Institut für Informatik
Lehr- und Forschungseinheit für Kommunikationssysteme und Systemprogrammierung

Ludwig-Maximilians-Universität München

Evaluation of Task Scheduling Algorithms and Wait-Free Data Structures for Embedded Multi-Core Systems

Master Thesis

Tobias Fuchs
Matrikel-Nummer 10393480

Betreuer	Dr. Karl Fürlinger (LMU)
	Dr. Tobias Schüle (Siemens AG)
Aufgabensteller	Prof. Dr. Kranzlmüller
Bearbeitungszeitraum	01. April 2014 - 14. Oktober 2014

Tobias Fuchs:
Evaluation of Task Scheduling Algorithms and Wait-Free Data Structures for Embedded Multi-Core Systems
October 14, 2014

SUPERVISORS:
Dr. Karl Fürlinger (LMU)
Dr. Tobias Schüle (Siemens AG)

LOCATION:
Munich, Germany

TIME FRAME:
April - October 2014

Ohana means family.
Family means nobody gets left behind, or forgotten.

— Lilo & Stitch

Dedicated to Corinna and my grandmother Gabriele.

"For a while" is a phrase whose length can't be measured.
At least by the person who's waiting.

— Haruki Murakami, *South of the Border, West of the Sun*

Abstract

Scaling computational power on embedded systems is subject to the general restrictions and laws observed in the last decade: Single core architectures having reached their physical limits, the importance of multi-core systems also grows in the embedded domain. Wait-free progress conditions appear to be a perfect match for the demand for bounded execution time in time-critical applications. However, strong progress guarantees are even more challenging to achieve when real-time constraints apply to the implementation of data structures. This thesis examines the state of the art of non-blocking and wait-free algorithmic paradigms with respect to their applicability in embedded- and real-time applications. Existing wait-free data structures are modified to enable their use in time-critical tasks and compared to prevalent lock-free alternatives in performance evaluation. In a complementary chapter, a comparison of work-stealing strategies gives an outlook from data- to task parallelism.

Declaration of originality

I hereby declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such.

Tobias Fuchs	City, Date

Acknowledgements

I sincerely thank my advisor Dr. Karl Fürlinger at the Ludwig-Maximilians-Universität München for his outstanding support and caring attitude throughout the course of this thesis.

I would like to express the deepest appreciation to the outstandingly skilled team at the multicore expert center at Siemens AG Corporate Technology. Results in this work would not have been possible without the persistent guidance and rich knowledge of my advisor Dr. Tobias Schüle and Dr. Christian Kern.

I owe my gratitude to Dr. Thomas Zander who introduced me to his project and gave me the opportunity to work on fascinating problems with such exceptionally bright minds.

Contents

1 Introduction

Among all computer systems that increasingly become a part of everyday life, embedded systems most probably have the highest impact on our personal safety. By definition, they interact with physical processes and therefore must meet real-time performance criteria. As technical components in risk-classified domains such as transportation and healthcare, embedded system applications are comprehensively audited for fault-tolerance, and characteristic constraints on resource consumption and execution times apply to the design of algorithms and data structures.

To harness the computational power of multi-core processors, applications must be parallelized, yet few existing libraries comply with constraints in real-time software applications. In addition, classical synchronization mechanisms such as locks introduce potential hazards such as deadlocks and starvation.

According to formal definition, wait-free algorithms provide the strongest possible fault-tolerance and guarantee an upper bound for their execution time.

1.1 Problem statement

Related work on wait-free data structures does not consider suitabilty for embedded systems and employ mechanisms such as garbage collection that conflict with wait-freedom or are not easily implemented using restricted resources. Application-ready implementations of wait-free concurrent data structures are therefore of high relevance for embedded system software engineering, especially for hard real-time applications.

In this master thesis, wait-free data structures and task scheduling algorithms shall be investigated regarding their suitability for embedded systems. This includes the following tasks:

- Literature research and analysis of existing approaches
- Development of appropriate algorithms and data structures
- Analytical derivation of upper bounds on the timing behavior and memory consumption
- Preparation of benchmarks and experimental evaluation on different hardware platforms
- Documentation and presentation of the results

1.2 Objectives and contributions

We present an overview of state of the art approaches and methodologies in wait-free data structures and algorithms, and practicable solutions for real-time and embedded system applications. In addition, intra-task scheduling is evaluated in variants of work-stealing strategies.

Characteristics of algorithms in real-time and embedded system applications are explained, and how they differ from general purpose software. Existing solutions are discussed with respect to their adequacy for real-time software.

Most relevant for practical use, this work contains application-ready implementations of a wait-free concurrent pool, queue and stack that are meeting requirements of time-critical applications.

For data structures, means of automated verification derived from their semantic definition are explained.

Evaluation of data structures in this work follows the principles of existing publications as close as possible to keep results comparable to related work. Differing from most existing performance comparisons, additional custom test scenarios emphasize metrics that are relevant to embedded- and real-time applications in particular, such as maximum latency and jitter. To illustrate the overall performance implications of wait-freedom, a popular lock-free implementation of each data type is added to the candidate set of its wait-free alternatives.

In addition, the benchmark framework implemented over the course of this thesis is explained in detail. Measurements in publications on lock-free and wait-free data structures are typically impossible to reproduce as little insight in the setup and implementation of the test tools is delivered. However, reproducible and precise performance measurements are only rendered possible by intricate detail, and ideally take platform-dependent behaviour into account.

1.3 Structure of this thesis

Concurrency in algorithms manifests in two main aspects: Data parallelism and task parallelism. Correspondingly, wait-free data structures and task scheduling are discussed in separate chapters of this thesis. Both start with an introductory section covering theoretical and technical foundations behind the problems and approaches presented. Existing publications that have been starting positions for own approaches are discussed in a separate section on related work in both chapters.

The sections following the review of related work describe own findings and results that have not been presented in previous publications. Verification and performance evaluations put own implementations in comparison to the status quo of lock-free algorithms.

2 Wait-free data structures

In this chapter, an introduction to progress definitions is provided along with other formal foundations for wait-free algorithms and data structures. This includes their theoretical relevance for requirements in embedded and real-time computing, and how wait-freedom is helping applications to guarantee task completion to critical deadlines.

In the following, concrete implementations of common data structures are presented with respect to their dependency on micro-instructions. Finally, these implementations will be verified and compared using benchmarks representing real-world demands.

2.1 Introduction

What guarantees could a data structure provide on the duration of its operations? For sequential algorithms, we consider duration of an operation as an unconditional consequence of its run-time complexity. Structural analysis gives reliable guarantees on worst-case complexity in the O domain, and estimating upper bounds for data structures is a daily routine in software engineering.

In concurrent data structures, the concept of duration unfolds into additional dimensions. Operations might depend on scheduler characteristics, and deadlock situations can delay the most efficient algorithm indefinitely. To account for these aspects, guarantees on execution time and completion of operations on concurrent data structures are defined in terms of *progress*.

In addition to their ranking by run-time complexity, concurrenct algorithms form a hierarchical structure of *progress guarantees*, with wait-freedom as the strongest guarantee in the classification.

This chapter summarizes the theoretical background of progress conditions in concurrent data structures, and the technical foundations of their realization.

2.1.1 Definition of wait-freedom

In the following, terms are used according to their common definitions in literature [HS11]: An *object* is an instance of a *data structure*, a container with a specific *state* that implements an abstract *data type*. The data type defines a set of *methods* available to manipulate it. An *operation* to an object is a single *execution* of a method according to the method's *signature*. Finally, data type *semantics* describe the methods' concrete effect on the object, i.e. how they transform the object from one well-defined state to another.

Using these definitions, an implementation of a concurrent object is informally described as wait-free if it guarantees that every process can complete any operation on the object [Her88]

1. within a finite number of execution steps, and
2. independent from scheduling and the progress of other processes.

This description implicitly includes unconditional fault-tolerance, as any process can complete its operation even if all other processes are aborted. Consequently, concurrency control in wait-free algorithms cannot be based on locks. *Non-blocking* implementations control concurrent access without exclusive access to any resource, which is necessary, but not sufficient for wait-freedom.

Application of the term wait-freedom varies in literature. However, a formal logic designed to verify progress properties of algorithms exists [Don06], and definitions of non-blocking progress conditions have been established in linear temporal logic (LTL) [PMS09].

An upper bound on number of execution steps for any operation is often presumed as a requirement for wait-freedom of shared objects. In fact, wait-freedom requires a *finite* number of execution steps. Existance of a *limit* for execution forms an additional, stronger *bounded wait-free* property. A hierarchy of progress properties is discussed in subsection 2.1.3, with explanations on their differentiation.

2.1.2 Motivation

Wait-freedom guarantees progress of a task independent from the scheduler strategy and activity of other processes. On the one hand, any activity by any other process can delay the completion of a wait-free procedure only for a limited number of steps. On the other hand, wait-free algorithms do not rely on progress of other processes. Whenever CPU time is assigned, they are guaranteed to progresss in their task.

This description alone might give the misleading impression of wait-freedom as a general performance improvement. The opposite can be true: For non-blocking implementations, any stronger progress guarantee is earned at the expense of throughput and average case performance. Stronger progress guarantees are bad advice when these metrics have priority and varying worst-case performance is acceptable.

Wait-free algorithms are a solution to adversities that cannot be outrun by efficiency. In a larger scale, they eliminate the common pitfalls in concurrent software engineering, not just by reacting to them properly, but by making their occurrence impossible. Prominent examples are:

Deadlocks Processes that cannot proceed because they are waiting for resources that are mutually held by other processes

Priority inversion Low-priority processes hold a lock required by a higher priority process

Convoying Several processes acquire locks in a similar order. If a slow process acquires a lock first, all other processes slow to the speed of the first one, neutralizing their scheduling priority.

Kill-sensitivity A thread is aborted before releasing an exclusive lock to a resource, for example.

Async-signal safety Signal handlers cannot use lock-based primitives, especially malloc and free

These formal guarantees are invaluable for real-time software applications. In this domain, priorities are shifted to fail tolerance and adherence to timing schedules. Then again, specific guidelines and restrictions apply for implementations of time-critical software, especially in embedded software engineering. As a result, most published algorithms cannot be employed in real-time applications in their original form, and have to be modified first. This chapter closes with characteristic constraints in embedded- and real-time software engineering, and how they relate to wait-free data structures and their implementation.

2.1.3 Theoretical foundations

Designing wait-free data structures is an ambivalent challenge between complex formal conditions as well as meticulous technical obstacles. This subsection gives an introduction to terms and definitions from theoretical aspects of concurrent algorithms, focusing on progress guarantees and correctness conditions.

Classification of progress

Herlihy and Shavit give concise and comprehensive definitions of progress conditions in their work *On the Nature of Progress* [HS11]. They explain the elementary vocabulary of progress in concurrent methods and continue to construct a unified model for progress conditions using just two aspects: Whether guaranteed progress is minimal or maximal, and how progress depends on scheduling. The combination of both aspects leads to an intuitive classification.

Considering any possible schedule of operations on a shared object, they define *minimal progress* as a condition that guarantees at least one thread to make progress, and *maximal progress* if progress is guaranteed for all threads.

For the second aspect, it is evident how blocking algorithms always depend on scheduling characteristics. Deadlock- and starvation-freedom in blocking algorithms must derive from the assumption that some thread will leave a critical section at some point in the schedule. The non-blocking wait-free and lock-free properties, on the other hand, hold as long as any thread is scheduled at any time, and do not depend on the order of threads in the schedule.

Wait-freedom, the strongest possible condition, is independent from scheduling and guarantees progress for all methods in any execution. The classification criteria correspond to formal definitions previously stated by Petrank in linear temporal logic [PMS09]:

Definition 2.1.1 (Wait-Freedom)**.** An *execution* $e \in P$ is wait-free if and only if e satisfies

$$\forall t \quad (GF\, sched(t)) \rightarrow (GF\, prog(t)).$$

A program P is wait-free if every execution $e \in P$ is wait-free.

Rephrased informally, program is wait-free if it makes progress in any scheduling of any execution at any time. There is no statement on *how long* a program would have to make progress until completion, yet. In related work, as mentioned earlier, wait-freedom is often used as synonym for a limited number of execution steps for all operations. This requires a further restriction of maximal progress. Petrank defines:

Definition 2.1.2 (Bounded Wait-Freedom)**.** An execution $e \in P$ is k-bounded wait-free if the execution satisfies

$$\forall t \quad GF_k^{sched(t)} \quad prog(t).$$

A program P is *bounded* wait-free if for any $n \in \mathbb{N}$, there exists a $k \in \mathbb{N}$ such that all executions in $P(n)$ are k-bounded wait-free.

It is worth mentioning that the upper bound of steps to completion k in any history is not necessarily constant and may depend on other parameters, such as the number of threads or a data structure's capacity. The limit has to exist as a concrete value for any specific execution, though [Her, p. 59].

Linearizability and serializability

Correctness conditions are not related to progress or wait-freedom in particular. They prove sequential consistency of concurrent operations in general, i.e. that their semantics are robust in any parallel execution.

Serializability is a correctness condition well-known from databases and distributed computing. Essentially, a scheduling of parallel operations is serializable if the effect of its actions would also result from their sequential execution[Pap79]. Then, a concurrent object is serializable if and only if all schedulings of parallel operations to it are serializable.

Herlihy defined *linearizeability* as an additional correctness condition for concurrent objects [Her91].

Both conditions relate to the problem that operations to an object execute over an interval of time. Without additional conditions granted, the object's state between invocation and completion of an operation is undefined, as its effect might only be partially applied.

A shared object is *linearizable* if every operation appears to take effect instantaneously at any time during its execution. The instant of effect is called *linearization point* of that operation [HW90]. It defines the point in time where the effect of a concurrent operation can be observed by other threads, and separates two observable, consistent states of the object, before and after the operation. This reminds of characteristics of atomic operations and atomicity in the transaction model.

Now, the essential detail is that two linearization points within the same object never occur simultaneously. Also, the amount of time spend in an operation before and after its linearization point is irrelevant. In conclusion, linearization points of concurrent operations indeed describe their equivalent sequential scheduling. In brief: If a linearization point can be found for every method of a shared object, it can safely be claimed to be linearizable.

Serializability can only be decided globally by means of possible execution histories of all involved transactions. Linearizability is a condition local to a process. This is convenient, as interactions of threads do not have to be examined when proving wait-free progress. Just like serializability, linearizability states that one or more sequential schedules with identical effect (*linearizations*) must exist for any history of operations on a data structure. The described implications of linearization points are an intuitive way to prove this, and an invaluable in communicaton on correctness.

For overlapping method calls, execution order is arbitrary and corresponds to the order of their linearization points. Otherwise, the real-time ordering of operations must be respected in their linearizations.

Serialization does not define when a transaction must take effect relative to the time of invocation and response. Linearizable operations are required to take effect between their invocation and response, which is a useful restriction for use in real-time applications.

2.1.4 Technical foundations

In this work, low-level implementation details are omitted in favor of explanations of algorithmic patterns and theory. Obviously, a comprehensive documentation of a CPU instruction set is readily available, while new paradigms in data structures are hard to elaborate.

This section covers technical foundations that are essential prerequisites for lock-free data structures, but does only explain details necessary to understand implementations presented in this work.

Fundamental atomic read-modify-write instructions

The vocabulary available to phrase non-blocking algorithms consists of a surprisingly small set of processor instructions. All of them realize atomic operations on one variable in a read-modify-write cycle.

The conditions that define whether a single atomic instruction is available on a specific platform do not follow a predictable pattern. Non-blocking algorithms are inseperably coupled with low-level computation, and many elegant approaches are known to fail on a wide range of platforms. In fact, more than just a few algorithms have been published that are known to be disfunctional on conventional processing architectures, as they depend on exotic instruction sets of special purpose architectures.

The following gives a brief description of the most significant read-modify write operations with their semantics, and the architectures that support them [JP06]:

`LL/SC` abbreviates the pair of instructions *load-linked* (LL) and *store-conditional* (SC). An additional operation *validate-link* (VL) is typically implied. LL/SC is available on ARM, MIPS, and Alpha architectures.

- `LL(R)` returns the value of register `R`.

- `SC(R,v)` changes the value in register `R` to v and returns true, if and only if no other process performed a successful `SC` since the most recent call of `LL` of the current process. Simply put, `SC` fails if the value of the register has changed since it has been read.
- `VL(R)` returns true if no other process performed a successful `SC` on register `R`, which allows to test a register value without changing it.

`RLL/RSC` is the reduced variant of load-linked / store-conditional. Semantics are weaker than LL/SC, as spurious failures are permitted for RSC where SC would succeed, and no shared variables must be accessed between the latest call of `RLL` and `RSC`.

`CAS` is an umbrella term for Compare And Swap instructions which modifiy a variable if and only if its current value is identical to an expected value. Nowadays, CAS is supported on Intel x386, x64 and most general purpose architectures with operands are restricted to pointer size.

- `CAS(R,e,n)` returns true and sets the value of `R` to `n` if the value in `R` is `e`. Otherwise, it returns false.
- `extended CAS` is identical to regular CAS but the expected value is passed by reference and set to the variable's previous value on success. This signature is most prevalent in academic literature and can be easily derived from regular CAS.
- `DWCAS`, ''Double-Wide" CAS, performs CAS on two adjacent memory locations and is available on most modern x86 and x64 architectures via opcode `CMPXCHG16B`.
- `DCAS` (`CAS2`), frequently confused with `DWCAS`, performs CAS on two independent memory locations. Despite of being used in some published algorithm designs, it only supported on Motorola 680x0.

`Fetch-and-Add` increments the value of a variable by a given offset and returns the result. This instruction always succeeds.

Both Compare-And-Swap and LL/SC might appear to execute the same task with different semantics: updating a value atomically if some other thread did not succeed first in doing so. The mechanism of a link that is invalidated when a register is changed is, however, fundamentally more powerful.

Compare-and-Swap only allows to test against a value. A use case requires to first read the value of a shared variable to a local copy. CAS is then called with the local variable as expected value and the new value to store. But what if between reading and updating the shared variable other threads changed the value, and eventually set it back to its original state? The comparison with the expected value passes, so the CAS-operation modifies the shared variable and succeeds, with no indication for intermediate change.

This situation is known as the *ABA problem* in multithreaded computing. Considering the scheduling in Figure 2.1 as an example.

If a value is changed from A to B, and finally back to A between reading and CAS-updating a value, no change is detected. Especially in data structures, atomic variables represent a global state, such as next-pointers in a linked list. Interleaving changes affect

Thread 1	Thread 2
A = read(atomicObj)	
	A = read(atomicObj)
new = A + 1	
	B = A + 22
	CAS(&atomicObj, &A, B) $\rightarrow$ true
	CAS(&atomicObj, &B, A) $\rightarrow$ true
CAS(&atomicObj, &A, new) $\rightarrow$ true	...

Figure 2.1: Interleaving operations of two threads illustrating the ABA problem

consistency, and in an ABA-hazard, the same value represents two different states of the data structure object.

This problem does not exist for LL/SC, as the first change from A to B invalidates an existing link to the register, and a subsequent SC fails. With only CAS available, hazard pointers or similar reclamation schemes as described in subsection 2.2.3 must substitute the load-linked mechanism.

Publications on non-blocking algorithms often are restricted to either CAS or LL/SC as a prerequisite. Is it possible to port an algorithm to platforms that do not provide atomic instructions it utilitzes?

In theory, CAS and LL/SC can be implemented by means of each other. A lock-free implementation of CAS from LL/SC is shown in listing 2.2 as an example, but no wait-free reverse implementation is known. Jayanti presented wait-free bidirectional constructions of LL/SC and CAS, but makes unrealistic assumptions on semantics of LL/SC with respect to spurious failures of LL/SC [Jay98].

Michael found a wait-free implementation of LL/SC from CAS that is wait-free, albeit in $O(n^2)$ worst case time complexity and $O(t^2 + k)$ space complexity for t threads and k shared objects [Mic04b].

As LL/SC greatly simplifies the implementation of concurrent data structures, research focused on efficient construction of LL/SC from CAS rather than the reverse construction of CAS from LL/SC.

```
bool CAS(*addr, e, n)
{
  atomic {
    if (*addr == e) {
      *addr = n;
      return true;
    }
    return false;
  }
}
```

Listing 2.1: Semantics of Compare-And-Swap

```
bool CAS_LLSC(*addr, e, n)
{
    do {
        if (LL(addr) != e)
          return false;
    }
    while (!SC(addr, n));
    // SC succedded
    return true;
}
```

Listing 2.2: Lock-free CAS from LL/SC

Memory consistency

Rarely do programmers have to worry how instructions in source code will eventually be executed on a concrete CPU architecture on micro-instruction level. Compilers and processing architecture have a long history of robust optimization techniques that are commonly trusted.

Again, the combination of non-blocking algorithms and embedded hardware poses a challenge. Memory operations on general purpose processors can safely be considered sequentially consistent under nearly all circumstances, and concurrent algorithms hence can rely on (semantically) sequential memory consistency. In contrast, instruction order on many prevalent architectures in the embedded system domain is only limited when explicity enforced by memory barriers. Otherwise, load- and store- instructions may be executed out-of-order and cause memory access hazards.

The following consistency models are common in literature:

Sequential consistency	guarantees all reads and all writes are in-order.
Relaxed consistency	essentially only prevents dependent loads, but allows some reordering of regular loads and stores.
Weak consistency	allows any reordering of reads and writes and is only limited by memory barriers.

