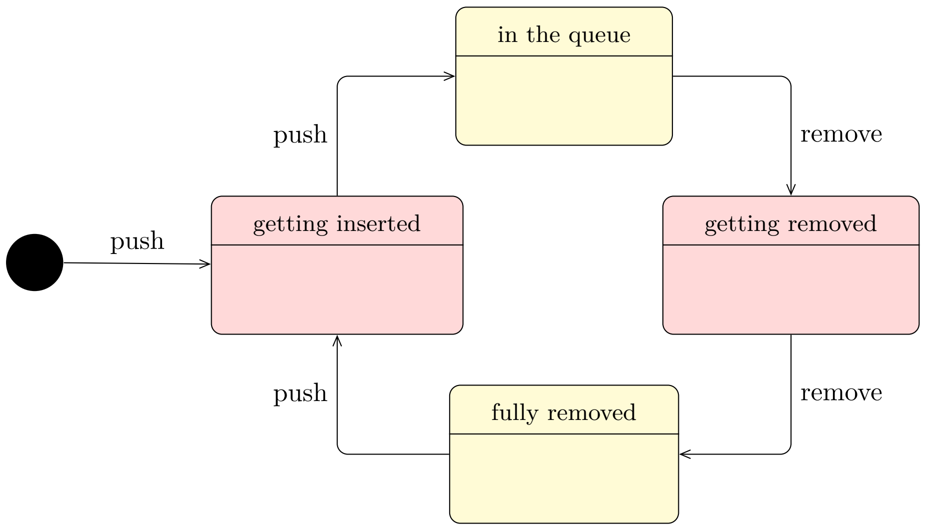

Figure 4.2: State-transition diagram for blocks in a thread_order_queue.

The implementations of the push and remove operations make heavy use of two helper functions: `make_marked` and `make_clean_marked`. The `make_marked` function (shown in Listing 4.39) takes a pointer to a `thread_control_block` and a `marked_ptr`, returning a new `marked_ptr` with the value of the given `thread_control_block` and the mark value of the given `marked_ptr` increased by `TagInc`. It is used to create new `marked_ptr` values when updating a block's `next` or `prev` pointers. The `make_clean_marked` function works just like `make_marked`, except that it implicitly resets the delete flag.

Listing 4.39: make_marked helper function

```
marked_ptr make_marked(thread_control_block* p, const marked_ptr& mark)
{
  return marked_ptr(p, (mark.mark() + TagInc) & MarkMask);
}
```

The implementation of the push method is shown in Listing 4.40.

Listing 4.40: Stamp-it's push

```cpp
void stamp_it::thread_order_queue::push(thread_control_block* block)
{
  // (1) - this release-store synchronizes-with the acquire-loads
  //       (7, 8, 20, 24, 25, 29, 31, 32)
  block->next.store(make_clean_marked(head, block->next), std::memory_order_release);

  marked_ptr head_prev = head->prev.load(std::memory_order_relaxed);
  marked_ptr my_prev;
  size_t stamp;

  for (;;)
  {
    marked_ptr head_prev2 = head->prev.load(std::memory_order_relaxed);
    if (head_prev != head_prev2)
    {
      head_prev = head_prev2;
      continue;
    }

    // fetch a new stamp and set the PendingPush flag
    // (2) - this seq_cst-fetch-add enforces a total order with (12)
    //       and synchronizes-with the acquire-loads (19, 23)
    stamp = head->stamp.fetch_add(StampInc, std::memory_order_seq_cst);
    auto pending_stamp = stamp - (StampInc - PendingPush);

    // (3) - this release-store synchronizes-with the acquire-loads (19, 23, 30)
    block->stamp.store(pending_stamp, std::memory_order_release);

    if (head->prev.load(std::memory_order_relaxed) != head_prev)
      continue;

    my_prev = make_clean_marked(head_prev.get(), block->prev);

    // (4) - this release-store synchronizes-with the acquire-loads
    //       (15, 17, 18, 22, 26)
    block->prev.store(my_prev, std::memory_order_release);

    // (5) - in this acq_rel-CAS the:
    //       - acquire-load synchronizes-with the release-stores (5, 21, 28)
    //       - release-store synchronizes-with the acquire-loads (5, 15, 18, 22)
    if (head->prev.compare_exchange_weak(head_prev, make_marked(block, head_prev),
                                         std::memory_order_acq_rel,
                                         std::memory_order_relaxed))
      break;
    // Back-Off
  }

  // reset the PendingPush flag
  // (6) - this release-store synchronizes-with the acquire-load (19, 23, 30)
  block->stamp.store(stamp, std::memory_order_release);

  // try to update our successor's next pointer
  // (7) - this acquire-load synchronizes-with the release-stores (1, 8, 27)
  auto link = my_prev->next.load(std::memory_order_acquire);
  for (;;)
  {
    if (link.get() == block ||
        link.mark() & DeleteMark ||
        block->prev.load(std::memory_order_relaxed) != my_prev)
      // our successor is in the process of getting removed,
      // or has been removed already -> never mind
      break;

    // (8) - in this CAS the:
    //       - release-store synchronizes-with the acquire-loads
    //         (7, 8, 14, 20, 24, 25, 29, 31, 32)
    //       - acquire-reload synchronizes-with the release-stores (1, 8, 27)
    if (my_prev->next.compare_exchange_weak(link, make_marked(block, link),
                                            std::memory_order_release,
                                            std::memory_order_acquire))
      break;

    // Back-Off
  }
}
```

We start by setting the `next` pointer to `head`, since we are always inserting blocks right after `head`. After that we load the current value of `head->prev`, which we will need later. In the next step we perform a `fetch_add` on `head->stamp`, thus incrementing `head`'s stamp and calculating the new stamp for the block we are about to insert. This new stamp value is adapted so that it has the `PendingPush` flag set before it is stored in our block. After that we write the previously loaded `prev` value in our block's `prev` pointer and attempt a CAS operation to update `head->prev` with our own block. Note that it is important that the `stamp` and `prev` fields of the block are written in this order; the details of why this is the case are explained in Section 4.4.6.8.

When the CAS is successful we have inserted our block in the `prev` list and can therefore reset the `PendingPush` flag. What remains now is that we update our successor's `next` pointer so that it references our newly inserted block. This is done in the final loop that simply performs the according CAS until either:

- the CAS operation was successful,
- the link is marked and can therefore not be updated,
- or some other thread has already updated the link to point to our new block.

The `remove` method (shown in Listing 4.41) first marks the `prev` and `next` pointers before calling `remove_from_prev_list` and (optionally) `remove_from_next_list`. Once the block is fully removed from the list we can set the `NotInList` flag and check if this block was the last one, i.e., if it was `tail`'s predecessor. If that is the case, we call `update_tail_stamp` to try to update `tail`'s `stamp` to that of the new last block.

Listing 4.41: Stamp-it's remove

```cpp
bool stamp_it::thread_order_queue::remove(marked_ptr n)
{
  // We need acq-rel semantic here to ensure the happens-before relation
  // between the remove operation and the reclamation of any node.
  //  - acquire to establish sychnronize-with relation with previous blocks
  //    that removed themselves by updating our prev.
  //  - release to establish synchronize-with relation with other threads
  //    that potentially remove our own block before we can do so.

  // (9) - in this acq_rel CAS the:
  //       - acquire-load of synchronizes-with the release-stores (21, 28)
  //       - release-store of synchronizes-with the acquire-loads (15, 17, 18, 22, 26)
  marked_ptr prev = set_mark_flag(block->prev, std::memory_order_acq_rel);
  marked_ptr next = set_mark_flag(block->next, std::memory_order_relaxed);

  bool fully_removed = remove_from_prev_list(prev, block, next);
  if (!fully_removed)
    remove_from_next_list(prev, block, next);

  auto stamp = block->stamp.load(std::memory_order_relaxed);
  // set the NotInList flag to signal that this block is no longer part of the queue
  block->stamp.store(stamp + NotInList, std::memory_order_relaxed);

  bool wasTail = block->prev.load(std::memory_order_relaxed).get() == tail;
  if (wasTail)
  {
    // Since the stamps of the blocks between tail and head are strictly increasing,
    // we can call update_tail_stamp with the next higher stamp (i.e., stamp+StampInc)
    // as the 'next best guess'.
    update_tail_stamp(stamp + StampInc);
  }

  return wasTail;
}
```

Marking the two pointers signals to other threads that this block is about to be removed, and at the same time prevents the pointers from being updated by some CAS operation from a thread that did not yet see the mark. In order to remove a block b from the `prev` list, the thread has to find its predecessor, i.e., the block c with the `prev` pointer pointing to b, and update c's `prev` pointer with the value of b's `prev` pointer. But it can of course happen that c's `prev` pointer is also marked and can therefore not be updated. In this case we have to find c's predecessor and help remove c before we can continue with the removal of b. By removing c, we get a new predecessor for b. So we can restart the loop and try to remove it again. The same idea is applied when removing a block from the `next` list.

Since a block b can only be removed from the `prev` list when its immediate predecessor is not marked, any marked immediate predecessor has to be removed before b can be removed. This leads to the following conclusion: Whenever a thread that tries to remove a marked block b encounters another block c which is supposed to come *after* b in `prev` direction (i.e., it was found by following the `prev` pointers starting from b), where $\text{stamp}_c > \text{stamp}_b$, or stamp_c has the `NotInList` flag set, and all blocks on the path from b to c are also marked, we can conclude that b has already been removed from both lists.

Since c was encountered *after* b in `prev` direction, it is supposed to have a lower stamp than b; it can only have a greater stamp if it was removed and reinserted. But since all blocks between c and b are marked, c could not have been removed without first removing all those blocks, including b. The same holds for the case when the `NotInList` flag is set, as the flag is only set once the block has been fully removed.

The `remove_from_prev_list` method (shown in Listing 4.42) does exactly what its name suggests: It removes the block b from the `prev` list. This is done in a lock-free manner by helping remove potentially marked predecessors of b as just described. The method keeps track of three different pointers:

prev this is a reference to the next *unmarked* block in `prev` direction, i.e., the block that we want to set as the new value for our predecessor's `prev` pointer. We get to this block by following our own `prev` pointer and the `prev` pointers of marked blocks (if any).

next this is a reference to some block that precedes our own block in `prev` direction. By following this block's `prev` pointer we should end up at our own block, unless some other thread has removed it already. This way we can efficiently find our immediate predecessor to update its `prev` pointer.

last this is a reference to a helper block that is used to remove potentially marked predecessors of our own block. When this pointer is not null, it should be the immediate predecessor of the *next* block.

So the order of the blocks in `prev` direction should be as follows: *last* (if it is set), *next*, our own block b, and *prev*. Each of these blocks can potentially be removed and reinserted at any time (except of course b, which can be removed by some other thread, but not reinserted). For *next* and *last* we have to consider this possibility and take appropriate actions. However, when we recognize that *prev* has been removed or maybe even reinserted, we can stop since we know that b must have been removed already as well (as previously described). The initial values of *prev* and *next* that we start from are passed to the method when it is called in `remove`. However, they are not just input but also output parameters, so we can use the latest values in the subsequent call to `remove_from_next_list` (if necessary).

Listing 4.42: Stamp-it's remove_from_prev_list

```
bool stamp_it::thread_order_queue::remove_from_prev_list(
  marked_ptr& prev, marked_ptr b, marked_ptr& next)
{
  const auto my_stamp = b->stamp.load(std::memory_order_relaxed);
```

```
marked_ptr last = nullptr;
for (;;)
{
  // check if the block is already deleted
  if (next.get() == prev.get())
  {
    next = b->next.load(std::memory_order_relaxed);
    return false;
  }

  auto prev_prev = prev->prev.load(std::memory_order_relaxed);
  auto prev_stamp = prev->stamp.load(std::memory_order_relaxed);

  // check if prev has been removed
  if (prev_stamp > my_stamp || // prev has been reinserted already
      prev_stamp & NotInList)  // prev has been removed
  {
    return true;
  }

  if (prev_prev.mark() & DeleteMark)
  {
    if (!mark_next(prev, prev_stamp))
    {
      // prev is marked for deletion, but mark_next failed because the stamp
      // of prev has been updated - i.e., prev has been deleted already (and
      // maybe even reinserted)
      // -> this implies that b must have been removed as well.
      return true;
    }
    // This acquire-reload is needed to establish a happens-before relation
    // between the remove operations and the reclamation of a node.
    // (17) - this acquire-load synchronizes-with the release-stores (4, 9, 21, 28)
    prev = prev->prev.load(std::memory_order_acquire);
    continue;
  }

  // We need need to obtain a consistent set of "prev" and "stamp" values
  // from next, otherwise we could wrongfully update next_prev's stamp in
  // save_next_as_last_and_move_next_to_next_prev, since we cannot be sure
  // if the "prev" value we see in the reload belongs to a block that is
  // part of the list.

  // (18) - this acquire-load synchronizes-with the release-stores (4, 5, 9, 21, 28)
  auto next_prev = next->prev.load(std::memory_order_acquire);
  // (19) - this acquire-load synchronizes-with the release-stores (2, 3, 6)
  auto next_stamp = next->stamp.load(std::memory_order_acquire);

  if (next_prev != next->prev.load(std::memory_order_relaxed))
    continue;

  if (next_stamp < my_stamp)
  {
      next = b->next.load(std::memory_order_relaxed);
      return false;
    }

    // Check if next has been removed from list or whether it is currently getting
    // inserted. It could be that the block is already inserted, but the PendingPush
    // flag has not yet been cleared. Unfortunately, there is no way to identify this
    // case here, so we have to go back yet another block. We can help resetting this
    // flag once we are sure that the block is already part of the list, which is
    // exactly what happens in save_next_as_last_and_move_next_to_next_prev.
    if (next_stamp & (NotInList | PendingPush))
    {
      if (last.get() != nullptr)
      {
        next = last;
        last.reset();
      }
      else
        // (20) - this acquire-load synchronizes-with the release-stores (1, 8, 27)
        next = next->next.load(std::memory_order_acquire);
      continue;
    }

    if (remove_or_skip_marked_block(next, last, next_prev, next_stamp))
      continue;

    // check if next is the predecessor of b
    if (next_prev.get() != b.get())
    {
      save_next_as_last_and_move_next_to_next_prev(next_prev, next, last);
      continue;
    }

    // unlink "b" from prev list
    // (21) - this release-CAS synchronizes-with the acquire-loads
    //        (5, 9, 15, 17, 18, 22, 26)
    if (next->prev.compare_exchange_strong(next_prev,
                                           make_marked(prev.get(), next_prev),
                                           std::memory_order_release,
                                           std::memory_order_relaxed))
      return false;

    // Back-Off
  }
}
```

The method essentially consists of a large loop that keeps track of the three mentioned blocks, while trying to find the direct predecessor of the block b that we want to remove. Once we have found that predecessor, we can try to update its `prev` pointer in order to remove b. There are several conditions that lead to the termination of this loop. In some of these cases we can conclude that b is already fully removed from *both* lists; this is signaled to the caller by returning `true`. In the other cases we know that b has been successfully removed from the `prev` list, but we still need to ensure that it is also removed from the `next` list; this is signaled to the caller by returning `false`. In the latter case we also have to ensure that `next` and `prev` point to blocks that allow `remove_from_next_list` to work correctly. The following paragraphs provide a more detailed explanation of the loop's code.

If the *prev* and *next* pointers point to the same block, *b* must have been removed from the `prev` list already. However, we do not know whether it is still part of the `next` list, so we set *next* to *b*'s `next` pointer to allow `remove_from_next_list` to start from there and return `false`.

Otherwise, we load *prev*'s `prev` pointer and `stamp`. If the loaded stamp value is greater than *b*'s `stamp` or has the `NotInList` flag set, we can conclude that *prev* must have been removed already and therefore *b* must have been removed too, so we can simply return `true`.

Otherwise, we check whether *prev*'s `prev` pointer is marked. If that is the case, we try to set the *delete mark* on the `next` pointer using the `mark_next` method (shown in Listing 4.43). This method performs a CAS loop trying to set the delete mark on *prev*'s `next` pointer as long as *prev*'s `stamp` matches the stamp value we previously read. When we detect that the `stamp` has changed, we can conclude that *prev* must have been removed already, so we return `true`. Otherwise, either the CAS operation succeeds or we recognize that some other thread has already set the delete mark, so we return `false`.

In the first case, i.e., when `mark_next` returns `true`, we can again conclude that *b* must have been removed, so `remove_from_prev_list` can return `true`. Otherwise, we set *prev* to the previously read `prev` pointer and restart the whole loop.

Listing 4.43: Stamp-it's mark_next

```cpp
bool stamp_it::thread_order_queue::mark_next(marked_ptr block, size_t stamp)
{
  // (31) - this acquire-load synchronizes-with the release-stores (1, 8, 27)
  auto link = block->next.load(std::memory_order_acquire);
  // We need acquire to synchronize-with the release store in push. This way it is
  // guaranteed that the following stamp.load sees the NotInList flag or some newer
  // stamp, thus causing termination of the loop.
  while (block->stamp.load(std::memory_order_relaxed) == stamp)
  {
    auto mark = link.mark()
    if (mark & DeleteMark ||
        // (32) - this acquire-reload synchronizes-with the release-stores (1, 8, 27)
        block->next.compare_exchange_weak(link,
                                          marked_ptr(link.get(), mark | DeleteMark),
                                          std::memory_order_relaxed,
                                          std::memory_order_acquire))
      return true;
  }
  return false;
}
```

In the next step we have to load a consistent set of `prev` and `stamp` values from *next* (the reason for this is explained in more detail in Section 4.4.6.8). For this we first load `prev`, followed by `stamp` and finally perform a reload of `prev`. When the reload of `prev` returns a different value we do not have a consistent set of values, so we simply restart the loop. Otherwise we can continue with checking if the *next*'s `stamp` is less than *b*'s stamp. If this is the case we can conclude that *b* must have been removed from the `prev` list, but we still have to ensure that it is also removed from the `next` list. So we again reset *next* to *b*'s `next` and return `false`.

Otherwise we check if the previously loaded stamp has the `NotInList` or `PendingPush` flag set. If that is the case we cannot use this block since it might not be part of the `prev` list. For the `NotInList` flag this is pretty obvious, but for the `PendingPush` flag this is a little more subtle; the flag signals that the block is currently getting inserted into the `prev` list, but with the information we have available at this time it is impossible to tell whether this has already happened or not. So we have no choice but to move *next* to the next known block in `next` direction. In case we have a valid *last* pointer, we use this one (as it is supposed to be the predecessor of *next* in `prev` direction), otherwise we take *next*'s `next` pointer and restart the loop.

Then we call `remove_or_skip_marked_block` (shown in Listing 4.44) with the values we just loaded from *next*. This method checks whether the *next* block is marked, and if that is the case, tries to remove it. But we can only remove it if we have a valid *last* pointer; remember, *last* is supposed to be the predecessor of *next*, so if it is set, we can perform a CAS trying to update *last*'s `prev` to the just loaded `prev` pointer from *next*. In case we have no *last* pointer, we have to move *next* to the next block in `next` direction.

Listing 4.44: Stamp-it's remove_or_skip_marked_block

```cpp
bool stamp_it::thread_order_queue::remove_or_skip_marked_block(
  marked_ptr& next, marked_ptr& last, marked_ptr next_prev, stamp_t next_stamp)
{
  // check if next is marked
  if (next_prev.mark() & DeleteMark)
  {
    if (last.get() != nullptr)
    {
      // check if next has "overtaken" last
      if (mark_next(next, next_stamp) &&
          last->prev.load(std::memory_order_relaxed) == next)
      {
        // unlink next from prev-list
        // (28) - this release-CAS synchronizes-with the acquire-loads
        //        (5, 9, 15, 17, 18, 22, 26)
        last->prev.compare_exchange_strong(next, make_marked(next_prev.get(), next),
                                           std::memory_order_release,
                                           std::memory_order_relaxed);
      }
      next = last;
      last.reset();
    }
    else
      // (29) - this acquire-load synchronizes-with the release-stores (1, 8, 27)
      next = next->next.load(std::memory_order_acquire);

    return true;
  }
  return false;
}
```

In case `remove_or_skip_marked_block` returns `true` we have a new value in *next* so we restart the loop. Otherwise, we continue by checking whether *next*'s `prev` pointer matches our *b*, i.e., if *next* is *b*'s predecessor in `prev` direction. If that is not the case we call `save_next_as_last_and_move_next_to_next_prev` (shown in Listing 4.45) and restart the loop.

The `save_next_as_last_and_move_next_to_next_prev` method tries to move *next* to the following block in `prev` direction, while keeping the old value of *next* in *last*. There is a special case that needs to be handled. It could happen that the next block in `prev` direction has successfully inserted itself into the list (obviously, otherwise we could not have found it by following the path of unmarked `prev` pointers), but still has the `PendingPush` flag set, i.e., it did not yet finish its `push` operation. Remember that we previously checked that *next*'s `stamp` does not have the `PendingPush` flag set, because otherwise we would have to dismiss the block as we could not determine whether it is already inserted? Now we can conclude that it is in fact part of the `prev` list, so we help the other thread resetting the `PendingPush` flag. This is necessary to ensure lock-freedom, as otherwise we would iterate infinitely because in the next iteration the previously mentioned check would hit again.

Listing 4.45: Stamp-it's save_next_as_last_and_move_next_to_next_prev

```
void stamp_it::thread_order_queue::save_next_as_last_and_move_next_to_next_prev(
    marked_ptr next_prev, marked_ptr& next, marked_ptr& last)
{
  // (30) - this acquire-load synchronizes-with the release-stores (3, 6)
  size_t next_prev_stamp = next_prev->stamp.load(std::memory_order_acquire);

  if (next_prev_stamp & PendingPush &&
      next_prev == next->prev.load(std::memory_order_relaxed))
  {
    // since we got here via an (unmarked) prev pointer next_prev has been added
    // to the "prev-list", but the PendingPush flag has not been cleared yet.
    // i.e., the push operation for next_prev is still pending
    // -> help clear the PendingPush flag
    auto expected = next_prev_stamp;
    const auto new_stamp = next_prev_stamp + (StampInc - PendingPush);
    if (!next_prev->stamp.compare_exchange_strong(expected, new_stamp,
                                                  std::memory_order_relaxed))
    {
      // CAS operation failed, i.e., the stamp of next_prev has been changed
      // since we read it. Check if some other thread cleared the flag already
      // or whether next_prev has been removed (and potentially readded).
      if (expected != new_stamp)
      {
        // the stamp has been updated to an unexpected value, so next_prev has
        // been removed already -> we cannot move to next_prev, but we can keep
        // the current next and last.
        return;
      }
    }
  }
  last = next;
  next = next_prev;
}
```

If, on the other hand, *next*'s `prev` does match *b*, we have found *b*'s predecessor, so we attempt a CAS on *next*'s `prev` with our current *prev*. If the CAS succeeds we have successfully removed *b* (as well as all the other marked blocks following *b* in `prev` direction on the path to *prev*) and can return `false`. Otherwise some other thread interfered, so we simply restart the loop and try again.

When `remove_form_prev_list` returns `true`, we know that our block has been fully removed from both lists already. Otherwise we still need to remove it from the `next` list as well; this is done in `remove_from_next_list` (shown in Listing 4.46).

