Multicore Scalability of Concurrent Objects

Dipl. Ing. Hannes Payer, Bakk. techn.
Department of Computer Sciences
University of Salzburg, Austria

Dissertation zur Erlangung des akademischen Grades
Doktor der technischen Wissenschaften
an der Naturwissenschaftlichen Fakultät der Universität Salzburg
unter der Betreuung von
Univ.-Prof. Dipl.-Inform. Dr.-Ing. Christoph Kirsch

Salzburg, July 2012
To my parents.
We study the design, implementation, performance, and scalability of concurrent objects on multicore systems. In the first part we analyze the apparent trade-off between adherence to concurrent data structure semantics and scalability based on a concurrent FIFO queue as running example. We quantitatively relax the sequential specification of a FIFO queue to a $k$-FIFO queue, which can be understood as queue where each element may be dequeued out-of-order up to $k - 1$ or as pool where each element is dequeued within a $k$-bounded number of dequeue operations. We present different $k$-FIFO queue implementations with constant $k$, workload-dependent $k$, and unbounded $k$, and show experimentally that there exist optimal and robust $k$ that result in best performance and scalability. Moreover, we introduce the notion of semantical deviation to quantify the difference in semantics between a run of a queue with relaxed FIFO semantics and a run of a regular FIFO queue on a given workload. We show in an extensive experimental evaluation that our $k$-FIFO queue implementations outperform and outscale many state-of-the-art concurrent queue and pool algorithms on different workloads and relate the performance results to the corresponding measured semantical deviations.

In the second part we focus on concurrent heap management systems. Firstly, we present the design and implementation of a real-time memory allocator called Compact-fit (CF) which provides temporal and spatial guarantees. We analyze the throughput, latency, and scalability versus memory fragmentation trade-off of CF in experiments. Secondly, we introduce hierarchical allocation buffers (HABs), a memory partitioning scheme for high-throughput heap management systems. We present an implementation of HABs in the Hotspot Java virtual machine and analyze the trade-off between throughput and scalability on one hand, and memory fragmentation on the other hand.
CONTENTS

Abstract v

Contents ix

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Concurrent Data Structures . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.2 Concurrent Heap Management Systems . . . . . . . . . . . . . . . . . . 3
1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.4 Chapter Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.4.1 Part I . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.4.2 Part II . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.5 Experimental Hardware Platforms . . . . . . . . . . . . . . . . . . . . 7

1 Concurrent Data Structures 9

2 Incorrect Systems: It’s not the Problem, 11

It’s the Solution 11

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2 Hardware . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.3 Hardware-Software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.4 Software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.4.1 Program Transformations . . . . . . . . . . . . . . . . . . . . . . . . 15
2.4.2 Concurrent Data Structures . . . . . . . . . . . . . . . . . . . . . . 15
2.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
# Contents

## 3 Quantitatively Relaxed Concurrent Data Structures 19

3.1 Introduction ................................................. 19

3.2 $k$-FIFO Sequential Specification .......................... 21

3.3 $k$-FIFO Queue Algorithms ................................. 22

  3.3.1 US $k$-FIFO Queue Algorithm .......................... 25

  3.3.2 Linearizability ......................................... 28

  3.3.3 Lock-freedom .......................................... 30

3.4 Related Work .............................................. 32

3.5 Experiments ............................................... 34

  3.5.1 Understanding $k$ ....................................... 34

  3.5.2 Performance and Scalability ......................... 36

  3.5.3 Dynamic $k$ ........................................... 37

3.6 Other quantitatively relaxed concurrent data structures 38

  3.6.1 $k$-Stack ............................................. 39

  3.6.2 $k$-counter ............................................ 46

3.7 Summary .................................................. 48

## 4 Performance, Scalability, and Semantics of Concurrent FIFO Queues 51

4.1 Introduction ............................................... 51

4.2 Scal Queues ............................................... 53

  4.2.1 Round-Robin Load Balancing .......................... 55

  4.2.2 Randomized Load Balancing ........................... 55

  4.2.3 Hierarchical Load Balancing ......................... 57

  4.2.4 Linearizability ....................................... 57

  4.2.5 Lock-freedom ......................................... 58

4.3 Semantical Deviation ...................................... 58

  4.3.1 Computing $H_L$ ....................................... 62

4.4 Experiments .............................................. 62

  4.4.1 Microbenchmarks ...................................... 63

  4.4.2 Macrobenchmarks ..................................... 67

4.5 Summary .................................................. 69

## II Concurrent Heap Management Systems 71

5 Concurrency and Scalability versus Fragmentation and Compaction with Compact-fit 73
## CONTENTS

5.1 Introduction ................................................. 73  
   5.1.1 Compact-fit ........................................... 76  
   5.1.2 Overview ............................................. 78  
5.2 Related Work .............................................. 80  
5.3 Non-incremental Compact-fit ................................. 82  
5.4 Probabilistic Analysis .................................... 86  
5.5 Incremental Compact-fit .................................... 88  
5.6 Complexity vs. Fragmentation ............................... 94  
5.7 Implementation ............................................ 96  
   5.7.1 Concurrent Non-incremental CF ....................... 97  
   5.7.2 Concurrent Incremental CF ......................... 98  
   5.7.3 Local vs. Global Size-classes ....................... 98  
5.8 Experiments ............................................... 98  
   5.8.1 Hardware Setup .................................... 99  
   5.8.2 Non-concurrent Non-incremental CF ............... 99  
   5.8.3 Concurrent Non-incremental CF ................. 104  
   5.8.4 Concurrent Incremental CF ..................... 107  
5.9 Summary ................................................ 108  

6 Hierarchical Allocation Buffers .............................. 109  
6.1 Introduction ............................................. 109  
6.2 PLABs, CLABs, TLABs ...................................... 111  
6.3 Implementation ........................................ 112  
   6.3.1 Operating System Support ....................... 114  
6.4 Related Work ........................................... 114  
6.5 Experiments ............................................ 115  
6.6 Summary ................................................ 118  

7 Conclusions and Future Work .................................. 119  
Team Contributions ........................................... 121  
Bibliography .................................................. 130
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Concurrent object performance and scalability in number of operations per time unit (throughput) with an increasing number of threads sharing the same concurrent object in an exemplified benchmark scenario</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Maximum achievable speedup of an application on a multicore system by Amdahl's Law with different fractions of parallel code $f$</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Multicore2 Intel-based server machine with four 10-core 2.0GHz Intel Xeon processors (2 hyperthreads per core)</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>“Metrics of Correctness”</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>$k$-FIFO sequential specification</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>Very high and low contention producer-consumer microbenchmarks with an increasing number of $k$ for different amounts of threads on a 40-core (2 hyperthreads per core) server machine</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Variable-load producer-consumer microbenchmarks with an increasing number of static $k$ versus a dynamically controlled $k$ on a 40-core (2 hyperthreads per core) server machine</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>$k$-stack sequential specification</td>
<td>39</td>
</tr>
<tr>
<td>3.6</td>
<td>Producer-consumer microbenchmarks on a 40-core (2 hyperthreads per core) server machine</td>
<td>45</td>
</tr>
<tr>
<td>3.7</td>
<td>Shared counter benchmarks on a 40-core (2 hyperthreads per core) server machine</td>
<td>48</td>
</tr>
<tr>
<td>4.1</td>
<td>Pool sequential specification</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Balancing quality of different random load balancers with increasing number of partial FIFO queues ($p$)</td>
<td>57</td>
</tr>
</tbody>
</table>
4.3 Performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine 64