Memory consistency is typically either ignored in algorithm-centered publications, or presumed to be sequential for didactic brevity in literature. In reality, sequential consistency that prevents any reordering of instructions on memory is not easily achieved. Code optimization in compilers might weaken consistency even before execution and must be explicitly disabled.

Strict sequential consistency is usually abandoned already in compilation, were a vast amount of optimization techniques involves reordering of memory reads and writes. Explicit keywords like `volatile` in C (not to be confused with the Java keyword with the same name) allow to annotate code sections that depend on exact instruction order.

Even with correct instruction order in the binary executable, the processor architecture might take the liberty to manipulate memory access order in execution. The following table lists an excerpt of memory reorderings on ARMv7, AMD64 and x86.

CPU	Ld/Ld	Ld after St	St/St	St after Ld	atomic after St	dep. Ld/Ld
ARMv7	yes	yes	yes	yes	yes	no
x86/x64	no	no	no	yes	no	no
AMD64	no	no	no	yes	no	no

As mentioned initially, memory access on general purpose architectures like x86/x64 can almost be treated as sequentially consistent. In contrast, ARMv7, the most relevant CPU architecture for embedded applications in this comparison, essentially knows no restrictions on order. This allows aggressive optimization techniques at runtime, but critical order must be enforced manually using *memory barriers*. Barriers (also: fences) are instructions that, once reached in execution, force a all CPUs to complete a specific kind of memory operation before proceeding. The barrier instruction itself is often written in verbatim inline assembly and must be guarded from compiler optimization. Typical memory barriers are:

SFENCE	forces all stores to memory to complete before the next store operation.
LFENCE	forces all loads from memory to complete before the next load operation.
MFENCE	forces any access to memory to complete any following memory access.
LOCK	implicitly has MFENCE as side-effect.

Finally, cache coherence and memory consistency might have less drastic consequences for implementation of concurrent algorithms after all: Non- blocking algorithms are designed to coordinate access to global states using atomic read-modify-writes. Atomic instructions automatically establish a full fence and clear the cache line of their operands. In most cases, these implicit effects suffice to avoid consistency hazards.

Memory contention and false sharing

Local caches reduce memory access time of processors by several orders of magnitude, but their contents must be kept consistent. For cache-coherent multiprocessors, local caches are organized in regions of equal size. These *cache lines* are invalidated as a whole if they differ from the latest value stored.

Slight differences in concurrent algorithms can make a considerable difference on how much a program benefits from processor caches. Consider a simple lock, a shared variable used for central coordination of threads. Assuming that the variable is read and modified by every thread in a loop. Any modification of the object's state invalidates at least one cache line on every CPU core, in effect causing a cache miss whenever the object is read.

A specific usage pattern known as *false sharing* causes avoidable cache invalidation and is notorious for degrading performance: Assuming two separate shared objects that are unrelated in an algorithm's model. If two threads operate on one of the objects exclusively, there are no mutual effects and no memory contention should occur. However, if the objects are stored in memory locations that fall into the same cache line, any modification of one object invalidates the cache line that includes the other. The typical countermeasure for false sharing is to instruct the compiler to align and pad objects into memory regions of the size of a cache line. The respective *cache alignment* and *cache padding* keywords vary between compilers.

The phenomenon of degrading memory access times and overhead from conflicting operations with an increasing number of threads is referred to as *contention*, an umbrella term for competing modifications to the same locations in memory. Effects of high contention rates, i. e. a high amount of threads accessing an object at the same time, are often overlooked in algorithm design, especially on simulated computing models. On real hardware, memory access notably affects performance to a degree that can only be estimated in micro-benchmarks [MS07, 1.1.1].

Restrictions in embedded- and real-time software engineering

Not all embedded systems host real-time applications, and not all real-time applications are deployed on embedded systems, of course. However, as the term suggests, embedded systems frequently interact with processes in the physical world. A control loop implemented in software periodically responds to its surrounding system. Hard real-time applications operate in high loop frequencies of 1 KHz and more, and even extremely rare or slight variances in their response frequency might build up unrecoverable instability or result in other functional failures, like loss of image data from a medical scanner.

The scheduled time of the next response is a critical deadline that must be met at all cost. Software engineering in the real-time domain is understandably governed by the concern for deterministic task execution.

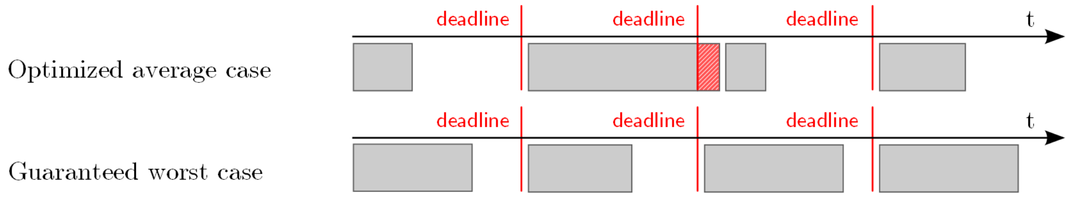

Figure 2.2: Periodical task execution times, optimized average case versus guaranteed worst case

As a consequence, algorithms employed in embedded- and real-time applications are optimized for guaranteed worst-case complexity as opposed to best average-case performance, as illustrated in Figure 2.2. Despite limited resources and the demand for low power consumption, predictable performance is preferred to optimizations of the common case when facing critical deadlines.

In addition, embedded devices often are physically integrated in systems with long maintenance intervals of possibly several years, or difficult to access, raising high stability and fault-tolerance to formal requirements. Therefore, common algorithmic practices might render unusable due to otherwise acceptable risks of failure.

Industrial guidelines for embedded- and real-time software engineering, such as MISRA-C and JSF C++, state variants of the following restrictions:

Dynamic memory allocation is guaranteed to introduce non-deterministic delays. Allocation complexity depends on available space and heap fragmentation, which cannot be avoided in general. To prevent delays in critical loops, a fixed-size memory area is allocated in the initialization phase of the application, before performing any time-critical tasks. The principle of worst-case optimization also applies to memory consumption: because later allocation at run time is forbidden, the potential maximum of memory required has to be allocated.

Garbage collection mechanisms obviously contradict determinism, but are also prone to memory leaks, especially when in continuous duty over long periods of time.

Standard libraries such as the C++ STL use dynamic memory allocation indirectly, and have to be replaced by alternatives specifically tailored to real-time constraints.

Monotonic counters are part of many algorithms, with the risk of overflow argued as astronomically low. In long-running systems, however, this probability steadily increases.

2.2 Related work

This section summarizes prior publications on wait-free data structures and foundations that are relevant to their design and implementation. Applicability of existing approaches and necessary modifications for real-time systems are discussed in brief.

2.2.1 Universal construction of wait-free data structures

In a work that was later awarded with the Dijkstra Price, Maurice Herlihy presented a novel systematic construction of wait-free algorithms in 1991 [Her91], three years after he conducted a proof that all algorithms can be implemented wait-free [Her88]. A universal construction procedurally transforms arbitrary sequential objects into wait- free counterparts. The wait-free algorithms produced by Herlihy's pioneer methods might not be known for their efficiency, but they represent a tangible example for the formal proof that wait-freedom can be achieved for arbitrary shared objects.

Since then, several further universal constructions have been developed. Recent work aims for improved performance of universally constructed algorithms. Fatourou obtained an efficient wait-free universal construction named *P-Sim* [FK11]. The proposed implementation of the P-Sim algorithm uses tagged pointers and therefore must be adapted for embedded and real-time applications. We evaluate a wait-free stack based on P-SIM in section 2.6,

Kogan and Petrank described the *fast-path / slow path* method [KP12], an implementation strategy where operations on a data structure are performed as a cheap lock-free operation first, falling back to a slower wait-free operation if it fails. Data structures based on this strategy showed performance results close to their lock-free counterparts.

Apart from achievements in practical wait-freedom, the fundamental theoretical implications of wait-freedom are subject to ongoing research. In 2012, Ellen, Faith and Fatourou proved that the property of disjoint access on a data structure is mutually exclusive to the wait-free property, with fundamental implications for future approaches in wait-free constructions [EFK$^+$12].

2.2.2 Simulation of wait-free data structures and the helping mechanism

Only recently, Timnat and Petrank presented a new variant to derive wait-free algorithms from existing lock-free implementations [TP14]. Universal constructions utilize small algorithmic building blocks to rework sequential objects gradually. In what they call a *wait-free simulation*, Timnat and Petrank rather employ a modular and, in some cases, less invasive refacturing scheme.

Wait-free simulation is an advancement from a helping scheme which is first known from a publication by John Turek at IBM in 1992 [TSP92]. The mode of thought in recently presented helper schemes follows the basic principle presented by Turek.

A single lock-free operation, such as adding or removing an element, first is transformed into linearizable, subsequent stages. The operation is then called a *normalized* algorithmn. These sub-steps are partial applications of an operation, but leave the data structure in a well-defined, consistent state. Also, operations are not executed directly, but only announced: the state of a pending modification is stored in an operation description object that then contains sufficient information to complete a single pending modification on the data structure. As a result, every thread can help complete any pending operation by performing its next sub-step.

The motivation behind this is to decouple operations from the thread that executes them. As an additional benefit, the state of an operation description preserves partial progress of an operation, comparable to safepoints as known from transaction management. In case of a conflicting access, an operation is retried starting from its last successful partial execution instead of repeating a single procedure from its beginning until it succeeds as a whole.

Acting in the spirit of wait-freedom, the helping scheme achieves that once a thread announced an operation, it is guaranteed to be completed even if the thread is never scheduled again.

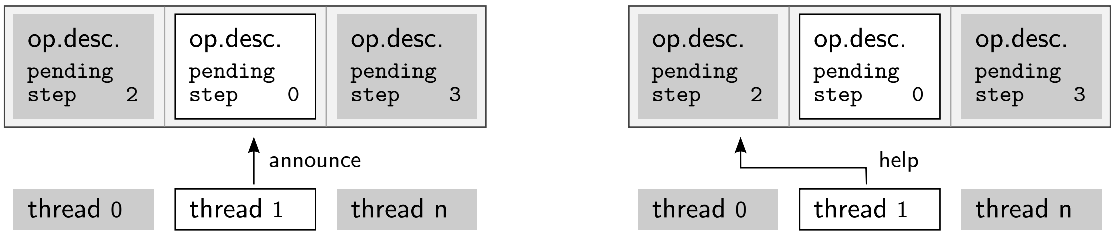

Figure 2.3: Announcing and helping operations in the general helping scheme

As every thread can only initiate one operation at any time at most, one operation description object is required for every thread. These are stored in a global array with every index dedicated to a specific thread. A thread can only modify the data structure by initializing a corresponding description object in the thread's dedicated slot in the array.

An operation description object represents a safepoints on a stable, intermediate states, and is thus only changed when one of their associated substeps has been completed. It is initialized in pending state and marked as non-pending once its last substep succeeded.

A thread only returns from its original operation call when its description object has been updated to the final, non-pending state. However, after they announced their own operation, threads engage in all pending modifications, helping older operations first. New operations on the data structure consequently cannot delay other ongoing operations by more than one substep. This is essential to the guarantee for each operation to be completed in a finite number of steps.

Timnat and Petrank published promising performance evaluations for simulated wait-free data structures. Their candidate algorithms have been implemented in Java and rely on garbage collection, however, and currently no wait-free garbage collector exists. With the intention to deploy wait-free algorithms on long-running embedded systems, this technical flaw is of high practical relevance.

To put it mildly, wait-free memory management offers many opportunities for future work; it has been aptly put as the achilles heel of wait-free data structures, and reasons for this will soon be evident in this chapter.

2.2.3 Memory management

A wide range of challenges in concurrent algorithms emerges from memory management within data structures: As memory resources are acquired and read by several threads simultaneously, it is not trivial to decide which accessor is responsible for acquisition and release of a specific memory region, especially when employing the helping pattern as described in subsection 2.2.2.

In software development for embedded systems, garbage collection usually is not provided by the runtime environment. Allocation and reclamation of memory therefore has to be implemented explicitly, and, in the context of this work, with respect to wait-free requirements.

Memory allocation

The problem of concurrent resource management has been reduced to an abstract data structure named *pool* by Udi Manber [Man86]. Essentially, a pool is an container of objects which abstracts their concrete allocation in memory.

In their book *Concurrent Data Structures*, Moir and Shavit discuss pools as a fundamental tool for concurrent lists and queues [MS07, 1-17]. They mention an array-based implementation and suggest more sophisticated approaches based on counting networks or diffracting trees [SZ96], but do not discuss wait-free variants.

It is important to point out that simulation of wait-free data structures from their lock-free implementation as described in [CER10] and [ZZY$^+$13] involves an operation description buffer. Therefore, wait-free algorithms for memory management cannot be constructed using these simulation techniques, as the construction would again rely on a wait-free pool. A scalable lock-free dynamic storage allocator is presented by Michael [Mic04c], which cannot be transformed to a wait-free variant for this reason.

Stellwag and Krainz specifically address wait-free storage allocation and present a design that is apparently suitable for real-time applications [SKSP] which is discussed in subsection 2.4.2

Wait-free memory allocation is rarely even addressed in related work. Most designs of wait-free algorithms are conceptually presented in Java and benefit from garbage collection, leaving wait-free memory management out of scope.

Memory reclamation

In addition to the problem of concurrent memory allocation, an indirect variant of the ABA problem is introduced when allowing shared memory objects to be reclaimed: After a shared address has been obtained, the referenced object might have been reclaimed and replaced by a new object instantiated at the same address.

A commonly used technique to guard referenced objects from reclamation objects is *reference counting*: A global counter variable is introduced for every shared memory block, which is incremented before referencing it, and decremented as soon as the contained object is no longer used by the respective thread. A shared memory region is consequently released when its associated reference counter reaches 0, as no thread is holding a reference to it at this point. Efficient solutions for lock-free algorithms are known [JP05] [GPST09], and a wait-free mechanism is conceptionally available [Sun05].

Regarding overflow of counters, reference counting methods formally do not solve the ABA problem but only make it unlikely to occur, as an identical counter value may refer to different references.

In a very popular work, Michael introduced *Hazard Pointers*, a patented scheme that allows wait-free memory reclamation [Mic04a, p. 492] relying on available atomic operations *CAS* or *LL/SC*: Each thread operating on a shared data structure owns a fixed amount of hazard pointers. A single active hazard pointer guards one reference at any time, so the amount of hazard pointers per thread depends on the maximum number of simultaneous guards needed in any operation.

When a thread releases a node for reuse, as when removing an element from a linked list, it is not deallocated but added to a local list of retired nodes. Nodes in this list are only deallocated if no other thread is holding a hazard pointer to it. A grace period is implicitly kept, as deallocation is only tried once the retired list reaches a certain size.

The hazard pointer algorithm itself is proven to be wait-free, but has only been presented as a solution in lock-free algorithms. The integration of hazard pointers in wait-free data structures is not trivial in general, and applicability for real-time applications needs additional considerations. How hazard pointers can be used in accordance with wait-freedom has to be examined for each individual case. An example is discussed in detail in subsection 2.5.2, which also includes a proof demonstrating how hazard pointers comply to real-time constraints.

In order to integrate hazard pointers in a data structure, an upper bound for the amount of guarded references for any operation has to exist, and to be known a-priori. Michael demonstrated hazard pointers in lock-free implementations of a list and a stack, where one hazard pointer per thread is sufficient, as no operation needs to guard more than one element. For some implementations of recursively defined data types, e.g. graphs and trees, a maximum amount of guarded nodes per operation does not inherently exist, though. This even poses a problem for much simpler data structures: the original deletion mechanism in Harris linked list requires an arbitrary amount of pointer guards [Har01].

A memory reclamation scheme that overcomes this limitation is known from Herlihy, Luchangco and Moir. Their algorithm *pass the buck* resembles hazard pointers in principle, [HLM02].

A class of reclamation strategies is based on *quiescent states*, i.e system states where an object is impossible to be referenced by another thread. The most prominent example is *Read-Copy-Update*, introduced by McKenney [MS98] and added to the Linux kernel in October of 2002. In RCU, multiple local copies of an object are kept, allowing wait-free reads in single-writer / multiple reader scenarios. Updating the shared object cannot be transformed to wait-free operations, however.

Fraser provided *Epoch-Based Reclamation* [Fra04] as another approach on reclamation involving quiescent states. It avoids expensive memory barriers as required for Hazard Pointers or Pass the Buck. Limbo lists are used to manage retired objects in this strategy, which can be implemented with lock-free, but not wait-free properties. Also, the *EBR* scheme itself is not strictly lock-free, as reclamation depends on progress of another thread and makes assumptions on the fairness of the scheduler. In epoch-based reclamation, no objects are deallocated as long as one thread is accessing the data structure.

Several other approaches for memory management in concurrent data structures have been published [Har01], which suffer from dependency on CPU architecture or a certain kind of scheduler.

More recently, Shin, Kim et al. introduced *Strata*, a wait-free memory reclamation scheme based on linked lists and chronological access [SKKE11], and evaluated their solution against user-space *RCU*.

2.2.4 Queues

Despite its unimposing functional capabilities, the concurrent queue is the functional core of many data structures. Queues are the most frequently discussed complex data type in publications in the wait-free domain, precisely because of their unsurprising yet indispensabel semantics.

The classic concurrent queue originates from Lesley Lamport in 1977 [Lam77]. The Lamport queue only supports synchronization for a single reader and writer but still is an undisputed solution for this scenario.

Michael and Scott introduced another celebrity among concurrent queue algorithms. The MS-queue is lock-free, and an imperative reference when discussing data structures with queue semantics.

Notable improvements on wait-free queue algorithms are quite recent, with the first practicable solution presented by Kogan and Petrank [KP11] in 2011, which employs a helper scheme with prioritized operation descriptions as described previously to resolve conflicting accesses.

The prioritization of pending operations is achieved using a monotonic counter: Each operation description contains a phase number, interpreted as the operation's timestamp. After a thread has announced its operation, it traverses the operation description and engages in every pending operation with a phase number less than or equal to its own. The phase value of a newly announced operation is thus intended to be greater, thus of lower priority, than phase values of older pending operations. This is argued to be achieved by traversing all elements in the operation description buffer to resolve the current maximum phase, incrementing it, and setting the result as a new operation's phase.

Kogan and Petrank complemented the wait-free queue with the *fast-path / slow-path* methodology [KP12]. It describes the simple yet effective algorithmic pattern to try wait-free operations as a fast, lock- free variant first, and only fall back to a slower wait-free path if it fails. In some benchmark scenarios, a fast-path variant of the Kogan-Petrank queue achieved throughput measurements close to lock-free candidates.

Nearly all published evaluations confirm notable performance gains of wait-free algorithms with an optimistic fast path.

2.2.5 Lists

Valois succeeded in the first implementation of a lock-free list that only requires prevalent compare-and-swap instructions [Val95]. Michael and Scott pointed out how the memory reclamation as presented by Valois is prone to an ABA race condition, and also suggested a solution [MS95]. The Valois list is still potentially causing memory leaks in the revised version, and is therefore not evaluated in this chapter.

Comparable to the Michael-Scott queue, Harris' linked list is an established reference for lock-free ordered lists [Har01]. The *Tim Harris algorithm* executes dequeue operations in two stages using tagged pointers. Nodes are first marked as logically deleted and phyisically deallocated afterwards. This process is not kill-tolerant as a node would never be physically deleted if a thread cancels its operation in the second phase. Michael revised Harris' algorithm and resolved pointer tags and vulnerability to cancellation using his hazard pointer scheme.

Michael improved on Harris' list in his publications on hazard pointers, where tagged pointers and the two stages in the deletion of nodes are replaced by his safe memory reclamation technique [Mic04a].

Timnat and Braginsky transferred the helper scheme and prioritization pattern from Kogan and Petrank's wait-free queue to Harris' linked list [TBKP12]. Although they again only evaluate an implementation in Java, their work presents a practicable design of a wait-free linked list.

An implementation for real-time applications demands additional effort, similar to Kogan and Petrank's queue.

2.2.6 Stacks

Treiber proposed a lock-free concurrent stack implementation in which he represents the stack as a singly-linked list. A top pointer for operations on the stack is modified atomically using Compare-and-Swap [DT86].

Jayanti and Petrovic presendet a wait-free stack that supports multiple producers but only a single consumer [JP05].

Hendler and Shavit applied an optimization method to a lock-free stack that benefits from LIFO order [HSY04]. The *elimination* paradigm considerably reduces contention in concurrent data structures and can be applied to any pair of methods that eliminate their effects mututally: An additional execution path mediates between two parallel `push` and `pop` operations directly so `pop` obtains its result from the value argument in `push` without coordination in the data object. This way, operations that otherwise might conflict can complete even faster compared to their sequential execution, because there is no need to query or modify the state of the data structure in the elimination path.

Moir optimized a lock-free FIFO queue and demonstrated that elimination is also beneficial when it is only applicable for specific combinations [MNSS05] of access collisions. A similar *combining* technique exists that coordinates colliding operations with identical semantics, but is not wait-free in its original design.

Timnat and Petrank mention they have constructed wait-free implementations of the Harris-Linked-List, a skiplist, and a tree, using the aforementioned simulation technique described in the same work [TP14], but did not apply their simulation technique to a stack, where contention is more frequent in comparison as producers and consumers logically operate on the same node. No algorithm code is provided for their data structures. These had to be modified for use in real-time applications in any case, as they rely on garbage collection.

A wait-free stack for multiple consumers and multiple producers is presented in section 2.6 of this work. The implementation builds upon Fatourou's universal construction scheme *P-Sim* which achieves improved scalability using *elimination*.