Listing 4.46: Stamp-it's remove_from_next_list

```
void stamp_it::thread_order_queue::remove_from_next_list(
  marked_ptr prev, marked_ptr removed, marked_ptr next)
{
  const auto my_stamp = removed->stamp.load(std::memory_order_relaxed);
  marked_ptr last = nullptr;
  for (;;)
  {
    // (22) - this acquire-load synchronizes-with the release-stores (4, 5, 9, 21, 28)
    auto next_prev = next->prev.load(std::memory_order_acquire);
    // (23) - this acquire-load synchronizes-with the release-stores (2, 3, 6)
    auto next_stamp = next->stamp.load(std::memory_order_acquire);

    if (next_prev != next->prev.load(std::memory_order_relaxed))
      continue;

    // check if next has been removed from list
    if (next_stamp & (NotInList | PendingPush))
    {
      if (last.get() != nullptr)
      {
        next = last;
        last.reset();
      }
      else
      {
        // (24) - this acquire-load synchronizes-with the release-stores (1, 8, 27)
        next = next->next.load(std::memory_order_acquire);
      }
      continue;
    }

    // (25) - this acquire-load synchronizes-with the release-stores (1, 8, 27)
    auto prev_next = prev->next.load(std::memory_order_acquire);
    auto prev_stamp = prev->stamp.load(std::memory_order_relaxed);

    // check if prev has a higher stamp than the block we want to remove.
    if (prev_stamp > my_stamp || prev_stamp & NotInList)
    {
      // due to strict order of stamps the prev block must have been removed already -
     and with it b.
      return;
    }

    // check if prev block is marked for deletion
    if (prev_next.mark() & DeleteMark)
    {
      // This acquire-load is needed to establish a happens-before relation
      // between the different remove operations and the reclamation of a node.
      // (26) - this acquire-load synchronizes-with the release-stores (4, 9, 21, 28)
      prev = prev->prev.load(std::memory_order_acquire);
      continue;
    }

    if (next.get() == prev.get())
      return;

    if (remove_or_skip_marked_block(next, last, next_prev, next_stamp))
      continue;

    // check if next is the predecessor of prev
    if (next_prev.get() != prev.get())
    {
      save_next_as_last_and_move_next_to_next_prev(next_prev, next, last);
      continue;
    }

    if (next_stamp <= my_stamp || prev_next.get() == next.get())
      return;

    auto new_next = make_marked(next.get(), prev_next);
    if (next->prev.load(std::memory_order_relaxed) == next_prev &&
        // (27) - this release-CAS synchronizes-with the acquire-loads
        //        (7, 8, 14, 20, 24, 25, 29, 31, 32)
        prev->next.compare_exchange_weak(prev_next, new_next,
                                         std::memory_order_release,
                                         std::memory_order_relaxed))
    {
      if ((next->next.load(std::memory_order_relaxed).mark() & DeleteMark) == 0)
        return;
    }
    // Back-Off
  }
}
```

This method is quite similar to `remove_from_prev_list`. It also keeps track of the same three pointers, where the initial values for *prev* and *next* are those that were returned by `remove_from_prev_list`. This allows us to continue from where we left, reducing the amount of work in many cases.

In this method we have to set *next* to the last unmarked block with a stamp greater than *b*'s `stamp`, and *prev* to the first unmarked block with a stamp less or equal to *b*'s `stamp` (both in `prev` direction), i.e., the two blocks that would be the predecessor and successor of *b* if *b* would still be part of the `prev` list. This entails that *next*'s `prev` pointer must reference *prev*. Once we have found these blocks, we can attempt a CAS to update *prev*'s `next` in order to finish removal of *b* from the `next` list. If the CAS succeeds, we have successfully removed *b*. However, we still have to make sure that the *prev* block has not been marked in the meantime. If this is the case we have to continue and help remove *prev* from both lists in order to maintain the previously described condition, which allows us to conclude that a block has been fully removed if we recognized that the successor block has been fully removed.

Otherwise we can return to `remove` and set the `NotInList` flag as we have now successfully removed *b* from both lists. In case our thread was the last one (i.e., if *b*'s `prev` pointer points to `tail`) we must update `tail`'s `stamp`; this is done in `update_tail_stamp` (shown in Listing 4.47).

Listing 4.47: Stamp-it's update_tail_stamp

```cpp
void stamp_it::thread_order_queue::update_tail_stamp(size_t stamp)
{
  // In the best case the stamp of tail equals the stamp of tail’s predecessor (in
  // prev direction), but we don’t want to waste too much time finding the "actual"
  // predecessor. Therefore we simply check whether the block referenced by tail->next
  // is the actual predecessor and if so take its stamp. Otherwise we simply use the
  // stamp that was passed (which is kind of a "best guess").

  // (14) - this acquire-load synchronizes-with the release-stores (8, 27)
  auto last = tail->next.load(std::memory_order_acquire);
  // (15) - this acquire-load synchronizes-with the release-stores (4, 5, 9, 21, 28)
  auto last_prev = last->prev.load(std::memory_order_acquire);
  auto last_stamp = last->stamp.load(std::memory_order_relaxed);
  if (last_stamp > stamp &&
      last_prev.get() == tail &&
      tail->next.load(std::memory_order_relaxed) == last)
  {
    if (last.get() != head)
      stamp = last_stamp;
    else
    {
      // Special case when we take the stamp from head - the stamp in head gets
      // incremented before a new block is actually inserted, but we must not use
      // such a value if the block is not yet inserted. By updating prev with an
      // incremented version a pending insertion would fail and cause a retry,
      // therefore enforcing the strict odering. However, since we are potentially
      // disturbing push operations, we only want to do this if it is "worth it",
      // i.e., if the stamp we read from head is at least one increment ahead of
      // our "next best guess".
      if (stamp < last_stamp - StampInc &&
          head->prev.compare_exchange_strong(last_prev,
                                             make_marked(last_prev.get(), last_prev),
                                             std::memory_order_relaxed))
        stamp = last_stamp;
    }
  }

  // Try to update tail->stamp, but only as long as our new value is actually greater.
  auto tail_stamp = tail->stamp.load(std::memory_order_relaxed);
  while (tail_stamp < stamp)
  {
    // (16) - this release-CAS synchronizes-with the acquire-load (13)
    if (tail->stamp.compare_exchange_weak(tail_stamp, stamp,
                                          std::memory_order_release))
      break;
  }
}
```

Ideally we want to update `tail`'s `stamp` to that of its current predecessor in `prev` direction. Unfortunately, finding this predecessor is not as simple as taking `tail`'s next pointer, since it could point to `head` (due to the predecessor not having finished its push operation) or to a block that could have been removed and potentially reinserted at the time we read its `stamp`. Of course we can detect such cases and try to find the actual predecessor by following the block's `prev` pointer, but we do not want to waste too much time for this. So we simply take `tail`'s `next` pointer, load that block's `stamp` as well as its `prev` pointer, and verify that the `prev` points to `tail` and `tail`'s `next` pointer is still unchanged. If so, perfect—we found a direct predecessor of `tail` so we can use the stamp value we just read. Otherwise, we stop searching and simply use the "next best guess", which is our own block's `stamp` plus the usual stamp-increment.

There is, however, one special case that we have to consider: When the predecessor we have found is `head`. When a block gets inserted into the queue it increases `head`'s `stamp`, stores the old value in its own `stamp` and then performs the CAS to insert itself into the `prev` list. Since the increment happens before the CAS, we could read the new `stamp` value in `update_tail_stamp`, but the reload of `prev` could still return the old value. Using this stamp value would result in `tail` having a greater stamp than the block that is about to get inserted, which is a clear violation of our invariant. To solve this issue we simply perform a CAS on `head`'s `prev` pointer in order to update the pointer's tag value. If the CAS succeeds, the CAS in the `push` operation will fail and the other thread will have to perform a retry. Otherwise the `push` operation was faster and `head` is therefore no longer the predecessor of `tail`. In this case we again simply stop wasting time and use the "next best guess".

Finally, we perform a simple CAS-loop, trying to update `tail`'s `stamp` as long as the new value we want to write is greater than the value we are trying to replace.

4.4.6.6 The `guard_ptr` class

The `guard_ptr` implementation is essentially identical to that of EBR (see Section 4.4.3.4).

4.4.6.7 The `region_guard` class

The `region_guard` implementation is essentially identical to that of NEBR (see Section 4.4.4.2).

4.4.6.8 Correctness

I will provide a few arguments to show that the `thread_order_queue` is lock-free and correct (i.e., the described invariants hold).

lock-freedom

Obviously, the `push` operation (see Listing 4.40) is lock-free. The first loop performs a CAS operation in order to insert the block into the queue. In case the CAS succeeds, we break out of the loop, otherwise we just restart the loop. The CAS can only fail if some other thread interfered—either by inserting or removing some block[12]. But in this case some other thread must have made progress.

The same argument can be applied to `update_tail_stamp` and `mark_next`. Both methods contain loops that perform CAS operations, and a failure of these operations can only be caused by progress in some other thread.

The `remove_from_prev_list` and `remove_from_next_list` are a bit more complex. Since they are quite similar, the following arguments can be applied to both methods. As mentioned before, both methods keep track of a *prev* and a *next* pointers. In each iteration we perform one of the following changes in case we have to restart the loop:

- move *prev* along the `prev` direction (in case *prev* is marked)
- move *next* along the `prev` direction (in case *next* is not *prev*'s predecessor)
- remove *next* from the `prev` list (in case *next* is marked and we have a *last* pointer)
- move *next* along the `next` direction (in case *next* is marked and we have no *last* pointer, or *next* has the `NotInList` or `PendingPush` flag set)
- nothing (in case the CAS to remove *b* failed)

The block *b* splits the `prev` list into two sublists: The sublist from `head` to *b*, and the sublist from *b* to `tail`. *next* points to a block in the first sublist and *prev* points to a block in the second sublist. New blocks are inserted at the beginning of the `prev` list (right after head), i.e., they become part of the first sublist. So the number of times we can move *next* in `prev` direction until we reach *b* is bounded by the number of entries in

[12]Theoretically this can also be caused by a spurious failure as we use a `compare_exchange_weak` operation, but these spurious failures do not spoil lock-freedom.

the first sublist, and the number of times we can move *prev* in `prev` direction until we reach `tail` is bounded by the number of entries in the second sublist.

The case where we have to move *next* back in `next` direction because it is marked and we have no valid *last* can be resolved by following *next*'s `next` pointer and from there move again along `prev`, while maintaining *last*. So the next time we encounter the same marked block, we will be able to remove it as we should now have a valid *last* pointer. In the worst case scenario we have to move along the `next` direction until *next* points to `head`, from where we can then start to move *next* along the `prev` direction, potentially removing any marked blocks. The case where *next* has the `PendingPush` flag set can be resolved in the same way.

This leaves us with the cases where *next* has the `NotInList` flag set or the CAS operation to remove *b* fails. But both cases can only occur when another thread changed the data structure in some way that it is no longer consistent with our thread's view. So unless some other thread interferes, for both methods, `remove_from_prev_list` and `remove_from_next_list`, it is guaranteed that at any time a thread is able to finish the method in a bounded number of steps.

Unfortunately, the block pointed to by *next* can be removed and reinserted at any time. Obviously, this spoils the previously mentioned bounds as with every reinsertion the block is put back right at the beginning of the `prev` list. However, this implies that the owning thread of this reinserted block has been able to finish its `remove` and subsequent `push` operation, i.e., it has made progress. Thus, the requirements for lock-freedom are fulfilled as it is guaranteed that at any time at least one thread makes progress: If there is no conflict with another thread, we can finish the operation in a bounded number of steps; otherwise, the interfering thread was able to make progress.

The main difficulty in the `thread_order_queue` data structure stems from the fact that each block can be removed and reinserted at any time. When a block gets reinserted, the delete marks in the `next` and `prev` pointers are cleared. So in contrast to the original dequeue by Sundell and Tsigas [ST05] we cannot rely solely on these marks, but have to establish more sophisticated checks to prevent invalid updates to these pointers.

All the relevant operations use only the three atomic member variables from the `thread_control_block` class: `prev`, `next` and `stamp`. But there are a lot of operations on these variables and, more importantly, the order in which these operations are performed on the variables plays an important role. In particular, the order in which different operations are performed on a single instance of these variables. Trying to describe all these possible variations and their implications in a formal way as shown in the previous sections would result in very large, complex and difficult to understand graphs. So instead, I resort to simple textual description for most of the relevant inter-operations. Similar to the formal notation, I will mark references in the text to atomic operations with the according number from the source code, e.g., *(6)* would reference the operation with the number 6 from the Stamp-it source code. In the source code each atomic operation is annotated with a number and a list of numbers referencing other operations that this operation synchronizes-with.

The `push` method

Suppose we want to insert the block *b* into the `thread_order_queue`. The *push* operation first sets *b*'s `next` pointer to `head` *(1)*, implicitly clearing the delete mark. Then it loads `head`'s `prev` pointer, performs the fetch-add on `head`'s `stamp` *(2)*, and stores the new value in *b*'s `stamp` *(3)*, implicitly clearing the `NotInList`, but setting the `PendingPush` flag. In the next step it stores the previously loaded `prev` pointer in *b*'s `prev` *(4)* and attempts a CAS on `head`'s `prev` *(5)* to finally insert *b* into the list. If this CAS fails, it simply restarts the loop and tries again. Otherwise, *b* is now part of the `prev` list, so it performs another write to *b*'s `stamp` to clear the `PendingPush` flag *(6)*.

Finally, it tries to update the `next` pointer of our successor (*prev*). It first performs an acquire-load on *prev*'s `next` *(7)*, followed by a loop in which it checks if *prev*'s `next` is marked, has already been set to point to *b*, or if the `prev` pointer of our block has been updated. The later case would imply that *prev* has been removed from the `prev` list, which in turn implies that the `next` list will be updated as part of that remove operation anyway. So if either of those checks is true, we immediately terminate the loop as there is nothing left to be done. Otherwise, we perform the CAS operation to update our *prev*'s `next` pointer *(8)*. If the CAS succeeds, it uses release-semantics to write the new value to `head`'s `prev`. If the CAS fails, it uses acquire-semantics for the reload. This acquire-reload synchronizes-with any release-store on `next` *(1, 8, 27)*, ensuring that we see any potential updates to our block's `prev` pointer in the next iteration.

All the store operations use release-semantic to ensure that any remove operations recognize the changes made to the block we are currently inserting. The CAS that inserts the block *(5)* uses `memory_order_acq_rel` and is therefore establishing a synchronize-with relation with all previous `push` operations (acquire-semantics), as well as any future `push` operations (release-semantics).

The `remove_from_prev_list` method

In the `remove_from_prev_list` method we look for the predecessor (*next*) of our block *b* and try to remove *b* by performing a CAS operation on that predecessor's `prev` pointer, setting it to some block (*prev*) that follows *b* in `prev` direction. So in order for `remove_from_prev_list` to work correctly we have to ensure that:

1. the CAS operation updates a block that is part of the `prev` list,
2. the CAS operation stores a pointer to a successor of *b* in `prev` direction that is still part of the `prev` list at the time the CAS operation is performed,
3. all blocks on the path in `prev` direction from *next* to *prev* are marked, i.e., no unmarked blocks get removed.

The CAS operation is only performed if *next*'s `prev` pointer is unmarked and points to *b*. The `prev` pointer being unmarked implies that *next* can only be "in the queue" or "getting inserted" (see Figure 4.2). To ensure that we are actually dealing with a block that is "in the queue", we load a consistent set of `prev` and `stamp` values from *next*: First we perform an acquire-load on `prev` *(18)*, followed by an acquire-load of `stamp` *(19)*, and finally a relaxed reload of `prev` to ensure that it matches the value returned by the previous load. When the first acquire-load *(18)* returns an unmarked value, it synchronizes-with the release-store in `push` *(4)*, ensuring that the following load returns the latest value of `stamp`, i.e., the value with the `PendingPush` flag which is written immediately before the `prev` pointer, or some newer value. The subsequent acquire-load of `stamp` *(19)* in turn synchronizes-with the release-store in `push` *(3, 6)*, and therefore ensures that we see the latest value written to `prev`, i.e., if the `stamp` value we read was written by a subsequent insert operation (i.e., the block was inserted, removed and reinserted again), we would recognize this since `prev` must have changed.

If the reload of `prev` does not match the previously read value, we simply restart the loop. Otherwise, we check whether `stamp` has the `NotInList` or `PendingPush` flag set. If this is the case, we cannot use this block, so we set *next* to *next*'s `next` pointer and restart the loop. Otherwise, we know that *next* is "in the queue" and that we have loaded the correct `prev` value.

In the next step we call `remove_or_skip_marked_block` with the just loaded values from *next*. This function checks whether *next*'s `prev` is marked, and if that is the case tries to remove *next* (if we have a valid *last*), or moves *next* to the next block in `next` direction. It is described in more detail later.

Assuming that *next*'s `prev` is not marked, we continue with checking whether *next*'s `prev` points to *b*, so we can safely attempt the CAS to remove *b* *(21)*, thus ensuring the first property. In case *next*'s `prev` does not point to *b*, we try to move *next* in `prev` direction (i.e., towards *b*). But in order to do so we must ensure that the new block does not have the `PendingPush` flag set. This is done in `save_next_as_last_and_move_next_to_next_prev`.

This approach also ensures the second property. A block can only remove itself by updating its immediate predecessor, which requires that this predecessor must not be marked. If it is marked, it has to be removed first. Suppose we want to remove the block *b* and our successor in `prev` direction is *prev*. According to the second property we must ensure that we only set *next*'s `prev` pointer to *prev* iff *prev* is still part of the `prev` list. Suppose that *prev* has been removed in the meantime. For *prev* to be removed it has to update its own predecessor (*b*), but since *b* is marked it first has to remove *b*. So when the owning thread of *b* performs a CAS to update *next*'s `prev` pointer to point to *prev* *(21)*, this can only succeed if *prev* is still part of the list. Otherwise, the remove operation of *prev* would have already removed *b* and the CAS would therefore fail since *next*'s `prev` pointer is no longer pointing to *b*. This is also the reason why it is sufficient to use relaxed loads when reading the `prev` and `stamp` values from *prev*. The third property also follows from this approach, since initially *prev* is set to the immediate successor of *b* in `prev` direction, and *prev* is only moved towards tail when it is marked.

The `save_next_as_last_and_move_next_to_next_prev` method

This method takes the current *next* and the previously loaded `prev` pointer from *next* (`next_prev`) as input parameters. It tries to move *next* to the block pointed to by `next_prev` while ensuring that the new block does not have the `PendingPush` flag set. We previously established that *next* is part of the `prev` list and that we loaded the correct `prev` pointer, i.e., the block pointed to by `next_prev` was also part of the `prev` list at that time. So when we load `next_prev`'s `stamp` *(30)* there can be two reasons for the `PendingPush` flag to be set: Either that block's thread was not yet able to finish its `push` operation, or the block has been removed in the meantime and is now getting re-inserted. To be able to tell the difference we perform an acquire-load on `next_prev`'s `stamp` *(30)*, and if it has the `PendingPush` flag set we perform a reload of *next*'s `prev` pointer, verifying that it still matches `next_prev`. The acquire-load *(30)* synchronizes-with the release-store in `push` that sets the `PendingPush` flag *(3)*. So in case the block has been removed and is now beeing reinserted, the reload of `prev` would return a different value, since the remove operation of `next_prev` must have updated *next*'s `prev` and this update is sequenced-before the release-store of `stamp` *(3)*.

This allows us to determine whether `next_prev` is still part of the `prev` list and, if necessary, help to reset its the `PendingPush` flag.

The `remove_or_skip_marked_block` method

This method takes four parameters: the current *next* pointer together with its `prev` and `stamp` values, as well as the current *last* pointer. The `prev` and `stamp` values have to reflect the state of *next* at a time it was "in the queue". When `next_prev` has the delete flag set and we also succeed in setting the delete flag in *next*'s `next` pointer, we attempt to remove *next* by updating *last*'s `prev` pointer *(28)*. *last* used to be our *next* pointer in a previous iteration and we obtained the new *next* by following its `prev` pointer, so our current *next* reflects the `prev` value of *last* at that time. Therefore, we can use our current *next* as expected value for the CAS *(28)*. It is guaranteed that the CAS *(28)* can only succeed iff neither *last* nor *next* have been removed in the meantime, since in either case *last*'s `prev` would have been updated.

The `remove_from_next_list` method

In this method we try to remove *b* by updating the `next` pointer of *prev* (the predecessor of *b* in `next` direction) to *next*. So in order for `remove_from_next_list` to work correctly we have to ensure that the CAS operation:

1. is performed on a block that is part of the `next` list,
2. stores a pointer to a successor of *b* in `next` direction that is still part of the `next` list at the time the CAS operation is performed.

We first perform the same set of steps as in `remove_from_prev_list` to obtain a consistent set of `prev` and `stamp` values from *next*, and verify that *next* does not have the `NotInList` or `PendingPush` flags set. Then we perform an acquire-load on *prev*'s `next` *(25)*, followed by a relaxed load of *prev*'s `stamp`. If *prev* was already reinserted (or is currently in the process of getting reinserted), the acquire-load *(25)* would synchronize-with the release-store in `push` *(1, 8)*, ensuring that the subsequent load of `stamp` would return the value with the NotInList flag set or some newer value. Thus, we would recognize that *prev* has been removed already and can therefore ensure the first property.

Otherwise, we have loaded the prev_next value we will need to update, but before we can do that we have to ensure that we have the right *next* block. For this we use the same checks and helper methods as in remove_from_prev_list (i.e., remove_or_skip_block and save_next_as_last_and_move_next_to_next_prev) to ensure that *next* is not marked and that it is in fact the predecessor of *prev*.

Finally we perform a reload of *next*'s prev pointer before we attempt the CAS. This is necessary to ensure that *next* has not been removed since we read its prev and stamp, so we do not update *prev* to point to a block that is no longer part of the next list (this ensures the second property). Assume that *next* has been removed. In this case the remove operation would have had to update *prev*'s next pointer as part of its own remove_from_next_list call. If the acquire-load of *prev*'s next pointer *(25)* returns the updated value, the load synchronizes-with the release-CAS *(27)*, so the reload of *next*'s prev would return a new value, signaling that we have to restart the loop. Otherwise we can safely attempt the CAS, since it will simply fail in case the load has returned an old value or *prev* has been removed in the meantime.

The **update_tail_stamp** method

In update_tail_stamp we try to find the immediate predecessor of tail and update tail's stamp to the same value as its predecessor. To that end we perform an acquire-load on tail's next pointer *(14)*, returning a pointer to the potential predecessor (*last*), followed by an acquire-load of *last*'s prev *(15)* and a relaxed-load of *last*'s stamp. The acquire-load of tail's next *(14)* synchronizes-with one of the two possible release-stores *(8, 27)*, ensuring that the subsequent loads from *last* return the updated values. This allows us to verify in the next step whether *last*'s stamp is actually greater than the value we want to update, as well as whether *last* is an actual predecessor of tail. If that is the case we check if *last* points to head. In that case we have to perform a CAS on head's prev, trying to set it to the previously loaded value with an updated tag. This is necessary to ensure we only use the stamp from head if there is no pending push operation that might insert a block with a lower stamp. If the CAS is successful, any pending push operation will have to perform a retry and therefore get a new (greater) stamp. Otherwise, some other thread interfered, so we simply resort to the “next best guess” that was passed to update_tail_stamp by the caller. If on the other hand *last* does not point to head, we can just use the previously loaded stamp.

All that remains is a simple CAS loop that tries to update tail's stamp, as long as the value we want to write is greater than the current value.

We still have to show that a node is only reclaimed when it is guaranteed that no thread is holding a reference to it. Assume, without loss of generality, that thread t_1 removes some node n from a data structure, fetches the current stamp from head, stores this stamp value in n and adds n to its internal retire-list. n can safely be reclaimed once all threads that were in a critical region at the time n was removed have left their respective critical region. `tail`'s `stamp` is less or equal to the `stamp` of the last thread. So n's `stamp` being less or equal to `tail`'s `stamp` implies that all threads currently inside a critical region (if any) have entered their respective critical region *after* n was removed, so n can safely be reclaimed.

When t_1 adds n to its internal retire-list, it performs a sequentially consistent load *(12)* to obtain the current `stamp` from `head`, and stores it in n. When t_2 inserts itself into the `thread_order_queue`, it performs a sequentially consistent fetch-add on `head`'s `stamp` *(2)* and stores the returned value in its own block. Since both operations are sequentially consistent, there is a total order. If the load *(12)* is ordered before the fetch-add *(2)*, it is clear that n was removed before t_2 entered its critical region and therefore t_2 does not block the reclamation of n. If the fetch-add *(2)* is ordered before the load *(12)*, then the load will return the new `stamp`, which implies that n's `stamp` will be larger than t_2's `stamp`, so n can only be reclaimed once t_2 has left its critical region.

What remains to be shown is that there exists a happens-before relation between the reclaim operation and any changes made to the reclaimed node. A node can only be accessed by a thread that is inside a critical region, so it suffices to show that there exists a happens-before relation between the reclaim operation and any remove operation of a thread with a lower stamp than the node that gets reclaimed.

To determinate if a node can be reclaimed, a thread obtains the `stamp` from `tail` using an acquire-load *(13)*. This load synchronizes-with the corresponding release-CAS in `update_tail_stamp` *(16)*, so in consequence we need a happens-before relation between this release-CAS and the remove operations of all blocks that had a lower stamp than the new stamp value. This relation will be established with the help of the `prev` list.