4.4 Semantical deviation of very high contention producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine 65

4.5 Semantical deviation of low contention producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine 66

4.6 Macrobenchmarks on a random graph with 100000 vertices and 100000000 edges with an increasing number of threads on the Intel-based 40-core server machine 67

4.7 Mandelbrot macrobenchmarks with block size 4x4 pixels and an increasing number of threads on the Intel-based 40-core server machine 68

5.1 Deallocation throughput, system latency, and memory fragmentation with different versions and configurations of Compact-fit 79

5.2 Size-class automaton with $\pi > 1$ 83

5.3 Size-class automaton with $\pi = 1$ 85

5.4 Probability of reaching compaction and worst-case fragmentation 87

5.5 The lifetime of a page 89

5.6 Incremental size-class automaton with $\pi > 1$ 91

5.7 Concurrent CF versions 96

5.8 Increasing allocation size microbenchmark 100

5.9 Rate-monotonic microbenchmark 101

5.10 Fragmentation microbenchmark 103

5.11 Allocation throughput of a single thread with decreasing partial compaction 103

5.12 Allocation throughput with an increasing number of threads 104

5.13 Allocation throughput for Hummingbird and Emacs 105

5.14 Memory usage and size-class fragmentation for Hummingbird and Emacs 106

5.15 Allocation throughput, system latency, and transient size-class fragmentation with decreasing compaction increments 107

6.1 Hierarchical allocation buffers (HABs) with three levels: on the highest level is the global heap, in the middle are the PLABs or CLABs $P1$ and $P2$, and on the lowest level are the TLABs $T1 - T4$ 110
LIST OF FIGURES

6.2 Speedup of using HABs with different TLAB and PLAB/CLAB configurations over the corresponding TLAB-only configurations of the unmodified Hotspot JVM. ........................................ 115

6.3 The data of Figure 6.2 but using the unmodified Hotspot JVM with the TLAB-growing strategy as common baseline. .......................................................... 116
LIST OF TABLES

5.1  Time complexity of malloc and free as well as worst-case system latency, memory size, and size-class fragmentation per CF configuration and size-class .................................................... 95

5.2  Performance: TLSF versus optimized, non-compacting CF (without abstract addressing) ................................................................. 107

6.1  Percentage of foreign allocations with different PLAB and TLAB configurations ............................................................. 117

6.2  Percentage of foreign allocations with different CLAB and TLAB configurations ............................................................. 118
CHAPTER 1

INTRODUCTION

We are living in the multicore era. Parallel computing units (cores) are ubiquitous these days. They can be found in big server machines as well as in small mobile devices. Applications have to be specifically designed to take advantage of the parallelism provided by such multicore systems.

Threads running on different cores of a multicore system may read from and write to concurrent objects through the globally shared memory. Simultaneous access to concurrent objects may have to be synchronized to avoid race conditions [30]. Memory access may become a critical bottleneck if many threads access at a high rate the same concurrent object. The result is degraded performance. Memory access bottlenecks may become even more significant with an increasing number of threads in an application and an increasing number of cores in the multicore system.

The challenge is to build applications that take advantage of parallel cores for better performance. This may be achieved by minimizing or even eliminating memory access bottlenecks. We are interested in designing and implementing concurrent objects that provide high performance and positive scalability on shared memory multicore systems. By performance we mean throughput measured in operations per time unit. Scalability is performance as a function of the number of threads in a system. Figure [L.1] depicts an exemplified benchmark scenario. The ideal result is linear scalability and high performance already with few threads. This is nevertheless an unlikely outcome on multicore systems where shared memory access is typically orders of magnitude slower than core computation. A still challenging yet more realistic outcome and our goal in particular is positive scalability, i.e., increasing performance with an increasing number of threads and high performance already with few threads. Achieving both performance and scalability is important since positive scalability but low performance with few threads may be even worse than negative scalability.
Figure 1.1: Concurrent object performance and scalability in number of operations per time unit (throughput) with an increasing number of threads sharing the same concurrent object in an exemplified benchmark scenario

Applications consist of code that can be executed in parallel and code that must be executed sequentially, e.g. due to synchronization. The scalability of applications is limited by Amdahl’s Law [5], which states that the degree to which we can speed up an application on a multicore system is limited by the fraction of sequential code execution time. Let $f$ be the fraction of parallel code execution time of an application and $n$ be the number of parallel cores of a multicore system then the maximum achievable speedup is $\frac{1}{(1-f)+\frac{f}{n}}$. The maximum achievable speedup by Amdahl’s Law with different values of $f$ and increasing $n$ is depicted in Figure 1.2.

Figure 1.2: Maximum achievable speedup of an application on a multicore system by Amdahl’s Law with different fractions of parallel code $f$

Since operations on concurrent objects may not be fully parallelized there is an intrinsic concurrency bottleneck in many applications using concurrent objects that gets
1. **INTRODUCTION**

Increasingly problematic with the rapidly increasing number of parallel cores. The key to high performance and positive scalability is parallelization with low sequential overhead. There are various trade-offs in application design that incorporate scalability. In this thesis we are going to investigate the trade-off between scalability and correctness in implementing certain concurrent data structures and the trade-off between scalability and memory fragmentation in real-time and high-throughput concurrent heap management systems.

1.1 **Concurrent Data Structures**

Making data structures concurrent typically involves some form of synchronization that relies on lock-based, lock-free, or even wait-free mechanisms. Traditionally, the challenge is to guarantee correctness of concurrent data structure implementations, that is, linearizability \[31\] while still providing scalability. However, even basic linearizable data structure implementations such as FIFO queues have negative scalability under high contention due to synchronization.

It turns out that strict sequential specifications of concurrent data structures are sometimes not needed. Consider, for example, a webserver which stores incoming requests in a shared FIFO queue running on a server machine with possibly hundreds of cores. Requests are dequeued and handled by worker threads at a later point in time. In such a scenario it is often not important to process the requests in perfect FIFO order as long as each element is eventually dequeued. Instead, it may be sufficient if the order in which the requests are handled is FIFO up to a constant that bounds the deviation from FIFO order for fairness.

This topic of the thesis is part of a recent trend towards scalable but semantically weaker concurrent data structures \[71\] acknowledging the intrinsic difficulties of implementing deterministic semantics in the presence of concurrency \[6\]. The idea is to address the multicore scalability challenge by leveraging non-determinism in concurrent data structure semantics for better performance and scalability. We therefore propose to relax the sequential specifications of concurrent data structures for trading off adherence to data structure semantics and scalability.

1.2 **Concurrent Heap Management Systems**

By heap management system \[35\] we refer to explicit or implicit heap management systems that manage the memory of an application heap. An application using an explicit heap management system explicitly calls the corresponding memory allocation
1.2 Concurrent Heap Management Systems

and deallocation methods of the heap management system. An application using an implicit heap management system relies on the heap management system to perform automatic deallocation of unused objects. Implicit heap management systems are also called garbage collected systems.

Memory fragmentation refers to the problem when free memory of the application heap can not be used for memory allocation. External memory fragmentation is the amount of contiguous pieces of memory in the application heap which are smaller than a given memory allocation request. External memory fragmentation can be reduced by performing memory compaction or using a better object placement strategy in the application heap for a given application. Internal memory fragmentation occurs when memory of the application heap is partitioned by the heap management system and parts of the partitioned memory can not be used by memory allocation requests even if they would be large enough. Internal memory fragmentation can be reduced by finding the right partitioning scheme for a given application.