Bar-Nissan evaluated elimination and combining for blocking and lock-free stack algorithms [BNHS11]. In performance evaluation of stack algorithms in this work, we examine if comparable improvements can be achieved while maintaining wait-free progress.

2.3 Verification and benchmark methodology

Meaningful evaluation of algorithms requires to ensure that candidate implementations realize identical semantics, and that identical conditions are used to measure their performance. The means of verifiying semantics and correctness and the setup of the benchmark suite is explained in this section.

2.3.1 Explicit-state model checking

In the software engineering process, testing has evolved as the most common and important method to verify the correctness of a software system against its specification. However, tests are limited to their concrete scenarios by nature and can only indicate correctness for exemplary executions. As conventional unit tests are agnostic of non-deterministic effects, they cannot be applied to specifications of parallel implementations. As testing every possible scheduling of a given set of transactions is not feasible, model-based verification is left as a viable option. In model checking, a system is abstracted as a finite-state automaton corresponding to the system's behavior which is represented and verified logically against a specification, represented by a set of formulas. Elaborating a formal model from an implementation manually - as well as the inverse process - is prone to error and thus can result in critical differences between the source code used in production and the verified model. In addition, code optimizations applied by the compiler can introduce hazards that are not evident in the original source code. Ideally, model checking operates on the compilation result as it is used in the software product's release.

For verification of the data structures presented in this chapter, a tool chain based on the *DiVine* explicit state model checker [BBH^{+}13] has been composed. Figure 2.4 illustrates its steps.

The *verification system* is a custom source code implementation designed to execute schedulings that lead to hazard situations in the examined data structure by calling parallel operations on a single instance of it. The system is compiled to LLVM bitcode using the *clang* compiler. In the linker stage, the C++11 standard library is replaced

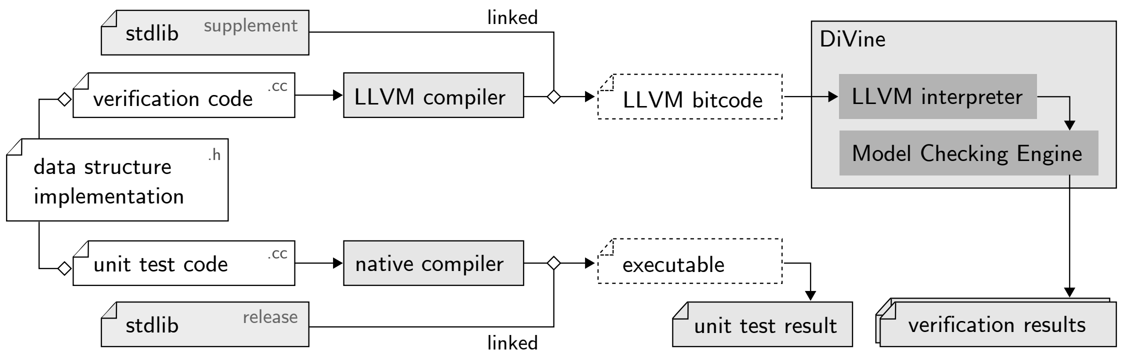

Figure 2.4: The verification tool chain

by substitutes provided by *DiVine* that allow detection of memory hazards within the C/C++ runtime and parts of the *STL*.

The logical model for a data structure is derived directly from the LLVM bitcode. For verification, *DiVine*'s integrated LLVM interpreter transforms the system to a *nondeterministic Büchi automaton* (NBA) considering all possible operation interleavings at the level of bitcode instructions. On the NBA's state space, a reachability test on every negated assertion is performed, as well as cycle detection for deadlocks [BK08, p. 159]. A verification run results in a counter-example for the first violated verification property, presented as an operation history of all threads.

DiVine's explicit-state model checking can only verify assertions that are reachable in at least one possible execution history of a program. Source code implementing a verification program resembles unit tests, with assertions on expected values in execution paths. Even with all possible interleavings explored, an incorrect implementation might pass verification if program states that would lead to failures are impossible to be reached in any schedule. It must therefore be argued individually that the verified system is comprehensive enough to produce schedules that provoke all hazard conditions to be tested.

On the other hand, it is advised to design a succinct verification system using the minimum amount of threads and smallest data structure capacity possible. Known as the phenomenon of *state space explosion*, the complexity of verified interaction schedules grows at least exponentially with the degree of parallelism and the amount of concurrent operations in the model. [BK08, p. 77]. Own findings and and related experiments [vdB13] show that rapid growth of the explored state space might render seemingly trivial verification setups impractical. Model checking of 5 or more threads operating on even an arguably trivial shared data structure is not feasible as of this writing [vdB13, p. 46], as its execution might take hours and days; higher degrees of concurrency have not been necessary for this work, however. Instead of monolithic, complex scenarios designed to cover all hazard situations, we define a series of scenarios that are verified individually. In conclusion, the following general process for designing a verification system for a data structure is recommended:

1. Declaring white-box conditions as assertions in the implementation of the data structure, i. e. preconditions, postconditions, and loop invariants
2. Identifying potential hazards in the data structure
3. Deriving critical schedules that would trigger every hazard condition

4. Implementation of a verification program that is capable of producing all critical schedules
5. Assertions on black box correctness, i. e. expected return values, of the data structure in the verification main program

Still, this procedure provides no guarantee that a positive verification result is *true positive* and evidence for correctness. As explained, a scenario might yield a false positive result if it is insufficient to produce hazard situations in the first place.

A principle known from test-driven development gives a pragmatic solution. Here, complete test scenarios are defined first and performed on a known incorrect implementation. All tests then are expected to fail. Similarily, we first use *DiVine* to detect a counter-evidence that demonstrates a known error in an implementation. This *true negative* result demonstrates how a hazard condition is in fact detectable. As a concrete example, a necessary pointer guard is removed from a data structure. Verification is supposed to disprove correctness and present a counter-example that depends on the missing guard.

As shown in Figure 2.4, a unit test suite is build in addition to the model checking chain. Unit tests are identical to the verification program, but configured with a higher amount of threads and a higher capacity of the data structure. They are executed as a smoke test before starting the *DiVine* chain, as state space exploration might take hours to complete. Once all candidate implementations of a data type have passed the same verification process, their semantics are considered correct and equivalent, a prerequisite for their meaningful comparison.

2.3.2 Benchmark methodology

The benefit of an implementation approach can only be reasoned when it is presented with means of a comprehensive comparison to its alternatives. Candidate implementations are embedded in a series of standardized scenarios which have been modeled from real-world load conditions. The resulting performance metrics aggregated from run-time measurements in each scenario give an intuitive understanding of fitness for common application characteristics.

Relevant performance metrics

A single test case is repeated for every combination of the defined parameter settings range, consequently testing scalability in all dimensions. These are at least the number of elements managed by the test instance and the number of threads operating on the instance. As only a single parameter is modified in every run, variations in performance can be reasoned easily.

Similar to verification systems, benchmark scenarios are modeled based on properties derived from semantics. However, the quality of a benchmark is defined by how closely it is related to real-world applications. Because of this, performance criteria and benchmark task definitions are not formally constructed but derive from common use cases. The following performance measures are a standard in micro-benchmarks of algorithms and data structures in general:

Latency	Timestamps are created right before calling an operation and after its completion. Operation latency is measured as the difference of these timestamps.
Jitter	The overall fluctuation margin of operation latencies.
Throughput	the total number of operations divided by their overall time to completion, measured in operations per second. Throughput is measured individually for every operation type.
Speedup	The relation of execution time depending on the number of threads. The same number of operations is executed by an increasing number of processor cores to examine how throughput scales with the degree of parallelization,

Evaluation of speedup follows the well-known *Amdahl's Law*. For parallelization with p as the proportion of a program that is executed in parallel using n processors, speedup is defined as:

$$\mathrm{Speedup}(n) = \frac{1}{(1-p) + p/n}$$

Concrete values of p allow to predict speedup for any degree of parallelization, but are unknown for evaluated algorithms a priori. We measure execution time for an identical problem size with an increasing number of cores n_k. The estimated proportion of parallelizable regions p_e is then derived from speedup s_e as:

$$p_e = \frac{1/s_e - 1}{1/n_k - 1}$$

Amdahl's law does not consider cache coherency and memory access in general, which affect estimations based on actual performance measurement. Gunther presented the *Universal Scalability Law*, which extends Amdahl's model of scalability by contention penalty $0 \leq \alpha$ and coherence penalty $\beta < 1$ [Gun93]:

$$C(n) = \frac{n}{1 + \alpha(n-1) + \beta \cdot n(n-1)}$$

These models are not to be used to curve-fit measurements, but help to discuss observed differences in performance of candidate algorithms. In particular, Gunther's law explains how throughput can actually worsen when a problem is distributed to an increasing number of cores. In fact, this phenomenon will later be observed in evaluation in several cases.

In literature and related work, performance evaluations of data structures typically rank candidates exclusively on average measures, such as the mean throughput $|op|/sec$, an arguably sensible initial performance indicator for container types. Publications on wait-free algorithms follow this convention and disregard latency measurements completely. All known publications on wait-free algorithms only discuss averaged throughput measurements [KP11] [ZZY$^+$13], despite guaranteed global progress and bounded time to completion being the actual benefit of wait-freedom.

For the data structures presented in the following, worst-case performance is interpreted from measurements in addition to average costs. To give an intuition for complexity overhead of wait-free implementations compared to their lock-free alternatives, verification and evaluation will also include popular lock-free implementations. The benchmark scenarios and the performance metrics include the methods typically found in related publications, disregarding their significance for wait-freedom or applicability on embedded systems. Some test scenarios are not even representative for any realistic use case, but pose an informal standard as they are used in many published evaluations. Results from these benchmarks are presented to allow comparison with results published in related work.

A consolidated metric of worst-case latency is not provided in any related work. Presumably, this is because operation latency highly depends on the test environment and hence measurements are difficult to reproduce. As a matter of fact, distribution of latency is the most valuable metric when deciding for an algorithm in embedded applications. More precisely, worst-case latency is the single criterion to decide if a candidate implementation is better suited than another to meet critical deadlines within hard-timed control loops. For this reason, experiments in this work are also performed on a complementary setup specifically configured for latency measurements.

Experimental setup

We conducted performance evaluation of candidate implementations in this work based on measurements of *throughput* and *latency*. Benchmarks have been performed on two system configurations, each chosen and configured for a specific performance metric:

AMD Opteron	48-core, 1.9 GHz AMD Opteron 6168, NUMA, Linux 3.11.0-26-generic kernel. Memory access adds non-deterministic latencies on this platform, but the high number of available cores allow **throughput measurements** varying in degree of parallelism.
ARM Cortex-A9	4-core, 996 MHz ARM Cortex-A9, SMP. Configured for **latency measurements**, with a minimal Linux installation and hardware interrupts disabled where possible.

The basis of evaluation are experiments executed on a benchmark suite that has been specifically designed for this work. Its mandatory parameters to execute a benchmark are:

`u`	the concrete implementation variant (unit) to test. Implicitly also defines available benchmark scenarios.
`S`	the benchmark scenario to execute.
$\mathtt{n}_\mathtt{T}$	the number of threads operating on the shared container in total, or
$\mathtt{n}_\mathtt{P}, \mathtt{n}_\mathtt{C}$	the number of producer and consumer threads, if applicable
`c`	the number of elements managed by the data structure, or container capacity
`i`	the number of iterations, configuring how many times each thread executes its operation sequence.

The benchmark suite provides platform-independent, reproducible time measurements, utilizing the most precise performance counters (native constant *Time Stamp Counter*) and wall-clock time available on each of the test platforms. Integrated self-test routines help to ensure correct functionality of internal methods used to measure performance on a specific test platform. Most importantly, self-tests detemine precision and accuracy of time measurements by comparing performance counter measurements of busy loops with elapsed wall-clock time. Operations are measured using both timestamp counters (`RDTSC` on AMD Opteron with monotonic, constant TSC, `PMU` on ARM) and the platform's native clock (`clock_gettime`) for a series of varying busy loops. Measurements of timestamp- and clock-based timers are expected to be identical, ensuring that fluctuations from expected measurements originate from `usleep`. A setup on a platform is considered adequate for performance tests if precision and accuracy is near-constant for any measured period.

Accuracy and precision of time measurements are tedious to verify and optimize for a specific architecture. The *Performance API* library [1] is a helpful starting point for stable measurements for a wide range of platforms. It is selectable in the benchmark suite as a substitute for custom performance counter implementations. Custom timing implementations gave marginally improved precision and reduced the constant latency overhead of measurements by approximately 40%.

As another prerequisite for meaningful measurements, influences by external processes during the execution of a benchmark and overhead from measurements must be eliminated. All scenarios utilize auxiliary data structures, such as containers to collect measurement data. Their instantiation or any other memory allocation within a scenario would interfere with measured operations. A benchmark run therefore consists of three isolated, sequential steps:

Initialization generates all test data in advance, and pre-allocates all auxiliary container instances to their maximum size required for the benchmark scenario. No further allocation occurs in the execution step to avoid non-determinism in execution times.

Execution starts all threads of the benchmark scenario. Measurements are stored unprocessed in pre-allocated, thread-local containers that provide $O(1)$ access.

Reporting collecting and processing measurement data from all threads after the scenario has completed. Writes measurements as unprocessed timestamps to sample files, and processed performance metrics derived from these samples to a summary file.

To avoid influence from external processes and to reduce the probability of outlier measurements caused by the system environment, active threads in an execution are pinned to a dedicated processor core. Benchmark scenarios are performed multiple times with individual results collected in individual data frames. Plots presented in this work have been generated directly from output of the benchmark suite using automated procedures in *GNU R*.

Benchmark code was compiled using the prevalent GNU Compiler Collection (GCC) version 4.8.2 with the -O3 flag for all algorithms.

Limitations of evaluation

Performance metrics are exclusively derived from time measurements. Memory consumption is not tested, but only deduced from the implementation in the verification

1 PAPI official website: http://icl.cs.utk.edu/papi/

phase. As all tested implementations solely rely on static memory allocation, worst-case upper bounds of memory consumption must be determined and asserted in verification.

It must be emphasized that latency measurements only describe *observed* latencies within a limited number of task iterations. Evaluation of maximum latency hence only refers to a limited set of measured operations within a single test scenario which only define a lower bound of theoretical worst-case execution time.

2.4 Pools

All data structures presented in the following sections are dynamic collections, i. e. their number of elements is variable; however, this must be achieved without heap allocation, following the guidelines described in section 2.1.4. In this section, pools are discussed as a solution for dynamic object allocation in real-time systems.

2.4.1 Definitions

In conventional software engineering terms, a *pool* describes a container providing reuse of its elements. Pools can be used to cache instances of types with expensive initialization, or to manage limited resource entities, such as blocks of memory.

In the context of this work, pools are used for dynamic allocation of shared objects. They manage a dedicated coherent range of memory, and provide indexed access to objects previously added to them them. Effectively, pools described in this section act as dynamic storage allocators for a specific object type: They transparently manage instantiation and reclamation of objects in static memory ranges, and eliminate memory fragmentation. Pools divide memory into segments of identical size which, in theory, reduces allocation to linear worst-case complexity.

Upper bounds of memory allocation complexity are substantial. The progress guarantee of any data structure that itself is implemented as a wait-free algorithm is restricted to the progress of its memory management mechanisms.

When threads acquire instances from a pool concurrenly, the pool implementation ensures that every instance is only obtained once. Differing from general dynamic storage allocation, a pool is bound to a specific element type and therefore operates on memory blocks with fixed size.

Formally, the term *concurrent pool* [Man86] refers to an abstract data structure representing a *multiset*, supporting the following operations:

`Add(p,x)`	add element x to pool p. No effect if x is already present.
`RemoveAny(p,y)`	remove any element from pool p and assign it to y.

To specialize this definition to the purpose of dynamic object allocation, the signature is extended by object pointers. As an object pool is coupled with a single object type, its elements occupy constant memory and their addresses can be represented by an index.

$$\texttt{newIndexPool()} = \texttt{p} \tag{2.1}$$
$$\texttt{newElement(p)} = \texttt{i} \tag{2.2}$$
$$\texttt{addIndex(p,removeAnyIndex(p))} = \texttt{p} \tag{2.3}$$
$$\texttt{removeAnyIndex(newIndexPool())} = \texttt{ERROR} \tag{2.4}$$
$$\texttt{addIndex(addIndex(p,i),i)} = \texttt{ERROR} \tag{2.5}$$

Figure 2.5: Semantics of the index pool data type (axiomatic specification)

`FreeObject(p, (i,e))`	add a tuple of an element e and its index i to pool p. Element values are arbitrary and may contain duplicates.
`AllocateObject(p, (i,e))`	remove any previously added element from pool p and return the element and its index in the tuple (e, i).

Functions *FreeObject* and *AllocateObject* correspond to *addElement* and *removeElement* as defined for the *multiset* data type. Allocating an elements is removing it from the set, releasing an element is adding it back. The dynamic object allocation consists of two concepts: an allocation strategy organizing elements in memory by index, and a data structure managing concurrent access to these indices.

Disregarding implementation, pools can be modeled as a combination of an indexed *list* of pool elements and a *set* of indices in the list. By obtaining a dedicated index from the set, the memory block in the list at this index position is reserved for the thread that obtained it. Conflicting accesses therefore only occur on the set of indices. Systematically, concurrent access to objects in a pool can be reduced to the problem of acquiring dedicated pool indices atomically, which then implicitly serialize operations on their referenced memory blocks.

Finally, the fundamental data structure crucial for concurrent dynamic object allocation is similar to a *set* of integers, with relaxed semantics of the $remove$ operations. As the value returned from an index pool is previously unknown to the caller, the operation $removeAny$ is specified instead, which removes and returns any element from the set.

`RemoveAnyIndex(p,i)`	Removes and returns any index from the pool
`AddIndex(p,i)`	Adds index i back to pool p

While an object pool has multiset semantics and may contain duplicates, an index pool corresponds to the *set* data type as its elements are unique: An element can only be freed if it has been acquired before. Correspondingly, an index can only be added back if it is not contained in the pool.

```
bool Lookup(const string & key, list<const T &>);
bool Insert(const string & key, const T & value);
```

Listing 2.3: Interface of the map data structure

Sets and maps share identical semantics when considering a map as a set of objects $(key \mapsto obj)$. . In case of a map, the user defines values of key and value, while on a pool, index values are set by the data structure.

```
index_t removeAny() {
  bool expected;
  for (index_t i = 0; i < capacity; ++i)
    {
    expected = false;
    if (CAS(reservationFlags[i], expected
    , true))
      return i;
    }
  }
}
```

```
void add(index_t index) {
  elements[i]         =
    element;
  reservationFlags[i] =
    false;
}
```

Figure 2.6: Implementation of acquisition and release in the array-based pool

2.4.2 Array-based wait-free pools

In a basic approach, the index pool stores its elements in an array, along with a second array of the same size containing atomic boolean objects. These can be modified using *CAS* and represent reservation flags. When an element is acquired from the pool, any reservation flag is atomically swapped from $false$ to $true$, and the element at the same index in the element array is returned. The flag is reset when this element has been added back to the pool.

It is an important detail to ensure that cache-aligned allocation is used for the array to prevent false sharing, which would harm performance.

The crucial problem is selecting an available reservation flag, and the search mechanism used. A possible wait-free solution is a linear search starting at the beginning of the reservation flag array.

On high allocation levels, the $O(n)$ complexity of the search is drastically impairing performance. Pools may have large capacity, so n is potentially big. Every acquired pool element increases the duration of every following acquisition, which is especially problematic when large coherent ranges in the pool are held for long-runnig tasks.

All of the few practicable wait-free memory management strategies implicitly involve a mechanism semantically equivalent to a wait-free index-pool. This also applies to solutions that are substantially different from from hazard pointers, such as reference counting algorithms [Sun05] [SKKE11].

Many straight-forward optimizations of this strategy are known for lock-free implementations [MS07]. A solution for wait-free dynamic storage allocation using a tree as index structure for logarithmic search has been proposed by Stellwag and Krainz who sketch a helping pattern functioning without a subjacent pool [SKSP]. The mechanism is not suitable as it requires a machine instruction that is not supported by any known architecture, though. Also, the authors explain it as not linearizable. We use a lock-free predecessor of the tree-based pool algorithm as a reference in evaluation.

Currently, no suitable wait-free solution for pools has been published that improves on the array-based approach. Especially when real-time constraints are taken into account, all approaches in related work are found inadequate. Improvements on the array-based pool aiming for reducing search complexity do not seem promising and feasible as of this writing.

However, indexed search is not the only thinkable solution: If the average amount of failed reservation attempts until successful acquisition is minimized by any means, worst case latency improves as well. Two approaches are derived from this assumption:

Diffuse search Linear, sequential search in the array is replaced by pseudo-randomized search patterns. Hypothetically, non-linear scans prevent repeated reservation attempts on coherent ranges of reserved cells.

Redundancy Implying a tradeoff at the expense of memory overhead, additional index elements are held available for every thread, similar to thread-specific storage.

2.4.3 Thread-dependent search patterns

In non-linear search, every thread traverses the reservation array in a different pattern. Here, an iteration function depending on the thread ID t is to be defined. As an example, the iteration could skip $t + 1$ indices, so thread 0 traverses reservation flags as before, thread 1 would test every second reservation flag, and so on. The iteration patterns could be further optimized to minimize conflicting accesses probabilistically.

This mechanism seems simple and apparently reduces the size of ranges that only consist of reserved cells. The altered search pattern evidently does not introduce situations where an operation could be delayed arbitrarily. When verifying for correctness, however, the algorithm is found to conflict with linearizabity.