In the `remove` operation, the very first step is to set the delete mark on the block's `prev` pointer *(9)*. This is done using an acquire-release-CAS operation. Any acquire-load on that block's `prev` pointer *(15, 17, 18, 22, 26)* that returns the marked value therefore synchronizes-with the CAS. On the other hand, the acquire-semantics ensures that a synchronize-with relation is established with any potential release operation *(21, 28)* that updated that block's `prev` as part of some earlier `remove` operation.

When a thread attempts to remove its block, there are two possible scenarios: Either it removes the block itself, or some other thread removes it as part of its own `remove` operation. Since we use the `prev` list to establish the required happens-before relation, we only care about the removal of a block from the `prev` list.

When a thread removes itself from the `prev` list, it updates its predecessor's `prev` pointer using a release-CAS *(21)*. So when that predecessor starts its own `remove` operation, the acquire-CAS that sets the delete mark *(9)* synchronizes-with this release-CAS *(21)*.

In order for a block to be removed by some other thread, that thread has to recognize that the block's `prev` pointer is marked. In `remove_from_prev_list` we obtain *prev*'s `prev` using a relaxed-load. If it has the mark flag set, we try to set the mark flag on `next`, and if that is successful, we perform another acquire-load on `prev` *(17)*. With regards to *next*, we already use acquire-semantics when we load its `prev` value *(18)*.

In the `remove_from_next_list` operation we use acquire-semantics to load *next*'s `prev` pointer *(22)*. With regards to *prev*, we first perform an acquire-load on its `next` *(25)*, and if it has the mark flag set, we perform an additional acquire-load on `prev` *(26)*.

So both CAS operations that remove a block from the `prev` list use release-semantics *(21, 28)*, and we always perform an acquire-load on the block's `prev` pointer before we try to remove it *(17, 18, 22, 26)*. This results in a chain of synchronize-with relations between the different operations on the `prev` pointers of the removed blocks. In `update_tail_stamp` we first perform the load on `tail`'s `next` *(14)* to get the current predecessor of `tail` (we refer to it as *last*), and then perform an acquire-load on *last*'s `prev` *(15)*. Therefore, this acquire-load *(15)* synchronizes-with the according release operation that was last performed on that `prev` pointer (as well as any release sequences on that object). A block that was a successor of *last* in `prev` direction will have removed itself by updating *last*'s `prev` pointer using a release-CAS. Thus the acquire-load *(15)* establishes a happens-before relation between the load and the removal of that block.

This also holds for the cases were we use the "next best guess" to update `tail`'s `stamp`. We use the "next best guess" instead of *last*'s `stamp` in the following situations:

1. When `tail` already has a greater `stamp`.

2. When *last*'s `prev` does not point to `tail`.

3. When `tail`'s `next` has been updated since the first load.

4. When *last* is `head` and the update of `head`'s `prev` fails.

In the first case some other thread was faster, so there is nothing left to do. In the other cases we cannot use *last*'s `stamp`, since we cannot be sure if it was `tail`'s predecessor at the time we loaded the `stamp` value. However, we know that it must have been `tail`'s predecessor at some point, because when we removed our own block we (or some other thread) had to remove it from the `next` list by updating our predecessor in `next` direction (i.e., `tail`). So the load of `tail`'s `next` *(14)* can either return the value we (or some other thread) wrote as part of the remove operation for our block, or some newer value. But a newer value could only be written by some other `remove` operation, or as the last step of some `push`. Either way, it is guaranteed that (i) *last* points to a block that was the immediate predecessor of `tail` at some point, and (ii) that it had a stamp value greater or equal to our "next best guess" at that time. Therefore, we can safely use our "next best guess" to update `tail`'s `stamp`, because the acquire-load of *last*'s `prev` still establishes the required happens-before relation.

Together with the release-CAS that updates `tail`'s `stamp` *(16)*, we finally have the happens-before relations between the reclamation of a node and the remove operations of all blocks that had a lower stamp.

4.5 Data structures

The following chapters describe the three lock-free data structures that were implemented and used in the benchmarks.

4.5.1 Lock-Free List-based Set

This is an implementation of Harris' list-based set [Har01]. It is based on the improved implementation by Michael [Mic02], but adapted to the previously described interface to allow the use of arbitrary reclamation schemes. The data structure is based on a totally-ordered set of `keys`, implemented as a sorted singly linked list. However, I will not go into too much detail about the algorithm itself, but focus on the usage of the reclamation scheme and the semantics of the atomic operations.

The definition of the `list` class is shown in Listing 4.48. The private section contains some boilerplate code that defines a few aliases for `concurrent_ptr`, `marked_ptr` and `guard_ptr` to make the rest of the code more readable.

Listing 4.48: Definition of list class

```cpp
template <class Key, class Reclaimer>
class list
{
public:
  list() = default;
  ~list();

  bool search(Key key);
  bool remove(Key key);
  bool insert(Key key);

private:
  struct node;

  using concurrent_ptr = typename Reclaimer::template concurrent_ptr<node, 1>;
  using marked_ptr = typename concurrent_ptr::marked_ptr;
  using guard_ptr = typename concurrent_ptr::guard_ptr;

  struct node : Reclaimer::template enable_concurrent_ptr<node, 1>
  {
    const Key key;
    concurrent_ptr next;
    node(Key k) : key(std::move(k)), next() {}
  };

  concurrent_ptr head;

  struct find_info
  {
    concurrent_ptr* prev;
    marked_ptr next;
    guard_ptr cur;
    guard_ptr save;
  };
  bool find(Key key, find_info& info, detail::backoff& backoff);
};
```

The `find` method (shown in Listing 4.49) is the centerpiece of the data structure; all other methods are built on top of it. It performs a linear search and returns a boolean value indicating whether a node with a matching key was found. In either case, by its completion, it guarantees that the `find_info` structure has captured a snapshot of a segment of the list including the node (if any) that contains the lowest key value greater than or equal to the input key, as well as a reference to the predecessor's `next` pointer. During the traversal of the list, whenever a marked node is encountered, it attempts to remove the node from the list using a CAS operation. If successful, the removed node is marked for reclamation.

Upon return, the snapshot in `find_info` holds the following values:

- `prev` holds a reference to the `concurrent_ptr` referencing `cur`. This can either be a reference to `head` or to the `next` pointer of some other node that is the predecessor of `cur`.
- `cur` holds a safe reference to the first node with a key value greater than or equal to the input key. If the list is empty, `cur` is set to `nullptr`.
- `next` holds a reference to the successor of `cur` (if any).
- `save` holds a safe reference to the predecessor of `cur` (if any).

The traversal requires at most two `guard_ptr`s at any time: one that references the current node (`find_info.cur`), if any, and that references its predecessor (`find_info.save`), if any [Mic04a].

Listing 4.49: Implementation of list::find

```
template <class Key, class Reclaimer>
bool list<Key, Reclaimer>::find(Key key, find_info& info, detail::backoff& backoff)
{
retry:
  info.prev = &head;
  info.next = info.prev->load(std::memory_order_relaxed);
  info.save.reset();

  for (;;)
  {
    // (1) - this acquire-load synchronizes-with the release-CAS (3, 4, 6)
    if (!info.cur.acquire_if_equal(*info.prev, info.next, std::memory_order_acquire))
      goto retry;

    if (!info.cur)
      return false;

    info.next = info.cur->next.load(std::memory_order_relaxed);
    if (info.next.mark() != 0)
    {
      // Node *cur is marked for deletion -> update the link and retire the element

      // (2) - this acquire-load synchronizes-with the release-CAS (3, 4, 6)
      info.next = info.cur->next.load(std::memory_order_acquire).get();

      // Try to splice out node
      marked_ptr expected = info.cur.get();
      // (3) - this release-CAS synchronizes with the acquire-load (1, 2)
      //       it is the head of a potential release sequence containing (5)
      if (!info.prev->compare_exchange_weak(expected, info.next,
                                            std::memory_order_release,
                                            std::memory_order_relaxed))
      {
        backoff();
        goto retry;
      }
      info.cur.reclaim();
    }
    else
    {
      if (info.prev->load(std::memory_order_relaxed) != info.cur.get())
        goto retry; // cur might be cut from list.

      Key ckey = info.cur->key;
      if (ckey >= key)
        return ckey == key;

      info.prev = &info.cur->next;
      info.save = std::move(info.cur);
    }
  }
}
```

The `insert` operation (shown in Listing 4.50) first creates a new node with the given key. It then continuously calls `find` to check if another node with the same key is already part of the list. If that is the case, the newly created node is deleted and the function returns false. Otherwise, it sets the new nodes `next` pointer to `find_info.next` and attempts to insert the new node before `find_info.cur` by performing a CAS operation on `find_info.prev`. If the CAS operation succeeds, then we have successfully inserted the node and therefore return `true`. Otherwise we restart the loop with a new call to `find`.

Listing 4.50: Implementation of list::insert

```
template <class Key, class Reclaimer>
bool list<Key, Reclaimer>::insert(Key key)
{
  node* n = new node(std::move(key));
  find_info info;
  detail::backoff backoff;
  for (;;)
```

```cpp
  {
    if (find(key, info, backoff))
    {
      delete n;
      return false;
    }
    // Try to install new node
    marked_ptr cur = info.cur.get();
    n->next.store(cur, std::memory_order_relaxed);

    // (4) - this release-CAS synchronizes with the acquire-load (1, 2)
    //       it is the head of a potential release sequence containing (5)
    if (info.prev->compare_exchange_weak(cur, n,
                                         std::memory_order_release,
                                         std::memory_order_relaxed))
      return true;

    backoff();
  }
}
```

The `remove` operation (shown in Listing 4.51) continuously calls `find` to check if the list contains a node with a matching key. If that is not the case it immediately returns false as there is nothing to do. Otherwise it performs a CAS operation to set the mark bit on `cur->next`. This mark bit serves two purposes: on the one hand it signals other threads that this node should be removed from the list, and at the same time it prevents other threads to update `cur->next` with a CAS operation (e.g., in an attempt to insert a node). If it fails to set the mark bit it simply restarts the loop. Otherwise it attempts yet another CAS to update `prev` with `next`, effectively removing `cur` from the list. If that CAS operation succeeds, it can thus mark `cur` for reclamation. Otherwise some other thread has interfered and it is not known if `prev` is the actual predecessor of `cur`. So it simply performs another call to `find` with the same key, causing a traversal of the list that removes any marked nodes, including `cur` in case it has not yet been removed by some other thread.

Listing 4.51: Implementation of list::remove

```cpp
template <class Key, class Reclaimer>
bool list<Key, Reclaimer>::remove(Key key)
{
  detail::backoff backoff;
  find_info info;
  // Find node in list with matching key and mark it for reclamation.
  do
  {
    if (!find(key, info, backoff))
      return false; // No such node in the list
    // (5) - this CAS operation is part of a release sequence headed by (3, 4, 6)
  } while (!info.cur->next.compare_exchange_weak(info.next,
                                                 marked_ptr(info.next.get(), 1),
                                                 std::memory_order_relaxed));

  // Try to splice out node
  marked_ptr expected = info.cur;
  // (6) - this release-CAS synchronizes with the acquire-load (1, 2)
  //       it is the head of a potential release sequence containing (5)
  if (info.prev->compare_exchange_weak(expected, info.next,
                                       std::memory_order_release,
                                       std::memory_order_relaxed))
    info.cur.reclaim();
  else
    // Another thread interfered -> rewalk the list to ensure reclamation of marked
    node before returning.
    find(key, info, backoff);

  return true;
}
```

4.5.1.1 Correctness

Michael already showed in [Mic04a] that only two hazard pointers are required to ensure a safe and correct execution of the algorithm. My implementation closely follows Michael's, except that I replaced the hazard pointers with the more generic `guard_ptrs`, so I will not provide any arguments in this regard.

What remains to be shown, however, is the correctness of the implementation with regards to the used memory semantics. In particular, we have to show that there is a happens-before relation between the insertion of a node and any subsequent access to the node's `key`, i.e.:

$$t_1\colon \texttt{n->key.store} \xrightarrow{\text{hb}} t_2\colon \texttt{cur->key.load}$$

`insert` creates a new node and sets `key` as well as `next` before it performs the CAS operation that inserts that node into the list (4.34). It uses release semantics for the CAS operation to ensure the required happens-before relation.

$$t_1 : \underbrace{\texttt{n->key.store} \xrightarrow{\text{sb}} \texttt{n->next.store}_{rlx} \xrightarrow{\text{sb}} \texttt{prev->cmpxchg\_weak}^{(4)}_{rel}}_{\texttt{insert}} \tag{4.34}$$

`remove` marks the `next` pointer of the returned node (if any) using a CAS operation and then tries to update the `next` pointer of the node's predecessor using another CAS (4.35).

$$t_1 : \underbrace{\texttt{cur->next.cmpxchg\_weak}^{(5)}_{rlx} \xrightarrow{\text{sb}} \texttt{prev->cmpxchg\_weak}^{(6)}_{rel}}_{\texttt{remove}} \tag{4.35}$$

`find` uses acquire semantics to obtain the `guard_ptr` to `cur` and then performs a relaxed load on `cur`'s `next` pointer. If it encounters that `cur` is marked, it performs another load of `next`, but this time using acquire semantics, and then attempts a release-CAS to remove the node from the list. Otherwise, it reads the node's `key` (4.36); this is the load operation for which we have to show that it happens after the initial store.

In most cases it is sufficient to use a relaxed load for `next`, as in the next iteration this is the value that is then used as the expected value in the call to `acquire_if_equal`, which in turn uses acquire semantics. However, when a marked node is encountered we have to perform an additional acquire-load to ensure a proper happens-before relation as explained later.

$$\begin{array}{l} \text{if cur is marked:} \\ t_1 : \underbrace{\texttt{prev->load}^{(1)}_{acq} \xrightarrow{\text{sb}} \texttt{cur->next.load}^{(2)}_{acq} \xrightarrow{\text{sb}} \texttt{prev->cmpxchg\_weak}^{(3)}_{rel}}_{\texttt{find}} \\ \text{if cur is not marked:} \\ t_1 : \underbrace{\texttt{prev->load}^{(1)}_{acq} \xrightarrow{\text{sb}} \texttt{cur->key.load}}_{\texttt{find}} \end{array} \tag{4.36}$$

Since the `insert` function uses a release-CAS to insert the new node (4.34), in the simplest case the acquire-load operation in `find` synchronizes with that release-CAS (4.37).

$$\begin{array}{c} \underbrace{t_1\texttt{: prev->cmpxchg\_weak}^{(4)}_{rel}}_{\texttt{insert}} \xrightarrow{\text{sw}} \underbrace{t_2\texttt{: prev->load}^{(1)}_{acq}}_{\texttt{find}} \implies \\ \underbrace{t_1\texttt{: n->key.store}}_{\texttt{insert}} \xrightarrow{\text{hb}} \underbrace{t_2\texttt{: cur->key.load}}_{\texttt{find}} \end{array} \tag{4.37}$$

However, things get a little more complicated when `remove` operations come into play. We have a release-CAS operation in `find` (3) and another one in `remove` (6) that remove a marked node from the list. When the acquire-load (1) in `find` reads the value written by one of those two operations, it therefore synchronizes with the respective CAS operation (4.38).

$$\begin{array}{c} \underbrace{t_1\texttt{: prev->cmpxchg\_weak}^{(3)}_{rel}}_{\texttt{find}} \xrightarrow{\text{rf}} \underbrace{t_2\texttt{: prev->load}^{(1)}_{acq}}_{\texttt{find}} \implies \\ t_1\texttt{: prev->cmpxchg\_weak}^{(3)}_{rel} \xrightarrow{\text{sw}} t_2\texttt{: prev->load}^{(1)}_{acq} \\ \\ \underbrace{t_1\texttt{: prev->cmpxchg\_weak}^{(6)}_{rel}}_{\texttt{remove}} \xrightarrow{\text{rf}} \underbrace{t_2\texttt{: prev->load}^{(1)}_{acq}}_{\texttt{find}} \implies \\ t_1\texttt{: prev->cmpxchg\_weak}^{(3)}_{rel} \xrightarrow{\text{sw}} t_2\texttt{: prev->load}^{(1)}_{acq} \end{array} \tag{4.38}$$

That leaves only the CAS operation that marks the `next` pointer of the node to be removed. This one uses `memory_order_relaxed`, but since it is a read-modify-write operation, it is part of a release sequence headed by the CAS operation that wrote the pointer's last value, i.e., either the release-CAS in `insert` (4) that inserts a node, or one of the release-CAS in `find` (3) or `remove` (6) that remove a node from the list.

Once a pointer is marked, it is guaranteed that its value will not change for the rest of the node's lifetime. So when `find` encounters a marked node, it can simply perform a reload of `next` with acquire semantics to establish a happens-before relation with the CAS operation that heads the release sequence (4.39).

$$\begin{gathered}
\underbrace{t_1\texttt{: prev->cmpxchg\_weak}^{(4)}_{rel}}_{\texttt{insert}}\underbrace{[t_2\texttt{: prev->cmpxchg\_weak}^{(5)}_{rlx}]}_{\texttt{remove}} \xrightarrow{\text{sw}} \underbrace{t_3\texttt{: prev->load}^{(2)}_{acq}}_{\texttt{find}} \\
\underbrace{t_1\texttt{: prev->cmpxchg\_weak}^{(3)}_{rel}}_{\texttt{find}}\underbrace{[t_2\texttt{: prev->cmpxchg\_weak}^{(5)}_{rlx}]}_{\texttt{remove}} \xrightarrow{\text{sw}} \underbrace{t_3\texttt{: prev->load}^{(2)}_{acq}}_{\texttt{find}} \\
\underbrace{t_1\texttt{: prev->cmpxchg\_weak}^{(6)}_{rel}}_{\texttt{remove}}\underbrace{[t_2\texttt{: prev->cmpxchg\_weak}^{(5)}_{rlx}]}_{\texttt{remove}} \xrightarrow{\text{sw}} \underbrace{t_3\texttt{: prev->load}^{(2)}_{acq}}_{\texttt{find}}
\end{gathered} \tag{4.39}$$

There can be an arbitrary number of concurrent `insert` and `remove` operations, leading to a combination of all the described synchronize-with relations engaging with each other, eventually establishing a happens-before relation between the initialization of a node's `key` and any subsequent access to that `key` by some other thread.

4.5.2 Lock-Free Queue

This is an implementation of a lock-free queue based on the proposal by Michael and Scott [MS96]. It implements the queue as a singly-linked list with `head` and `tail` pointers. `head` always points to a dummy node, which is the first node in the list, and `tail` points to either the last or second last node in the list. The nodes are connected via `next` pointers, so the first node that actually contains a value is the one pointed to by `head.next`.

The definition of the `queue` class is shown in Listing 4.52. The private section contains some boilerplate code that defines a few aliases for `concurrent_ptr`, `marked_ptr` and `guard_ptr` to make the rest of the code more readable.

Listing 4.52: Definition of queue class.

```
template <class T, class Reclaimer>
class queue {
public:
  queue();
  ~queue();

  void enqueue(T value);
  bool try_dequeue(T& result);
private:
  struct node;

  using concurrent_ptr = typename Reclaimer::template concurrent_ptr<node, 0>;
  using marked_ptr = typename concurrent_ptr::marked_ptr;
  using guard_ptr = typename concurrent_ptr::guard_ptr;

  struct node : Reclaimer::template enable_concurrent_ptr<node>
  {
    T value;
    concurrent_ptr next;
  };

  alignas(64) concurrent_ptr head;
  alignas(64) concurrent_ptr tail;
};
```

The `enqueue` operation (shown in Listings 4.53) first creates the new node before it enters a loop to insert this node into the linked list. In the loop it first reads the `tail` pointer and checks if `tail`'s next pointer is null. If that is not the case some other thread has inserted a node, but did not yet update `tail`, i.e., `tail` is pointing to the second last node. So we try to help the other thread finishing its operation and then retry to insert our own node. Otherwise we can perform a CAS operation to insert our node. If the CAS is successful we have inserted our node, but we still need to perform another CAS to update `tail` to point to our newly inserted node. However, we can safely ignore a failure of this last CAS, as this would just mean that another thread was faster in helping to update `tail`.

Listing 4.53: Implementation of queue::enqueue

```
template <class T, class Reclaimer>
void queue<T, Reclaimer>::enqueue(T value)
{
  node* n = new node{};
  n->value = std::move(value);

  detail::backoff backoff;

  guard_ptr t;
  for (;;)
  {
    // Get the old tail pointer.
    t.acquire(tail, std::memory_order_relaxed);

    // Help update the tail pointer if needed.
    auto next = t->next.load(std::memory_order_relaxed);
    if (next.get() != nullptr)
    {
      marked_ptr expected(t.get());
      tail.compare_exchange_weak(expected, next, std::memory_order_relaxed);
      continue;
    }

    // Attempt to link in the new element.
    marked_ptr null{};
    // (1) - this release-CAS synchronizes-with the acquire-load (2).
    if (t->next.compare_exchange_weak(null, n, std::memory_order_release,
                                      std::memory_order_relaxed))
      break;

    backoff();
  }
```

```cpp

  // Swing the tail to the new element.
  marked_ptr expected = t.get();
  tail.compare_exchange_strong(expected, n, std::memory_order_relaxed);
}
```

The `try_dequeue` operation (shown in Listing 4.54) tries to remove the node pointed to by `head` from the linked list. Since `head` always points to a dummy node we are actually interested in the value of the node pointed by `head.next`. If the remove operation succeeds, the `next` node's value is written to the `result` out-parameter, the `next` node now becomes the new dummy node pointed to by `head` and the function returns `true`. Otherwise the function returns `false` and `result` remains unchanged.

In the implementation we first acquire a `guard_ptr` to `head` and then another `guard_ptr` to the `head`'s `next`. Before continuing, we check if `head` has changed in the meantime, in which case we have to restart. The acquired guard to next being null means that the queue is empty, thus we simply return `false`. Otherwise we check if `head` equals `tail`, and if that is the case, there must be a pending `enqueue` operation that inserted the node, but did not yet update `tail`. So we help with that and restart from the beginning. If `head` is not equal to `tail` we can read `next`'s value and try a CAS operation to set `head` to `next`, effectively removing the node currently pointed to by `head`. Thus, if the CAS operation is successful, we can mark the old dummy node for reclamation and return `true`.

Listing 4.54: Implementation of queue::try_dequeue

```cpp
template <class T, class Reclaimer>
bool queue<T, Reclaimer>::try_dequeue(T& result)
{
  detail::backoff backoff;

  guard_ptr h;
  for (;;)
  {
    // Get the old head and tail elements.
    h.acquire(head, std::memory_order_relaxed);

    // Get the head element's successor.
    // (2) - this acquire-load synchronizes-with the release-CAS (1).
    auto next = acquire_guard(h->next, std::memory_order_acquire);
    if (head.load(std::memory_order_relaxed).get() != h.get())
      continue;

    // If the head (dummy) element is the only one, return false to signal that
    // the operation has failed (no element has been returned).
    if (next.get() == nullptr)
      return false;

    marked_ptr t = tail.load(std::memory_order_relaxed);

    // There are multiple elements. Help update tail if needed.
    if (h.get() == t.get())
27     {
28       tail.compare_exchange_weak(t, next, std::memory_order_relaxed);
29       continue;
30     }
31 
32     // Save the data of the head’s successor. It will become the new dummy node.
33     result = next->value;
34 
35     // Attempt to update the head pointer so that it points to the new dummy node.
36     marked_ptr expected(h.get());
37     if (head.compare_exchange_weak(expected, next, std::memory_order_relaxed))
38       break;
39 
40     backoff();
41   }
42 
43   // The old dummy node has been unlinked, so reclaim it.
44   h.reclaim();
45 
46   return true;
47 }
```

4.5.2.1 Correctness

The correctness of this algorithm was already shown by Michael [MS96]. What remains to be shown, however, is that our usage of the reclamation scheme is correct and, since we use the weaker acquire/release semantics, that there is a happens-before relation between the `enqueue` operation adding a node and a `try_dequeue` operation removing the same node. Or, to be more precise, that there is a happens-before relation between the operation that writes the value into the node (line 5 in Listing 4.53) and the operation that reads that value (line 33 in Listing 4.54).

The `enqueue` operation acquires a `guard_ptr` on `tail`, which allows us to safely access to the `next` pointer and eventually perform the CAS to insert the new node; the `guard_ptr` is held until the operation returns, thus ensuring that it can not be reclaimed.