Concurrent heap management systems consist of concurrent objects which hold the information needed to maintain the used and free heap memory. For example, allocated objects and free memory chunks may be organised in concurrent lists for fast retrieval. These concurrent object may have to be implemented efficiently to provide scalability for memory allocation intensive applications.

A key observation is that scalability in concurrent heap management systems may be achieved by reducing the degree of sharing of its concurrent objects by making them thread-, processor-, or core-local. Such a partitioning of the application heap typically results in an increase in internal memory fragmentation. Thread-local concurrent objects are not shared between threads and therefore do not require synchronization. Processor- and core-local concurrent objects are used by the threads running on a given processor or core. They are shared between multiple threads and therefore require synchronization. Contention on them may be lower in comparison to a single globally shared concurrent object since just the threads running on a given processor or core may use them.

In this thesis we present two concurrent heap management systems for two different application scenarios. First, we focus on real-time concurrent heap management systems where heap management operations have to be predictable in time to provide temporal guarantees. Moreover, it may also be required in real-time systems that memory fragmentation is predictable, which means that it can be computed in constant time whether a memory allocation of a given size can be performed. This can be achieved by performing memory compaction. Again, scalability may be achieved
1. INTRODUCTION

by reducing the degree of sharing of concurrent objects. We present in this thesis a memory allocator with memory allocation and deallocation methods predictable in time and predictable memory fragmentation. Furthermore, we discuss an approach for incremental memory compaction of allocated objects where a single object is moved incrementally in memory to reduce latency of the memory compaction operation while maintaining a predictable bound on memory fragmentation.

Next, we consider a garbage collected system [35] for memory allocation intensive applications with high-throughput requirements. High throughput may be achieved by using thread-local [78] or processor-local [26] allocation buffers at the expense of internal memory fragmentation which may increase linearly with the number of threads or processors.

1.3 Contributions

This thesis summarizes some of the publications I co-authored. The contributions of the co-authors are listed in detail in the Team Contributions Section. The following contributions related to concurrent data structures are presented in this thesis:

- We present an overview of some of the state-of-the-art work in the engineering of hardware and software systems where traditional correctness requirements are relaxed, usually for higher performance and lower resource consumption but possibly also for other non-functional properties such as more robustness and less cost. We define different “metrics of correctness” for different correctness relaxations on hardware and software level. The results were published in [40].

- We present the definition of a $k$-FIFO queue sequential specification. Logically, a $k$-FIFO queue can be understood as queue where each element may be dequeued out-of-order up to $k - 1$ or as pool where each element is dequeued within a $k$-bounded number of dequeue operations. We present different $k$-FIFO queue implementations with constant $k$, workload-dependent $k$, and unbounded $k$. We show in an extensive experimental evaluation that our $k$-FIFO queue implementations outperform and out-scale many state-of-the-art concurrent queue and pool algorithms on different workloads. Moreover, to quantify the difference in semantics between a queue with relaxed FIFO semantics and a regular FIFO queue we introduce the notion of semantical deviation as a metric of correctness. Intuitively, when running a given queue implementation on some workload, semantical deviation keeps track of the number of dequeue operations that overtook older
1.4 Chapter Overview

This thesis summarizes some of the publications I co-authored working on research problems related to scalability of concurrent objects on multicore systems.
1. INTRODUCTION

1.4.1 Part I

In Part I of the thesis we discuss the trade-off between performance, and scalability versus correctness in implementing concurrent objects.

In Chapter 2 we motivate the approach of relaxing the correctness requirements of concurrent objects for higher performance and scalability. We discuss different relaxations of hardware and software components and define for each of them a “metric of correctness”.

In Chapter 3 we introduce the sequential specification of a $k$-FIFO queue. We present fast and scalable algorithms that implement bounded- and unbounded-size, lock-free, linearizable $k$-FIFO queues. We show in experiments that our $k$-FIFO queue implementations outperform and out-scale many state-of-the-art concurrent queue and pool algorithms on different benchmarks. Moreover, we present preliminary results of a quantitatively relaxed stack ($k$-stack) and a relaxed shared counter ($k$-counter).

In Chapter 4 we present the Scal framework that implements fast and scalable, lock-free, linearizable $k$-FIFO that provide workload-dependent $k$ (in the number of threads) and unbounded $k$. Hence, Scal queues can be seen as pools. Moreover, we introduce the notion of semantical deviation as a metric of correctness. We show in experiments that the Scal $k$-FIFO queue implementations outperform and out-scale many state-of-the-art concurrent queue and pool algorithms on different micro- and macrobenchmarks.

1.4.2 Part II

In Part II of the thesis we focus on heap management systems and discuss the trade-off between scalability, performance, latency, and memory fragmentation.

In Chapter 5 we present the design and implementation of CF, a real-time heap management system which provides temporal and spatial guarantees. We analyze its throughput, latency, and scalability versus memory fragmentation trade-off in experiments.

In Chapter 6 we introduce HABs, present an implementation of HABs in the Hotspot JVM, and analyze there the trade-off between throughput and scalability versus memory fragmentation.

1.5 Experimental Hardware Platforms

We run the experiments presented in this thesis on different hardware platforms and refer to them using the following abbreviations:
1.5 Experimental Hardware Platforms

- **Embedded1**: a Gumstix Connex board running a XScale PXA 255 CPU with 400MHz and 64MB of memory.
- **Embedded2**: a Gumstix Vertex board running a XScale PXA 270 CPU with 600MHz and 128MB of memory.
- **MultiCore1**: an AMD-based server machine with two 4-core 2.0GHz AMD Opteron processors and 16GB of memory running Linux 2.6.24 with real-time preemption patches applied.
- **MultiCore2**: an Intel-based server machine with four 10-core 2.0GHz Intel Xeon processors (40 cores, 2 hyperthreads per core), 24MB shared L3-cache, and 128GB of memory running Linux 2.6.39. The hardware architecture is depicted in Figure 1.3.

![Figure 1.3: Multicore2 Intel-based server machine with four 10-core 2.0GHz Intel Xeon processors (2 hyperthreads per core)](image-url)
Part I

Concurrent Data Structures
CHAPTER 2

INCORRECT SYSTEMS: IT’S NOT THE PROBLEM, IT’S THE SOLUTION

We present an overview of some of the state-of-the-art work in the engineering of digital systems (hardware and software) where traditional correctness requirements are relaxed, usually for higher performance and lower resource consumption but possibly also for other non-functional properties such as more robustness and less cost. The work presented here is categorized into work that involves just hardware, hardware and software, and just software. In particular, we discuss work on probabilistic and approximate design of processors, unreliable cores in asymmetric multi-core architectures, best-effort computing, stochastic processors, accuracy-aware program transformations, and relaxed concurrent data structures. As common theme we identify, at least intuitively, “metrics of correctness” in each piece of work which appear to be important for understanding the effects of relaxed correctness requirements and their relationship to performance improvements and resource consumption.

2.1 Introduction

We acknowledge the emergence and advocate the study of relaxed, possibly quantitative approaches to describing and establishing the correctness of digital systems. The notion of hardware or systems software either computing the correct result for a given input or not has been the dominating principle in systems engineering for a long time whereas other areas of computer science such as scientific computing and machine learning as well as audio, video, and image processing, to name a few, have adopted relaxed notions of correctness early on. While the promise in special-purpose areas is typically higher performance and lower power consumption at a bounded loss in quality the effect of
2.1 Introduction

relaxed notions of correctness in systems may add other non-functional properties to
the list such as robustness as well as production and development cost. Yet systems
engineering tolerating, beware, incorrect results has up until recently been a rather
secluded topic.