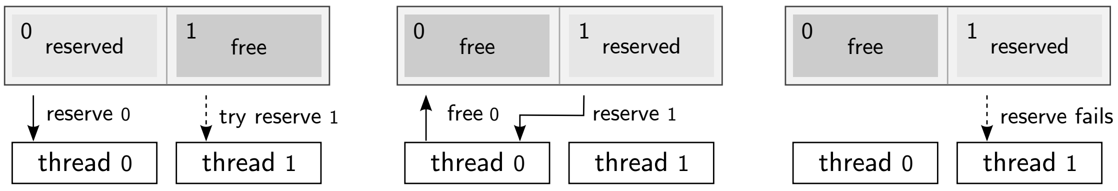

Figure 2.7: Structure of the array-based pool with thread-specific compartments

An example using two threads and two cells illustrates this conflict when threads iterate using different search patterns. As shown in Figure 2.7, both threads try to reserve and release one of two available cells. There is a possible schedule where thread 0 reserves cell 0, and thread 1 starts its search at cell 1 and is about to reserve it, but is preempted. Thread 0 continues to release cell 0 and reserve cell 1. When thread 1 is activated, it fails to reserve cell 1.

This pattern can be applied in reverse when thread 1 should try to reserve cell 0 after his failed reservation attempt. In any sequential history, both threads would always succeed in reserving a cell. Consequently, a reservation algorithm based on different search patterns is not linearizable.

This leaves the second approach, where more cells are held available for every thread.

2.4.4 A compartment-based index pool

The following modification on the array-based pool does not reduce complexity of scans in acquisition of elements, but at least improves performance for some use cases.

Expensive contentions on reservation flags can be avoided when definining ranges in the pool that are allocated by a dedicated thread exclusively. The pool is thus divided into a public range that is identical to the conventional array-based pool, and thread-specific *compartments*. As opposed to thread-local storage, elements in compartments are also accessible to all other threads, but can only be acquired by the compartment's owner. The total size of the pool's private range is proportional to its capacity and defined by a factor k.

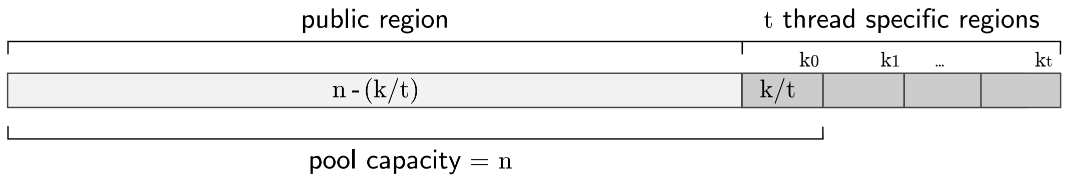

Figure 2.8: Structure of the array-based pool with thread-specific compartments

This simple addition eliminates competition for pool elements up to a specific number of allocated elements per thread, depending on the pool's capacity and its partitioning defined by k. The layout of a *compartment pool* is illustrated in Figure 2.8. Evidently, introducting compartments increases the memory overhead of the array-based pool. Its capacity of n elements must be entirely available not to all threads in total, but to every single thread. For t threads and n_k elements in every compartment, the actual number of elements held in the pool increases by $n_k \cdot (t-1)$. If all elements in compartments have been acquired, the actual pool capacity of $n + n_k \cdot (t-1)$ is made available.

2.4.5 Verification

For the array-based pool implementations, we found no apparent hazard situation. Its mechanism is so simple, however, that violation of correctness conditions are easily detectable in straight-forward verification setups. As an example, disproving linearizability of thread-specific search patterns in the array-based pool only required two threads and two elements capacity. The compartment-based implementation was verified for different partitionings of public and thread-specific ranges.

The following schedulings of enqueue (E) and dequeue (D) operations have been verified for all pool implementations using DiVine:

```
(E||D);E;E;D;D
E;(D||(E;D;D;E))
E;E;(D||(D;E);D)
```

Similar setups have also been used for verification of memory barriers in the Michael-Scott queue in related work [vdB13].

2.4.6 Benchmarks

A common-place scenario for allocation strategies is **enqueue-dequeue pairs** (e.g. [SKSP]), where threads each allocate and immediately deallocate memory with a fixed size for a given number of iterations. The corresponding use case is a highly parallelizable task distributed to threads with short modification cycles. This applies to some image processing algorithms.

Finally, **reader-** and **writer-intensive race** are commonly used to evaluate memory allocation efficiency under imbalanced work loads for reading and writing (e.g. [SKKE11]).

In the **fill-up** test, threads allocate single memory blocks in a loop until no more memory is available. Results from fill-up tests did not lead to new findings, as the scenario is implicitly covered by enqueue-dequeue pairs with preallocation.

Custom scenarios

In order to provide results that are meaningful for realistic applications in particular, benchmarks and setup variations have been designed to measure performance in situations that are known as challenging in this domain.

In real-time applications, long-running procedures typically allocate resources at startup in bulk instead of smaller chunks on demand at run time. The scenario **enqueue-dequeue bulk** is designed to reflect this use case: Threads acquire and release a large amount of pool elements in parallel as opposed to the enqueue-dequeue pairs scenario where a single element is allocated and released per iteration. For comparable results, the total amount of operations is to be constant in every execution, so the number of allocations per iteration is $i_a = n / i$. The impact of allocation bulk size on throughput and latency of the periodical operations is evaluated. Supposedly, effects of auxiliary index structures in the pool implementation are more noticeable when removing and adding elements in bulk.

Allocators also tend to decrease in performance when their managed resources run low. We introduce an additional benchmark parameter **pre-allocation ratio**, which sets the percentage of the pool capacity that is allocated before executing a benchmark. This modification has no effect in the fill-up scenario, but should yield insightful results when running enqueue-dequeue pairs with gradually increasing allocation level.

For benchmarks on pools, the reader- and writer-intensive race scenarios are adapted and combined into a single scenario *allocator/deallocator race*. Here, n_P allocator threads run in parallel to n_C deallocator threads, each acquiring or releasing i_a elements respectively. The number of active threads in total $n_T = n_P + n_C$ is constant, starting with a single allocator thread ($n_P = 1$). The allocation / deallocation balance is gradually shifted from release-intensive to acquisition-intensive in every execution, until a single deallocator thread runs in parallel to $n_T - 1$ allocator threads.

As the total number of operations is constant, we expect that measurements demonstrate how latency and throughput depend on aquisition / release balance.

Pool elements must be acquired first before they can be released, so a deallocator thread reserves i_a elements in the initialization phase, and adds them back to the pool in execution. As an unintended side effect, preallocation ratio then varies depending on allocator / deallocator balance, as the total number of preallocated elements ranges from $i_a \cdot (n_P = 1)$ to $i_a \cdot (n_P = n_T - 1)$. To take preallocation out of the equation, deallocator threads each acquire $(n_T - 1) \cdot i_a / n_P$ elements in initialization phase in every exection, so the total amount of preallocated elements is constant.

2.4.7 Evaluation

In every scenario, standard measurements as described in section 2.3.2 are recorded for every operation call on the data structure. For throughput measurements, the experiments with their test parameters are:

test	n_T **threads**	**capacity** c	**pre-allocation** r	i **iterations**	i_a **alloc /** i
ED-P	2-32	100 k	0 - 0.9	2500	190
ED-B	2-32	$i_a \cdot n_T$	0	100 k / i_a	1 - 1000
ADR	1:32-32:1	320 k	$32 \cdot i_a = 0.5$	1	5000

Results from the original enqueue-dequeue-pairs scenario are summarized in Figure 2.9. The array-based algorithm shows superior throughput compared to the lock-free implementation when no elements are pre-allocated. This is explicable by their reservation mechanism: Flags in the reservation array are reset to their original state when elements are immediately released.

For i_a elements allocated and released in each iteration of t threads, at most $t \cdot i_a$ reservation flags have to be traversed for removing an element from the pool. This upper bound is very unlikely to be reached, the actual amount of failed reservation attempts is lower by orders of magnitude. As threads only operate on the head of the array, the scan complexity of $O(n)$ does not come to effect. Repeapting the scenario with an increasing number of preallocated elements eliminates this bias. As expected, the amount of removed elements has no effect on performance of the lock-free pool, while throughput of the wait-free implementation gradually decreases by orders of magnitude.

In the same plot, performance of the array-based pool with no preallocated elements initially increases proportional to the number of cores. Throughput gradually drops after reaching a sweet-spot at 12 threads. Gunther's law explains this phenomenon: The lookup-time in the array-pool's scan routine directly relate to the number of failed reservation attempts. As more threads compete for the same atomic objects, memory contention increases proportionally.

Speedup of the variants depending on the degree of parallelism has substantially different characteristics for both implementations. Most notably, an asymptotic limit for throughput is observed for the tree-based pool. We assume that this is an effect of contention rate as explained earlier, as cache miss rate increases with the number of threads operating on the pool.

Related to the Universal Scalability Law, we define a general term for this choke point effect for later reference in comparable situations:

Definition 2.4.1 (Central point of conflict)**.** An atomic variable that is affected in any invocation of a method f and thus causes a lower bound for contention penalty in this method is a single point of conflict in f. An atomic variable s is a *global* point of conflict if all methods that change the state of a data structure have s as a common central point of conflict.

A data structure with n_s central points of conflict can guarantee to complete at most n_s operations in parallel. By this definition, the atomic counter in the root node of the lock-free pool's tree index is a global point of conflict. It is easily argued that in a tree where atomic variables manage access to child nodes, every node is a local point of conflict for operations to its descendants. From a probabilistic view, a node on level l in such trees with branching factor d contributes to $1/(d \cdot l)$ of overall contention penalty.

In the tree pool implementation used for evaluation, a threads decides on which branch to descend with a deterministic left-first strategy, so leaf elements are reserved from left to right. A thread-specific descending pattern could distribute accesses to all children of any node, and hereby reduce contentions by a linear rate proportional to the branching factor d. Analogical to the optimization approach previously described for array-based pools, this approach would have to be verified for linearizability.

Plots in Figure 2.12 represent measured throughput of the tested pools in the allocator/deallocator race scenario. The lock-free implementation descends and modifies a tree index in allocation as well as deallocation. Adding an element back to the array pool has constant complexity and is embarrassingly parallel, so it never causes contention. Operation `Add` is evidently entirely parallelizable, and throughput measured depending on the number of deallocators grows with the expected speedup rate n_p.

In conclusion, thread-specific compartments can account for drastic improvements of both latency and throughput, even if they span just a small fraction of an array pool's capacity. This improvement does not hold for the general case and has no effect on maximum latency. The compartment scheme cannot be argued as an ideal conclusive solution, of course. Then again, its performance is nearly impossible to beat in use cases where threads acquire and release few elements with high frequency, especially when memory overhead is not a concern.

The fact that the simple semantics of an index pool cannot be trivially transformed to a more efficient, wait-free, and linearizable algorithm is counter-intuitive to say the least. The complexity of such a transformation does not relate to semantics but from additional restrictions that prevent the use of otherwise common patterns like hazard pointers and counters.

2.5 Queues

Concurrent queues, containers with First-In-First-Out (FIFO) semantics, are one of the most fundamental and widely studied concurrent data structures in literature. As a fundamental building block of inter-process and inter-thread communication, queue algorithms are essential components in multi core applications in general.

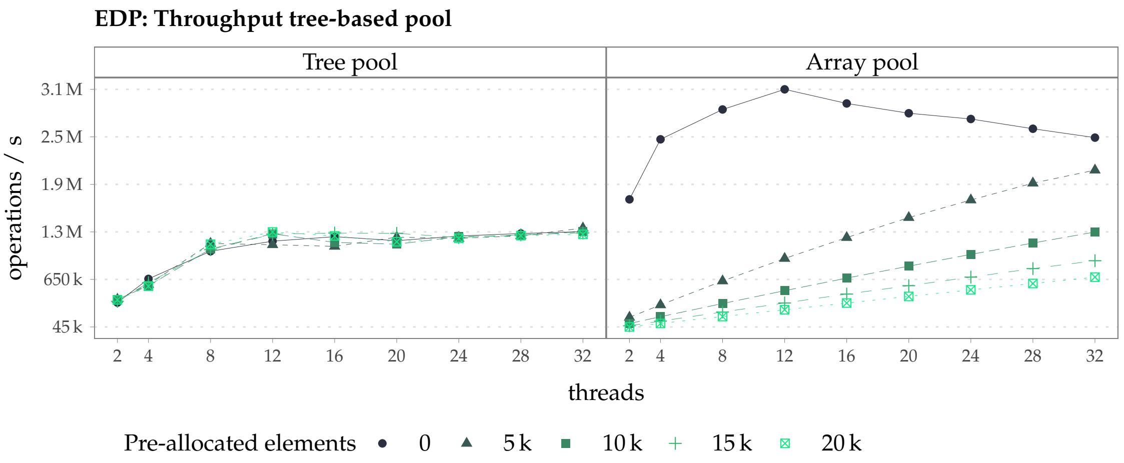

Figure 2.9: Throughput of wait-free array-based pool and lock-free tree-based pool in scenario *enqueue/dequeue-pairs*, by preallocation ratio

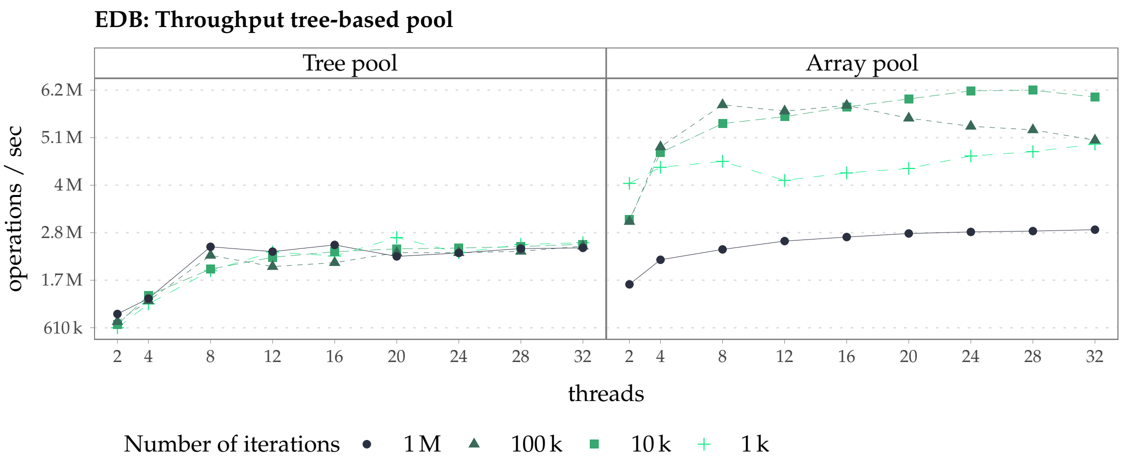

Figure 2.10: Throughput of wait-free array-based pool and lock-free tree-based pool in scenario *enqueue/dequeue-bulk*, by number of iterations

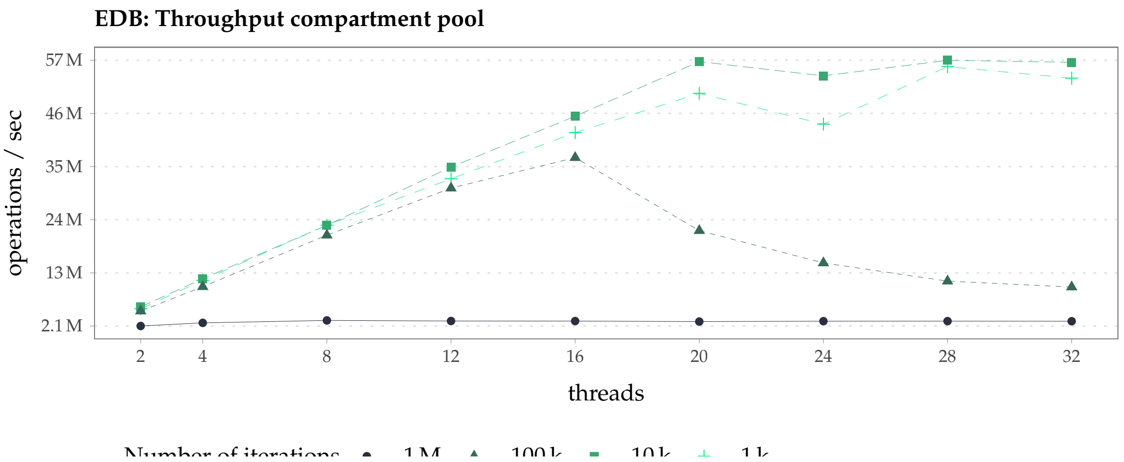

Figure 2.11: Throughput of wait-free array-based compartment pool in scenario *enqueue/dequeue-bulk* with $K = 5$, by number of iterations

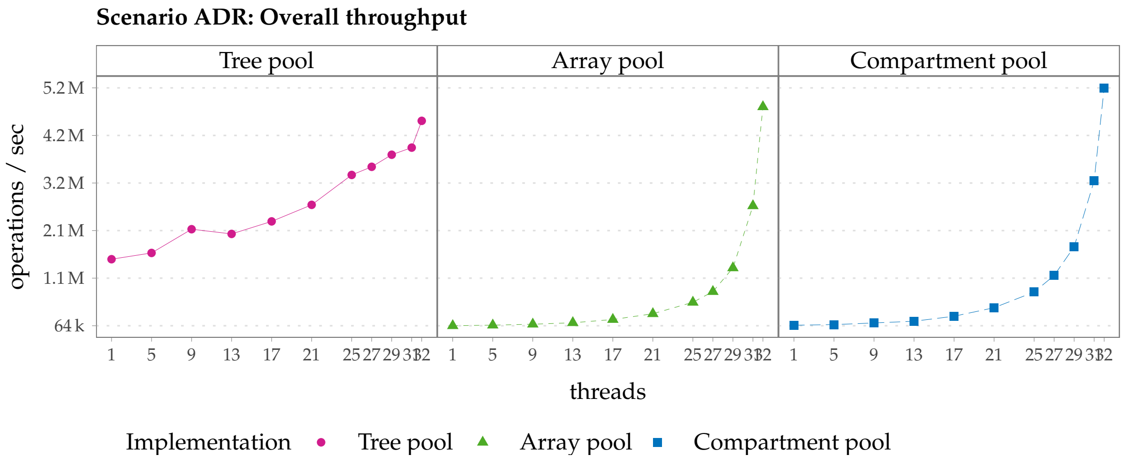

Figure 2.12: Throughput of wait-free array-based pool and lock-free tree-based pool in scenario *allocator/deallocator race*, for increasing number of producers and decreasing number of consumers, by number of producers

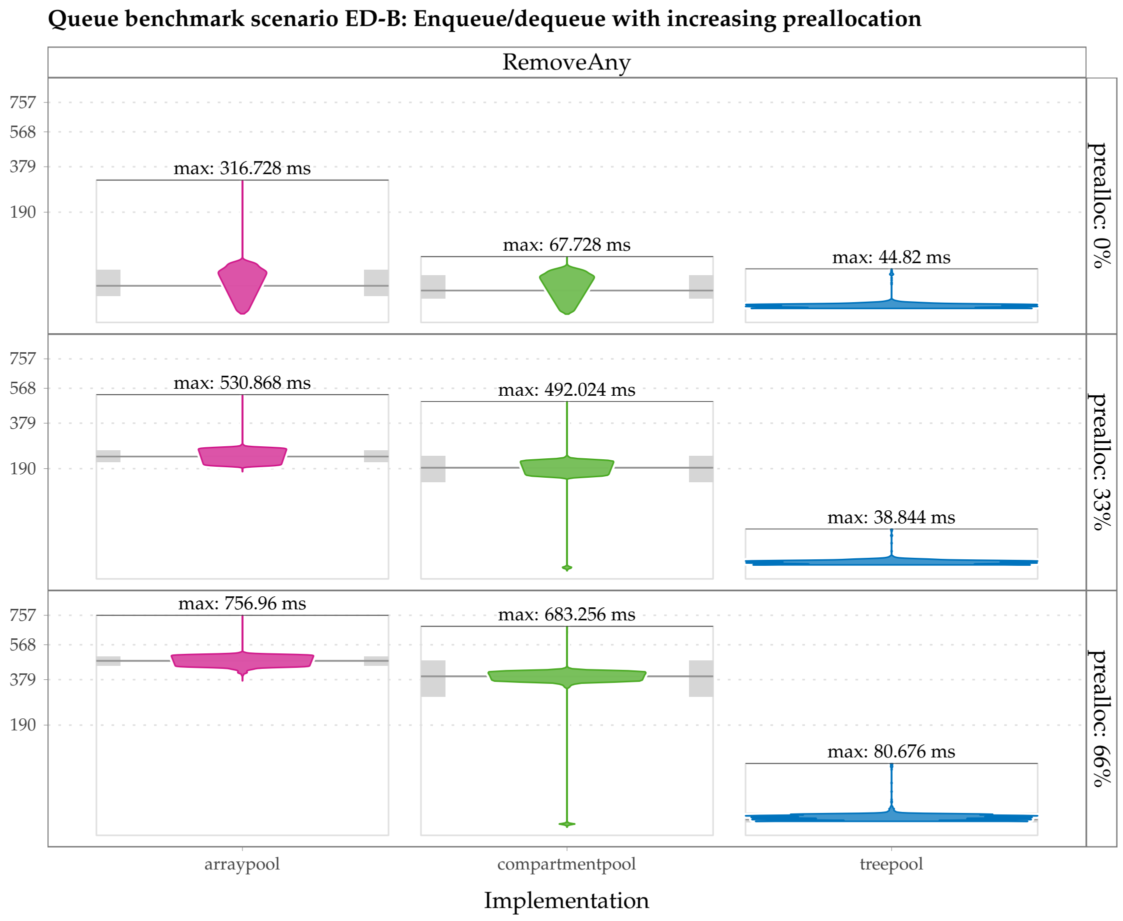

Figure 2.13: Observed latencies in pool benchmark scenario *enqueue/dequeue-pairs* with preallocation ratio from 0-66%, pool capacity of 300000 objects, 1000 iterations

2.5.1 Definitions

The sequential queue data type is well known and semantics need little clarification. It provides the following operations:

`Dequeue(q, e, s))`	add element *e* to queue *q*. Returns a boolean value *s* indicating whether the operation was successful.
`Enqueue(q, (e,s))`	remove an element from queue *q* and assign its value to *e*. Returns a tuple of *e* and a boolean value *s* indicating whether the operation was successful.