The `try_dequeue` operation acquires a `guard_ptr` on `head`, which again allows us to safely access `next`, for which we acquire yet another `guard_ptr`, so we can safely read `next->value`. `try_dequeue` requires two `guard_ptr`s, because for non-empty lists it always operates on two nodes: the dummy node currently pointed to by `head` as well as the node with the actual value, and therefore both nodes have to be protected from reclamation.

Regarding the happens-before relation we have to show that:

$$t_1\texttt{: n->value.store} \xrightarrow{\text{hb}} t_2\texttt{: n->value.load()}$$

`enqueue` stores the value in the newly created node before it performs the CAS operation that inserts that node into the queue (4.40).

$$t_1 : \underbrace{\texttt{n->value.store} \xrightarrow{\text{sb}} \texttt{t->next.cmpxchg\_weak}^{(1)}_{rel}}_{\texttt{enqueue}} \tag{4.40}$$

$$t_2 : \underbrace{\texttt{h->next.load}^{(2)}_{acq} \xrightarrow{\text{sb}} \texttt{next->value.load()}}_{\texttt{try\_deuque}} \tag{4.41}$$

The `next` pointer of newly created nodes is initialized with `nullptr` and it can only be updated once. It thus follows that there are only two possible scenarios when some thread reads `next`: either the returned value is `nullptr`, then the queue appears to be empty for this thread (4.42), or the returned value is a valid pointer, then the acquire-load in `try_dequeue` synchronizes-with the release-CAS in `enqueue` (4.43).

$$\begin{gathered}\underbrace{t_1\texttt{: n->next.store(nullptr)}}_{\text{initialization of } n} \xrightarrow{\text{rf}} t_2\texttt{: h->next.load}^{(2)}_{acq} \implies \\ \text{Thread } t_2 \text{ reads null, so the queue appears to be empty.}\end{gathered} \tag{4.42}$$

$$\begin{gathered}t_1\texttt{: t->next.cmpxchg\_weak}^{(1)}_{rel} \xrightarrow{\text{rf}} t_2\texttt{: h->next.load}^{(2)}_{acq} \implies \\ t_1\texttt{: t->next.cmpxchg\_weak}^{(1)}_{rel} \xrightarrow{\text{sw}} t_2\texttt{: h->next.load}^{(2)}_{acq} \implies \\ t_1\texttt{: t->next->value.store} \xrightarrow{\text{hb}} t_2\texttt{: h->next->value.load()}\end{gathered} \tag{4.43}$$

It is therefore sufficient to use acquire/release semantics for the operations on the `next` pointers, all other atomic operations can be fully relaxed.

4.5.3 Lock-Free Hash Map

This is an implementation of a lock-free hash map based on the proposed algorithm by Michael [Mic02]. It is basically a container with a fixed number of list-based sets (as described above) where each list-based set represents a *bucket*.

However, the interface is slightly different because the benchmark that uses this hash map stores rather big data in it, so we want to be able to access the data directly. For this reason the interface does not return a copy of the data, but a `guard_ptr` to the internal node. This way the user can access the node's key and value without copying it as the `guard_ptr` protects the node from being reclaimed, even if it got removed from the hash map in the meantime.

The definition of the `hash_map` class is shown in Listing 4.55.

Listing 4.55: Definition of hash_map class.

```
template <class Key, class Value, class Reclaimer, size_t Buckets>
class hash_map
{
public:
  struct node;

```

```cpp
private:
  using concurrent_ptr = typename Reclaimer::template concurrent_ptr<node, 1>;
  using marked_ptr = typename concurrent_ptr::marked_ptr;

public:
  hash_map() = default;
  ~hash_map();

  using guard_ptr = typename concurrent_ptr::guard_ptr;

  struct node : Reclaimer::template enable_concurrent_ptr<node, 1>
  {
  public:
    const Key key;
    const Value value;
  private:
    concurrent_ptr next;
    node(Key k, Value v) : key(std::move(k)), value(std::move(v)), next() {}
    friend class hash_map;
  };

  guard_ptr search(Key key);
  bool insert(Key key, Value value);
  bool insert(Key key, Value value, guard_ptr& entry);
  bool remove(Key key);

private:
  concurrent_ptr buckets[Buckets];

  struct find_info
  {
    concurrent_ptr* prev;
    marked_ptr next;
    guard_ptr cur;
    guard_ptr save;
  };
  bool find(Key key, concurrent_ptr& head, find_info& info, detail::backoff& backoff);
};
```

It is quite similar to the definition of the list based set (see Listing 4.48), the main difference is that the `node` and `guard_ptr` definitions are public and that `search` and `insert` have different signatures; `search` returns a `guard_ptr` to the node and there is an overload for `insert` that has an additional out-parameter of type `guard_ptr` that is set to the newly created node when the insert operation succeeds.

4.6 Benchmarks

I have implemented the same set of list and queue micro-benchmarks that Hart et al. describe in [HMBW07]. While my implementation differs in several aspects, I tried to keep it close enough so that the results should be roughly comparable.

The tests are set up as follows. The main thread spawns N child threads and starts a timer. Every child thread performs operations on the data structure that is currently being measured until the timer expires. Upon timer expiry the child threads are stopped and the parent thread calculates the average execution time per operation by summing up the runtime of each child thread and its number of performed operations.

The following parameters can be set for every benchmark:

`-reclaimer` – the reclamation scheme that shall be used in the benchmark.

`-threads` – the number of threads that will concurrently perform benchmark operations. The default value is 4.

`-trials` – the number of trials the benchmark shall be executed. The default value is 8.

`-runtime` – the runtime of each trial in milliseconds. The default value is 10000.

`-memory-samples` – the number of samples that shall be taken during the runtime of a single trial. The default value is zero, i.e., by default this feature is disabled. When this is set to a value greater than zero, additional samples are automatically taken at the beginning (before threads are given the start signal) and at the end (after all threads have signaled that they are finished) of a trial. Therefore, when this parameter is set to n, for each trial a total of $n + 2$ samples are taken with an average runtime of `runtime`$/(n + 1)$ milliseconds between two consecutive samples.

There are a number of additional benchmark specific parameters which are described in the following chapters.

The experiments are *throughput oriented* in the following sense. The main thread spawns p child threads and waits from them to signal that they are fully initialized and ready to start. Once all threads are ready, the main thread starts a timer and sets a global start flag. Every child thread performs operations on the data structure under scrutiny. Upon timer expiry the main thread clears the global flag and waits for all child threads to stop. Then it calculates the average execution time per operation by summing up the runtime of each child thread and its number of performed operations.

When the taking of memory samples has been activated, the main thread regularly wakes up during the trial to gather some data and record it. This data consists of the process' *resident set size* (which corresponds to the used physical memory), the runtime, and optionally the number of allocated and reclaimed nodes. The function that is used to get the resident set size was taken from David Robert Nadeau [Nad12]. The recording of allocated and reclaimed nodes is disabled by default, but can be activated by defining the `TRACK_ALLOCATIONS` macro. When this is activated, the reclaimers will track the number of constructions and destructions of all `enable_concurrent_ptr` instances. To avoid contention between different threads, this is done using thread-local counters which are connected via a linked list, to enable easy calculation of the total sum over all threads when recording samples.

The function executed by each child thread consists of two nested loops, where the outer loop continues as long as the global flag is set. The inner loop performs i

iterations (i is benchmark specific) and in each iteration a benchmark specific operations is performed. For the Queue and List benchmark, this inner loop is inside the scope of a `region_guard` instance, thus allowing amortization of the costs of all the operations inside the loop, in case the reclamation scheme supports it. A pseudocode implementation of this inner loop is shown in Listing 4.56. When the inner loop finishes, the thread adds i to a thread-local counter of the total number of performed operations and continues with the next iteration of the outer loop.

Listing 4.56: Per-thread pseudocode for running a benchmark.

```
while (keep running flag is set)
{
  region_guard region;
  for i from 1 to n do
  {
    key = random key;
    op = random operation;
    d = data structure;
    op(d, key);
  }
}
```

For EBR and NEBR `UpdateThreshold` is set to 100, i.e., each thread tries to update the global epoch after 100 critical region entries since it last observed a new epoch. The threshold for the local retire-list in HPBR is calculated as $100 + \sum_{i=0}^{p} K_i * 2$ where p is the number of threads and K_i is the number of hazard pointers for the thread with index i. In case of the `static_hazard_pointer_policy` K_i is constant for all threads, which simplifies the calculation. However, the `dynamic_hazard_pointer_policy` allows threads to have a dynamic number of hazard pointers, so the calculation has to take this into account. This is explained in more detail in Section 4.4.2.2.

4.6.1 GuardPtr benchmark

This benchmark is used to measure the base cost of creating and destroying `guard_ptr` instances. The threads perform $i = 100$ iterations, where in each iteration a `guard_ptr` instance to a single shared node is created and, since it is running out of scope, immediately destroyed. The fact that all instances reference a single shared node represents the worst case scenario for LFRC, while for all other schemes this is an irrelevant detail.

4.6.2 Queue benchmark

This is a synthetic micro-benchmark based on the queue data structure. The threads perform $i = 100$ iterations and in each iteration performs a random `push` or `pop` operation. Both operations have equal probability, thus the data structure size kept roughly constant throughout a single trial.

The only possible parameter for this benchmark is `--elements` which defines the number of elements the queue should be prefilled with.

4.6.3 List benchmark

This is a synthetic micro-benchmark based on the list data structure. It supports the following parameters:

`-elements` – defines the number of elements the list should be prefilled with. The default value is 10.

`-modify-fraction` – defines the ratio of modify operations (i.e., insert and delete) to all operations; in the remaining text this parameter is also referred to as *workload*. The default value is 0.5.

The threads perform $i = 100$ iterations and in each iteration a random `insert`, `remove` or `search` operation is performed. The probability for the modifying operations is set via the command line parameter.

4.6.4 HashMap benchmark

This benchmark was inspired by a real-world application. Suppose you have an optimization algorithm for a complex combinatorial problem with several hundred parameters. The algorithm generates thousands of possible solutions that each have to be simulated in order to calculate an objective, which the algorithm tries to maximize.

The simulation of a solution is quite complex and computationally intensive. Fortunately, it is possible to split it into m smaller *blocks* which can be simulated independently, thus producing m partial results. The total objective can then be calculated efficiently based on these partial results. Unfortunately, it is not possible to incrementally calculate the total objective, so all block results have to be accessible at the same time.

Each solution has a large number of parameters that can be varied, but not every parameter influences every block. Thus, block results calculated for one solution may be reused for other solutions. To improve efficiency of the simulation, these block results are therefore *cached*.

Unfortunately, these block results are rather big, so we do not want to copy the whole data block when we find a cache entry. Instead we want to get a safe reference (i.e., a `guard_ptr`) to the cache entry itself. Since the block results require quite a lot of memory, the total number of entries in the result-cache has to be limited. If the limit is exceeded, the number of entries in the result-cache has to be reduced. For the reduction a simple queue with the keys of all entries is used, resulting in a simple first-in-first-out policy. The keys of newly inserted entries are pushed into the queue. To reduce the number of entries in the result-cache, the next key is popped from the queue and the corresponding entry is removed from the result-cache. This is repeated until the number of cached entries no longer exceeds the limit.

Since all block results have to be accessible at the same time, this implies that they are all blocked for reclamation. And since there is no upper bound on the number of blocks, this problem cannot be solved with the classic hazard pointer scheme with a static number of hazard pointers. Instead, a variation with the `dynamic_hazard_pointer_policy` (see Section 4.4.2.2) is used.

In contrast to the other benchmarks, this one uses two scopes, each with its own `region_guard`. In the first one all the cache lookups and calculations are performed, while the second one is used to reduce the size of the result-cache.

This is a highly simplified description of the original problem, but it suffices to create a simple benchmark with similar properties that set this benchmark apart from the usual micro-benchmarks:

- there is no upper bound on the number of nodes that are *intentionally* blocked from reclamation,
- the average lifetime of each `guard_ptr` is relatively long compared to other use cases, due to the fact that the computation of the final objective is computationally intensive and has runtime complexity linear in the number of blocks,
- the memory footprint of each node is significant, putting additional pressure on the reclamation scheme to reclaim nodes efficiently and in a timely manner.

CHAPTER 5

Experimental Results & Performance Analysis

This chapter presents the results from the performance analysis based on the benchmarks described in Section 4.6. These benchmarks were used to investigate the performance impact of various parameters like the number of threads, the workload or the traversal length, as well as the schemes' efficiency in reclaiming retired nodes. All experiments were run on several machines with different architectures to compare their impact on the performance of the various schemes. The CSV files with the raw results, as well as the scripts to run the benchmark and to analyze the results are available on GitHub: `https://www.github.com/mpoeter/emr-benchmarks`.

Unless stated otherwise, each benchmark was performed with 30 trials, each with eight seconds runtime. Most of the benchmarks focus on performance and calculate the *average runtime of a single operation* for each trial. Every thread calculates the average operation runtime by dividing the overall runtime that it was actively running the benchmark, by the total number of operations it performed. The total average runtime per operation is then calculated as the average of these per-thread runtime values.

It is important to note that all 30 trials are performed sequentially within the same process. This is especially important in case of the HashMap benchmark, as the hash-map is retained over the whole runtime. This means that a result calculated in the first trial can be found in the hash-map and reused in a subsequent trial. For this reason, performance will be worse at the beginning, while the hash-map is in the "warm up phase", but will improve over time when it becomes filled and more items can be reused. But it is also possible that in the other benchmarks previous trials have some impact on later ones, e.g., due to an already initialized memory manager, left over orphans that can now be adopted and reclaimed, etc. It was a deliberate design decision to run all trials in the same process as this more closely reflects a real world situation.

Four machines with different (micro-)architectures were used to run the benchmarks. A list of these machines with their respective specifications is shown in Table 5.1.

Table 5.1: Machines

AMD	CPUs	4x AMD Opteron(tm) Processor 6168
	Frequency	max. 1.90GHz
	Cores/CPU	12
	SMT	–
	Hardware Threads	48
	Memory	128GB
	OS	Linux 4.7.0-1-amd64 #1 SMP Debian 4.7.6-1 (2016-10-07) x86_64 GNU/Linux
	Compiler	gcc version 6.3.0 20170205 (Debian 6.3.0-6)
Intel	CPUs	8x Intel(R) Xeon(R) CPU E7- 8850
	Frequency	max. 2.00GHz
	Cores/CPU	10
	SMT	2x
	Hardware Threads	160
	Memory	1TB
	OS	Linux 4.7.0-1-amd64 #1 SMP Debian 4.7.6-1 (2016-10-07) x86_64 GNU/Linux
	Compiler	icpc version 17.0.1 (gcc version 6.0.0 compatibility)
XeonPhi	CPUs	1x Intel(R) Xeon Phi(TM) coprocessor x100 family
	Frequency	max. 1.33GHz
	Cores/CPU	61
	SMT	4x
	Hardware Threads	244
	Memory	16GB
	OS	Linux 2.6.38.8+mpss3.8.1 #1 SMP Thu Jan 12 16:10:30 EST 2017 k1om GNU/Linux
	Compiler	icpc version 17.0.1 (gcc version 5.1.1 compatibility)
SPARC	CPUs	4x SPARC-T5-4
	Frequency	max. 3.60GHz
	Cores/CPU	16
	SMT	8x
	Hardware Threads	512
	Memory	1TB
	OS	SunOS 5.11 11.3 sun4v sparc sun4v
	Compiler	gcc version 6.3.0 (GCC)

On all platforms the standard memory manager from `libc` was used, except on Sparc, where I used `jemalloc` [Eva06]. The reason for this is that the `libc` implementation of `malloc` and `free` in Solaris uses a global lock. Newer versions of Solaris bring their own scalable memory manager `libumem`[1]. Tests with `libumem` started promising, but soon revealed sporadic but severe performance drops when running with about 200 threads or more. I suspect these issues to be caused by large numbers of cross-thread deallocations. These performance issues and their analysis is described in more detail in Section 5.6.

5.1 Lock-Free reference counting

As described in Section 4.4.1, LFRC has been implemented with two extensions: an optional padding to avoid false sharing between the reference counter and other data members, and an optional local free list with a bounded number of entries. This analysis compares the performance of different configurations in various scenarios. To this end, all the previously described benchmarks have been run with four different LFRC configurations on all the machines; the results are shown in the Figures 5.1, 5.2, 5.3 and 5.4. The following variations of LFRC were used:

- unpadded – the original LFRC proposal without padding.
- padded – the original LFRC proposal with padding to avoid false sharing between the reference counter and the node's payload.
- unpadded-20 – like unpadded, but with a local free list with max 20 entries.
- padded-20 – like padded, but with a local free list with max 20 entries.

The results are quite interesting as there is no overall "best" configuration. Instead, the performance of the different configurations varies with both, the data structure as well as the CPU architecture. However, in almost all cases at least one of the other configurations is significantly faster than the original, unpadded LFRC.

LFRC is often criticized for its bad performance, but as the results show there is some potential to improve that. One of the main reasons LFRC's performance suffers lies in the fact that in order to acquire a safe reference to some node n, a thread has to increment a shared counter in n, and again decrement the counter when the reference is dropped. So many threads performing increment/decrement operations on the same node lead to high contention and potential cache misses. This problem lies in the nature of how LFRC works, so there is not really room for improvement here. The only possible attempt is to reduce the number of cache misses by inserting the appropriate padding between the reference counter and the node's payload; especially Intel benefits from this approach.

But as can be seen from the results, in some cases also the internal free-list can cause significant performance issues that can be reduced by using a small thread-local free-list; especially on XeonPhi. For future work it might be interesting to investigate this further, e.g., whether FIFO/LIFO approaches for the thread-local and/or the global free list perform differently, as well as other approaches that would potentially reduce contention on the global free-list like, e.g., letting push and pop operate on separate lists.

[1]`https://blogs.oracle.com/ahl/number-11-of-20:-libumem`

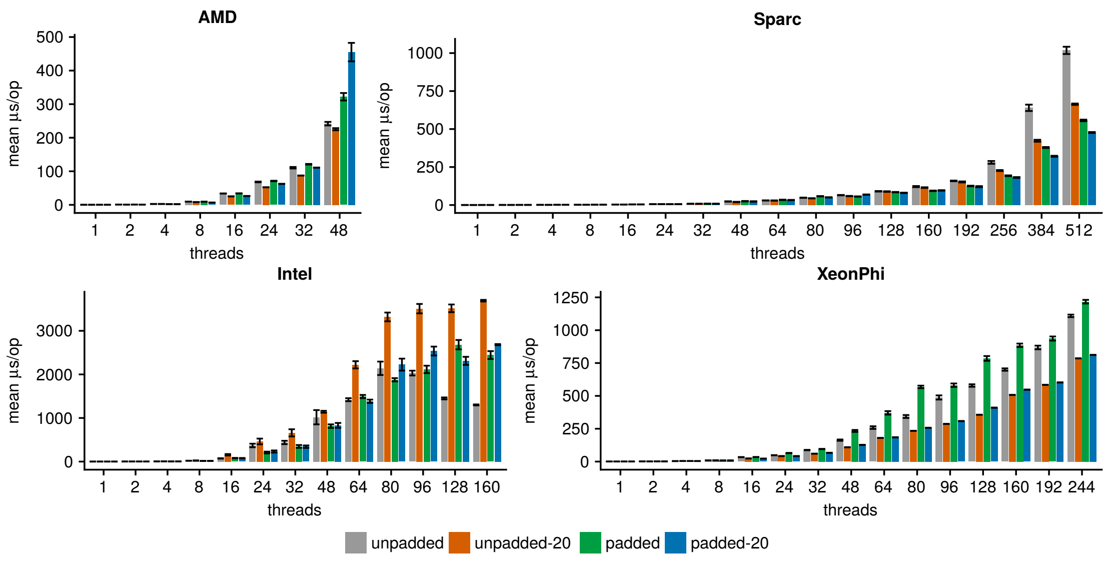

Figure 5.1: Performance comparison of different LFRC configurations with the Queue benchmark.

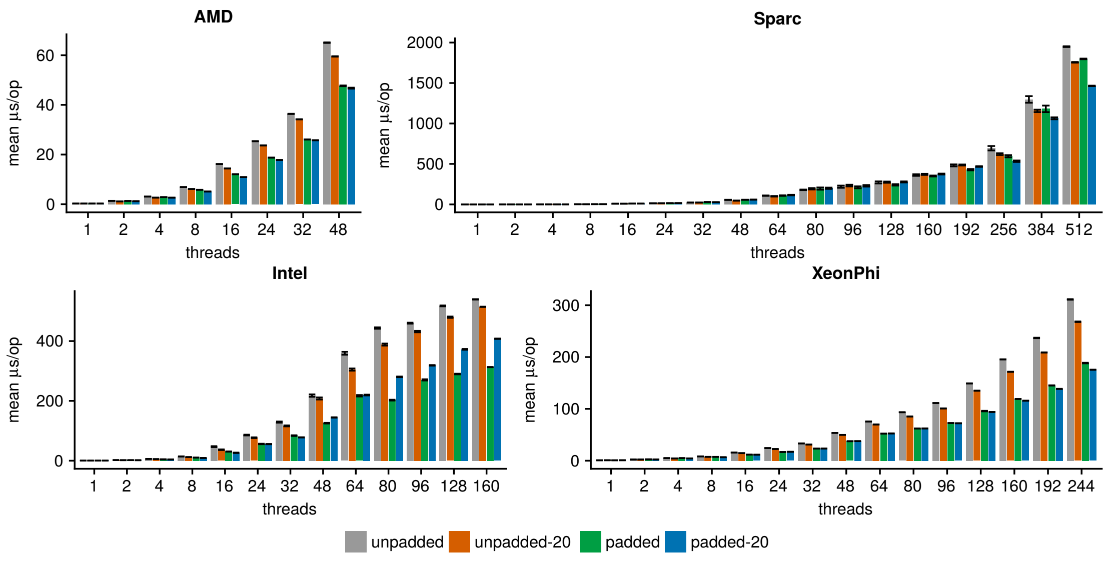

Figure 5.2: Performance comparison of different LFRC configurations with the List benchmark 10 elements and a workload of 20%.

5.2 Stamp-it base performance

As previously described, the adapted version of Stamp-it does not provide an upper bound on the number of steps for entering or leaving a critical region. This section presents

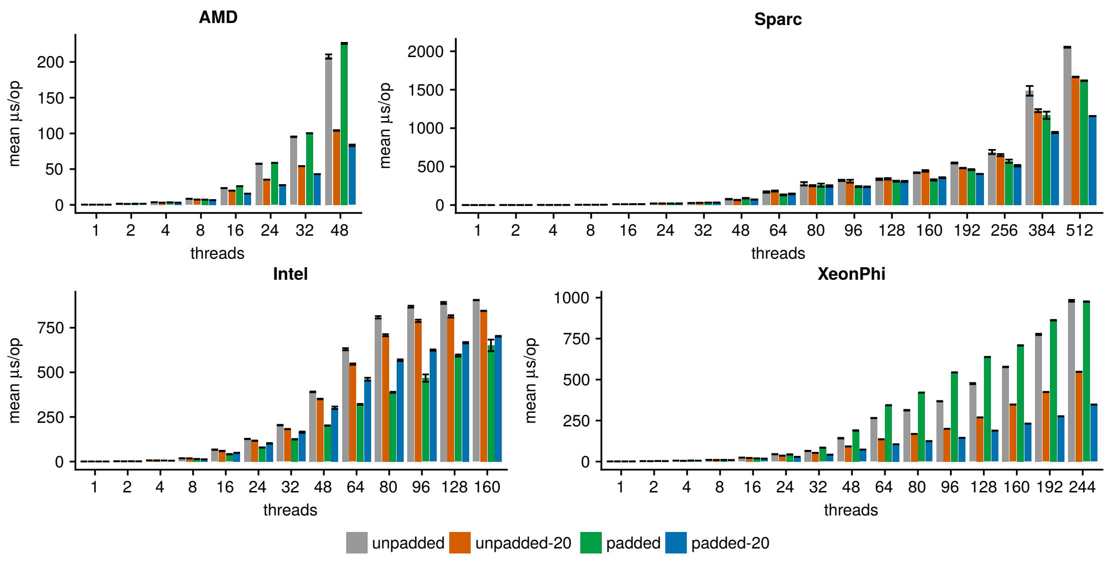

Figure 5.3: Performance comparison of different LFRC configurations with the List benchmark 10 elements and a workload of 80%.