The discrete nature of mathematics relevant in digital systems is probably a promis-
ing factor to look at for an explanation. Clearly, constructing a digital artifact and then
showing that it produces results good enough for general (as opposed to special) pur-
pose rather than results that are simply right or wrong is difficult in the presence of
discrete semantics. Just modelling systems that involve both discrete and continuous
concepts and then arguing about their properties is already a challenge [28]. Yet we feel
that the emergence of relaxed notions of correctness in systems engineering is a sign of
a maturing field taking a turn with new potential for the current and next generation of
computer scientists and engineers. And it is not just about improving robustness and
reducing cost but also about being able to utilize the emerging generations of systems,
some increasingly parallel, using hopefully less energy so that some of the limitations
of traditional design may eventually be overcome.

The common theme of this chapter is to identify “metrics of correctness” in each
example and discuss their properties and possibly ways to obtain quantities in them.
Note that this is an extremely short list of work far from being complete or even
representative. The material cited here should only be seen as a hopefully reasonable
starting point for finding other related work not cited here.

We work with three types of metrics for quantifying quality and performance of
computation as well as error degree of systems whose design has been relaxed such
that they may make mistakes, as shown in Figure 2.1. An error metric quantifies
the degree of errors introduced by a relaxed system. A quality metric quantifies the
computational quality produced by an application running on a relaxed system. We
say that an application is error-tolerant on a relaxed system if the quality it produces
increases whenever the system makes less errors, i.e., if the degree of errors in the
computed results is proportional to the degree of errors introduced by the system.
 Relevant performance metrics are here execution time and power consumption. We say that an application is error-scalable if its performance increases whenever the system may make more mistakes.

### 2.2 Hardware

An important source of complexity in hardware design is reliability of computation, e.g. in arithmetic units and mechanisms such as hardware-based error detection and correction. Unreliable hardware design allows to reduce that complexity potentially providing benefits such as lower production cost, lower test and verification cost, smaller form factors, higher performance, and lower power consumption.

So-called probabilistic design as well as approximate design are two unreliable hardware design principles [48, 49] for trading off reliability of computation and power consumption [61]. The idea of probabilistic design is to develop hardware that produces an output value for a given input value with a certain probability. Experimental data obtained on actual hardware shows that there is a monotone relationship between the probability of correct computation and power consumption [17], suggesting the probability of correct computation as quality metric. Approximate design results in hardware that deterministically produces, for a given input value, an output value that may, however, be incorrect. Similarly, experimental data obtained on actual hardware shows that there is a monotone relationship between arithmetic error and power consumption [18]. Here, the arithmetic error is an obvious candidate for a quality metric.

Unreliable hardware may also help increase hardware parallelism since unreliable hardware may require significantly less space than reliable hardware. An asymmetric multi-core architecture where a small number of reliable cores is combined with a large number of unreliable cores is an example of a design with a higher degree of parallelism than a conventional design of the same size [47]. Intuitively, the number of unreliable cores may be useful as error metric, and inversely even as quality metric if the quality of computations on unreliable cores deteriorates monotonically with the number of unreliable cores.

### 2.3 Hardware-Software

The correctness of conventional software typically relies on hardware that returns deterministic output values for any given input values. Best-effort computing [16, 33] is a system design methodology for taking advantage of combinations of unreliable
hardware and error-tolerant software to gain higher performance and lower power consumption. Errors introduced by unreliable hardware may be tolerated by certain types of software or handled by higher-level software layers. The challenge here is to divide applications into parts that tolerate errors and parts that do not. Such applications can also take advantage of the previously mentioned asymmetric multi-core architectures where application parts that tolerate errors run on unreliable cores [16, 47].

Stochastic processors [57, 70] produce so-called stochastically correct values, as with probabilistic design, through simplified hardware design, which may again enable higher performance and lower power consumption. Higher-level software layers may handle incorrect values either by tolerating the error or by detecting and correcting the error [75].

Control divergence in control-flow graphs makes branch prediction difficult and unreliable and may thus decrease performance. Branch herding performed by hardware or software reduces control divergence by forcing threads to take only a subset of all possible paths through the control-flow graph which may result in higher performance [75]. The degree of branch herding is an error metric whose inverse may also serve as quality metric if eliminating any branch always maintains or increases the error of the output. In this case, the application tolerates branch herding.

Detecting and correcting errors may incur high overhead which may eliminate the performance gains of stochastic processors. Therefore, it might be beneficial to detect just certain types or certain numbers of errors and correct them to stay within given error boundaries [75]. Here an error metric may be the number of errors that get detected and to which degree they get corrected. Again, the inverse may serve as quality metric if the actual error of the output is monotone in the error metric.

A key enabler of stochastic processors may be automatic transformation tools [70, 75] that generate error-tolerant versions out of regular applications, i.e., a transformed application may produce results at a quality that increases, at least within certain boundaries, whenever stochastic processors make less errors.

2.4 Software

Relaxing software specifications may result in higher performance and lower power consumption, and may even increase reliability and robustness of software [67]. We distinguish relaxation techniques based on program transformations and concurrent data structure design.
2. INCORRECT SYSTEMS: IT'S NOT THE PROBLEM, IT'S THE SOLUTION

2.4.1 Program Transformations

For certain types of applications, accuracy-aware program transformations \cite{82} may generate error-tolerant code that may perform better than the original code. Here are three examples.

Substitution transformations \cite{82} replace parts of a program with code that computes approximations of the output computed by the original parts but with less computational overhead. The approximate versions of the code are given and relaxed to different degrees in terms of some error metric. The program is error-tolerant if the approximations are compositional in terms of some quality metric.

Sampling transformations \cite{82} work with code that computes output from a given set of elements. The transformed code performs the same computation as the original code but only on a subset of the elements obtained by some sampling policy. The code is error-tolerant if the quality of the result improves with larger subsets.

Loop perforation \cite{73} is another type of program transformation which transforms a loop into a new loop where only a subset of the original loop iterations is performed. Decreasing the subset size of loop iterations decreases the runtime of executing the loop but also increases the error of the loop output.

2.4.2 Concurrent Data Structures

Concurrent data structures require synchronization to implement the exact specification of their sequential counterpart in a concurrent environment. However, synchronization operations may incur high overhead and prevent program code from executing in parallel. We discuss two approaches which may decrease synchronization overhead and increase parallelism, either by reducing contention on synchronization bottlenecks or by eliminating synchronization operations entirely, at the expense of adherence to exact data structure semantics. As a result, relaxed versions of concurrent data structures emerge which may perform and scale better on increasingly parallel hardware, and still be tolerated by certain applications.

In terms of performance and scalability, the goal is to achieve throughput in terms of number of data structure operations per second that is higher than of conventional designs (high performance) and grows with the number of concurrent units such as threads, for example, sharing the same data structure, to more threads than with conventional designs (positive scalability). Both, negative scalability even with high performance for low numbers of threads and positive scalability but with low performance are undesirable.
2.4 Software

Reducing Contention on Synchronization Bottlenecks

Our own work presented in detail in Chapter 3 and Chapter 4 is on relaxing the semantics of concurrent data structures by reducing contention on synchronization bottlenecks. We achieve that by relaxing the sequential specification of a concurrent data structure. Consider, for example, a regular first-in-first-out (FIFO) queue where elements are enqueued at the queue tail and dequeued at the queue head. The problem with this specification is that it leaves little room for optimization in concurrent implementations [55, 27] so that scalability in the presence of high contention on the queue may still be limited to relatively low numbers of threads. Instead of maintaining the original specification of a FIFO queue, we propose to relax the specification to what we call a $k$-FIFO queue with $k \geq 1$, which may dequeue elements out of FIFO order up to $k - 1$. We refer to the introduced incorrectness as semantical deviation. Intuitively, the semantical deviation keeps track of the number of dequeue operations necessary to return oldest elements and the age of dequeued elements. Hence for the concurrent data structures with relaxed sequential specifications the error metric is semantical deviation. Another interesting aspect of semantical deviation is the problem of measuring it. Clearly, improved performance and scalability comes at the expense of increased semantical deviation, which may or may not be tolerated by applications using queues.