A concurrent queue must be linearizable to a sequential queue with FIFO semantics [MS07]. Linearizability permits arbitrary execution order of overlapping operations [Her, p. 57] as their linearization points may occur anywhere between their invocation and response. In effect, FIFO element order is not necessarily maintained within small intervals or even impossible to argue for high contention.

2.5.2 Adapting the Kogan-Petrank wait-free queue for embedded systems

A practicable wait-free queue for an arbitrary number of enqueuers and dequeuers has been presented by Kogan and Petrank based upon the helping pattern [KP11]. Their algorithm is a construction from the Michael-Scott queue, which is represented as a single-linked list.

Mechanisms in the Kogan-Petrank queue closely resemble the wait-free simulation technique presented recently by the same authors. Enqueue- and dequeue operations are transformed into three subsequent sub-steps: A preparation method announcing the operation, a method that applying an operation to the queue that also contains its linearization point, and a final cleanup procedure passing the operation's result to its owner. The full implementation is listed in Appendix B of this work.

The pseudo code in the original publication could be directly reimplemented in C++, but some aspects of the data structure's design do not comply with restrictions of real-time and embedded software:

Dynamic memory management Underlying containers are allocated dynamically and have to be replaced by pools with limited capacity.

Garbage collection The published design originates from an implementation in Java and therefore requires automatic memory management.

Atomic instructions Compare-And-Swap is used on operation description objects which are 64 bit wide. This instruction is not universally supported.

Monotonic counter To help operations in FIFO order, operation descriptions contain a monotonic phase counter that is prone to overflow.

As an alternative to garbage collection, Kogan and Petrank suggest hazard pointers. They argue that wait-freedom of the queue algorithm was maintained as the safe memory reclamation scheme is wait-free. This argument is incomplete: While the hazard pointer algorithm achieves wait-freedom, aperations establishing a hazard pointer guard is a lock-free routine, as explained in subsection 2.2.3 of this work.

Dynamic memory allocation can be easily replaced by pools in principle, but a maximum capacity must be known for every pool. With hazard pointers guarding queue nodes, some amount of pool elements is retained as retired nodes. The element pool's capacity must be extended by the maximum number of retired nodes, which is unknown.

Eventually, we successfully in ported the Kogan-Petrank queue to an implementation in C++ that respects all mentioned restrictions and meets the following requirements:

C++03
The queue data structure must be implemented C/C++ following the C++03 language standard, and must not rely on architecture-specific conditions.

Memory reclamation
The data structure and its memory reclamation mechanisms must be wait-free.

Storage allocation
Memory must only be acquired via automatic or static allocation within object initialization, i. e. in the constructor.

Counters
An alternative for the phase counter has to be found that does not utilitze a monotonic counter.

This required modifications on the original algorithms which are discussed in detail in the following.

Integration of hazard pointers in the Kogan-Petrank queue

Michael describes the following state of an allocatable node that can guarded by a hazard pointer:

Allocated	Acquired, but not inserted into a data structure
Reachable	Reachable by following links from the root of a data structure
Removed	Not reachable, but still being accessed by the removing thread
Retired	Flagged as retired by the removing thread
Free	The node's memory is available for allocation.
Unavailable	The node's memory is used by an unrelated object.

The `Free` and `Unavailable` states must be published to all threads when they engage in concurrent operations of a guarded node. Most relevant for ARM architectures, reordering of instructions must be taken into account whenever a hazard pointer guard is established.

Accesses to atomics, such as operation description objects, implicitly flush the cache line they are loaded into, which conveniently prevents false sharing. Our implementation builds upon atomic primitives from the EMBB library which automatically perform a full fence before and after read-modify-write operations. Still, specific coding conventions must be followed to prevent reordering.

Consider the code snippet in Figure 2.14. in line 9 depend on states of `head` and `next`, and appear to be safe: According to the source code, guards on both variables are in effect before changing the object's state. The atomic read on `next` was put before the loop body to cache its value, and it will only be guarded if the state of `head` is unchanged to avoid an unnecessary guard. This paradigm is common practice with the intent to minimize active guards and avoid expensive atomic instructions within a loop. As a

```
 1 //   ...
 2     aptr head = Head.Load();
 3     aptr next = head->Next.Load();
 4     while(cond) {
 5       hp.GuardPointer(0, head);
 6       if (head == Head.Load()) {
 7 // -- mfence --------------------
 8         hp.GuardPointer(1, next);
 9         ModifyState(head, next);
10         hp.ReleaseGuard(1);
11       }
12       else {
13         hp.ReleaseGuard(0);
14       }
15     }
16 //   ...
```

```
 1 //   ...
 2     aptr head = Head.Load();
 3     aptr next = head->Next.Load();
 4     while(cond) {
 5       hp.GuardPointer(0, head);
 6       hp.GuardPointer(1, next);
 7       if (head == Head.Load()) {
 8 // -- mfence --------------------
 9         ModifyState(head, next);
10         hp.ReleaseGuard(1);
11       }
12       else {
13         hp.ReleaseGuard(0);
14       }
15     }
16 //   ...
```

Figure 2.14: Use cases of hazard pointers and effect of atomic operations on reordering

```
//   ...
    pointer p = node.Next;
    // -- possible change of node.Next --
    while(hp.GuardPointer(p) && p != node.Next) {
        hp.ReleaseGuard(p);
    }
//   ...
```

Figure 2.15: A retry-loop guarding a reference using hazard pointers

consequence, there is no memory barrier between guarding and passing next, though. Any reordering of operations within the loop body could result in an access hazard. This can be easily corrected by a small change: The guard operation on next is moved before reading from head, so the implicit memory barrier in line 7 ensures that the guard state is published to all threads.

The Hazard Pointers technique introduces an inconvenient problem when used in wait-free algorithms in particular: As a pointer value can be changed by another thread before the its guard is taking effect, it has to be re-checked after calling Guard. In the examples given by Michael, this is implemented by retrying in an infinite loop as illustrated in Figure 2.15, a solution that conflicts with wait-free requirements.

For wait-free algorithms, additional effort is needed to guarantee guarding will only fail for a limited number of times. For integration in the Kogan-Petrank queue, retry loops are avoided by taking advantage of its proven linearizability.

An obvious ABA-hazard are pointers to queue nodes in operation descriptions. Nodes are only deallocated in dequeue operations, and guarding the pointer to the element to be dequeued is sufficient in this case. After the guard is established, a CAS on the operation description of the current dequeue is performed. There is no need to check whether the node pointer has changed:

- If the CAS succeeds, no other thread can be engaged in any operation affecting this element. Consequently, no other thread could have changed the pointer before its guard was in effect, so the guard is valid.

- If it fails, the operation description has changed. This, however, strictly implies that another helper thread completed the dequeue first. The guarded pointer value is invalid, but also became irrelevant: The operation that lost the CAS is cancelled and releases the guard.

Redesign of operation prioritization in the Kogan-Petrank queue

Kogan and Petrank mention the possibility of an overflow of the phase value, as it is incremented monotonically. An overflow of the phase would not lead to an inconsistent state of the queue, but could lead to arbitrary delay of an operation, given very specific circumstances.

In Kogan and Petranks pseudo code, also phase values of inactive operation descriptions are considered when obtaining the maximum phase. This is surprising, as completed operations cannot affect the state of the queue. The original work gives no explanation for this algorithmic detail. Presumably, the intention is to increment phase values monotonically. A specific counter value will not reocurr in any operation that is announced in a later phase when following the pseudo code in the publication.

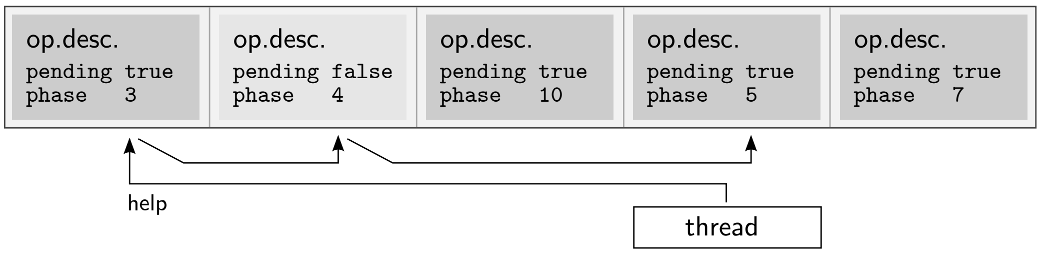

Figure 2.16: Order of helping operations by phase in the Kogan-Petrank queue

A monotonically incremented phase counter provides an obvious guarantee for correct prioritization. On the other hand, considering that ABA conditions might arise from overflow, it is highly undesirable. Kogan and Petrank mention the possibility of an overflow of the phase value, but consider it unlikely for a wide integer type.

When modifying the algorithm so phase values of completed operations are ignored, the phase count might fall back to any prior value. If no prior operation is in pending state, a new operation is initiated with phase value starting over at 1. In experiments, this reduced the maximum phase value used in long-running scenarios by several orders of magnitude, rendering overflow of the phase counter less likely. However, it must be ensured that this modification does not harm correctness.

To give a sketch for a proof of correctness of the Kogan-Petrank queue using the modified phase counter:

1. In the queue's initial state, prioritization is irrelevant for the first announced operation.
2. Assuming any amount of operations on the Kogan-Petrank queue have been completed so all operation descriptions are in non-pending state.

3. In this case, the queue's state is indiscernable from its initial state and prioritization is irrelevant.
4. Otherwise, a phase value will only be reused if and only if all pending operation have a lower phase.
5. Consequently, if a phase value is reused for an operation, all operations will be helped before executing this operation.
6. In conclusion, prioritization is intact.

This intuition has been confirmed by verification using DiVine, but even when allowing the counter to fall back, overflow is still possible.

It is thinkable to use a kind of windowed arithmetic ("bounded counter") for phase numbers in order to employ a comparison operator that is robust against overflow. Finally, we found that prioritization also can be achieved with strategies that replace the phase entirely.

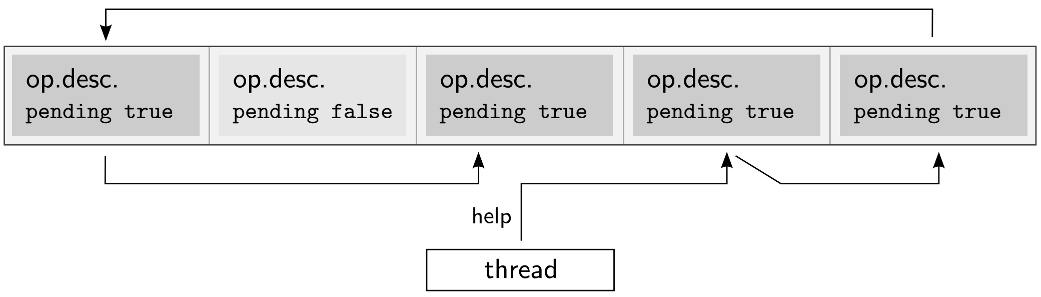

Figure 2.17: Order of helping operations without phase in the modified Kogan-Petrank queue

In the final implementation, the phase scheme is removed, which also eliminates the need to resolve its maximum for every enqueue and dequeue operation. Instead, every thread iterates pending operations in the global array starting at the index succeeding its own, until it reaches its own operation. Consequently, every other thread is helped first, disregarding the age of its operation announcement. The C++ implementation of the modified help procedure is shown in Listing 2.4 and illustrated in Figure 2.17.

```
      if (!desc.Pending) continue;
      if (desc.Enqueue)
        HelpEnqueue(tId % numThreads);
      else
        HelpDequeue(tId % numThreads);
    }
  }
  void DeleteNodeCallback(index_t releasedNodeIndex) {
    nodeIndexPool.Add(releasedNodeIndex);
  }
};
```

Listing 2.4: Implementation of the modified help procedure in the wait-free Kogan-Petrank without phase counter

A different condition for prioritization is achieved: When a thread t_0 traverses operations its helping loop, it possibly helps operations first that have been announced after its own. But likewise, any helped thread t_i is guaranteed to first complete the operation announced by t_0 before their own, so prioritization is intact even for high contention.

Wait-freedom and linearizability are maintained. The probability for an operation to be helped monotonically increases while at least one thread is engaging in an operation.

This principle is not restricted to the Kogan-Petrank queue. It applies to all wait-free data structures that employ the operation state helping scheme, including any algorithm that follows the wait-free simulation technique.

Defining an upper bound for memory overhead of hazard pointers

In embedded system application development, dynamic memory allocation at run-time is forbidden as it introduces non-determinism. Data structure are therefore allocated statically, based on their worst-case memory consumption. Therefore, upper bounds for memory consumption are required for every data structure, possibly depending on their initialization arguments like container capacity, or any other parameter known at boot-time.

For the Hazard Pointers technique, memory is allocated for the global array of hazard pointers and the thread-local lists of retired nodes; the latter pose a problem.

The mechanisms behind the hazard pointers scheme need to be explained in more detail for a solution: Whenever a thread's retired list exceeds a threshold, it performs the scan routine to free retired nodes that have no associated active guard in the global hazard pointer list. The maximum amount of all nodes in the threads' retired lists must be held available for allocation in addition to the queues capacity, as they are not yet freed, but also unused.

Upper bounds for the number of nodes in retired lists contribute to the memory footprint of any data structure employing hazard pointers, and include the maximum amount of nodes at any time that are *eligible* for reuse, but not deallocated yet. Michael presents a maximum of the number of retired nodes that are *not eligible* for reuse (guarded), and a lower bound for nodes freed in a deallocation scan. The following definitions are given:

$$\begin{aligned} &K && hazardpointersperthread \\ &N = K \cdot T && hazardpointersintotal \\ &R = \Omega(N) = 1.25 \cdot K \cdot T && \text{threshold} \end{aligned} \tag{2.6}$$

with N hazard pointers in T threads. The threshold R defines the size of the local retired list that triggers the `Scan` procedure. The lower-bound function $\Omega(x)$ is usually defined as $x \cdot 1.25$ in implementations.

From the definition of the hazard pointers' `Scan` routine, we derive the following upper memory bounds:

$$UB_C = C \cdot \text{size}(E) \qquad \text{container capacity} \quad (2.7)$$

$$UB_{RG} = K \cdot T \cdot \text{size}(HPRecord) \qquad \text{max. guarded nodes in a retired list} \quad (2.8)$$

$$UB_{RF} = R \cdot \text{size}(HPRecord) - UB_{RG} \quad (2.9)$$

$$\text{retired nodes eligible for reuse} \quad (2.10)$$

$$UB = T^2 \cdot 1.25 \cdot K + UB_C \quad (2.11)$$

$$\text{hazard pointers in total} \quad (2.12)$$

With UB as an upper bound for memory consumption overhead in a data structure with capacity C that require at most one guard for every element of type E, such as the Kogan-Petrank queue.

Finally, UB is the maximum number of nodes that must be available for allocation and thus the minimum element pool capacity in the modified wait-free Kogan-Petrank queue.

In general, a maximum amount of guarded nodes for any operation has to be known to employ hazard pointers. Michael demonstrated hazard pointers in lock- free implementations of a list and a stack in his original publication on hazard pointers [Mic04a]. In these cases, at most two hazard pointer per thread is sufficient, as no operation needs to guard more than two node indexes at any time. For some implementations of e.g. graphs and trees, a maximum amount of guarded nodes per operation does not inherently exist, though.

2.5.3 Benchmarks

Publications on queue algorithms present results from the same common benchmark scenarios for decades. The **enqueue/dequeue pairs** scenario as described for pools in the last section has originally been defined for queue data structures, but applies to set semantics in general.

The most prevalent metric in evaluation of queues derives from FIFO order: when used for buffered communication between threads, how long will it take to transfer a specific amount of elements?

In the **buffer** benchmark scenario, producer threads add a set of elements to a shared queue. In parallel, consumer threads continuously try to dequeue elements until all elements have been transferred to consumers. The scenario corresponds to common real-world use cases where computation is distributed to parallel tasks yielding partial results.

Buffer performance is then evaluated for overall throughput as the amount of elements transferred per second, and buffer latency, the amount of time elements spent in the buffer.

Also, some publications on queue and stack data structures mention a **random 50/50** scenario. Threads initially acquire some elements and randomly either release or acquire a single element in a loop; performance is measured in operations per second.

Custom scenarios

FIFO buffer latency highly depends from the buffer's fill rate. For an empty queue, a single element can be enqueued and dequeued immediately, yielding minimum buffer latency. Results in parallel scenarios may depend from scheduling, however. Latency increases significantly if producer threads are executed predominantly, even if this inbalance only occurs once and for a limited time span in the entire benchmark run: Once a large amount of elements has been enqueued without interleaving dequeue operations, fill rate increases and measured latency of all following elements is raised by a near-constant time interval until the fill rate drops back to 0 again.

Related work does not consider influences from scheduling, presumably because fair scheduling is assumed and evaluation is based on averaged measurements. For a reasonably high number of iterations, measurements will level out in mean and median.

Fluctuation of maximum latency due to slight differences in scheduling cannot be eliminated this way, and even worsens for higher numbers of iterations. Scenarios have to be specifically designed for measuring maximum latency, but fortunately, a slight modification to the buffer scenario is sufficient: At the end of the iteration loop of a producer threads, it is put to sleep for a short period of time. The producers' *think time* allows consumer threads to reduce fill rate, and imitates real-world use cases even better: Producer threads spend time on computing a result before it is put in a buffer.

In our implementation of the buffer benchmark, producers enqueue a series of indexed queue elements which contain their respective enqueue timestamp. The same amount of consumers threads continously try to remove an element from the queue and measure *buffer latency* as the difference of an element's timestamp and the point in time it has been removed from the queue.

The time measured from enqueueing the first element until dequeueing the last element is overall *throughput*, represented as elements per second.

2.5.4 Evaluation

test	n_T **threads**	**capacity** c	**pre-allocation** r	i **iterations**	i_a **allocs /** i
ED-P	2-32	100 k	0 - 0.9	2500	190
ADR	1:32 - 32:1	320 k	$32 \cdot i_a = 0.5$	1	5000
BUF	2:2 - 16:16	$n_P \cdot i_a$	0	10 k	8

The evaluated queue implementations allocate nodes from a pool, so performance of allocation is supposed to affect execution time especially of enqueue operations. We add two test candidates `koganpetrank-pl-tp` and `michaelscott-ap`, a Kogan-Petrank queue allocating nodes from a lock-free tree-based pool, and a Michael-Scott queue with an instance of the wait-free array pool. This allows to compare performance of the queue algorithms on eye level, without implicitly benchmarking their allocation mechanism. Furthermore, the impact of pool algorithms on the performance of data structures using them becomes apparent.

The Michael-Scott queue has two central points of conflict according to the previous definition in Figure 2.4.7 for enqueue- and dequeue operations if a queue instance holds more than one element. As a consequence, performance of either operation should not be affected by the other. In principle, this also applies to the Kogan-Petrank queue, as it is a construction of the Michael-Scott queue. But the helping pattern makes a difference in the wait-free implementation: Any enqueue operation possibly helps pending dequeue operations first, and vice versa. Consequently, both methods access their central points of conflict mutually which might affect contention under high load.