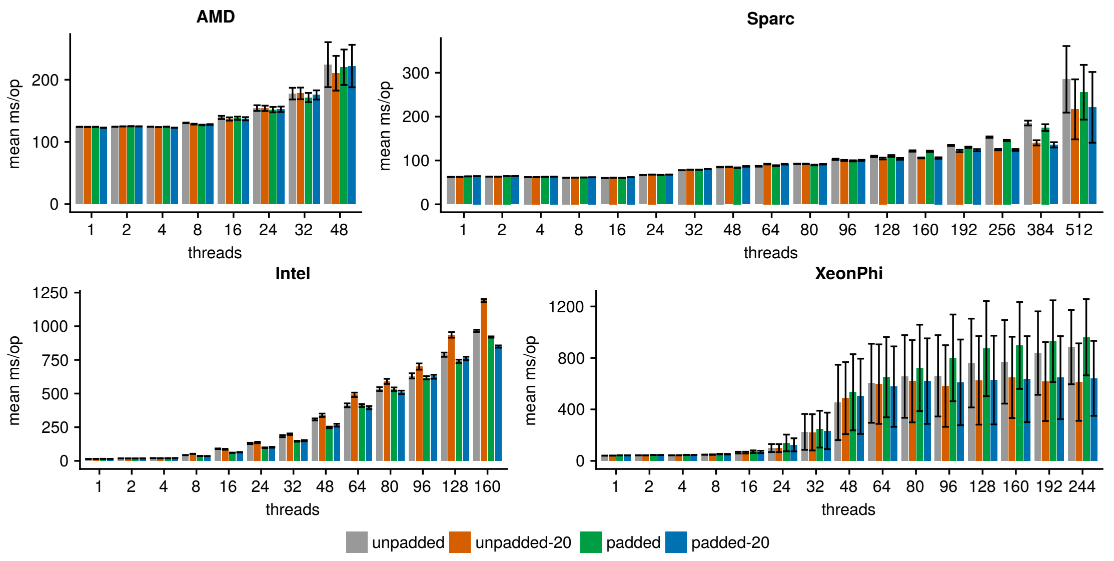

Figure 5.4: Performance comparison of different LFRC configurations with the HashMap benchmark.

the results of an analysis for the effective average number of steps for each operation. The `stamp_it` implementation has been extended with optional thread-local performance counters that keep track of the number of retries in `push` and `remove`, thus allowing to calculate the average number of iterations per operation. Since the data structure is based on a doubly linked list, the `remove` operation builds on two other operations `remove_from_prev` and `remove_from_next` to remove the node from both directions; the number of retries for these two operations is measured separately. The performance counters are disabled by default and can be enabled by defining the `WITH_PERF_COUNTER` macro.

The benchmarks were run as previously described, but instead of average time per operation the average number of iterations in `push`, `remove_from_prev` and `remove_from_next` has been measured. The results for the various benchmarks are shown in Figures 5.5, 5.6, 5.7, 5.8 and 5.9.

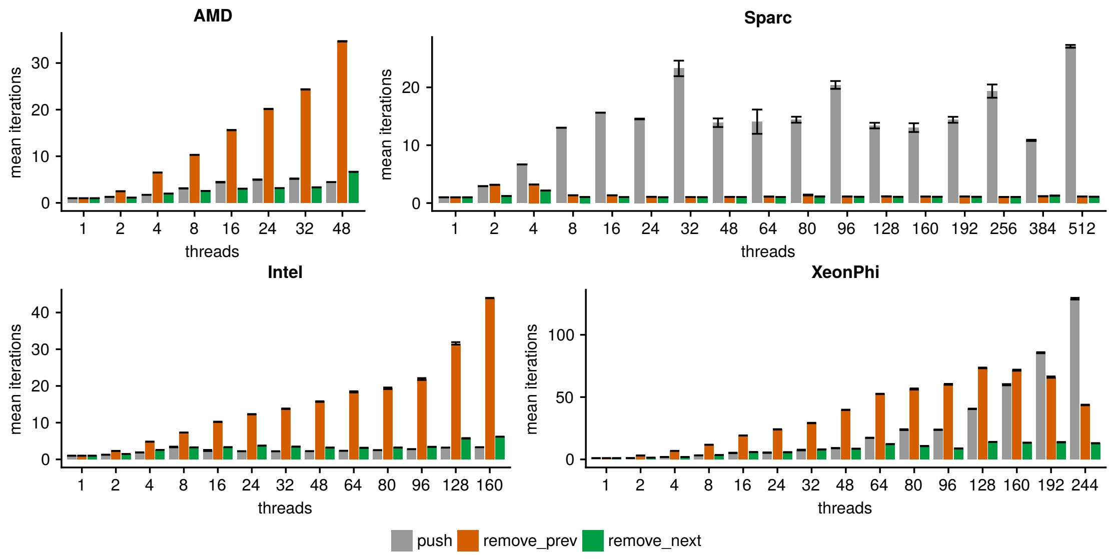

Figure 5.5: Mean number of iterations for the respective operations in the GuardPtr benchmark.

The GuardPtr benchmark is the most interesting, since this is kind of a "stress test", i.e., it simulates the worst case scenario where all threads just insert and immediately remove themselves from the `thread_order_queue`. Essentially, this scenario tests the scalability of the `thread_order_queue` data structure itself. As can be seen in Figure 5.5, the average number of iterations is less than the number of threads in all cases, suggesting that even in this worst case scenario the expected average runtime complexity is $O(p)$.

Interestingly, the behavior differs significantly between the various architectures. For AMD, Intel and the XeonPhi the results are dominated by the number of iterations in `remove_from_prev`. On XeonPhi the number of iterations in `push` increases significantly once the number of threads is greater than 120. The reason for this is probably the SMT based architecture with 61 physical cores and the way instructions are scheduled [Rah13]. For SPARC the situation is the complete opposite: The number of threads has almost no impact on the number of iterations in the remove-methods, instead the number of iterations in `push` is increasing, but varies significantly.

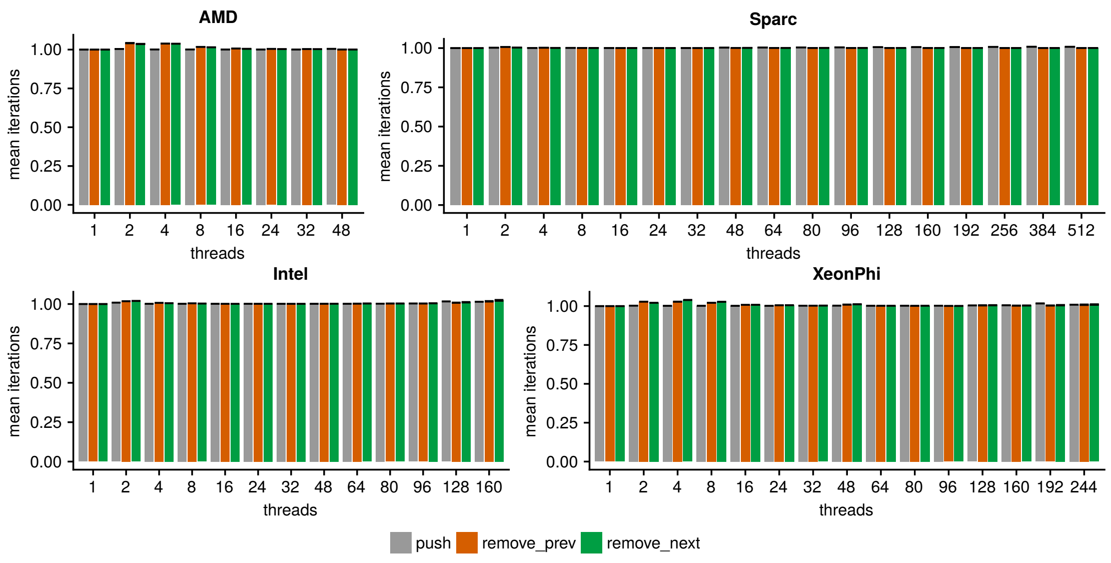

Figure 5.6: Mean number of iterations for the respective operations in the Queue benchmark.

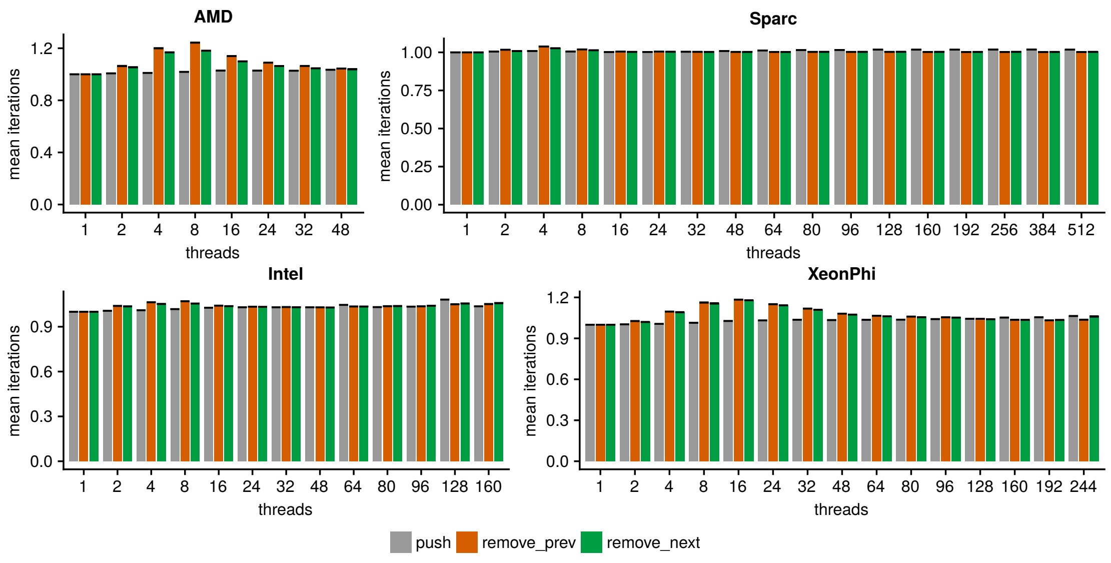

Figure 5.7: Mean number of iterations for the respective operations in the List benchmark with a workload of 20%.

It is likewise interesting to see how the data structure performs under "normal" conditions. As can be seen in Figures 5.6, 5.7, 5.8 and 5.9, which give results for the Queue, List and HashMap benchmarks, the number of threads has almost no measurable impact on the number of iterations for all three methods: The numbers are virtually

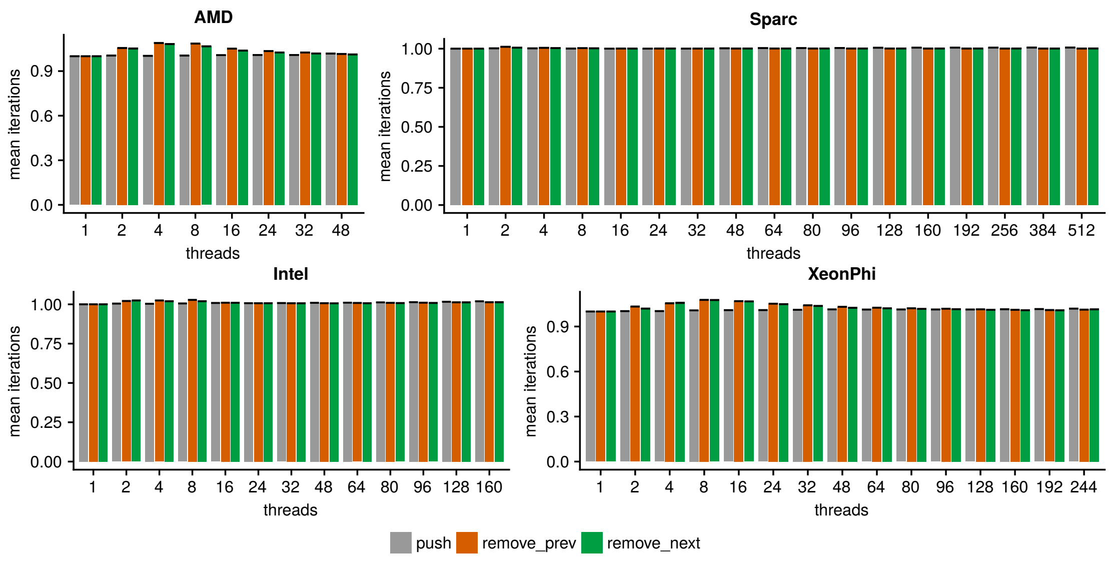

Figure 5.8: Mean number of iterations for the respective operations in the List benchmark with a workload of 80%.

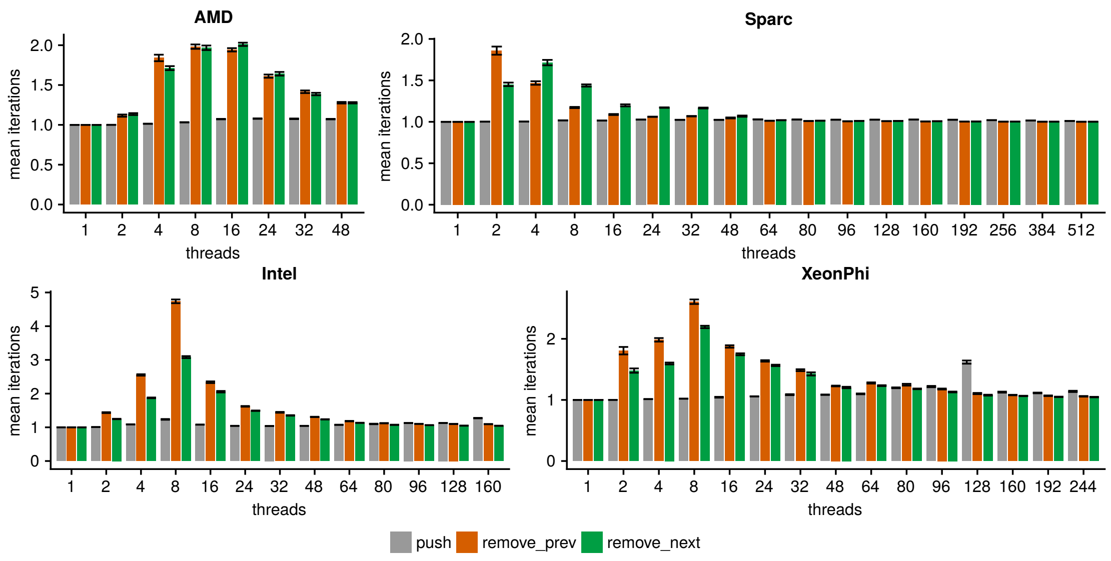

Figure 5.9: Mean number of iterations for the respective operations in the HashMap benchmark.

constant, with a few outliers in the HashMap benchmark (shown in Figure 5.9). There is small increase in the number of iterations in the remove-methods on all platforms around 4-16 threads that decreases again with a growing number of threads.

5.3 Base costs

This analysis measures the base costs of the schemes. The experiments are performed with a single thread to eliminate contention on the used data structure, so resulting performance differences are caused solely by the overhead of creating and releasing `guard_ptr` instances. It also includes the GuardPtr benchmark (described in Section 4.6.1) to measure the pure overhead of creating and releasing a `guard_ptr` instances without any other operations involved.

All benchmarks except HashMap were run on all machines using a single thread, 30 trials and eight seconds runtime. The results are shown in Figure 5.10; "List reads" corresponds to the List benchmark with a workload of 0% (i.e., read-only) and "List writes" corresponds to the List benchmark with a workload of 100% (i.e., all operations are either insert or delete). The number of elements for the List and Queue benchmarks was left at the default value of 10.

The HashMap benchmark was excluded here because in comparison to the other benchmarks it has a very high runtime dominated by the simulated calculations; the overhead for allocating and releasing `guard_ptr`'s is rather irrelevant. The reclamation schemes still have a very big impact on the performance of the HashMap benchmark, but mainly due to the difference in how efficiently retired nodes can actually be reclaimed. This aspect is discussed in more detail in Section 5.7. Stamp-it performs very poorly in

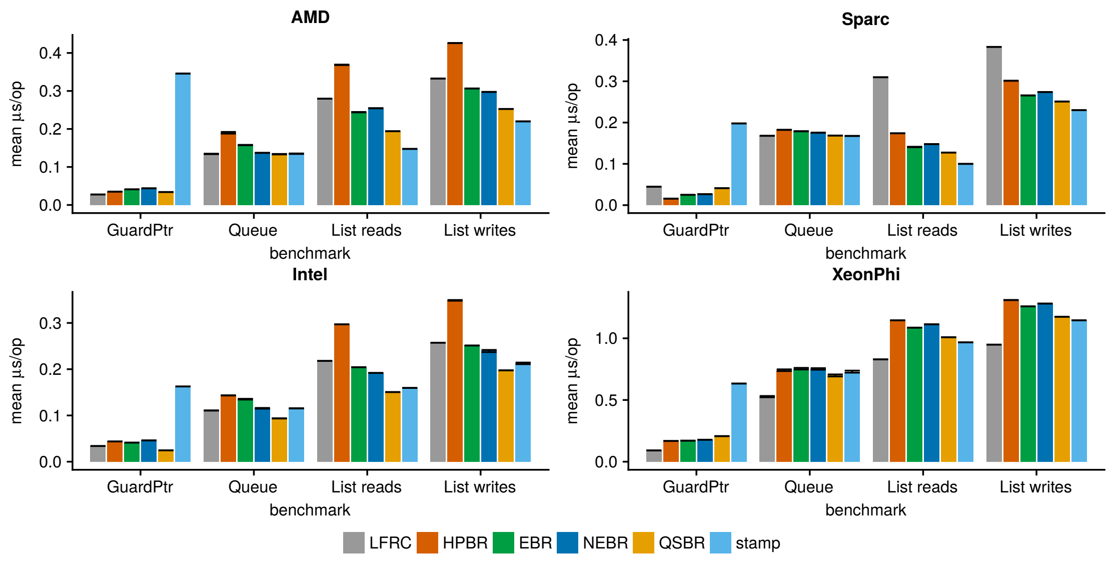

Figure 5.10: Base costs of the various schemes in single thread runs.

the GuardPtr benchmark, due to the more expensive operations to insert and remove the thread from the internal queue. But the results show that there is hardly any trace of this overhead in the other benchmarks; in some cases it is the fastest of all schemes. This is due to the fact that, just like NEBR and QSBR, Stamp-it also uses the `region_guard` concept to amortize the cost of these insert and remove calls over a larger number of operations.

What can also be seen is that there are significant differences between Sparc and the Intel based architectures. On Sparc, LFRC is significantly slower than HBPR. The results of the other benchmarks will corroborate this observation.

5.4 Scalability with workload

This analysis uses the List benchmark to examine the workload's impact on the reclamation schemes by gradually increasing the read-to-update ratio of the performed operations from read-only to update-only.

When pure read-only operations are used, no nodes get reclaimed, so the schemes only differ in the performance overhead of acquiring and releasing the necessary `guard_ptr` instances. With an increasing number of update operations, the performance overhead for acquiring and releasing the `guard_ptr` instances stays the same (we still have to search the list the same way as for read-only operations). But the more update operations are performed (specifically delete operations), the more impact on the overall performance is caused by the reclamation of retired nodes. The benchmark was run in four different configurations:

- one thread; one element (see Figure 5.11)
- one thread; 25 elements (see Figure 5.12)
- 32 threads; one element (see Figure 5.13 and 5.15)
- 32 threads; 25 elements (see Figure 5.14 and 5.16)

Each configuration was run with 30 trials and a runtime of eight seconds. For LFRC the configuration with padding and a local free-list with 20 entries was used; based on the results from Section 5.1 it seemed to be the overall best choice for this scenario. As can be seen by the results of the various configurations, the workload by itself seems to have no significant impact on the performance of the reclamation schemes; within each configuration and architecture, all schemes exhibit roughly the same slope, i.e., the relative performance difference between the schemes stays more or less the same, regardless of the workload. Hart et al. came to the same conclusion in their experiments [HMBW07]. This is not entirely unexpected, since insert and remove operations still require the same lookup to be performed as in a search operation. The only exception is LFRC, which actually shows a performance improvement on Sparc in the configuration with one element and 32 threads (see Figure 5.13), but it starts out with a huge gap to the other schemes. It is not entirely clear why LFRC can improve its performance, but I suspect it is due to the way of how LFRC reuses reclaimed nodes.

In the base cost analysis we saw that LFRC seems to incur a higher overhead on the Sparc architecture. Figure 5.11 shows the results for the configuration with one element and one thread. In this configuration HPBR performs worst in almost all

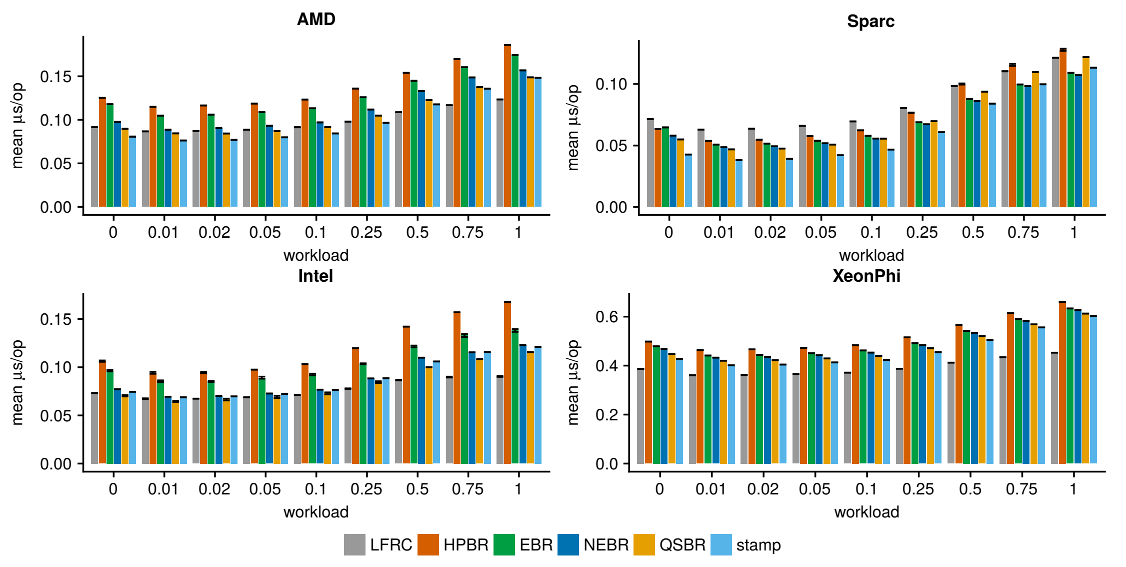

Figure 5.11: Effect of varying workload on a lock-free list with one element, one thread.

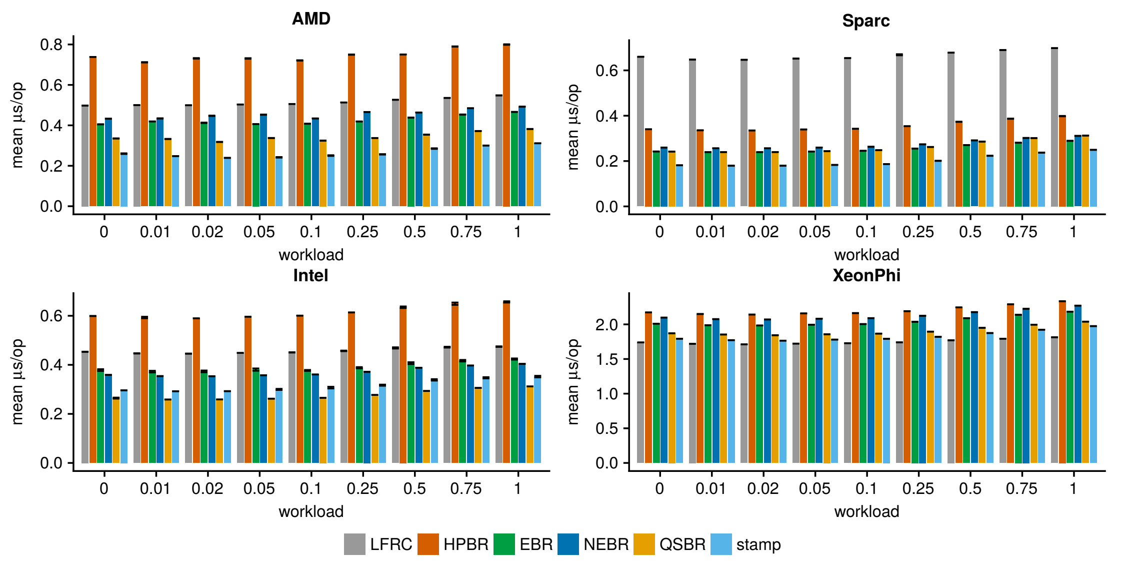

Figure 5.12: Effect of varying workload on a lock-free list with 25 elements, one threads.

cases, while LFRC on the other hand is almost always fastest, or at least on par with the fastest scheme—with the exception of Sparc, where LFRC performs worse than HPBR in virtually all scenarios. This pattern also emerges from the results of all other configurations, which corroborates the observation from the base cost analysis that LFRC performs worse on Sparc, and is thus less well suited for this architecture.