Eliminating Synchronization Bottlenecks

Synchronization bottlenecks can be eliminated by eliminating the corresponding synchronization operations. This approach may lead to race conditions of which some may result in effects such as data duplication or loss which may nevertheless still be tolerated by certain applications.

Idempotent work-stealing queues are distributed queues, one per thread, where a thread may either dequeue an element from its local queue without synchronization or dequeue (steal) an element from the queue of another thread with synchronization [56]. Queue elements may be returned multiple times instead of just once because of races between unsynchronized local dequeue and synchronized global steal operations. Intuitively, the number of races or, even more accurate, the amount of element duplication may be useful as error metric and inversely even as quality metric if the quality of computation deteriorates monotonically with element duplication. Moreover, error and quality metric may also be related to the number of involved threads since a larger number of threads increases the probability of races.

Another example of in fact full elimination of synchronization is a space-subdivision tree construction algorithm that does not use any synchronization operations and yet
provide a well-defined and consistent tree state that may be good enough for some applications [68]. The race conditions that may occur result in subtree losses which reduces the amount of data held by the tree. Similar to the idempotent work-stealing queues, the number of races or, again even more accurate, the amount of subtree loss may serve as a useful error metric and inversely as a quality metric if the quality of computations deteriorates monotonically with subtree loss. Again, both error and quality metric may also be monotone in the number of involved threads since more threads make races more likely.

2.5 Summary

Relaxed and possibly quantitative notions of correctness are likely to play an increasingly important role in systems engineering. We believe that one of the key challenges is identifying “metrics of correctness” such that the effects of more errors gracefully, as opposed to abruptly, degrade the quality of a system and yet translate into higher performance and lower resource consumption. The idea seems to apply in virtually any area of systems engineering.
There is a trade-off between performance and correctness in implementing concurrent data structures. Better performance may be achieved at the expense of relaxing correctness, by redefining the semantics of data structures. We address such a redefinition of data structure semantics and introduce based on a concurrent FIFO queue as running example fast and scalable algorithms that implement bounded- and unbounded-size, lock-free, linearizable $k$-FIFO queues with empty (and full) check [39]. Logically, a $k$-FIFO queue can be understood as queue where each element may be dequeued out-of-order up to $k - 1$ or as pool where each element is dequeued within a $k$-bounded number of dequeue operations. We show experimentally that there exist optimal and robust $k$ that result in best performance and scalability. We then demonstrate that our algorithms outperform and outscale many state-of-the-art concurrent queue and pool algorithms on different workloads. Moreover, we demonstrate a prototypical controller which aims at identifying optimal $k$ automatically at runtime for best performance. Finally, we demonstrate that quantitative relaxations may be also applied to other concurrent data structures such as stacks and shared counters for better performance and scalability [29].

3.1 Introduction

Concurrent data structures may be a performance and scalability bottleneck and thus prevent effective use of increasingly parallel hardware [71]. There is a trade-off between adherence to concurrent data structure semantics and scalability. A way to improve scalability and performance of concurrent data structures is to relax their se-
mantics. The semantics is given by some notion of equivalence with sequential behavior. The equivalence is determined by a consistency condition, most commonly linearizability \[31\], and the sequential behavior is inherited from the sequential version of the data structure. Therefore, relaxing the semantics of a concurrent data structure amounts to either weakening the consistency condition (linearizability being replaced with sequential consistency or quiescent consistency) or redefining (relaxing) its sequential specification. Our approach is based on quantitatively relaxing the sequential specification of concurrent data structures. In \[29\] we introduced a systematic and formal framework for obtaining new data structures by quantitatively relaxing existing ones.

We introduce two algorithms that implement bounded-size (BS) and unbounded-size (US), lock-free, linearizable $k$-FIFO queues with empty (and full) check. Logically, enqueueing an element into a $k$-FIFO queue appends the element at the end of the queue. A $k$-FIFO queue can then be understood as queue where each element may be dequeued out-of-order up to $k-1$ or as pool where each element is dequeued within a $k$-bounded number of dequeue operations. Thus a 1-FIFO queue is a regular FIFO queue and a $k$-FIFO queue provides bounded fairness \[21\] for finite $k$. The details are discussed in Section 3.2.

The BS algorithm maintains a bounded-size array of elements that is dynamically partitioned into segments of size $k$ while the US algorithm maintains an unbounded list of segments. Thus up to $k$ enqueue and $k$ dequeue operations may be performed in parallel. The US algorithm simplifies for $k=1$ to an algorithm that implements a lock-free FIFO queue similar to the lock-free Michael-Scott FIFO queue (MS) \[55\] but without a sentinel node. The details are discussed in Section 3.3. Logically, the idea of segments is similar to the idea of segments in the Segment Queue (SQ) \[4\]. On data structure (but not code) level SQ is related to the US algorithm, which improves upon SQ in performance and scalability through less overhead and reduced contention and in features through a linearizable empty check. We discuss the relation to SQ in detail in Section 3.4.

In Section 3.5 before presenting a detailed performance analysis of our algorithms relative to a variety of concurrent FIFO queue and pool algorithms, we show experimentally that there exist optimal and robust $k$ that result in best performance and scalability. Interestingly, performance generally increases with $k$ but only up to a certain point which is determined by a tradeoff between degree of parallelism and management overhead. Our algorithms outperform and outscale all other algorithms that we considered in almost all threading and contention scenarios. Furthermore, we discuss a prototypical controller that adjusts $k$ dynamically and automatically at runtime outperforming
### 3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

**enqueue**

\[ enqueue_k(e)(q, o) = (q \cdot e, o) \quad \text{(L1)} \]

**dequeue**

\[
\text{dequeue}_k(e)(q, o) = \begin{cases} 
(q', o') & \text{if } e = \text{null}, q = \varepsilon \\
(q', o') & \text{if } q = e_1 \ldots e_{i-1} e_i e_{i+1} \ldots e_n, \\
q' = e_1 \ldots e_{i-1} e_{i+1} \ldots e_n, & \text{if } e = e_i, \\
1 \leq i \leq \min(k - o(e_1), n), & \text{if } o' = F(o, i) \\
\text{error} & \text{otherwise} \end{cases} \quad \text{(L2-L6)}
\]

\[ F(o, i)(e_j) = \begin{cases} 
o(e_j) & \text{if } j \geq i \\
o(e_j) + 1 & \text{otherwise} \end{cases} \quad \text{(L5-L6)}
\]

Figure 3.1: \( k \)-FIFO sequential specification

any static and manual \( k \) configuration on the workload that we considered.

In Section 3.6 we discuss preliminary results of applying quantitative relaxations to a concurrent stack (\( k \)-stack) and a shared counter (\( k \)-counter). We show in experiments that the \( k \)-stack and \( k \)-counter implementations outperform and outscale all other algorithms that we considered in our performance evaluation.

#### 3.2 \( k \)-FIFO Sequential Specification

We define a sequential specification of the \( k \)-FIFO queue algorithms to facilitate a high-level discussion here and a linearizability argument in Section 3.3. The formal definition of the sequential specification is shown in Figure 3.1. Similar to a regular FIFO queue, a \( k \)-FIFO queue provides an enqueue and a dequeue operation.