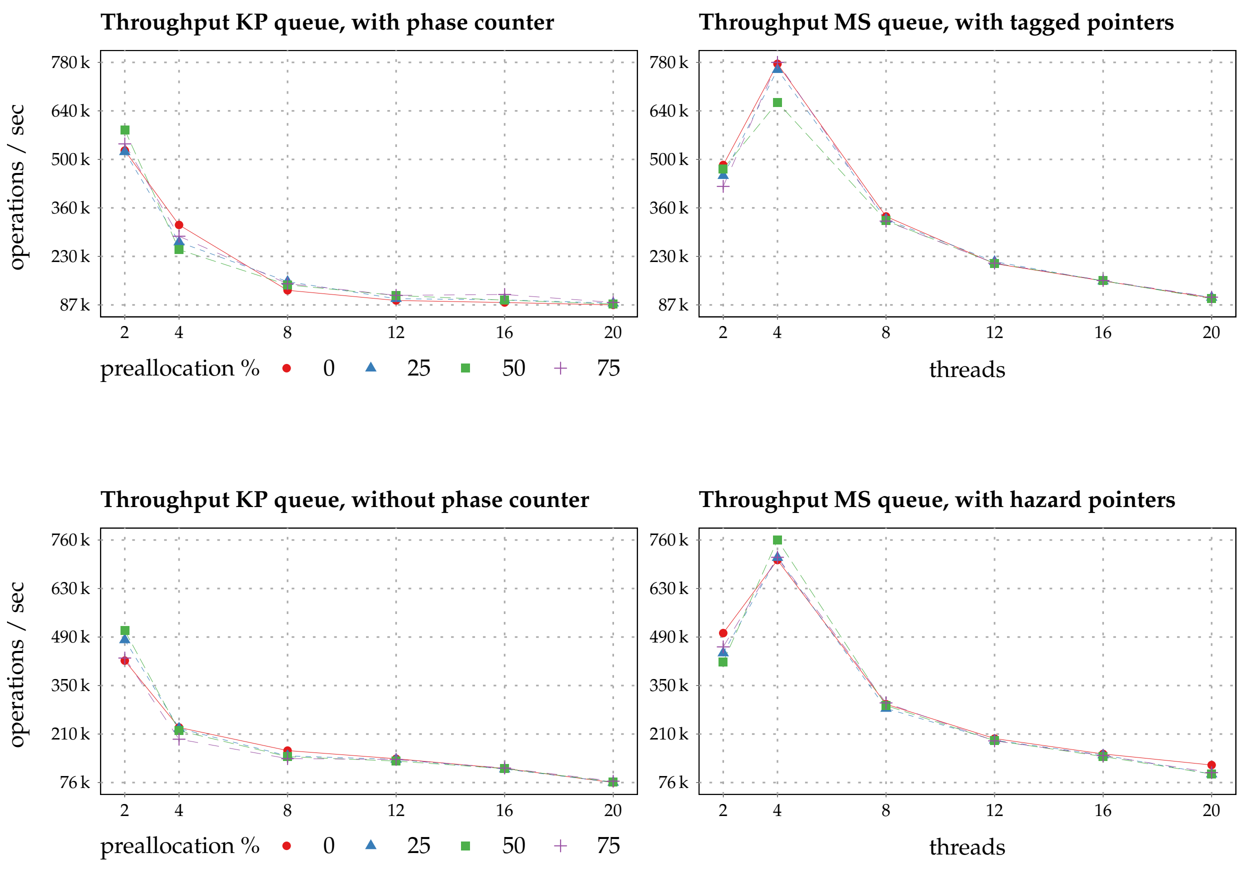

Figure 2.18: Throughput of the modified Kogan-Petrank queue without phase, and modified Michael-Scott queue with hazard pointers, by preallocation

The principle of the *allocator/deallocator race* scenario seems ideal to put this intuition to the test. Unfortunately, it is impracticable for test of queues using the array-based pool. As described in the previous section, the test setup requires to pre-allocate about half of the pool capacity. To little surprise, execution time of allocation occluded the performance of the actual queue operations in experiments on queues that use the array-based pool. Therefore, we use tree-based pools for both queue instances in this scenario. This modification gives meaningful measurements displayed in Figure 2.20, but memory management and consequently all operations in the Kogan-Petrank queue are formally not wait-free.

As an alternative, the *buffer* scenario allows to evaluate for different numbers of enqueuers and dequeuers, too. Surprisingly, all queue implementations are en par when

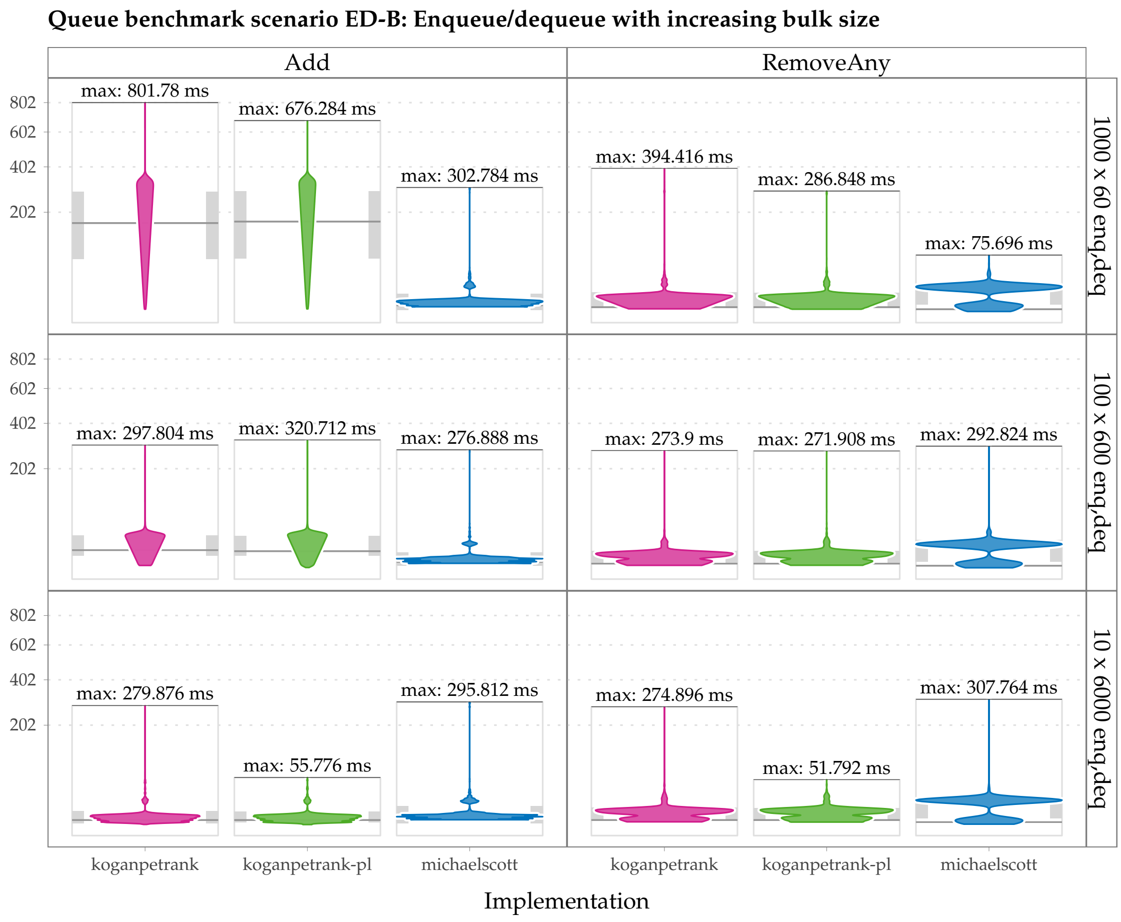

Figure 2.19: Observed latencies in queue benchmark scenario *enqueue/dequeue-pairs* with preallocation ratio from 0-66%, queue capacity of 300000 objects, 1000 iterations

comparing maximum latency in this test configuration. On closer inspection of the distribution of latencies illustrated in Figure 2.19, the wait-free Kogan-Petrank queue even achieves lower buffer latencies for most elements, especially in the two producers / two consumers setup. The helping pattern is a logical explanation: Unlike the Michael-Scott queue, all threads help both enqueue- and dequeue operations whenever they modify the queue. In fact, upper bound and median of measured buffer latency of the Kogan-Petrank queue even improve when more threads are active in the tested configurations. Presumably, this is because announced operations are reached sooner when more threads traverse the operation description array.

2.6 Stacks

Like pools and queues, stack data structures implement bag semantics and are frequently used to represent pools in literature. The well-known operations of the stack data type are:

`Push(s,e)`	Puts an element *e* on stack *s*
`Pop(s,e)`	Removes and returns element *e* from stack *s*

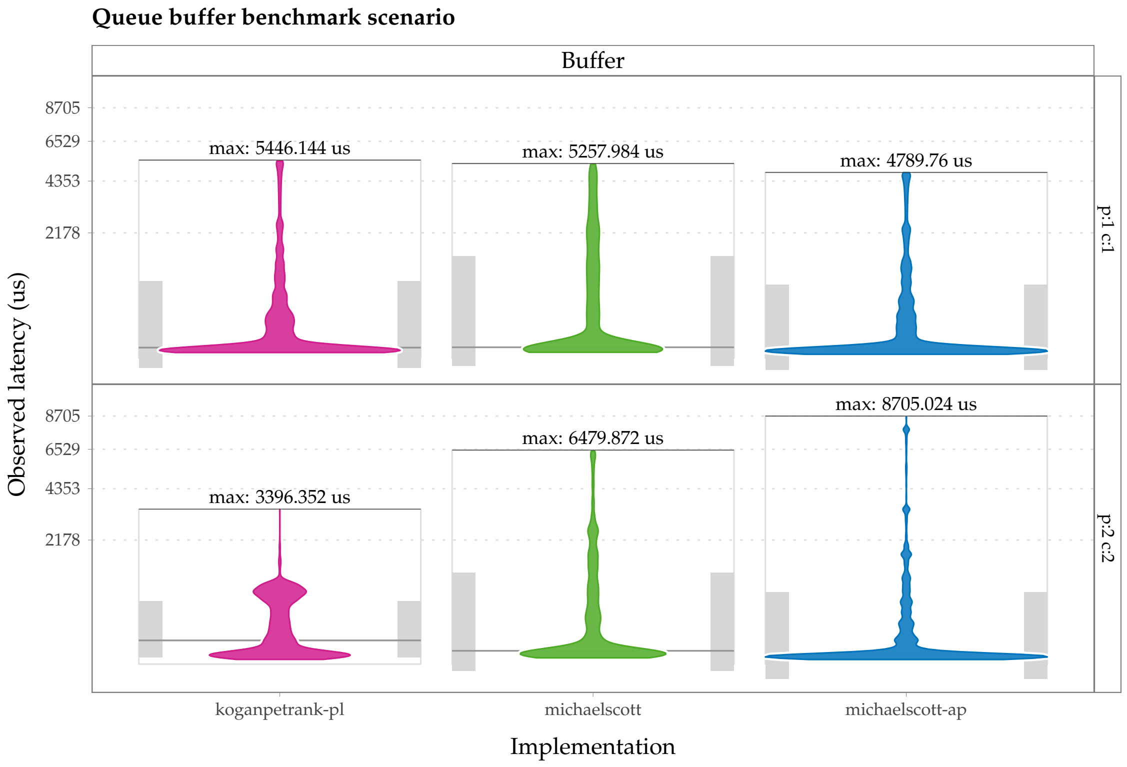

Figure 2.20: Observed latencies in queue benchmark scenario *buffer* with 1/1 and 2/2 producer/consumer threads

Literature and related work do not agree on the behaviour of concurrent stacks, especially when trying to remove an element from an empty stack, or how LIFO consistency should be maintained under high load.

We define semantics for the stack data type as the following axiomatic specification:

$$isEmpty(emptyStack()) = TRUE \tag{2.13}$$
$$isEmpty(push(e,s)) = FALSE \tag{2.14}$$
$$pop(s = emptyStack()) = (FALSE, Undefined, s) \tag{2.15}$$
$$pop(push(e,s),s) = (TRUE, e, s) \tag{2.16}$$
$$push(pop(e,s),s) = (TRUE, e, s) \tag{2.17}$$

Considering element order, we follow the definition of linearizability which allows arbitrary execution order for operations that overlap in a history.

2.6.1 An efficient wait-free stack for multiple producers and consumers

It is tempting to derive a stack data structure from the Kogan-Petrank queue, now that a suitable implementation is available. However, the helping procedure in the wait-free queue is not easily adapted for LIFO semantics.

One could transform Treiber's lock-free stack using the wait-free simulation technique explained in subsection 2.2.2. A core element of its design is an instance of the Kogan-

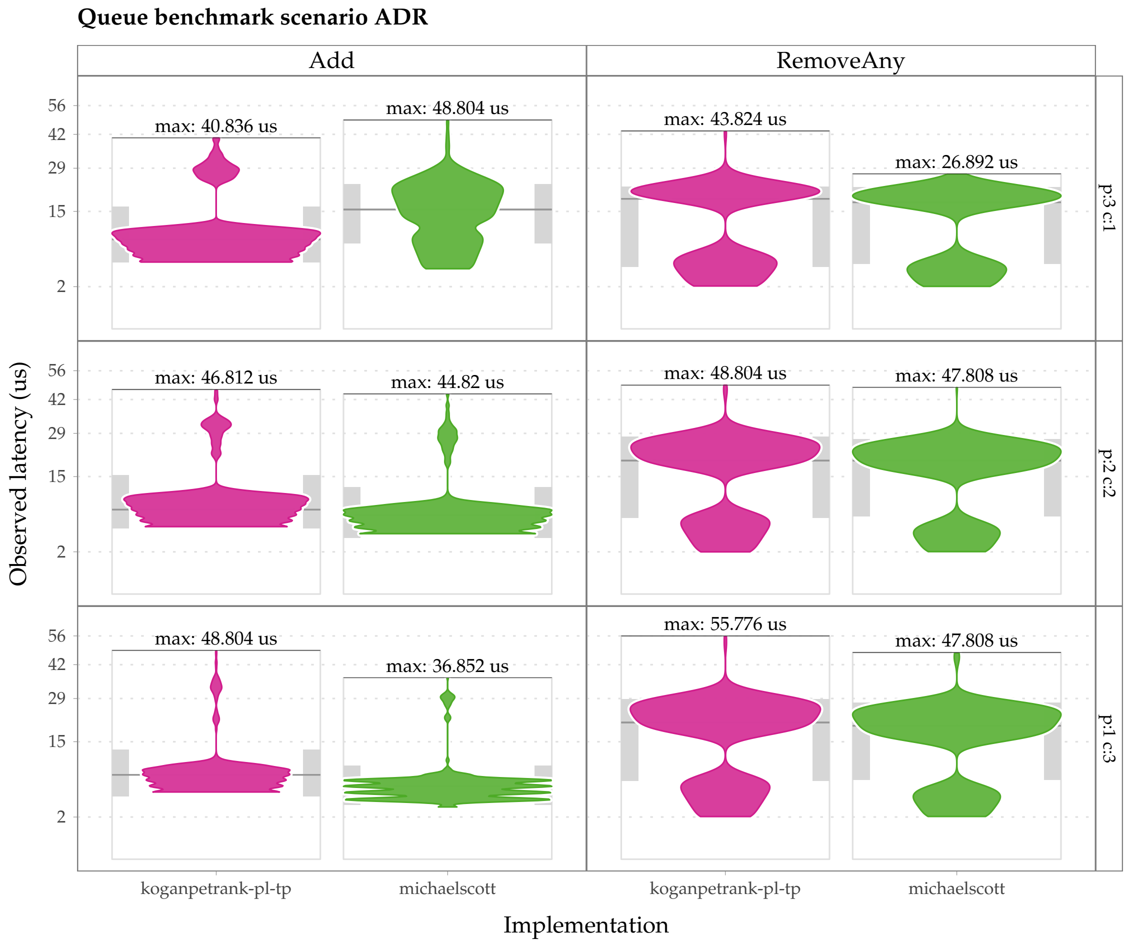

Figure 2.21: Observed latencies in queue benchmark scenario *allocator- / deallocator race* with varying balance of allocator- and deallocator threads

Petrank queue as a container of operation descriptions to achieve wait-free, linearizable access to announced operations. As all operations interact with the operation buffer, performance of any algorithm based on this method is limited by the underlying wait-free queue.

In general, semantics of the stack introduce a drawback considering efficient parallel access: Concurrent modifiying accesses interfere in every operation as accessors operate on the same head reference, which is a single point of contention as described in subsection 2.4.7. Evidently, the contention rate is thus considerably higher as compared to a concurrent queue where enqueue- and dequeue-operations do not conflict if the queue is non-empty.

Semantics in Equation 2.13 describe that methods `push` and `pop` are idempotent in composition, that is: the state of a stack object is effectively unchanged if two subsequent operations do not invoke the same method. Obviously, a call to `pop` reverts the effect of an immediately preceeding call to `push` on the stack. It is desireable to employ the elimination scheme explained in subsection 2.2.6 which is known to improve scalability of stack algorithms in related work. Fatourou's wait-free universal construction *P-Sim* achieves scalability based on elimination, and it is argued that it can be directly applied to stack algorithms in its original publication [FK11].

We implemented a wait-free stack for multiple enqueuers and dequeuers following

Fatourou's methodology. Compared to Kogan and Petrank's wait-free simulation, P-Sim describes a more flexible and concise interface.

The construction is essentially contained in a central function `ApplyOperation` that accepts a generic reference to a shared object, and a reference to a function. The latter implements a modification on it in a sequential manner, i. e. disregarding concurrency. A any concurrent operation on the shared object. In Fatourou's helping mechanism, threads announce their activity by setting a dedicated bit in a global atomic variable that serves as a bit field. This method is in effect identical to the announcement of pending operation descriptions in the wait-free queue. Using an atomic variable, however, thread can easily obtain a snap shot of all pending operations at the time of their own announcement.

In the helping phase of the wait-free stack, all ative push operations are completed before all pending pop operations, which also realizes eliminiation of these operations. The complete implementation in C++ is listed in section B.2. Pointer tags have been replaced by Michael's hazard pointers scheme.

Different from the Kogan-Petrank queue, threads do not perform own or helped operations on the shared object, but on a local copy of its global state. Once all operations are applied, they try to replace the global state with their modified local copy atomically using CAS. A successful CAS is the algorithms linearization point. A thread has to retry this copy-modify-apply cycle only once, as its operation is guaranteed to be completed by another thread if a second CAS fails.

While this is an elegant design, *SIM* and its variant *P-SIM* require that the object's state is contained in a CAS-compatible variable. This is easily achieved for a stack, and also for a queue's head and tail pointers. However, the state of data structure objects with higher order, such as trees and graphs, is not limited and therefore not compatible with existing atomic read-modify-write instructions. As data structures exist that cannot be implemented using *P-SIM*, it is actually not a *universal* construction due to technical restrictions.

As common in related work, the publication of the *P-SIM* algorithm does not include wait-free memory reclamation. In fact, memory is not reclaimed at all in the presented pseudo code. Also, allocation and deallocation need an additional undo-methodolody in *P-SIM*: Whenever a thread allocates an element for a `Push` operation, it must be freed if the final CAS on the global state fails. Likewise, elements cannot be freed in `Pop` immediately, as the overall operation might fail.

In the modified implementation listed in Appendix B of this work, allocation and deallocation is delayed and only performed after the linearization point. For this, the capacity of the stack's shared element pool must be increased by the maximum amount of thread operating on the stack, as every thread can only enqueue at most one element at any time.

2.6.2 Verification

Most verification scenarios defined for the queue data type can be reused for stacks as both implement bag semantics.

Assertions on element order are straight-forward in tests for queue objects: It suffices to enqueue timestamp elements and verify total order of dequeued elements. Verification

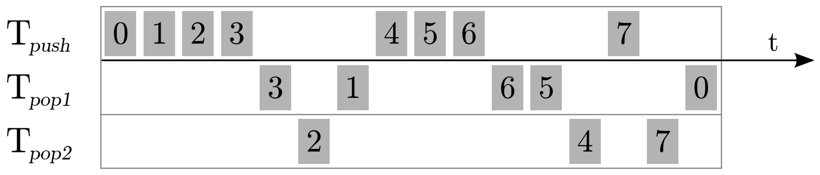

Figure 2.22: Schedule demonstrating dequeued element order on a concurrent SP/MC stack

of LIFO order requires a more complex technique. The observed order of dequeued elements from a concurrent stack can seem arbitrary despite being correct, even for just one enqueuer and one dequeuer. An example of a schedule with observed dequeued elements is illustrated in Figure 2.22. The order of elements potentially even degenerates to FIFO order, i. e. in a schedule that is alternating between enqueue- and dequeue operations.

A decision on correct temporal element order for more than one consumer thread seems even more complex. There is no apparent pattern that could be used within the consumer threads dequeue-loops to identify out-of-order cases. Monotonic time-stamps could serve as sequence numbers, but are not equidistant.

A viable solution for run-time assertions on LIFO element order is sketched in Figure 2.23. Producer threads put elements on the stack that contain a tuple (p, seq) containing the producer thread's identifier and a sequence number that is local to the producer thread and incremented for every element. Consumer threads cannot assert on a complete series of sequence numbers, but an assertion on the maximum dequeued sequence number is an equivalent solution. Every dequeued element dequeued in iteration i then must contain a sequence number $seq_{deq}[p][i]$:

1. that is smaller by exactly one increment than the sequence number in the last dequeued element from the same producer thread p
2. that is greater than all sequence numbers from the same producer thread dequeued in previous iterations.

This test does not require any synchronization mechanism between the threads. For an element dequeued in iteration i with sequence value seq_i, the assertion is defined as:

$$\text{assert}\left(\text{seq}_\text{i} = \text{seq}_\text{deq}[p][i-1] - 1 \quad \vee \quad \text{seq}_\text{i} > \text{seq}_\text{max}[p]\right)$$

After all threads, the series of all dequeued sequence numbers is simply asserted against gaps and duplicates to verify that all elements have been enqueued and dequeued.