Naturally, LFRC performs significantly worse with a growing number of threads as

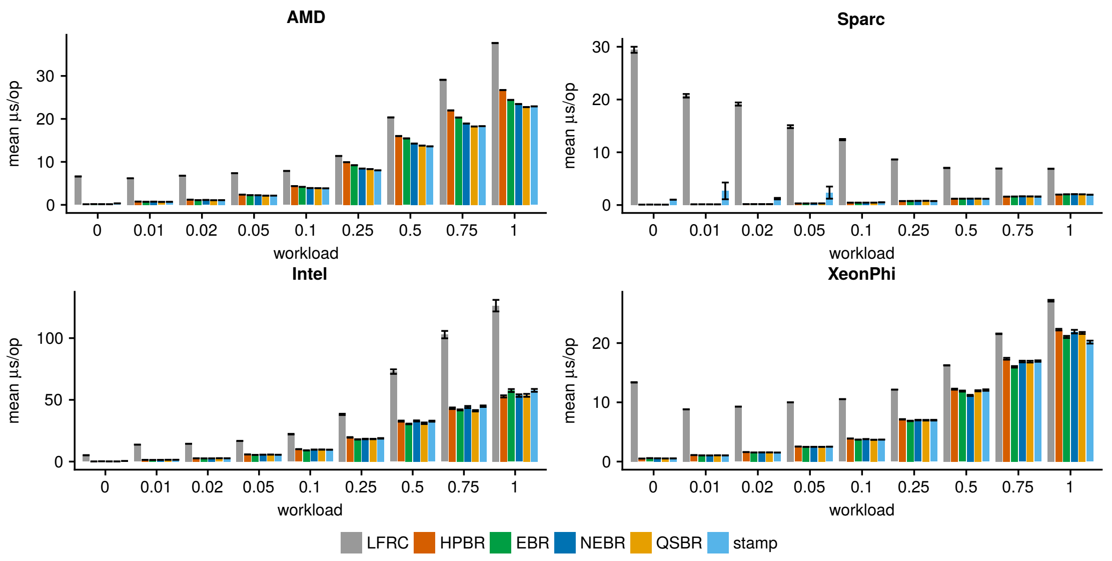

Figure 5.13: Effect of varying workload on a lock-free list with one element, 32 threads.

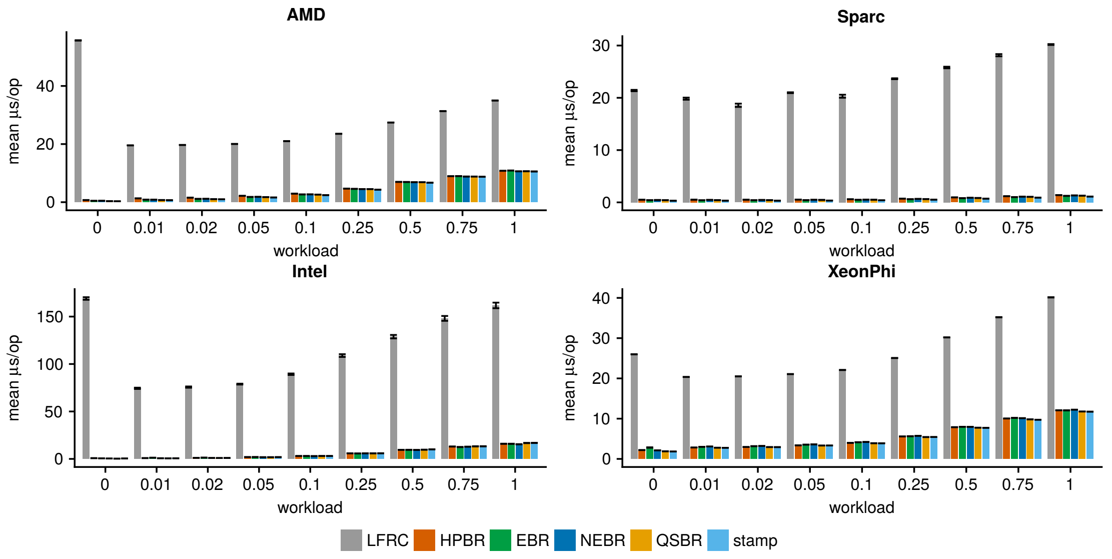

Figure 5.14: Effect of varying workload on a lock-free list with 25 elements, 32 threads.

can be seen in Figures 5.12 and 5.14. What is quite interesting, though, is that in the scenario with a single element (see Figure 5.12), on Sparc the performance of LFRC is actually *increasing* with a higher workload; the other schemes and architectures do not show such an effect. Since these results are dominated by the rather bad performance of LFRC, Figures 5.15 and 5.16 show the same results with LFRC excluded. What can be seen in Figure 5.15 is that in the configuration with 32 threads and a single element, in the

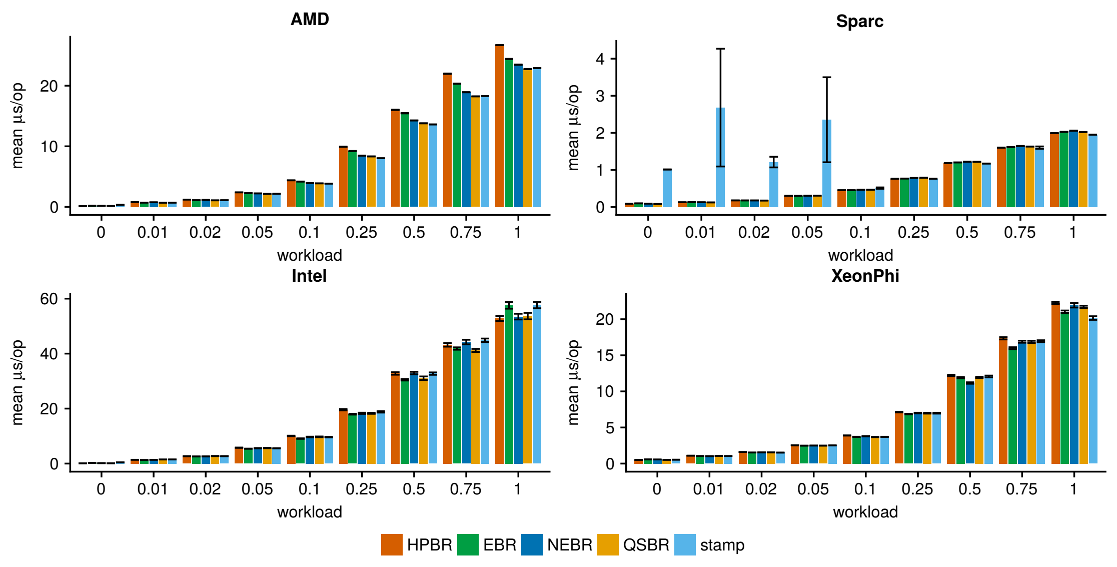

Figure 5.15: Effect of varying workload on a lock-free list with one element, 32 threads without LFRC.

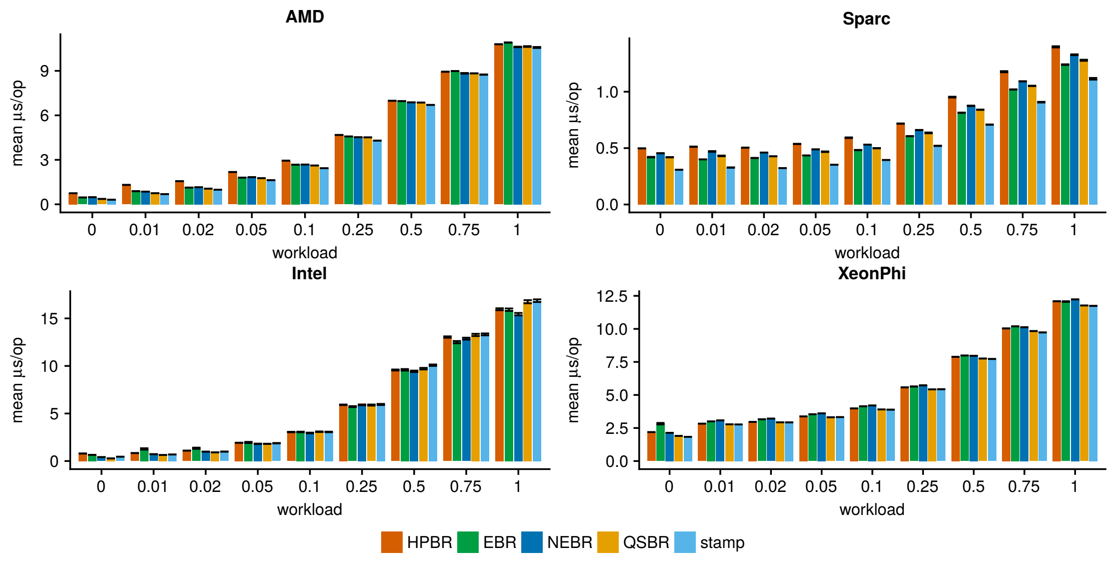

Figure 5.16: Effect of varying workload on a lock-free list with 25 elements, 32 threads without LFRC.

first scenarios, which have a low workload, on Sparc Stamp-it performs significantly worse than the other schemes. But with an increased workload this performance difference completely vanishes. The reason for this is the higher overhead in Stamp-it's enter and leave functions. With only a single element and a low workload, this overhead dominates the total work each thread is handling. By increasing the workload, this overhead becomes much less relevant, while at the same time efficient reclamation of the removed elements becomes more important. So in the scenarios with higher workload Stamp-it shows much better performance. Obviously, an increased number of elements also reduces the relevance of this overhead. The configuration with 32 threads and 25 elements even shows inversed results (see Figure 5.16); in this configuration Stamp-it clearly outperforms all the other schemes on Sparc. Interestingly, the other architectures are largely unaffected and show no such bias.

5.5 Scalability with traversal length

The number of elements in a list can also have an impact on how good the different reclamation schemes perform. This analysis examines this impact by varying the number of elements the list gets initialized with at the start of each trial from zero to 1000. It is also run in four different configurations, each with 30 trials and a runtime of eight seconds:

- one thread; workload of zero (i.e., read-only) (see Figure 5.17)
- one thread; workload of 50% (see Figure 5.18)
- 32 threads; workload of zero (i.e., read-only) (see Figures 5.19 and 5.21)
- 32 threads; workload of 50% (see Figure 5.20 and 5.22)

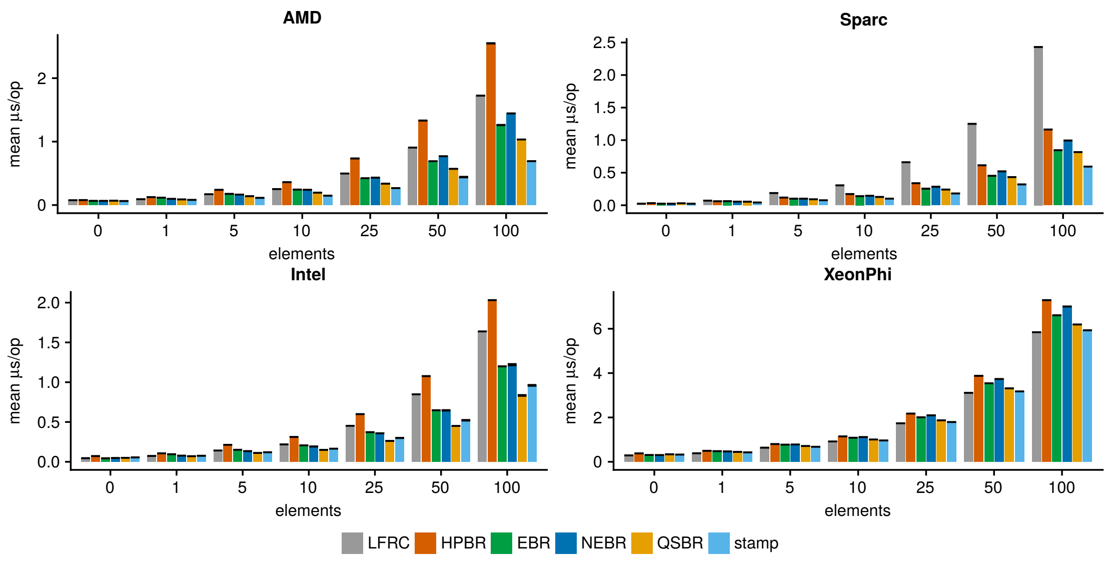

Figure 5.17: Effect of varying traversal length on a read-only lock-free list with one thread.

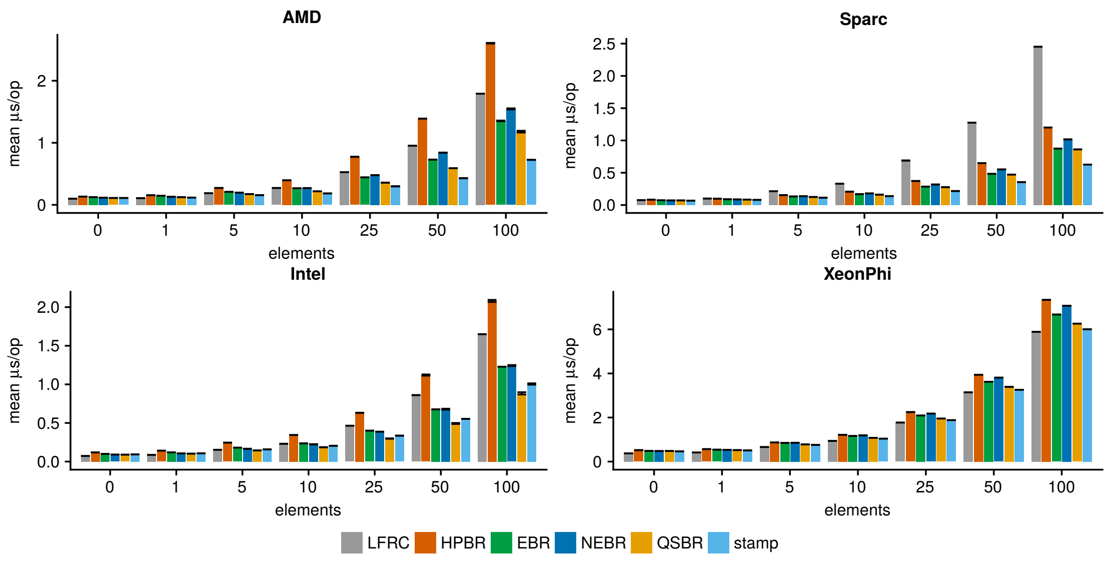

Figure 5.18: Effect of varying traversal length on a lock-free list with one thread and a workload of 50%.

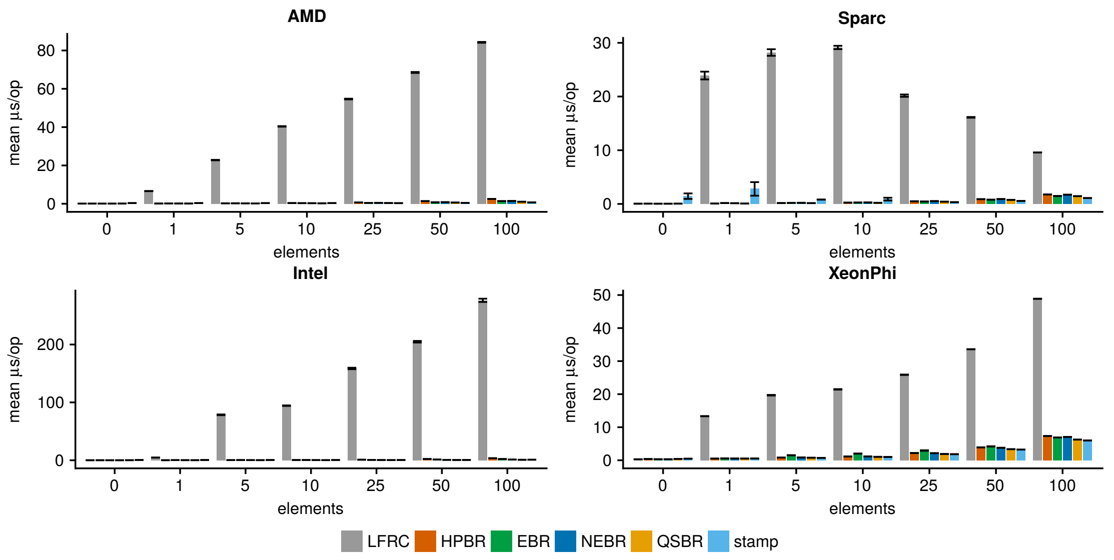

Figure 5.19: Effect of varying traversal length on a read-only lock-free list with 32 threads.

The single threaded results for a read-only list (see Figure 5.17) and a workload of 50% (see Figure 5.18) look almost identical. This corroborates the observations from the previous analysis that the workload has no significant impact on the performance of the reclamation schemes.

What can be seen from the results of the single thread configurations is that with

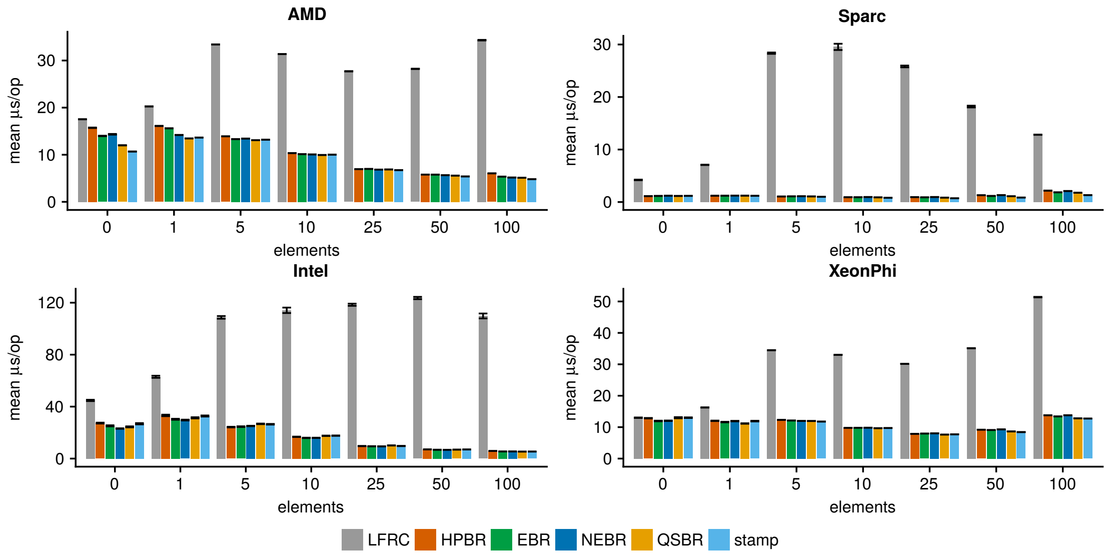

Figure 5.20: Effect of varying traversal length on a lock-free list with 32 threads and a workload of 50%.

an increasing traversal length the performance of LFRC and HPBR degrades. This is expected since, due to their design, these schemes have a per-element overhead. It is interesting, though, that this effect varies in intensity, depending on the respective architecture.

When looking at the results of the 32 thread configurations (Figures 5.19 and 5.20), LFRC's runtime degrades dramatically—especially in the read-only case. Therefore Figures 5.21 and 5.22 show the same results, but with LFRC excluded. From these results one can see that in the read-only configuration the additional overhead of HPBR is highly significant (at least on AMD and Intel), but becomes negligible when looking at the results with 50% workload. Similar to the results discussed in the previous Section, Stamp-it performs rather poorly on the read-only list with a small number of elements (see Figure 5.21). With a larger number of elements, or an increased workload (see Figure 5.22), its performance significantly improves so that in some configuration on Sparc, it again clearly outperforms all other schemes. In the read-only configuration (Figure 5.21) AMD and Intel show similar effects, but much less significant; XeonPhi seems to be completely unaffected.

For the other schemes the results suggests that the traversal length has only a small impact on their respective performance. This is not unexpected since EBR, NEBR, QSBR and Stamp-it do not require per-element operations like HPBR or LFRC.

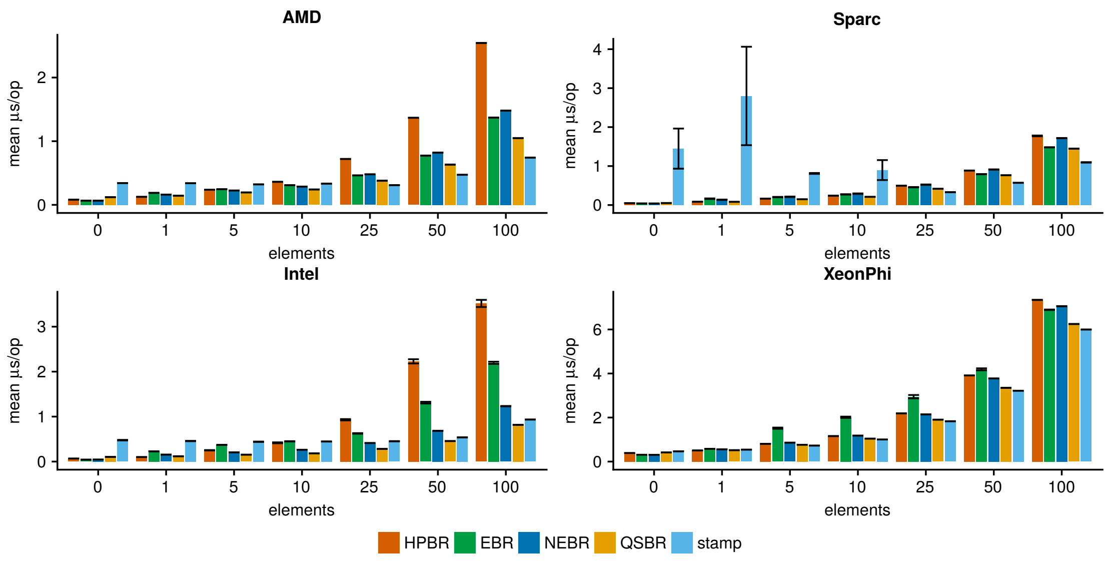

Figure 5.21: Effect of varying traversal length on a read-only lock-free list with 32 threads without LFRC.

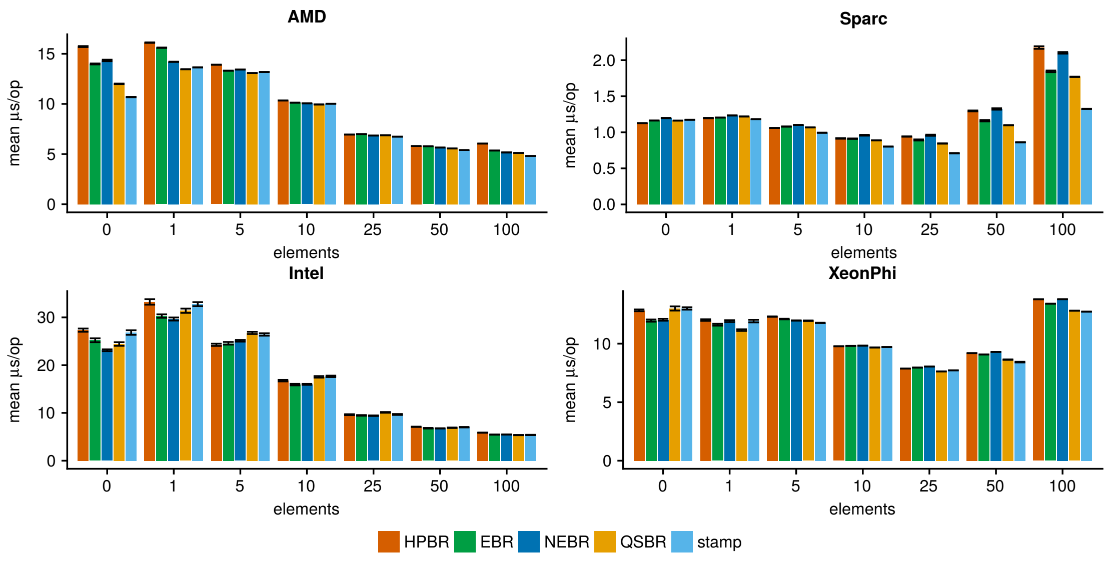

Figure 5.22: Effect of varying traversal length on a lock-free list with 32 threads and a workload of 50% without LFRC.