Let the tuple \((q, o)\) denote the state of a \( k \)-FIFO queue where \( q \) is the sequence of \( n \) queue elements with \( n \geq 0 \) and \( o \) is a map from queue elements \( e_i \in q \) to counters \( o(e_i) \geq 0 \) indicating how often a queue element was overtaken by younger elements. We refer to \( o \) as the lateness map and to the counters \( o(e_i) \) as lateness counters. The initial, empty state of the \( k \)-FIFO queue is \((\varepsilon, o)\), where \( o \) is an empty map.

The enqueue operation is a function from queue states and queue elements to queue states. It logically inserts an element at the end of the queue. The lateness counter of an enqueued item is initially set to zero. The dequeue operation is a function from queue states and queue elements or the null return value, which indicates an empty queue, to queue states. It logically removes an element \( e_i \) among the elements \( e_1 \ldots e_n \) from
3.3 $k$-FIFO Queue Algorithms

The queue where $1 \leq i \leq \min(k - o(e_1), n)$ if the queue is not empty, and otherwise returns null. The lateness map $o$ is maintained with every enqueue operation and updated with every dequeue operation that returns an element $e_i$ by incrementing the lateness counters $o(e_j)$ of the elements $o_j \in q$ with $j < i$. Hence, the lateness counters decrease monotonically from $e_1$ to $e_n$. Intuitively, with a $k$-FIFO queue up to $k - 1$ younger elements may be dequeued before a given element, where each dequeued element is not younger than the $k$-oldest element, and null is only returned when the queue is empty. In particular, dequeueing an element $e_i$ from a $k$-FIFO queue $e_1 \ldots e_n$ with $k > 0$ may take anywhere between $l$ and $u$ dequeue operations that may return the elements $e_1 \ldots e_{i-1}e_{i+1} \ldots e_{u+1}$ where $l = \lfloor \frac{i}{k} \rfloor \cdot k$ and $u = \min(l + k, n) - 1$.

Thus a 1-FIFO queue is a regular FIFO queue and a $k$-FIFO queue provides bounded fairness [21] for finite $k$. For brevity we use the terms enqueue and dequeue with queues as well as pools.

We present the algorithms of the lock-free bounded-size (BS) and unbounded-size (US) $k$-FIFO queues for $k > 0$. The pseudo code of the algorithms is depicted in Listing 3.1 and Listing 3.2. The occurrence of the ABA problem is made unlikely through version numbers. We refer to values enhanced with version numbers as atomic values. The gray highlighted code is only used in the BS version. We present the general idea of the algorithms followed by a detailed discussion of the BS algorithm. We then discuss the US algorithm by outlining its differences to the BS algorithm and finally show informally linearizability with respect to the $k$-FIFO sequential specification.

When implementing a linearizable FIFO queue the head and tail pointers may become scalability bottlenecks. The principle idea of the BS and US $k$-FIFO queue algorithms is to reduce contention on the head and tail pointers by maintaining an array (BS) or a list (US) of so-called segments each consisting of $k$ slots, instead of maintaining an array or list of plain queue elements. A slot may either point to null indicating an empty slot or may hold a so-called item, which is our implementation concept for enqueued elements. An enqueue operation is served by the tail segment and a dequeue operation is served by the head segment. Hence, up to $k$ enqueue and $k$ dequeue operations may be performed in parallel.

The BS $k$-FIFO queue algorithm is based on an array of atomic values of a given size. For simplicity we restrict size to be a multiple of $k$. The queue tail and queue head pointers are also atomic values. Both initially point to the slot at index zero.

The enqueue method is depicted in Listing 3.1. Given an item representing an
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

Listing 3.1: Enqueue operation of lock-free bounded-size (BS) and unbounded-size (US) $k$-FIFO queue algorithms. Gray highlighted code is only used in the BS version

```c
bool enqueue(item):
    while true:
        tail_old = get_tail();
        head_old = get_head();
        item_old, index = find_empty_slot(tail_old, k, TESTS);
        if tail_old == get_tail():
            if item_old.value == EMPTY:
                item_new = atomic_value(item, item_old.version + 1);
                if CAS(&tail_old->segment[index], item_old, item_new):
                    if committed(tail_old, item_new, index):
                        return true;
                    else:
                        if tail_old.value + k == head_old.value:
                            if segment_not_empty(head_old, k):
                                if head == get_head():
                                    return false;
                            else:
                                advance_head(head_old, k);
                                advance_tail(tail_old, k);
                return true;
            else:
                if tail_old->segment[index] != item_new:
                    head_current = get_head();
                    tail_current = get_tail();
                    item_empty = atomic_value(EMPTY, item_new.version + 1);
                    if in_queue_after_head(tail_old, tail_current, head_current):
                        return true;
                    else if not_in_queue(tail_old, tail_current, head_current):
                        if !CAS(&tail_old->segment[index], item_new, item_empty):
                            return true;
                        else: //in queue at head
                            head_new = atomic_value(head_current.value, head_current.version + 1);
                            if CAS(&head, head_current, head_new):
                                return true;
                            if !CAS(&tail_old->segment[index], item_new, item_empty):
                                return true;
                    return false;
```

element to be enqueued, the method returns `true` when the `item` is successfully inserted and `false` when the queue is full. First the method tries to find an empty slot in the tail segment which is located in between the indices [tail, tail + k] using the `find_empty_slot` method (line 5). The `find_empty_slot` method randomly selects an index in between [tail, tail + k] and then linearly searches at most TESTS ≥ 1 array locations for an empty slot starting with the selected index wrapping around at index
3.3 \( k \)-FIFO Queue Algorithms

In our experiments we observed that \( \text{TESTS} = k \) results in the best performance. Afterwards the \texttt{enqueue} method checks if the \( k \)-FIFO queue state has been consistently observed by checking whether \texttt{tail} changed in the meantime (line 6) which would trigger a retry. If an empty slot is found (line 7) the method tries to insert the \texttt{item} at the location of the empty slot using a compare-and-swap (CAS) operation (line 9). If the insertion is successful the method verifies whether the insertion is also valid by calling the \texttt{committed} method (line 10), as discussed below. If any of these steps fail a retry is performed. If no empty slot is found in the current tail segment the \texttt{enqueue} method tries to increment \texttt{tail} by \( k \) using CAS (line 19) and then retries. Hence in the worst-case only \( \text{TESTS} \) slots may be used in a segment if \( \text{TESTS} < k \). If \texttt{tail} cannot be incremented without overtaking \texttt{head} (line 13) and the segment to which \texttt{head} points is empty (line 14) the method tries to increment \texttt{head} by \( k \) using CAS (line 18). If this segment is not empty and \texttt{head} did not change in the meantime (line 15) the queue is full and \texttt{false} is returned (line 16).

The \texttt{committed} method (line 21) validates an insertion. It returns \texttt{true} when the insertion is valid and \texttt{false} when it is not valid. An insertion is valid if the inserted item already got dequeued at validation time by a concurrent thread (line 22, 30, 39) or the tail segment where the item was inserted is in between the current head segment and the current tail segment but not equal to the current head segment (line 27). If the tail segment where the item was inserted is not in between the current head segment and the current tail segment (line 29) the method tries to undo the insertion using CAS (line 30). If the tail segment where the item was inserted is equal to the current head segment a race with concurrent dequeueing threads may occur which may not have observed the insertion and may try to advance the \texttt{head} pointer in the meantime. This would result in loss of the inserted item. To prevent that the method tries to increment the version number in the \texttt{head} atomic value using CAS (line 34). If this fails a concurrent dequeue operation may have changed \texttt{head} which would make the insertion potentially invalid. Hence after that the method tries to undo the insertion using CAS (line 36). False is returned (line 38) if the insertion was undone.