2.6.3 Evaluation

For performance analysis of the stack algorithms, we apply benchmark scenarios that have earlier been defined for pools and queues:

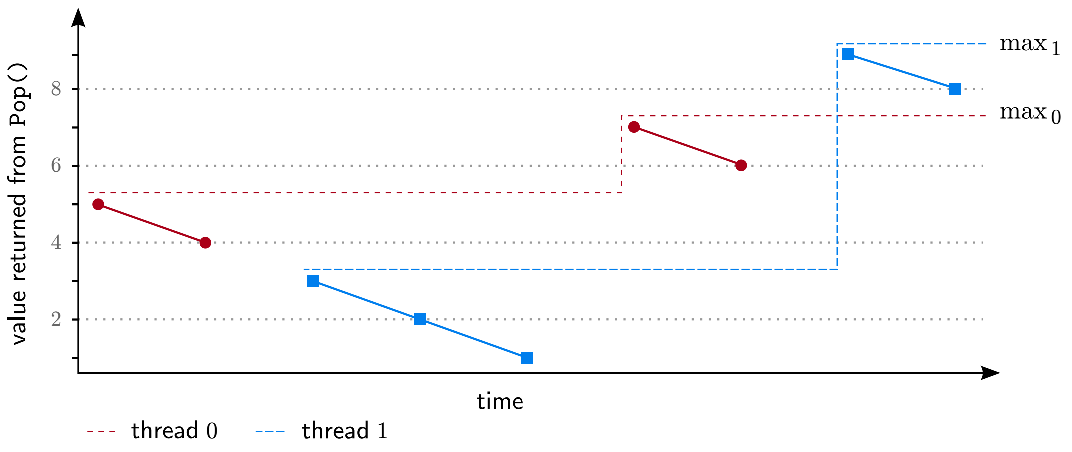

Figure 2.23: Maximum value constraints in verification of concurrent LIFO order for a series of monotonically incremented values

test	n_T threads	capacity c	pre-allocation r	i iterations	i_a allocs / i
ED-P	2-32	100 k	0	60 k	190
BUF	2:2 - 14:14	$n_P \cdot i_a$	0	10 k	8

Effects of elimination that is only integrated in the wait-free stack are evident. The lock-free stack shows coomparable constant overhead and achieves higher completion rates if only a few threads are active. Already for 4 threads, contention is high enough to observe benefits from elimination.

Both implementations can be considered en par in latency measurements of operations `Add` and `RemoveAny` summarized in Figure 2.25. These results are different from previous experiments with the Kogan-Petrank queue, where latency was notably higher compared to a lock-free queue implementation. Consequently, the effective constant overhead of the wait-free operations is arguably low compared to other constructions such as wait-free simulation, despite additional complexity due to wait-free progress guarantees.

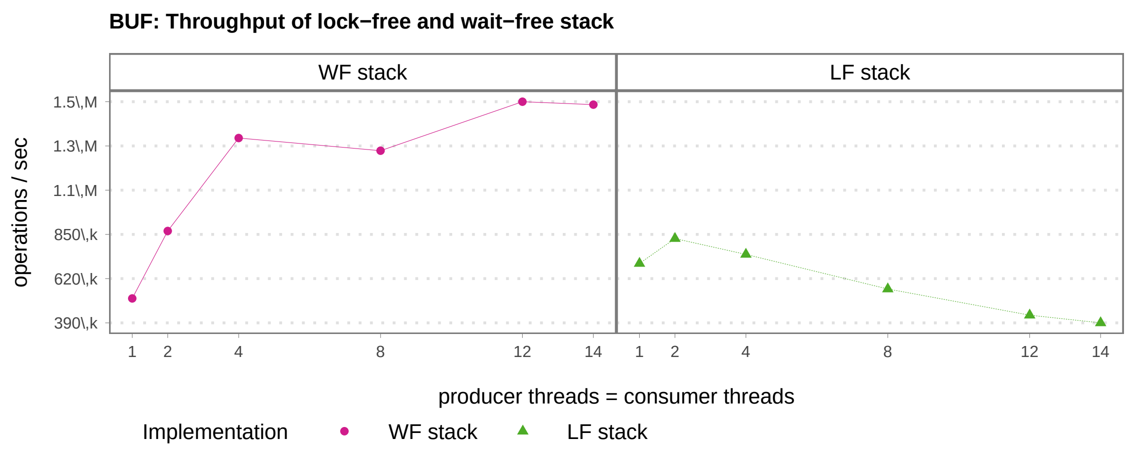

Figure 2.24: Throughput in stack benchmark scenario *buffer*, 200000 iterations

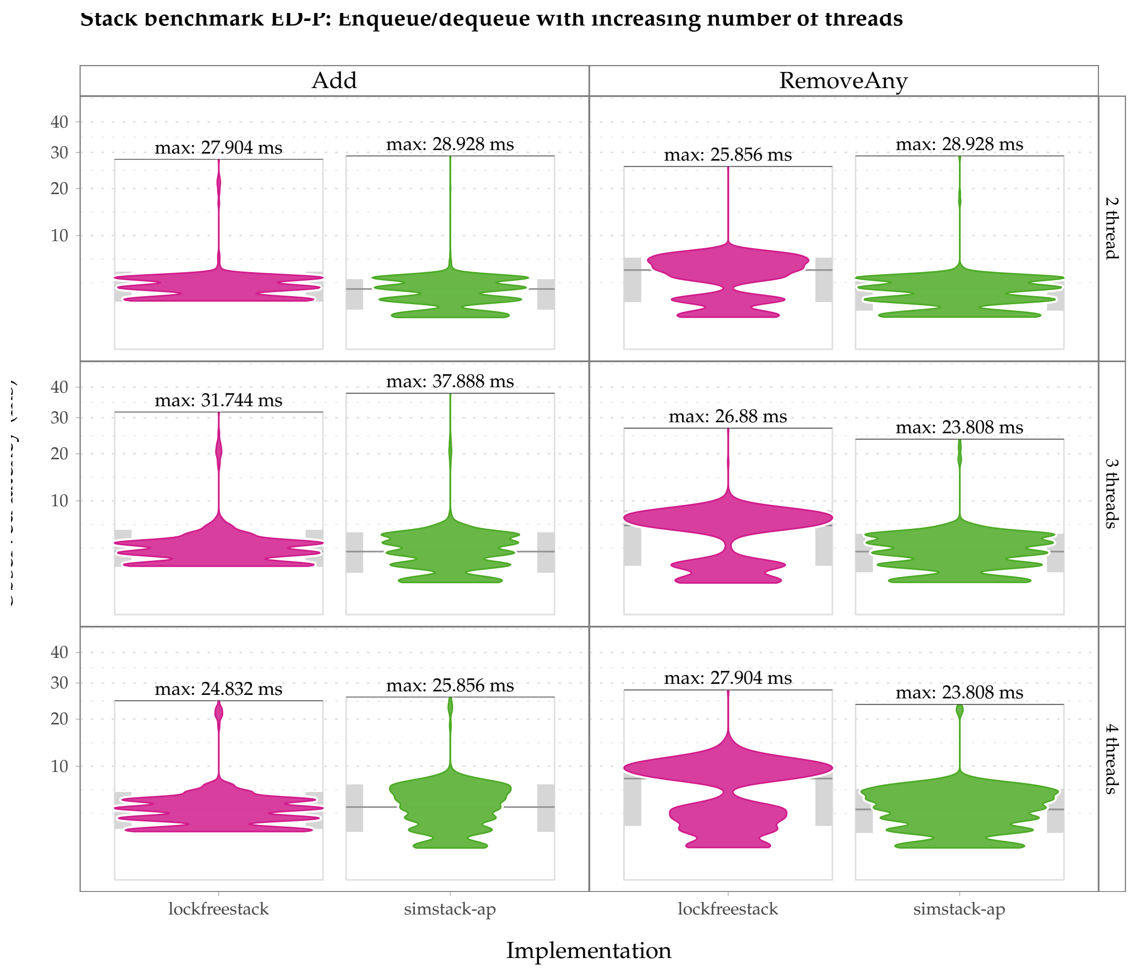

Figure 2.25: Observed latencies in stack benchmark scenario *enqueue/dequeue pairs*, 60000 iterations

3 Task Scheduling

Different from scheduling on operating system level, intra-task scheduling coordinates the execution of short, light-weight tasks in worker threads within a single process in user space. This chapter reviews related work on *work stealing* and *work dealing*, two common paradigms in this context. Finally, we present a variant of work-stealing that integrates task priorities in a practicable way.

3.1 Introduction

Most concurrency frameworks such as *OpenMP*[1] provide integrated patterns for parallelization of small program blocks. For example, iterations of loops annotated as *parallel for* are unfolded into single tasks and distributed to multiple worker threads. With many task groups competing for CPU time, user-space scheduling mechanisms coordinate their execution similar to system-wide scheduling within the kernel.

The next subsection describes fundamental principles of task-scheduling that are common in literature, followed by a summary of related work on specific approaches.

3.1.1 Technical foundations

As a common principle of work distribution on application level, fine-granular tasks are instantiated as objects consisting of a the task body, typially a closure in the sense of a lambda-function, and the task context. Tasks are only executed by a number of *worker threads* in user space. Worker threads are created during initialization of the concurrency framework and each assigned to a dedicated CPU core. Their life cycle only ends with termination of the program.

To schedule a task object for execution, it is enqueued in a worker thread's publicly accessible *task queue*. This abstraction and short-living tasks allows to reassign tasks between worker threads (and thus, CPU cores) easily, but with additional overhead.

The basic principle of **work stealing** is to occupy idle cores with scheduling, and to allow busy cores to focus on finishing their work. For scheduling fully-strict computations This mode of thought has been proven as optimal [Pol13]. Workers obtain the next task to execute either by dequeuing from the bottom of their local double-ended queue, or by stealing from the bottom of other workers' task queues. In the latter case, the threads are named *thief* and *victims* respectively.

The term *work sharing* refers to load-balancing strategies where a scheduler actively migrates tasks from processors under high load to those with capacity available [BL99].

1 Official OpenMP Website: http://openmp.org

It is sometimes beneficial to prevent worker threads from migration. For this, they are configured with an affinity mask that instructs the operating system to execute them on a given set of cores preferably. This practice is also referred to as *CPU pinning*.

3.1.2 Related Work

Various paradigms for work sharing are known from related work, with work stealing being the most prevalent. Compared to work sharing, Blumofe argues that work stealing leads to less frequent migration of tasks, as worker threads only request tasks when they have capacity available [BL99]. Supposedly, the locality of work sharing is inferior in the common case. In addition, workers deal tasks from the top of queues in work sharing. Work stealing ensures the oldest task is dequeued when stealing, and with it potentially a coherent set of child tasks that it could spawn. This has a positive effect on locality and reduces the frequency of (expensive) stealing [BJK^{+}94].

Hendler and Shavit discussed the effects of cache locality on performance of real-time systems and introduced *work dealing* as a suggested solution. Their method derives from work-stealing, and borrows the concept of worker threads and task queues. As opposed to work stealing, the latter are not owned by a single thread, the number of task queues is not limited. An arbitrarily defined policy determines the queue a newly generated task is added to, and from which queue an idle thread will retrieve the next task to execute.

Herlihy raises the objection that under high load, exchange of tasks might rarely succeed in work dealing. In this case, the mediation scheme has no benefit but only creates additional overhead [Her, 16.4].

Nogueira focuses on work-stealing in real-time systems in a recent publication, but does not provide empirical results [Pol13]. He describes an approach utilizing a global earliest-deadline-first queue, where processor- specific queues are used as priority queues of queues of tasks, and idle processors steal from the priority schedule upon all processors. As an alternative, a constant bandwidth server processes high-prioritized tasks in parallel to a work-stealing scheduler.

Additional schemes exist, such as *work balancing*, where worker threads actively exchange tasks periodically [Her, p.].

3.2 Work-stealing with prioritization

The original description of work-stealing and related work do not consider preferred execution of tasks by priority. The approaches presented by Nogueira are promising, but difficult to implement as portable components. As a pragrmatic solution, we build upon a conventional design with worker threads that actively steal tasks from victims. We introduce prioritization in the worker loops and stealing methodolgy as follows:

Worker threads maintain a queue for every priority level provided by the runtime framework. Tasks in the queue with highest priority will be executed first, as displayed in

Figure 3.1. Considering this modification, more specific strategies can be described that define how and when tasks are stolen from victim threads.

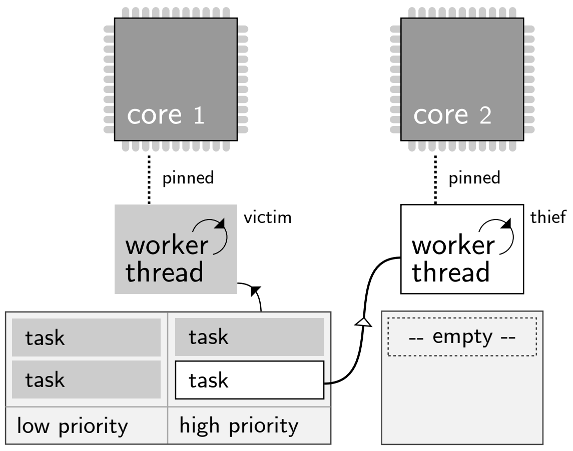

Figure 3.1: Work stealing with task queues and pinned worker threads

Our implementation is based on the MTAPI standard and part of the current release 0.2.0 of the Embedded Multicore Building Blocks (EMBB) library [2].

In the default configuration of the scheduler, task migration follows the so-called *Local First* principle: stealing only occurs if queues on all priority levels are empty, minimizing overhead in task migration. The alternative *Victim High Priority First* allows to steal from a victim queue with highest possibly priority as soon as the local queue with equal priority is empty.

2 EMBB project website: http://github.com/siemens/embb

4 Summary

Referring to the problem statement in introduction, this last chapter summarizes results and points out the most significant findings obtained in the course of this work.

4.1 Revisiting the objective

In this thesis, we discussed the state of the art of wait-free data structures and thoroughly examined related work for their suitability in embedded- and real-time applications.

Existing designs of a wait-free queue and stack algorithm have been adapted, resulting in implementations of a wait-free queue and stack for an arbitrary number of threads. While doing so, we described techniques and best practices that allow to achieve wait-free progress guarantees, and to adapt general purpose data structure designs to constraints on resource consumption and fault-tolerance.

Publications on wait-free algorithms leave memory management out of scope and refer to hazard pointers as a proposed solution. We pointed out that hazard pointers do not guarantee wait-freedom when employed, and also demonstrated how safe memory reclamation can be integrated in the Kogan-Petrank queue and the P-SIM universal construction. Allocation has been entirely replaced by pools presented in this work, and no counter mechanisms are required. The resulting modified designs comply with guidelines and constraints in real-time applications.

We discussed viable methodologies of evaluation and verfication in detail, and explained common pitfalls, and how to avoid them. Benchmark scenarios have been derived from data type semantics and real-world use cases in real-time applications. Obtained results allow evaluation of worst-case performance criteria of and are easily applied to designs presented in prior or future work.

4.2 Recommendations

Evaluation in this work demonstrated that effective optimization techniques can improve throughput and latency of wait-free algorithms, rivaling performance of their lock-free alternatives. This, however, only applies when allocation overheads have similar complexity. Pools are an essential component in all dynamic containers regardless of their semantics, and in effect limit their efficiency. A slightly optimized wait-free pool has been discussed, but worst-case performance of wait-free allocation could not be achieved below linear complexity. The most challenging restrictions apply for the implementation of pools that are suitable for dynamic memory allocation within containers; known optimization techniques thus cannot be applied without profound modifications.

With a wait-free pool algorithm that itself does not rely on dynamic memory allocation or techniques permitted by embedded software constraints, applicability of wait-free data structures in general would greatly improve. The modified wait-free stack presented in this work could serve as a foundation of an efficient pool that benefits from the elimination strategy. For this, safe memory reclamation must be redesigned to eliminate the need for underlying memory management. With a promising solution on its way, it is recommended to follow upcoming releases of the open source EMBB project.

Bibliography

[BBH$^+$13] J. Barnat, L. Brim, V. Havel, J. Havlicek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[BJK$^+$94] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system, 1994.

[BK08] C. Baier and J.P. Katoen. *Principles of Model Checking*. Mit Press, 2008.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[BNHS11] Gal Bar-Nissan, Danny Hendler, and Adi Suissa. A dynamic elimination-combining stack algorithm. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, pages 544–561, Berlin, Heidelberg, 2011. Springer-Verlag.

[CER10] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 335–344, New York, NY, USA, 2010. ACM.

[Don06] Brijesh Dongol. Formalising progress properties of non-blocking programs. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering*, ICFEM'06, pages 284–303, Berlin, Heidelberg, 2006. Springer-Verlag.

[DT86] International Business Machines Corporation. Research Division and R.K. Treiber. *Systems Programming: Coping with Parallelism*. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[EFK$^+$12] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 115–124, New York, NY, USA, 2012. ACM.

[FK11] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 325–334, New York, NY, USA, 2011. ACM.

[Fra04] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[GPST09] Anders Gidenstam, Marina Papatriantafilou, Hakan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Trans. Parallel Distrib. Syst.*, 20(8):1173–1187, August 2009.

[Gun93] Neil J Gunther. A simple capacity model of massively parallel transaction systems. In *CMG-CONFERENCE-*, pages 1035–1035. COMPSCER MEASUREMENT GROUP INC, 1993.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[Her]

[Her88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 276–290, New York, NY, USA, 1988. ACM.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[HLM02] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures. Technical report, Mountain View, CA, USA, 2002.

[HS11] Maurice Herlihy and Nir Shavit. On the nature of progress. In Antonio Fernández Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 313–328. Springer Berlin Heidelberg, 2011.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[Jay98] Prasad Jayanti. A complete and constant time wait-free implementation of cas from ll/sc and vice versa. In Shay Kutten, editor, *Distributed Computing*, volume 1499 of *Lecture Notes in Computer Science*, pages 216–230. Springer Berlin Heidelberg, 1998.

[JP05] Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS '05, pages 408–419, Berlin, Heidelberg, 2005. Springer-Verlag.

[JP06] Prasad Jayanti and Srdjan Petrovic. Efficiently implementing a large number of ll/sc objects. In JamesH. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin Heidelberg, 2006.

[KP11] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. *SIGPLAN Not.*, 46(8):223–234, February 2011.

[KP12] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, feb 2012.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11), 1977.

[Man86] Udi Manber. On maintaining dynamic information in a concurrent environment. *SIAM J. Comput.*, 15(4):1130–1142, nov 1986.

[Mic04a] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[Mic04b] Maged M. Michael. Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas. In *In DISC 2004, vol. 3274 of LNCS*, pages 144–158, 2004.

[Mic04c] Maged M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, June 2004.

[MNSS05] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM.

[MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical report, Rochester, NY, USA, 1995.

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy-update: Using execution history to solve concurrency problems, 1998.

[MS07] M. Moir and N. Shavit. Concurrent data structures. In D. Metha and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 47–14 – 47–30, 2007. Chapman and Hall/CRC Press.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[PMS09] Erez Petrank, Madanlal Musuvathi, and Bjarne Steesngaard. Progress guarantee for parallel programs via bounded lock-freedom. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, New York, NY, USA, 2009. ACM.

[Pol13] Polytechnic Institute of Porto (ISEP-IPP). *On the use of Work-Stealing Strategies in Real-Time Systems*, Berlin, Germany, 2013.

[SKKE11] Eunhwan Shin, Inhyuk Kim, Junghan Kim, and Young Ik Eom. Strata: Wait-free synchronization with efficient memory reclamation by using chronological memory allocation. In *Proceedings of the 2011 International Conference on Computational Science and Its Applications - Volume Part V*, ICCSA'11, pages 217–231, Berlin, Heidelberg, 2011. Springer-Verlag.

[SKSP] Philippe Stellwag, Jakob Krainz, and Wolfgang Schröder-Preikschat. A wait-free dynamic storage allocator by adopting the helping queue pattern. In *Proceedings of the 9th IASTED International Conference*, volume 676, page 79.

[Sun05] Hakan Sundell. Wait-free reference counting and memory management. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, pages 24.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[SZ96] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, nov 1996.

[TBKP12] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 309–310, New York, NY, USA, 2012. ACM.

[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 '14, pages 357–368, New York, NY, USA, 2014. ACM.

[TSP92] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '92, pages 212–222, New York, NY, USA, 1992. ACM.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM.

[vdB13] F.I. van der Berg. Model checking llvm ir using ltsmin. Master's thesis, University of Twente, December 2013.

[ZZY$^+$13] Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael Spear. Practical non-blocking unordered lists. In Yehuda Afek, editor, *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 239–253. Springer Berlin Heidelberg, 2013.