5.6 Scalability with threads

This analysis studies the effect of increasing the number of threads that share a single instance of some data structure, under the assumption that the number of threads is less or equal than the number of hardware threads (i.e., CPU cores including SMT). I did not experiment with oversubscribed cores.

As already discussed, the `libc` implementation of `malloc` and `free` on Sparc uses a global lock, therefore rendering it useless for this analysis. The first alternative would have been `libumem`, which is part of all new Solaris versions, but it showed some severe performance drops by a factor of ∼20 (in the Queue benchmark) to ∼250 (in the List benchmark) in some trials. This happened only in these two benchmarks (i.e., the HashMap benchmark was not affected), and only with Stamp-it when running with more than 200 threads. The behavior was not deterministic, i.e., it only happened sporadically and it did not always affect the same trials, but it could be reproduced on a regular basis.

Unfortunately, this phenomenon completely vanished when I tried to profile it with the tools from the Solaris Development Studio. It was simply impossible to reproduce this behavior when the `collect` tool was attached to the process. What the heap statistics from this tool did show, though, was that Stamp-it reclaimed 5-30 times more nodes than the other reclamation schemes (this aspect is analyzed in more detail in Section 5.7). Other experiments showed that this issue could longer be reproduced when the actual reclamation was disabled, i.e., the whole algorithm remained untouched, only the `delete_self` call was disabled. These observations let me confidently conclude that the problem was not the reclamation scheme itself, but the memory manager. In particular I suspect the large number of cross-thread deallocations to be the cause of this behavior.

In search of an alternative memory allocator I first tried `Hoard` [BMBW00], which seems to have similar performance as `libumem`, but also showed performance drops of the same magnitude; they occurred much less frequently though. `TCMalloc` [GM11] dropped out since it does not support Sparc. Eventually I found `jemalloc` [Eva06], which does support Sparc. It seems to be slightly slower, but showed robust performance without any performance drops. Based on these observations I selected `jemalloc` since it seems to be best suited for scenarios with such a large number of threads. However, I only performed a very small set of tests that do not allow to draw a final conclusion. A more in-depth analysis of various memory allocators for architectures with such a large number of cores would be very interesting, but is outside the scope of this work. Figure 5.23 shows the performance of the reclamation schemes in the Queue benchmark. Surprisingly, LFRC performs by far best on Sparc and on XeonPhi, but is by far worst on Intel. On AMD, HPBR has a huge performance drop when running with the maximum number of threads. A similar effect can be seen by the other schemes as well, but much less significant. Apart from these exceptions, all schemes seem to scale largely equally good in this scenario.

For the results of the List benchmark (Figures 5.24 and 5.25) LFRC has been excluded since we already established that it performs poorly in this scenario, especially with a larger number of threads. On AMD, Intel and XeonPhi all schemes are more or less on par, but on Sparc EBR and NEBR show a significant degradation when the number of threads grows beyond 128. What is surprising, though, is that in all those cases NEBR constantly performs worse than EBR. This is quite unexpected, since NEBR was designed

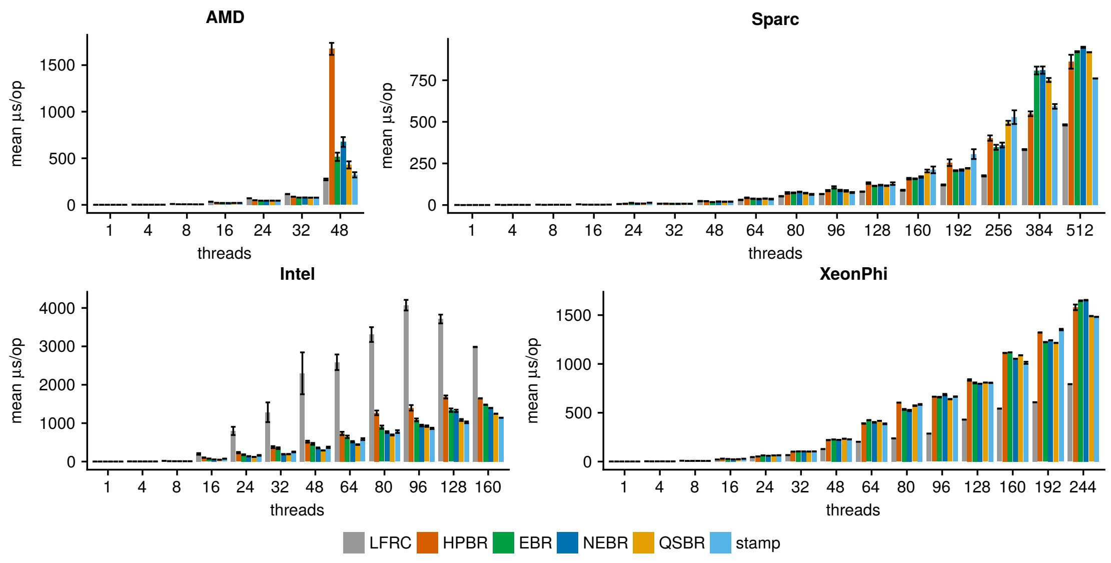

Figure 5.23: Performance of the Queue benchmark with varying number of threads.

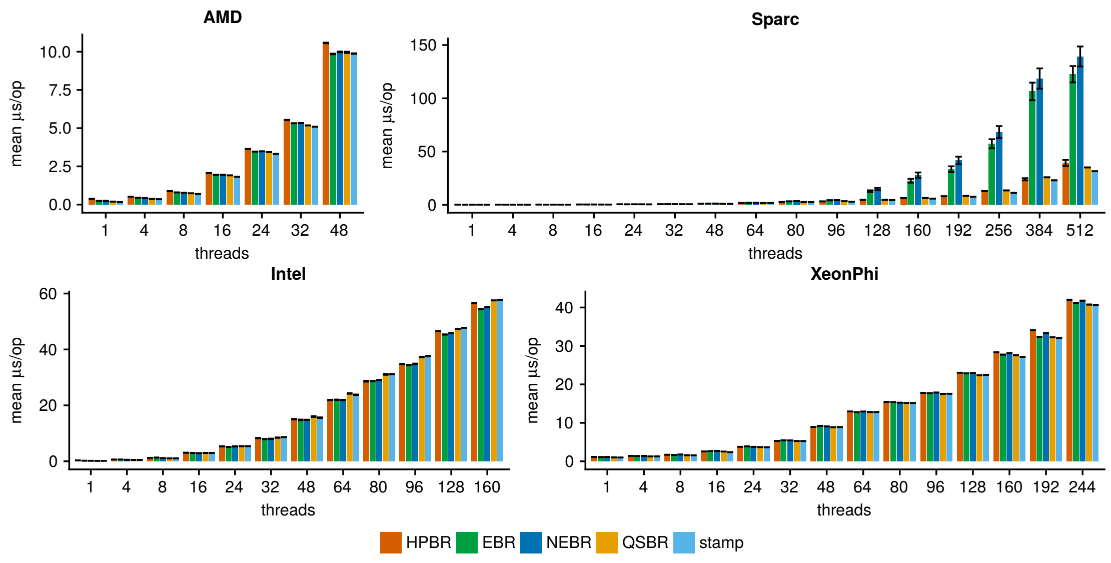

Figure 5.24: Performance of the List benchmark with 10 elements, a workload of 20% and a varying number of threads (without LFRC).

to have less overhead than EBR. I did not investigate the reasons for this in more detail, but one assumption is that this might be caused by a larger number of unsuccessful attempts to update the global epoch, which is caused by NEBR's design of larger critical regions.

Finally, the results for the HashMap benchmark are shown in Figure 5.26. Obviously,

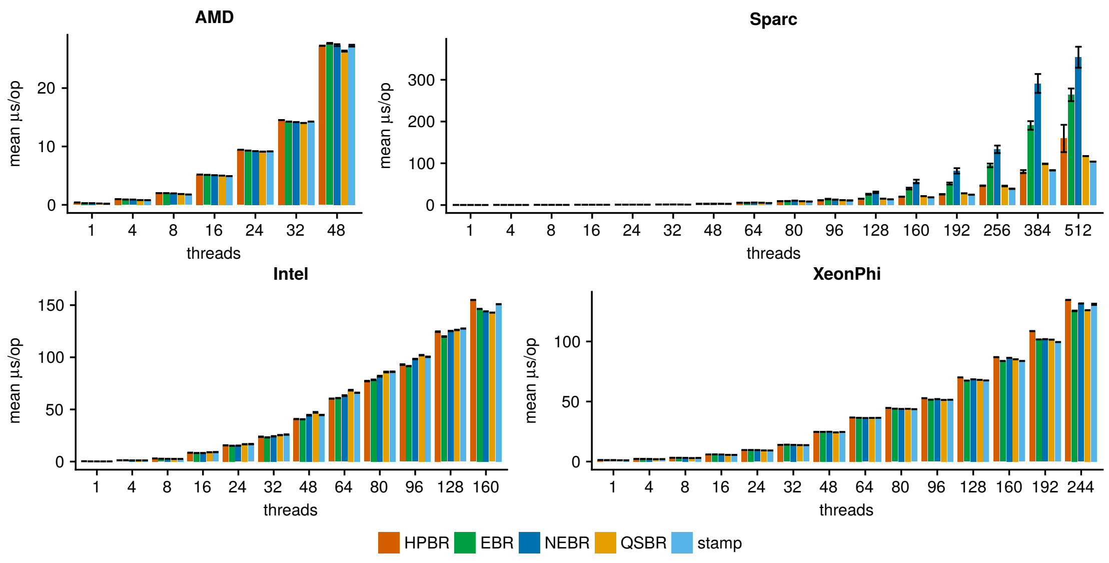

Figure 5.25: Performance of the List benchmark with 10 elements, a workload of 80% and a varying number of threads (without LFRC).

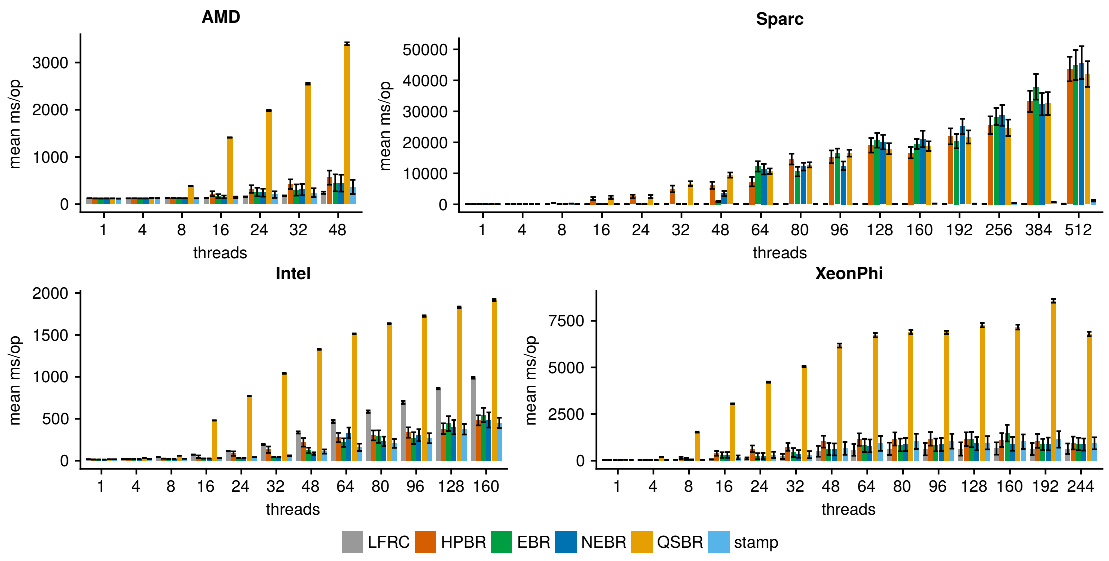

Figure 5.26: Performance of the HashMap benchmark with varying number of threads.

QSBR scales very poorly in this scenario and therefore its performance is very bad on all the architectures. The results with QSBR excluded are shown in Figure 5.27. On AMD, EBR, NEBR and Stamp-it scale almost perfectly, while LFRC's and HPBR's performance starts to degrade once the number of threads grows beyond 16. On Intel, LFRC scales very poorly while all other schemes scale more or less equally good, but not

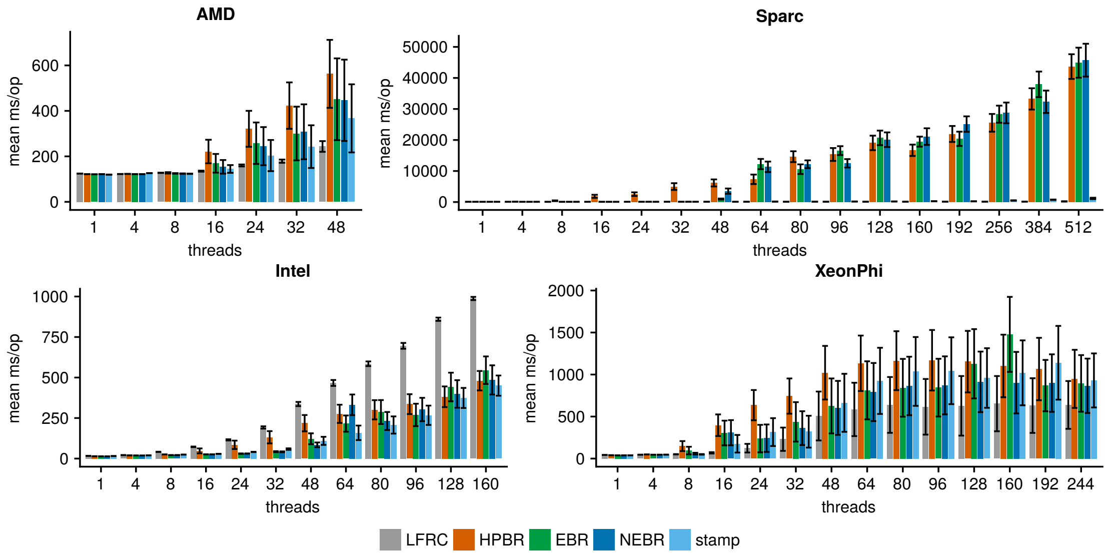

Figure 5.27: Performance of the HashMap benchmark with varying number of threads (without QSBR).

as good as on AMD. On XeonPhi on the other hand, LFRC scales best, while HPBR's performance starts degrading with more than 16 threads, but it again improves with more than 128 threads. The other schemes continuously lose performance when the number of threads grows from 16 to ∼80, but then stays more or less the same. But the biggest surprise are obviously the results on Sparc. Here, the runtime of HBPR, EBR and NEBR degrades dramatically and shows significant variance, while LFRC and Stamp-it scale almost perfectly. With 512 threads the performance difference between LFRC/Stamp-it and the other schemes is a factor of ∼4000. The reason for this will become clear when we look at the results of the reclamation efficiency analysis in the next section.

5.7 Reclamation efficiency

This analysis focuses on how efficiently (fast) the various reclamation schemes actually reclaim retired nodes. An increased reclamation efficiency can drastically reduce memory pressure, which in turn can have a significant impact on the overall performance. Nonetheless, this aspect is usually disregarded in most analyses of concurrent reclamation schemes.

To measure reclamation efficiency I have implemented the previously described allocation tracking—there are thread-local performance counters that track the number of allocated and reclaimed nodes. By calculating the difference we get the number of unreclaimed nodes, which is our measurement for efficiency; a smaller number of unreclaimed nodes means that the reclamation scheme works more efficiently.

The plots in this analysis show the development of the number of unreclaimed nodes over time. Each configuration is run with five trials, each with a runtime of eight seconds. During each trial a total of 50 samples are collected, including the additional samples at the beginning and the end of each trial. Therefore a sample is recorded approximately every 163ms. Since the benchmarks are randomized, each configuration with the five trials is run 20 times to account for any fluctuation in the measured samples. The plots show the smoothed conditional means of the measured samples of those 20 runs over the number of samples recorded during each run.

For reclamation efficiency, reference counting is the "golden standard". In contrast to other schemes there is no delay: A node is reclaimed immediately when the last thread drops its reference to that node. So in all the plots, LFRC can bee seen as the baseline against which all other schemes have to be measured. One has to keep in mind, though, that LFRC is not a general reclamation scheme, since the reclaimed nodes are not returned to the memory manager, but stored in the internal free-list.

The results are shown in Figures 5.28, 5.29, 5.30, and 5.31. What can be seen in all scenarios is that HPBR's efficiency is inversely proportional to the number of threads. This is due to the fact that the threshold for the number of unreclaimed nodes is quadratic in the number of threads as explained in Section 2.4. This is the case even for the Queue benchmark (Figure 5.28) and List benchmarks (Figures 5.29 and 5.30), even though the number of hazard pointers per thread is constant. In the HashMap benchmark (Figure 5.31) a dynamic number of hazard pointers is used, which makes the situation even worse.

The implementation allows to customize the calculation of this threshold, so for future work it might be interesting to analyse how a different threshold would affect reclamation efficiency and what impact this would have on the performance.

In the Queue and List benchmarks on AMD we can see a small bump in the number of unreclaimed nodes during the first trial for all reclamation schemes except LFRC and HPBR. After the first trial they all recover and perform comparably for the rest of the benchmark. It is not clear what causes this behavior as I did not investigate further.

Apart from this behavior and the previously described issue of HPBR with a large number of threads, the results for the Queue and List benchmarks are not too surprising; all schemes perform more or less comparably. In the Queue benchmark QSBR performs somewhat worse on Intel and XeonPhi, but this is not unexpected as QSBR is less well suited for update heavy scenarios.

In the HashMap benchmark (Figure 5.31) we can see that QSBR basically fails completely to reliably reclaim nodes on all the architectures. The number of nodes is constantly increasing and does not even go down at the end of the trials when all threads are stopped. This is also the reason why QSBR showed such bad performance in the previous analysis in Section 5.6.

For HPBR we can also see a consistent increase in the number of unreclaimed nodes over time, even though this number sharply drops right at the beginning of a new trial, but also increases again very rapidly. The only exception is Sparc, where no such drop occurs and the number of nodes is increasing all the time. The other schemes all perform

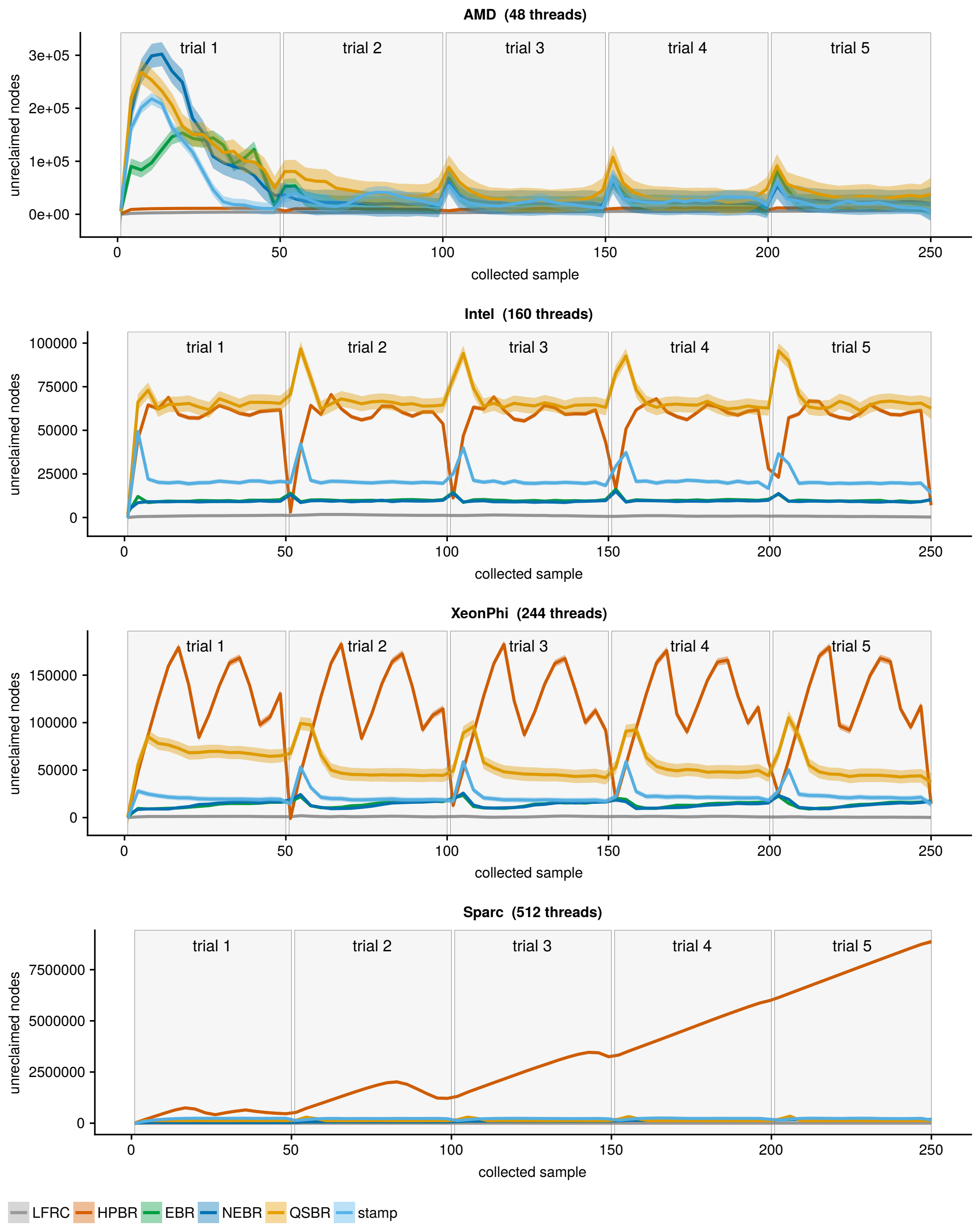

Figure 5.28: Number of unreclaimed of nodes over time in the Queue benchmark.