The \texttt{dequeue} method is depicted in Listing 3.2. It returns an \texttt{item} if the queue is not empty, and returns \texttt{null} if the queue is empty. Similarly to the \texttt{enqueue} method the \texttt{dequeue} method first tries to find an item in between the indices \([\texttt{head}, \texttt{head} + k]\) using the \texttt{find_item} method (line 43). The \texttt{find_item} method randomly selects an index in between \([\texttt{head}, \texttt{head} + k]\) and then linearly searches for an item starting with the selected index wrapping around at index \(\texttt{head} + k - 1\). Afterwards the \texttt{dequeue} method checks if the queue state has been consistently observed by checking whether
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

Listing 3.2: Dequeue operation of lock-free bounded-size (BS) and unbounded-size (US) $k$-FIFO queue algorithms

```c
item dequeue():
  while true:
    head_old = get_head();
    item_old, index = find_item(head_old, k);
    tail_old = get_tail();
    if head_old == get_head():
      if item_old.value != EMPTY:
        if head_old.value == tail_old.value:
          advance_tail(tail_old, k);
          item_empty = atomic_value(EMPTY, item_old.version + 1);
          if CAS(&head_old[index], item_old, item_empty):
            return item_old.value;
          else:
            if head_old.value == tail_old.value:
              return null;
          advance_head(head_old, k);
```

head changed in the meantime (line 45) which would trigger a retry. If an item was found (line 46) the method first checks whether head equals tail (line 47). If this is the case the method tries to increment tail by $k$ to prevent starvation of items in the queue and to provide a linearizable empty check. Afterwards the method tries to remove the item using CAS (line 50) and returns it if the removal was successful (line 51). Otherwise a retry is performed. If no item is found, head equals tail, and tail did not change in the meantime null is returned indicating an empty queue (line 54). Otherwise the method tries to increment head by $k$ (line 55) and performs a retry.

Note that to hold $n$ items the BS $k$-FIFO queue has to consist of at least $\lceil \frac{n}{\text{TESTS}} \rceil \times k + k$ atomic values. The additional segment is necessary to avoid that enqueue and dequeue operations operate on the same segment which prohibits starvation of items in the head segment and therefore provides bounded lateness $k$.

3.3.1 US $k$-FIFO Queue Algorithm

The US $k$-FIFO queue algorithm differs from the BS version in the implementation of the committed, advance_tail, and advance_head methods. The gray highlighted code in Listing 3.1 is not used in the US version since there is no full state. Hence the enqueue method always returns true.

The US $k$-FIFO queue algorithm implements a FIFO queue of segments, called segment queue. An enqueue operation is served by the tail segment. When this segment is full a new segment is added to the tail. A dequeue operation is served by the head
3.3 \textit{k-}\textit{FIFO} Queue Algorithms

Listing 3.3: Segment queue algorithm

```c
1  global head;
2  global tail;
3
4  void init():
5    new_ksegment = calloc(sizeof(ksegment));
6    head = atomic_value(new_ksegment, 0);
7    tail = atomic_value(new_ksegment, 0);
8
9  atomic_value get_head():
10    return head;
11
12  atomic_value get_tail():
13    return tail;
14
15  void advance_tail(atomic_value tail_old):(0)
16    next_ksegment = tail_old->next;
17    if tail_old == get_tail():
18      if next_ksegment.value != null:
19        tail_new = atomic_value(next_ksegment.value, tail_old.version + 1)
20        CAS(&tail, tail_old, tail_new);
21      else:
22        ksegment = calloc(sizeof(ksegment));
23        new_ksegment = atomic_value(ksegment, next_ksegment.version + 1)
24        if CAS(&tail_old->ksegment.next, next_ksegment, new_ksegment):
25          tail_new = atomic_value(ksegment, tail_old.version + 1)
26          CAS(&tail, tail_old, tail_new);
27
28  void advance_head(atomic_value head_old):
29    head_next_ksegment = head_old->next;
30    if head_old == get_head():
31      tail_current = get_tail();
32      tail_next_ksegment = tail_current->next;
33      if head_old.value == tail_current.value:
34        if tail_next_ksegment.value == null:
35          return;
36        CAS(&tail, tail_current, tail_next_ksegment);
37        head_old->deleted = true;
38        head_new = atomic_value(head_next_ksegment.value, head_old.version + 1);
39        CAS(&head, head_old, head_new);
```

segment. When this segment is empty it is removed from the segment queue except if it is the only segment in the queue.

Any FIFO queue algorithm may be used to implement the segment queue. We developed a lock-free FIFO queue which performs better than the lock-free Michael-Scott FIFO queue (MS) \cite{55} when used as segment queue in the US \textit{k-}\textit{FIFO} queue algorithm. We implemented the segment queue such that there is always at least one
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

segment, even if it is empty, in the queue, which enables fast and direct access to the head and tail segments. The pseudo code of our segment queue is shown in Listing 3.3.

The `advance_tail` method adds a new segment to the segment queue if `tail` did not change in the meantime, i.e., `tail` still points to `tail_old`. The `advance_head` method removes the head segment if it is not the only segment in the queue and if it did not change in the meantime, i.e., `head` still points to `head_old`. Before removing the head segment it marks that segment as deleted. The `committed` method in the US version is similar to the BS version. It decides based on whether the current head segment is marked as deleted or not if it has to undo the insertion, instead of performing a range check.

Listing 3.4: Unbounded-size (US) 1-FIFO queue algorithm (get_head, get_tail, advance_head, and advance_tail are implemented by the pseudo code of the segment queue in Listing 3.3 with \( k = 1 \))

```c
bool enqueue(item):
    while true:
        tail_old = get_tail();
        head_old = get_head();
        item_old = tail_old->item;
        if tail_old == get_tail():
            if item_old.value == EMPTY:
                item_new = atomic_value(item, item_old.version + 1);
                if CAS(&tail_old->item, item_old, item_new):
                    return true;
            else:
                advance_tail(tail_old);
        item dequeue():
        while true:
            head_old = get_head();
            item_old = head_old->item;
            tail_old = get_tail();
            if head_old == head:
                if item_old.value != EMPTY:
                    item_new = atomic_value(EMPTY, item_old.version + 1);
                    if CAS(&head_old->item, item_old, item_new):
                        return item_old.value;
                else:
                    if head_old.value == tail_old.value:
                        return null;
            advance_head(head_old);
```

The US algorithm simplifies for \( k = 1 \) to an algorithm that implements a lock-free FIFO queue similar to the MS algorithm but without a sentinel node. In particular, the empty US 1-FIFO queue contains an empty 1-segment in which the first enqueued
element is stored, in contrast to MS. Subsequent dequeue operations may lead to a queue with an empty 1-segment at the head but only because 1-segments are removed lazily (which may also be done eagerly avoiding empty 1-segments altogether in a non-empty queue). The pseudo code is depicted in Listing 3.4.

### 3.3.2 Linearizability

We now prove that the $k$-FIFO queue implementation is correct for the relaxed FIFO semantics depicted in Figure 3.1.

**Proposition 3.3.1.** The $k$-FIFO queue algorithm is linearizable with respect to the $k$-FIFO sequential specification.