Appendices

A List of Figures

B Code listings

Source code repositories

- Official EMBB release and development code base:
  https://github.com/siemens/embb/
- Branch with wait-free containers and benchmark implementations for this thesis (also contains R files used to generate plots):
  https://github.com/fuchsto/embb/tree/benchmark

B.1 Implementation of the modified Kogan-Petrank queue

```
template<typename T,
         typename ValuePool = WaitFreeArrayValuePool<bool, false>, [...] >
class WaitFreePhaselessQueue {
private:
  typedef union OperationDesc_u {
    struct OperationDesc_s {
      bool Pending : 1;
      bool Enqueue : 1;
      index_t NodeIndex : 30;
    } fields;
    atomic_t Raw;
  } OperationDesc;
  static const index_t QUEUE_SIZE_MAX = 0x3FFFFFFE;
  static const size_t NUM_GUARDS = 2;
  CACHE_ALIGN Atomic<index_t> headIdx
  CACHE_ALIGN Atomic<index_t> tailIdx;
  size_t size, numThreads, node_pool_size;
  Atomic<index_t> * operations;
  ValuePool nodeIndexPool;
  Node_t * nodePool;
  inline size_t retiredListMaxSize(size_t nThreads) {
    return static_cast<size_t>(1.25 *
      static_cast<double>(nThreads) * static_cast<double>(num_guards)) + 1;
  }
public:
  WaitFreePhaselessQueue(size_t size, size_t numThreads) :
    size(size),
    numThreads(numThreads),
    node_pool_size(retiredListMaxSize(numThreads) * numThreads + (size + 1)),
    nodeIndexPool(0, node_pool_size)
  {
    nodePool = nodeAllocator.allocate(node_pool_size);
    for (size_t i = 0; i < node_pool_size; ++i) {
      Node_t nullNode;
      nodePool[i] = nullNode;
    }
    Node_t sentinelNode;
    int sentinelNodePoolIndex = nodeIndexPool.RemoveAny();
    // Guard sentinel node from deletion for the
    // life time of the queue:
    hp.GuardPointer(0, sentinelNodePoolIndex);
    headIdx.Store(sentinelNodePoolIndex);
    tailIdx.Store(sentinelNodePoolIndex);
    operations = operationDescriptionAllocator.allocate(numThreads);
    for (size_t accessorId = 0; accessorId < numThreads; ++accessorId) {
      OperationDesc op(NONPENDING, NOOP, UndefinedIndex);
      operations[accessorId].Store(op.Raw);
    }
  }
  ~WaitFreePhaselessQueue() {
    T val;
    while (TryDequeue(val)) {}
    nodeAllocator.deallocate(nodePool, node_pool_size);
    operationDescriptionAllocator.deallocate(operations, numThreads);
  }
  bool TryEnqueue(T const & element) {
    index_t accessorId = Node_t::UndefinedIndex;
    Node_t poolNode;
    int nodeIndex = nodeIndexPool.RemoveAny();
    if (nodeIndex < 0) { return false; }
    hp.GuardPointer(0, nodeIndex);
    Node_t newNode(element, accessorId);
    nodePool[static_cast<index_t>(nodeIndex)] = newNode;
    OperationDesc enqOp(PENDING, ENDQUEUE, nodeIndex);
    operations[accessorId].Store(enqOp.Raw);
    Help();
    HelpFinishEnqueue();
    hp.GuardPointer(0, UndefinedGuard);
    return true;
  }
  bool TryDequeue(T & retElement, size_t accessorId) {
    OperationDesc curOp(operations[accessorId].Load());
    OperationDesc newOp(PENDING, DEQUEUE, UndefinedIndex);
    index_t curOpRaw = curOp.Raw;
    operations[accessorId].CAS(curOpRaw, newOp.Raw));
    Help();
    HelpFinishDequeue(); // Reload own operation description
    curOp = OperationDesc(operations[accessorId].Load());
    index_t nodeIdx = curOp.NodeIndex;
    Node_t & node   = nodePool[nodeIdx];
    if (nodeIdx == Node_t::UndefinedIndex) { return false; }
    index_t deqIdx  = node.NextPoolIdx();
    retElement = nodePool[deqIdx].Value();
    hp.RetireNode(nodeIdx);
    return true;
  }
private:
  void HelpEnqueue(size_t accessorId) {
    while (OperationDesc(operations[accessorId].Load()).Pending) {
      index_t lastIdx   = tailIdx.Load();
      Node_t & lastNode = nodePool[lastIdx];
      index_t nextIdx   = lastNode.NextPoolIdx();
      // Last node still is tail:
```

```
      if (lastIdx == tailIdx.Load()) {
        if (lastNode.NextIsNull()) {
          if (IsPending(accessorId)) {
            OperationDesc opDesc(operations[accessorId].Load());
            if (lastNode.Next.CompareAndSwap(nextIdx, opDesc.NodeIndex)) {
              HelpFinishEnqueue();
              return;
            }
          }
        } else {
          HelpFinishEnqueue();
        }
      }
    }
  }
  void HelpFinishEnqueue() {
    index_t lastIdx   = tailIdx.Load();
    Node_t & lastNode = nodePool[lastIdx];
    if (!lastNode.NextIsNull()) {
      index_t nextIdx   = lastNode.NextPoolIdx();
      Node_t & nextNode = nodePool[nextIdx];
      index_t helpAID   = nextNode.EnqueueAID();
      OperationDesc helpOp(operations[helpAID].Load());
      if (lastIdx == tailIdx.Load() &&
        helpOp.NodeIndex == nextIdx) {
        OperationDesc newOp(NONPENDING, ENQUEUE, nextIdx);
        index_t helpOpRaw = helpOp.Raw;
        operations[helpAID].CompareAndSwap(
          helpOpRaw,
          newOp.Raw);
        tailIdx.CompareAndSwap(lastIdx, nextIdx);
      }
    }
  }
  void HelpDequeue(index_t accessorId) {
    while (IsPending(accessorId)) {
      index_t firstIdx = headIdx.Load();
      index_t lastIdx  = tailIdx.Load();
      Node_t & first   = nodePool[firstIdx];
      if (firstIdx != headIdx.Load()) { continue; }
      if (firstIdx == lastIdx) {  // Queue might be empty
        if (first.NextIsNull()) { // Queue is empty
          hp.ReleaseGuard(0, firstIdx);
          OperationDesc curOp(operations[accessorId].Load());
          if (lastIdx == tailIdx.Load() && IsPending(accessorId)) {
            // Undefined index signals failed dequeue
            OperationDesc newOp(NONPENDING, DEQUEUE, UndefinedIndex);
            index_t curOpRaw = curOp.Raw;
            operations[accessorId].CompareAndSwap(curOpRaw, newOp.Raw);
          }
        } else {
          HelpFinishEnqueue();
        }
      } else {
        OperationDesc curOp(operations[accessorId].Load());
        index_t nodeIdx = curOp.NodeIndex;
        if (!IsPending(accessorId)) { break; }
        if (firstIdx == headIdx.Load() && nodeIdx != firstIdx) {
          hp.GuardPointer(0, firstIdx);
          OperationDesc newOp(PENDING, DEQUEUE, firstIdx);
          index_t curOpRaw = curOp.Raw;
          if (!operations[accessorId].CAS(curOpRaw, newOp.Raw)) {
            continue;
          }
        }
        index_t curDeqAID = Node_t::UndefinedIndex;
        first.DequeueAID().CAS(curDeqAID, accessorId)
        HelpFinishDequeue();
      }
    } // while pending
  }
  void HelpFinishDequeue() {
    index_t firstIdx   = headIdx.Load();
    Node_t & first     = nodePool[firstIdx];
    index_t nextIdx    = first.NextPoolIdx();
    hp.GuardPointer(0, nextIdx);
    index_t accessorId = first.DequeueAID().Load();
    if (accessorId != Node_t::UndefinedIndex) {
      OperationDesc curOp(operations[accessorId].Load());
      if (firstIdx == headIdx.Load() && nextIdx != UndefinedIndex) {
        OperationDesc newOp(NONPENDING, DEQUEUE, curOp.NodeIndex);
        index_t curOpRaw = curOp.Raw;
        operations[accessorId].CAS(curOpRaw, newOp.Raw);
        headIdx.CAS(firstIdx, nextIdx);
      }
    }
  }
  void Help() {
    size_t nHelp  = numThreads;
    tid_t ownId   = threadId();
    tid_t startId = (ownAccessorId + 1) % numThreads;
    for (tid_t tId = startId; nHelp > 0; ++tId, --nHelp) {
      OperationDesc desc(
        operations[tId % numThreads].Load());
      if (!desc.Pending) continue;
      if (desc.Enqueue)
        HelpEnqueue(tId % numThreads);
      else
        HelpDequeue(tId % numThreads);
    }
  }
  void DeleteNodeCallback(index_t releasedNodeIndex) {
    nodeIndexPool.Add(releasedNodeIndex);
  }
};
```

Listing B.1: Implementation of the modified wait-free Kogan-Petrank queue with safe memory reclamation and without phase counter

B.2 Implementation of the modified wait-free stack

```
template<typename T>
class WaitFreeSimStackNode {
private:
  typedef embb::containers::internal::WaitFreeSimStackNode<T> self_t;
public:
  typedef int index_t;
  typedef struct Element_s{
    VOLATILE index_t next;
    T value;
  } Element;
};

template<
  typename T           = uint32_t,
  T UndefinedValue     = 0xFFFFFFFF,
  size_t LocalPoolSize = 64,
  class ElementPool    = IndexedObjectPool< WaitFreeSimStackNode<T>::Element
    >
  class StateIndexPool = WaitFreeCompartmentValuePool< bool, false,
    LocalPoolSize >
>
class WaitFreeSimStack
{
private:
  static const unsigned int MAX_THREADS = 32; // 64 on x64
  typedef T Object;
  typedef T RetVal;
  typedef T OperationArg;
  typedef internal::WaitFreeSimStackNode<T> Node_t;
  typedef internal::WaitFreeSimStackNode<T>::Element Element_t;
  typedef internal::WaitFreeSimStackNode<T>::index_t ElementPointer_t;
  typedef int32_t atomic_int_t;    // int64_t on x64
  typedef uint32_t atomic_uint_t;  // uint64_t on x64
  typedef uint32_t bitword_t;      // uint64_t on x64
  typedef embb::base::Atomic<bitword_t> AtomicBitVectorValue;
  typedef struct ObjectState_s {
    bitword_t applied;
    typename internal::WaitFreeSimStackNode<T>::index_t head;
    T ret[MAX_THREADS];
  } ObjectState;
  typedef struct StackThreadState {
    CACHE_ALIGN bitword_t mask;
    bitword_t toggle;
    bitword_t bit;
    ElementPointer_t localObjectStateIndex;
    unsigned int backoff;
  } StackThreadState;
  typedef uint32_t bitfield_t;
  typedef embb::base::Atomic< bitfield_t > atomic_bitfield_t;
  typedef bool state_t;
  static const bitword_t BitWordZero = 0U;
  static const bitword_t BitWordOne  = 1U;
  static const int MAX_BACK = 0x000FFF;
  /// Null-pointer representation for hazard pointers
  static const ElementPointer_t UndefinedGuard = 0;
  /// Global stack pointer state
  embb::base::Atomic<ElementPointer_t> stackStateIndex;
  AtomicBitVectorValue atomicTogglesVector;
  /// Capacity of the queue instance.
  size_t size;
  /// Size of thread-local state pools.
```

```
size_t localPoolSize;
/// Allocator for object states
embb::base::Allocator<ObjectState> objectStateAllocator;
/// Allocator for announced operation arguments (ArgVal)
embb::base::Allocator<OperationArg> operationArgAllocator;
/// Pool for node elements
ElementPool elementPool;
/// Pool for indices in object state array. Stack state indices are
/// guarded by hazard pointers.
StateIndexPool stateIndexPool;
/// Bitset recording which thread already initialized their local state
uint32_t threadRegistry;
/// Allocator for the threads' local state (will be replaced
/// by thread-specific variable)
embb::base::Allocator<StackThreadState> stackThreadStateAllocator;
/// Array of thread-specific states of the stack object
StackThreadState * threadStates;
/// Index of initial object state in stack states array
ElementPointer_t initialStateIndex;
/// Index of initial element in stack
ElementPointer_t initialElementIndex;
/// Thread-specific, incremental random seed
embb::base::ThreadSpecificStorage<long> randomNextTss;
CACHE_ALIGN VOLATILE
  OperationArg * operationArgs;
CACHE_ALIGN VOLATILE
  ObjectState * stackStates;

void initStackThreadState(
  StackThreadState * threadState,
  unsigned int accessorId) {
  threadState->localObjectStateIndex = 0;
  threadState->mask     = 0;
  threadState->mask    |= (BitWordOne << static_cast<bitword_t>(accessorId)
  );
  threadState->bit      = 0;
  threadState->bit     ^= (BitWordOne << static_cast<bitword_t>(accessorId)
  );
  threadState->toggle   = 0;
  threadState->toggle   = ~threadState->mask + 1; // 2s complement negation
  threadState->backoff  = 1;
  // Initialize thread-specific random seed:
  randomNextTss.Get() = 1;
}
inline RetVal ApplyOperation(
  StackThreadState * threadState,
  OperationArg arg,
  unsigned int accessorId) {
  unsigned int numPushOperations;
  unsigned int numPopOperations;
  bitword_t diffs;
  bitword_t toggles;
  bitword_t pendingPopOperations;
  ElementPointer_t stackStateIndexNew;
  ElementPointer_t stackStateIndexCurr;
  // Stores pool indices of pushed elements for rollback.
  // At most MAX_THREADS push operations have to be rolled
  // back.
```

```
    ElementPointer_t pushElementIndices[MAX_THREADS];
    // Stores pool indices of removed elements for deallocation.
    // At most MAX_THREADS removed elements have to be stored.
    ElementPointer_t popElementIndices[MAX_THREADS];
    // This thread's local stack object state
    ObjectStateUnpadded * localStackState;
    // The current global stack object state
    ObjectStateUnpadded * globalStackState;
    if (threadState->localObjectStateIndex < 0) {
      EMBB_THROW(embb::base::ErrorException,
        "Invalid state index");
    }
    // Prepare thread state:
    threadState->bit    ^= (BitWordOne << static_cast<size_t>(accessorId));
    threadState->toggle  = ~threadState->toggle + 1; // 2s complement
    negation
    localStackState = (ObjectStateUnpadded *)(
      &stackStates[threadState->localObjectStateIndex]);
    // announce the operation
    operationArgs[accessorId] = arg;
    // Toggle accessorId's bit in atomicTogglesVector, Fetch&Add acts as a
    // full write-barrier
    atomicTogglesVector.FetchAndAdd(threadState->toggle);
    for (int spin = 0; spin < 2; ++spin) {
      // Random backoff if enabled:
      if (backoffEnabled) {
        VOLATILE int k;
        VOLATILE long backoff_limit = RandomRange(
          static_cast<long>(threadState->backoff >> 1u),
          static_cast<long>(threadState->backoff));
        for (k = 0; k < backoff_limit; k++) {
          ;
        }
      }
      // read reference to struct ObjectState
      stackStateIndexCurr = stackStateIndex.Load();
      // read reference of struct ObjectState in a local variable
      // localStackState
      globalStackState = (ObjectStateUnpadded *)(
        &stackStates[stackStateIndexCurr]);
      // Get all applied operations at this point:
      diffs = globalStackState->applied;
      // Determine the set of active processes
      diffs ^= threadState->bit;
      // If this operation has already been applied, return
      if ((diffs >> accessorId) & 1) {
        break;
      }
      // Copy global state to local state:
      *localStackState = *globalStackState;
      toggles = atomicTogglesVector.Load();
      if (stackStateIndexCurr != stackStateIndex.Load()) {
        continue;
      }
      // Intersection of applied and toggled operations:
      diffs = localStackState->applied ^ toggles;
      numPushOperations    = 0;
      numPopOperations     = 0;
```

```
 pendingPopOperations = 0;
 // Apply all operations that have been announced before this point
 while (diffs != BitWordZero) {
  int threadId = bitSearchFirst(diffs);
  diffs ^= BitWordOne << threadId;
  T announced_arg = operationArgs[threadId];
  if (announced_arg == UndefinedValue) {
   // == POP =======
   // Collect pending pop operatins in bit vector:
   pendingPopOperations |= (BitWordOne << threadId);
  }
  else {
   // == PUSH ======
   // Perform push operation and store pool index of
   // pushed element for rollback:
   pushElementIndices[numPushOperations] =
    PushOperation(localStackState, threadState, announced_arg, threadId);
   numPushOperations++;
  }
 }
 // Apply pending pop operations:
 while (pendingPopOperations != BitWordZero) {
  int threadId = bitSearchFirst(pendingPopOperations);
  pendingPopOperations ^= (BitWordOne << threadId);
  // Perform pop operation on local stack state and
  // store indices of removed elements. These will be
  // freed if the operation succeeded:
  popElementIndices[numPopOperations] =
   PopOperation(localStackState, threadState, threadId);
  numPopOperations++;
 }
 // Update applied operations of local stack state:
 localStackState->applied = toggles;
 // Store index in pool where localStackState will be stored in
 // new stack pointer's index field:
 stackStateIndexNew = threadState->localObjectStateIndex;
 if (stackStateIndexNew < 0) {
  EMBB_THROW(embb::base::ErrorException,
   "Invalid state index");
 }
 // Guard index of current object state:
 hp.GuardPointer(0, stackStateIndexCurr);
 if (stackStateIndexCurr == stackStateIndex.Load() &&
   stackStateIndex.CompareAndSwap(
    stackStateIndexCurr,
    stackStateIndexNew)) {
  ElementPointer_t lastLocalObjectStateIndex =
   threadState->localObjectStateIndex;
  // Reserve new index from index pool:
  bool dummy;
  int objectStateIndex = stateIndexPool.Allocate(dummy);
  if (objectStateIndex < 0) {
   EMBB_THROW(embb::base::NoMemoryException,
    "Failed to allocate index for object state");
  }
  // Assign new index of local object state:
  threadState->localObjectStateIndex = objectStateIndex;
```

```
        // Operation succeeded, reduce backoff limit:
        threadState->backoff = (threadState->backoff >> 1) | 1;
        // Free elements removed in pop operations:
        for (int rb = 0;
             static_cast<unsigned int>(rb) < numPopOperations;
             ++rb) {
          // Paranoia check that initial element is not freed:
          if (popElementIndices[rb] != initialElementIndex) {
            elementPool.Free(popElementIndices[rb]);
          }
        }
        // Retire index of this thread's last local object state:
        hp.EnqueuePointerForDeletion(lastLocalObjectStateIndex);
        // Release guard on index of replaced object state:
        hp.GuardPointer(0, UndefinedGuard);
        // Return value from local thread state:
        return localStackState->ret[accessorId];
      }
      else {
        // Release guard on index of current object state:
        hp.GuardPointer(0, UndefinedGuard);
        if (backoffEnabled && threadState->backoff < maxBackoff) {
          // Operation failed, reduce backoff limit:
          threadState->backoff <<= 1;
        }
        // Free elements allocated during failed push operations:
        for (int rb = 0;
             static_cast<unsigned int>(rb) < numPushOperations;
             ++rb) {
          // Paranoia check that initial element is not freed:
          if (popElementIndices[rb] != initialElementIndex) {
            elementPool.Free(pushElementIndices[rb]);
          }
        }
      }
    }
    // Return the value from the state stored at the current object
    // state index:
    return stackStates[stackStateIndex.Load()].ret[accessorId];
  }
public:
  // Constructor
  WaitFreeSimStack(size_t size)
  : size(size),
    elementPool(
      // Add capacity for elements allocated in
      // unsuccessful push operations that will be
      // freed after when the operation succeeded:
      size + (MAX_THREADS * threadLocalPoolSize),
      Element_t()),
    stateIndexPool(
      internal::ReturningTrueIterator(0),
      internal::ReturningTrueIterator(
        (localPoolSize * numThreads * hp.GetRetiredListMaxSize()) +
        (localPoolSize * numThreads) + 1)),
    threadRegistry(0u) {
    bool flag;
    initialStateIndex = stateIndexPool.Allocate(flag);
    DEPENDANT_TYPENAME Node_t::Element initialElement;
    initialElementIndex = elementPool.Allocate(initialElement);
    // Guard sentinel stack node throughout the
    // lifetime of the stack instance:
    hp.GuardPointer(0, initialStateIndex);
    operationArgs = operationArgAllocator.allocate(
      numThreads);
    stackStates = objectStateAllocator.allocate(
      (localPoolSize * numThreads) + 1);
    threadStates = stackThreadStateAllocator.allocate(
      numThreads);
    // Initialize global stack pointer state:
    stackStateIndex.Store(initialStateIndex);
    stackStates[initialStateIndex].head    = initialElementIndex;
    stackStates[initialStateIndex].applied = 0;
    atomicTogglesVector = 0;
  }
  /// Pop method
  bool TryPop(T & retValue) {
    unsigned int tId;
    StackThreadState * threadState = &threadStates[tId];
    int threadBitMask = (1 << tId);
    if ((threadRegistry & threadBitMask) == 0) {
      // Initialize local state for this thread
      initStackThreadState(threadState, tId);
      threadRegistry |= threadBitMask;
    }
    retValue = ApplyOperation(threadState, UndefinedValue, tId);
    if (retValue == UndefinedValue) {
      return false;
    }
    return true;
  }

  bool TryPush(const T & element) {
    unsigned int tId;
    StackThreadState * threadState = &threadStates[tId];
    int threadBitMask = (1 << tId);
    if ((threadRegistry & threadBitMask) == 0) {
      // Initialize local state for this thread
      initStackThreadState(threadState, tId);
      threadRegistry |= threadBitMask;
    }
    ApplyOperation(threadState, element, tId);
    return true;
  }
private:
  /// Returns the pool index of the removed element
  inline ElementPointer_t PopOperation(
    ObjectStateUnpadded * stack,
    StackThreadState * threadState,
    int accessorId) {
    // Do not free elements here as this operation
    // might have to be rolled back
    ElementPointer_t headCurr = stack->head;
    if (headCurr != initialElementIndex) {
      DEPENDANT_TYPENAME Node_t::Element headNode =
        elementPool[static_cast<size_t>(headCurr)];
      stack->ret[accessorId] = headNode.value;
      stack->head = headNode.next;
      return headCurr;
    }
    else {
      // Head is initialElementIndex, so stack is
      // empty:
      stack->ret[accessorId] = UndefinedValue;
    }
    return UndefinedGuard;
  }
  /// Returns the pool index of the added element
  /// to allow rollback of push operations.
  inline ElementPointer_t PushOperation(
    ObjectStateUnpadded * stack,
    StackThreadState * threadState,
    T arg,
    int /* accessorId */) {
    typename embb::containers::internal::WaitFreeSimStackNode<T>::Element n;
    int poolIndex = elementPool.Allocate(n);
    ElementPointer_t elementIndex = static_cast<ElementPointer_t>(poolIndex);
    n.value = arg;
    n.next  = stack->head;
    elementPool[static_cast<size_t>(elementIndex)] = n;
    stack->head = elementIndex;
    return elementIndex;
  }
};
```

Listing B.2: Implementation of the wait-free stack with safe memory reclamation