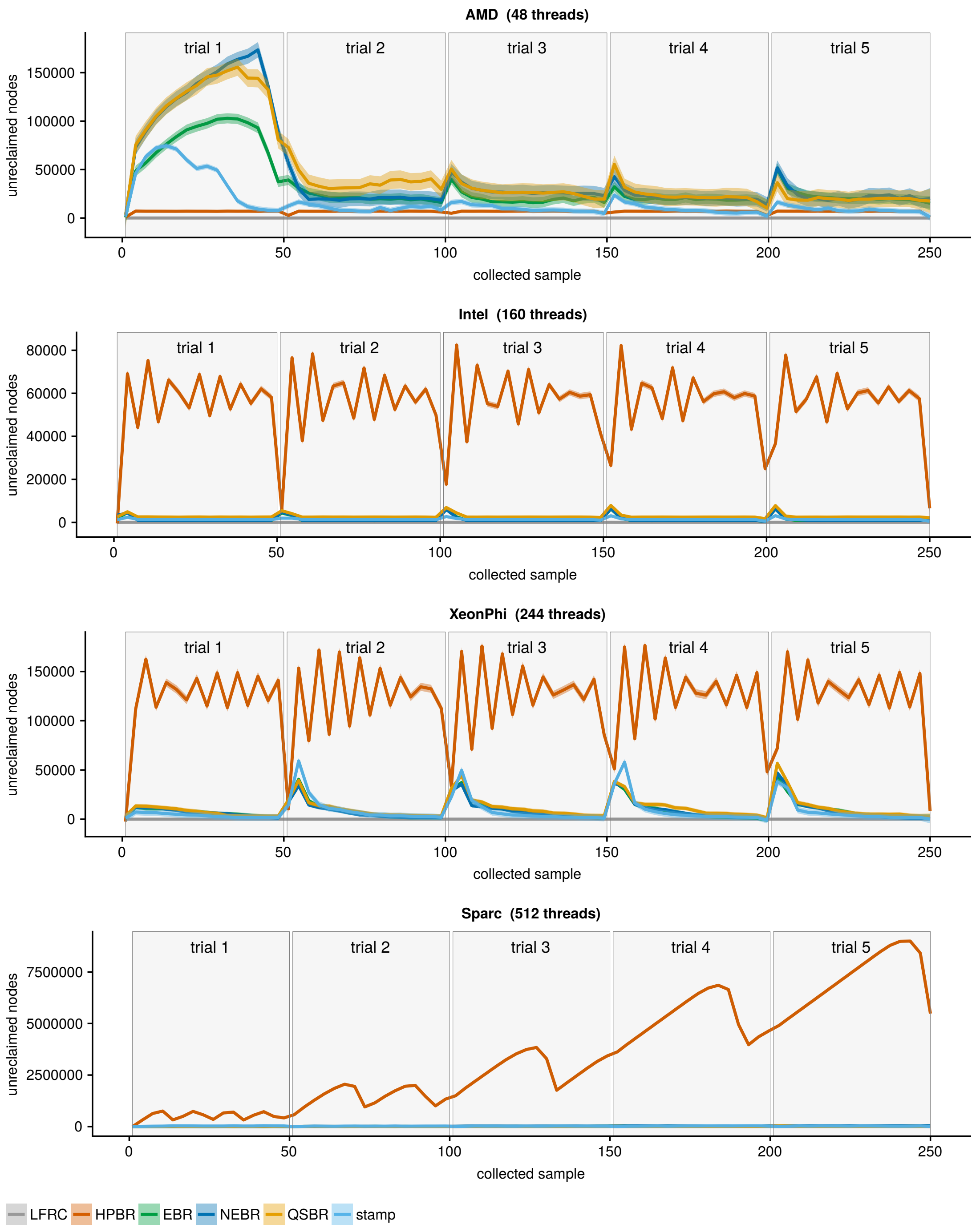

Figure 5.29: Number of unreclaimed nodes over time in the List benchmark with 10 elements and a workload of 20%.

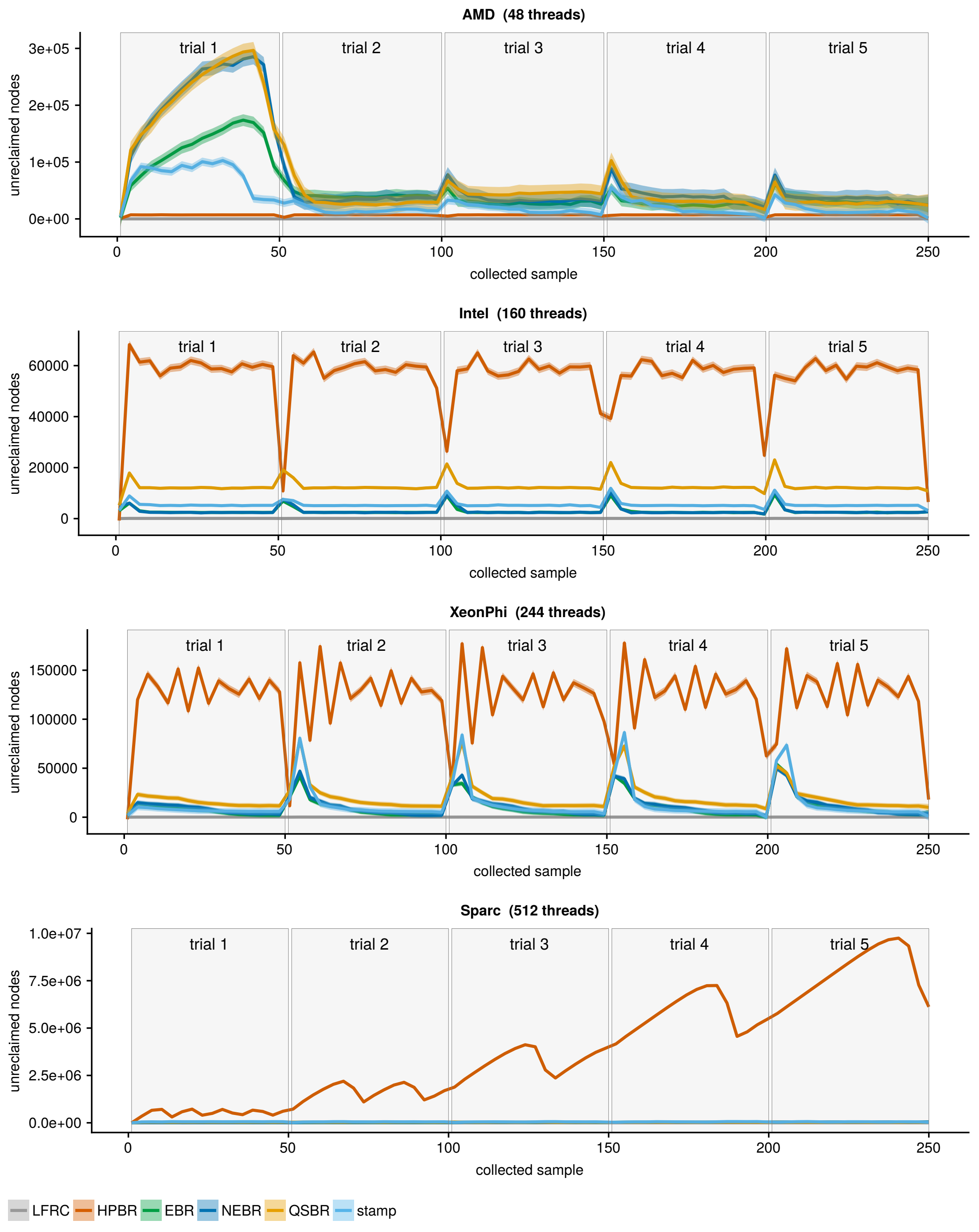

Figure 5.30: Number of unreclaimed nodes over time in the List benchmark with 10 elements and a workload of 80%.

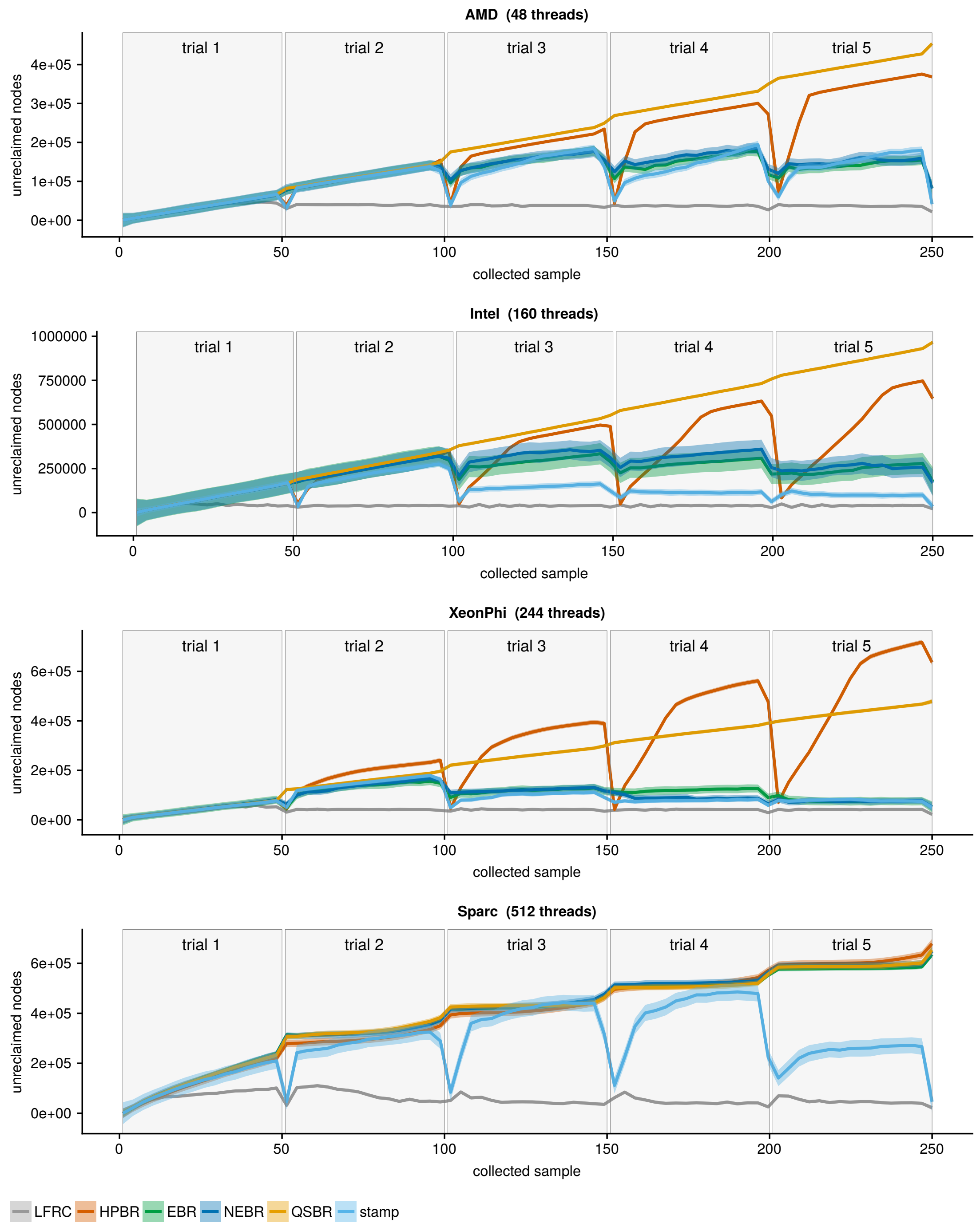

Figure 5.31: Number of unreclaimed nodes over time in the HashMap benchmark.

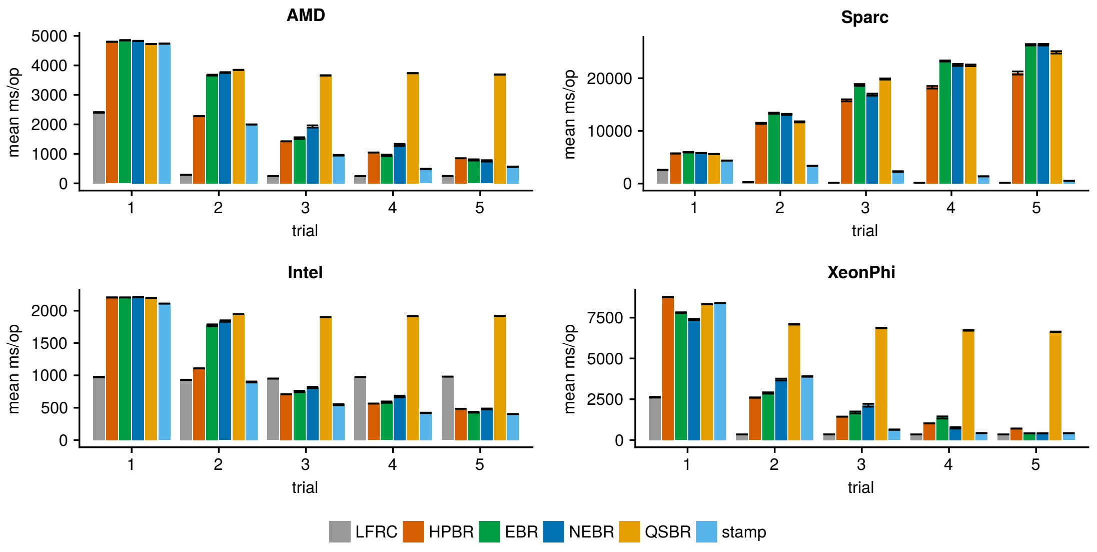

Figure 5.32: Development of runtime over trials (i.e., time) in the HashMap benchmark.

relatively good on all architectures; the exception again being Sparc. On Sparc HPBR, EBR, NEBR and QSBR are all performing equally bad. The number of unreclaimed nodes is constantly increasing and does not even go down at the end of the trials when all threads are stopped. This effect is probably caused by the fact that in these schemes every thread is responsible for reclaiming its own retired nodes. In Stamp-it we know if there is some other thread lagging behind, so we can add nodes to a global list and let that thread take responsibility for reclaiming them. This allows Stamp-it to more reliably reclaim nodes, especially at the end of each trial.

The failure to efficiently reclaim nodes increases memory pressure, which has a direct impact on the runtime. Figure 5.32 shows the development of the runtime over the five trials. On Sparc we can see that the runtime of HPBR, EBR, NEBR and QSBR is increasing with each trial, while LFRC and Stamp-it is decreasing. On the other architectures, runtime is decreasing for all schemes except QSBR. This would be the expected behavior since more results can be reused once the hash-map has been filled.

CHAPTER 6

Conclusion and Future Work

This thesis addresses the problem of efficient memory reclamation for concurrent data structures on shared-memory system. Since this is a critical component in almost all concurrent data structures, there already exists a substantial amount of work on this topic. I have presented an extensive overview of the current state of the art on this topic and introduced a new scheme called *Stamp-it*. I have also presented an adapted version of the generalized C++ interface proposed by Robison [Rob13] and discussed the implementations of six reclamation schemes, including Stamp-it, following this generalized interface. As basis for the correctness arguments of the various implementations, a brief introduction on the topic of memory models with a focus on the C++11 memory model has been provided. Finally, I have presented a large scale experimental study, comparing the performance of the implemented reclamation schemes on four different architectures in various scenarios.

6.1 Conclusion

Memory reclamation is a critical component in almost all shared-memory, concurrent data structures and algorithms. Naturally, there exists a large number of proposed solutions, many of which have been presented in Chapter 2. With *Stamp-it* this thesis has introduced a new general purpose memory reclamation scheme with attractive features. To the best of my knowledge, this is the first non-reference counting based scheme that does not have to scan all other threads to determine reclaimability of a node.

Publications on lock-/wait-free algorithms or data structures usually assume a sequentially consistent memory model in their explanations and correctness proofs; and publications on memory reclamation schemes pose no exception in this regard. Nevertheless, the memory model is crucial when it comes to actually implementing such a reclamation scheme; at least when you want to achieve best performance by avoiding sequential consistency in cases where it is not required. This thesis provides a brief introduction to the topic of memory models, with a focus on the C++11 memory model. The implementations are built using this model and, based on the defined semantics, I have tried to tune performance by relaxing all atomic operations as far as possible without sacrificing correctness. According correctness arguments for all implementations have also been presented.

In many cases, the authors of publications on this topic do not provide access to the implementations that were used in the experiments. And when they do, the implementations are usually primarily for proof of concept and therefore highly simplified, e.g., they assume a fixed number of threads, can only handle allocations of specific types, or do not pay enough attention to the memory model (in the case where they care about the memory model at all). For the implementations presented in this thesis I have decided to diverge from this practice, and instead provide implementations that are generic (i.e., can handle an arbitrary number of threads and arbitrary types), portable and fully compliant with the C++ standard. In short, they are designed in a way that should make them suitable to be used in real applications out-of-the-box. The full source code is made available on GitHub (`https://github.com/mpoeter/emr`).

A large scale experimental study has been presented, comparing the performance of the six implemented reclamation schemes on four different architectures (AMD, Intel, XeonPhi and Sparc) in various scenarios using three lock-free data structures (queue, list and hash-map). The used systems have hardware supported thread counts ranging from 48 to 512, allowing me to run the experiments at a larger scale than usually seen in publications on this topic. The empirical results show that Stamp-it matches or outperforms the other analyzed reclamation schemes in almost all cases. In addition to the usual throughput-oriented experiments I have also presented a number of experiments that analyze the reclamation efficiency, i.e., how quickly retired nodes are actually getting reclaimed. This aspect is often disregarded, but as the results of these experiments show it can have a significant impact on the overall performance.

For the purpose of full transparency and easy reproducibility I have put all results, together with the scripts that were used to run and analyze the experiments, on GitHub as well (`https://github.com/mpoeter/emr-benchmarks`).

6.2 Future Work

For future work I plan to add more implementations of other reclamation schemes and include them in the benchmark results. The first candidate for this is DEBRA [Bro15], since it is one of the newer proposals that also provides all the properties laid out in Section 4. I also want to extend the set of benchmarks to cover more data structures like skip-lists or binary search trees.

With regards to Stamp-it, it might be interesting to look for other data structures that could replace the doubly linked list, i.e., data structures that have less overhead while providing all the required properties. In this context I might also try to relax some of these properties (e.g., use a partial order instead of a strict order for thread entries) in order to reduce contention on the data structure.

Bibliography

[AB10] Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, August 2010.

[AB11] Sarita V. Adve and Hans-J. Boehm. Memory models. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1107–1110. Springer, 2011.

[ABC$^+$08] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT)*, pages 245–254, New York, NY, USA, 2008. ACM.

[ABFR11] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage collection for multicore numa machines. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, pages 51–57, New York, NY, USA, 2011. ACM.

[ABH$^+$04] Andrei Alexandrescu, Hans-J. Boehm, Kevlin Henney, Doug Lea, and Bill Pugh. Memory model for multithreaded C++. Document WG21/N1680=J16/04-0120; available at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf`, September 2004.

[AEH$^+$14] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*, pages 25:1–25:14, New York, NY, USA, 2014. ACM.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Transactions on Computer*, 29(12):66–76, 1996.

[AGW14] Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 385–395, New York, NY, USA, 2014. ACM.

[ALMS15] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 123–132, New York, NY, USA, 2015. ACM.

[BA08] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78. ACM, June 2008.

[BB16] J.F. Bastien and Hans-J. Boehm. Fail or succeed: there is no atomic lattice. C++ standards committee paper, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0418r1.html`, August 2016.

[BGHZ16] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 349–359, New York, NY, USA, 2016. ACM.

[BKP13] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 33–42, New York, NY, USA, 2013. ACM.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGOPS Oper. Syst. Rev.*, 34(5):117–128, November 2000.

[Boe04] Hans-Juergen Boehm. An almost non-blocking stack. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 40–49, New York, NY, USA, 2004. ACM.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005.

[Boe16] Hans-J. Boehm. Temporarily deprecate memory_order_consume. C++ standards committee paper, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0371r0.html`, May 2016.

[BP11] Anastasia Braginsky and Erez Petrank. Locality-conscious lock-free linked lists. In *Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN)*, pages 107–118, Berlin, Heidelberg, 2011. Springer-Verlag.

[Bro15] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 261–270, New York, NY, USA, 2015. ACM.

[BS93] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4 (SEDMS)*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[C++12] Stefanus Du Toit C++ Standards Committee. Working draft, standard for programming language C++. C++ standards committee paper, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`, January 2012.

[Cop95] James O. Coplien. Curiously recurring template patterns. *C++ Report*, 7(2):24–27, February 1995.

[CP15a] Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 260–279, New York, NY, USA, 2015. ACM.

[CP15b] Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 254–263, New York, NY, USA, 2015. ACM.

[DG16] Tudor David and Rachid Guerraoui. Concurrent search data structures can be blocking and practically wait-free. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 337–348, New York, NY, USA, 2016. ACM.

[DHLM11] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 99–108, New York, NY, USA, 2011. ACM.

[Dij65] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, Technological University of Eindhoven, 1965.

[DMMJ01] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 190–199, 2001.

[DMS+12] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, New York, NY, USA, 2010. ACM.

[Eva06] Jason Evans. A scalable concurrent malloc(3) implementation for freebsd, 2006.

[FDSZ01] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on Java^{TM} Virtual Machine Research and Technology Symposium - Volume 1 (JVM)*, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[Fra04] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge Computer Laboratory, 2004.

[GM11] Sanjay Ghemawat and Paul Menage. Tcmalloc : Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, January 2011.

[GPST09] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, Aug 2009.

[Gup89] Rajiv Gupta. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 54–63, New York, NY, USA, 1989. ACM.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 265–279, London, UK, UK, 2002. Springer-Verlag.

[HLM02] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 339–353, London, UK, UK, 2002. Springer-Verlag.

[HM92] Maurice Herlihy and J. Elliot B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

[HM01] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying gc without stopping the world. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI)*, pages 48–57, New York, NY, USA, 2001. ACM.

[HMBW07] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, December 2007.

[IBM83] IBM. Ibm system/370 extended architecture, principles of operation. publication no. SA22-7085, 1983.

[Int16] Intel Corporation. *Intel® 64 and IA-32 Architectures Software DeveloperâĂŹs Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, September 2016.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computer*, C-28(9):690–691, Sept 1979.

[MA04] Scott Meyers and Andrei Alexandrescu. C++ and the perils of double-checked locking. *Doctor DobbâĂŹs Journal*, Jul 2004.

[McK05] Paul E. McKenney. Memory ordering in modern microprocessors. *Linux Journal*, 30:52–57, 2005.

[Mic02] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 73–82, New York, NY, USA, 2002. ACM.

[Mic04a] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[Mic04b] Maged M. Michael. Method for efficient implementation of dynamic lock-free data structures with safe memory reclamation, June 3 2004. US Patent App. 10/308,449.

[MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical report, University of Rochester, Rochester, NY, USA, 1995.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, New York, NY, USA, 1996. ACM.

[MS98] Paul E. Mckenney and J. D. Slingwine. Read-copy update: using execution history to solve concurrency problems, parallel and distributed computing and systems. In *Proceedings of the 1998 International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, NV, USA, 1998.

[MSS12] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. Technical report, `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`, October 2012.

[MW16] Maged M. Michael and Michael Wong. Hazard pointers - safe resource reclamation for optimistic concurrency. C++ standards committee paper, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0233r0.pdf`, February 2016.

[MWM16] Paul McKenney, Michael Wong, and Maged Michael. A concurrency toolkit for structured deferral or optimistic speculation. C++ standards committee paper, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0232r0.pdf`, February 2016.

[Nad12] David Robert Nadeau. C/c++ tip: How to get the process resident set size (physical memory use). `http://nadeausoftware.com/articles/2012/07/c_c_tip_how_get_process_resident_set_size_physical_memory_use`, July 2012.

[Nel16] Clark Nelson. Dynamic memory allocation for over-aligned data. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0035r4.html`, June 2016.

[Pet12] Erez Petrank. Can parallel data structures rely on automatic memory managers? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, pages 1–1, New York, NY, USA, 2012. ACM.

[PFPS07] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: A real-time garbage collector for multiprocessors. In *Proceedings of the 6th International Symposium on Memory Management (ISMM)*, pages 159–172, New York, NY, USA, 2007. ACM.

[PPS08] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 33–44, New York, NY, USA, 2008. ACM.

[PT17] Manuel Pöter and Jesper Larsson Träff. A new and five older concurrent memory reclamation schemes in comparison (Stamp-it). Technical report, Vienna University of Technology, Research Group Parallel Computing, 2017. arXiv:1712.06134.

[PT18] Manuel Pöter and Jesper Larsson Träff. POSTER: Stamp-it, amortized constant-time memory reclamation in comparison to five other schemes. In *Proceedings of the 2018 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018. To appear.

[PZM$^+$10] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 146–159, New York, NY, USA, 2010. ACM.

[R.12] James R. Transactional synchronization in haswell. Blog, `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, February 2012.

[Rah13] Reza Rahman. Intel® Xeon Phi™core micro-architecture. `https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture`, May 2013.

[Rob13] Arch D. Robison. Policy-based design for safe destruction in concurrent containers. C++ standards committee paper, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3712.pdf`, August 2013.

[Sco13] Michael L. Scott. *Shared-Memory Synnchronization.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publisher, 2013.

[SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A primer on Memory Consistency and Cache Coherence.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publisher, 2011.

[SPA92] SPARC International Inc. *The SPARC Architecture Manual Version 8 Revision SAV080SI9308*, 1992.

[SSO$^+$10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[ST05] Håkan Sundell and Philippas Tsigas. *Lock-Free and Practical Doubly Linked List-Based Deques Using Single-Word Compare-and-Swap*, pages 240–255. Springer Berlin Heidelberg, 2005.

[Sun05] Håkan Sundell. Wait-free reference counting and memory management. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 24b–24b, April 2005.

[TP14] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 357–368, New York, NY, USA, 2014. ACM.

[Val95] John D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, May 1995.