**Proof.** The linearization point [31] of the enqueue method that inserts an item is the successfully executed CAS in line 9 if the committed method in line 10 subsequently returns true. In the BS version, the queue may be full. The linearization point of the full check is in line 5 if the tail segment was found to be full and subsequently $\text{tail} + k$ equals $\text{head}$ (line 13), an item was found in the head segment (line 14) and the head pointer did not change in the meantime (line 15).

The linearization point of the dequeue method that returns an item is the successfully executed CAS in line 50. The linearization point of the empty check is in line 43 if no item was found in the head segment and subsequently $\text{head}$ did not change in the meantime (line 45) and points to $\text{tail}$ (line 53).

The BS and US $k$-FIFO queue algorithms are linearizable with respect to the $k$-FIFO sequential specification if for each obtained concurrent history [31] there is a sequential history in which concurrent methods are ordered according to their linearization points and these linearization points are within the $k$-FIFO sequential specification. This is equivalent to requiring that for the linearization point of a dequeue method, the value of the item must satisfy the $k$-FIFO sequential specification.

Without loss of generality, we assume that each enqueued element is unique. A segment $s'$ is reachable from a segment $s$ if either $s'=s$ or $s'$ is reachable from $s->\text{next}$. An item $i$ is in the $k$-FIFO queue, if $\text{enqueue}(i)$ has already committed and there exists a segment reachable from the head segment containing a slot whose value is $i$. Note that reachability is important, i.e., only having a slot containing the item is not enough to guarantee that the item is logically in the $k$-FIFO queue, because the slot could be on a segment which already got removed by a concurrent thread.

The correctness argument is based on the following facts.

1. An item is enqueued exactly once. This is a consequence of our unique-items
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

assumption and the control flow of \texttt{enqueue}(i), the only method that can modify a slot to contain \texttt{i}.

2. An item is dequeued at most once. If an item \texttt{i} is in the \(k\)-FIFO queue, then it can only be removed once, because of 1. and the existence of a unique statement which replaces \texttt{i} with \texttt{EMPTY}.

3. If in the BS \(k\)-FIFO queue an enqueue method returns \texttt{false}, then during its execution, there must be a state where at least \(n\) items have been enqueued successfully in the BS \(k\)-FIFO queue consisting of \(\lceil \frac{n}{\text{TESTS}} \rceil \times k + k\) atomic values. Since returning \texttt{false} is without any side-effect, it suffices to prove the existence of that logically full state. If all \(\lceil \frac{n}{\text{TESTS}} \rceil \) + 1 segments are in use then in all segments except for the head segment there must be \texttt{TESTS} items. The call to \texttt{segment not empty} is only performed if no empty slot was found in the tail segment, \texttt{tail} + \(k\) equals \texttt{head}, and \texttt{tail} still points to the same segment after that. \texttt{head} is read before checking the slots in tail segment. If the head segment contains elements and \texttt{head} did not change after that check, then \texttt{tail} cannot not be advanced. Hence, the BS \(k\)-FIFO queue is full.

4. If a dequeue method returns \texttt{null}, then during its execution, there must be a state at which there are no items in the \(k\)-FIFO queue. Since returning \texttt{null} is without any side-effect, it suffices to prove the existence of a state which corresponds to a logically empty queue. A slot in a segment is used exactly once for insertion, since either a dequeue method adds a new segment to the \(k\)-FIFO queue if \texttt{head} points to \texttt{tail} or an enqueue method adds a new segment to the \(k\)-FIFO queue if it encounters a full tail segment. Hence, if \texttt{find item} did not find an item in the head segment and if \texttt{head} still points to \texttt{tail} after calling \texttt{find item} then the head segment is still the only used segment and \texttt{find item} observed the empty state.

5. An item \texttt{j} cannot be dequeued before an item \texttt{i}, if they are both in the \(k\)-FIFO queue, and \texttt{i}, \texttt{j} are in segments \(s\), \(s'\), respectively, with \(s'\) reachable from \(s\) and \(s' \neq s\). The segment \(s'\) can become the head segment only after the segment \(s\) has been removed by some dequeue operation. Moreover, if a segment becomes unreachable from the head segment, it remains unreachable. These two observations imply that the linearization point of \texttt{dequeue}(i) must precede the linearization point of \texttt{dequeue}(j) which can only happen when \(s'\) is a head segment.

6. An item \texttt{i} in the \(k\)-FIFO queue is dequeued only if it is one of the \(k - l\) oldest items in the \(k\)-FIFO queue, where \(l\) is the current lateness of the oldest item. By oldest we mean least recently enqueued. Lateness of an item is the number of dequeue operations that returned items younger than the given item. Assume \texttt{i} is dequeued from the \(k\)-FIFO queue at the current moment in time and at that point the oldest
item is $t$ with current lateness $l$. This means that ever since $t$ is the oldest item in the $k$-FIFO queue, there have been $l$ dequeue operations performed none of which removed $t$. Let $j$ be any of these $l$ dequeued items. Since $t$ is the oldest item in the $k$-FIFO queue it is in the head segment. Since $j$ is removed before $t$, by 5, it must have also resided in the head segment. For the same reason, also $i$ is in the head segment prior to its removal. Hence, at the moment in which $\text{dequeue}(i)$ happens, there are at most $k - l$ items in the head segment.

Now, 5. shows that the sequential behavior obtained by ordering methods according to their linearization points satisfies the $k$-FIFO queue sequential specification. Thus, any concurrent execution generated by the given algorithm is linearizable with respect to the $k$-FIFO sequential specification. Observe that already 1. and 2. show that the $k$-FIFO queue has pool semantics.

\begin{proof}

\end{proof}

### 3.3.3 Lock-freedom

An algorithm is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps [30]. This liveness property ensures that at least one thread in a system makes progress at any time. We show informally that the BS and US $k$-FIFO queue algorithms are lock-free, c.f. [55].

**Proposition 3.3.2.** The BS and US $k$-FIFO queue algorithms are lock-free.

**Proof.** In the BS and US $k$-FIFO queues an enqueue operation only loops in the following cases:

- The condition in line 6 fails. This is the case if $\text{tail}$ is changed concurrently by some other thread in line 19, after the tail segment has been observed as full, or in line 48, to prohibit starvation of items in the head segment. A retry that happens because other threads observed the tail segment as full (and thus did advance $\text{tail}$) does not cause starvation, because $\text{tail}$ would have to be fixed in the local thread anyways (causing a retry). Both types of retries are bounded by one loop iteration. If $\text{tail}$ changes subsequently, other threads were able to perform enqueue or dequeue operations.

- The condition in line 7 never becomes $\text{true}$. This is the case if the segment is always full, meaning that other threads did succeed in enqueueing items.

- The inserting $\text{CAS}$ in line 9 fails. This is the case if either the item or the version number of the item in the slot changed. The item changes if some other thread was
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

able to concurrently enqueue an item, indicating overall progress. Furthermore the CAS also fails (due to the version number) if an item has been concurrently enqueued and dequeued in this slot. This is either the case if some threads have been faster and concurrently performing an enqueue and dequeue operation, leaving the item back as null, or some enqueue operation had to be undone (see next point). In both cases the system makes progress.

• The committed method returns false in line 10. This is the case if an operation successfully enqueued an item, and then successfully undid the operation by removing it. The undo step is required if both, head and tail, have been moved while an item has been inserted and the inserted item is not in the range of those two pointers. However, moving head and tail is only done while enqueueing and dequeueing items, indicating overall progress.

• (Only BS.) The condition in line 13 is true and the one following in line 14 is false. This is the case if the queue is observed as full while the head segment is empty. It is then necessary to make this head segment usable again (for insertion) by advancing head. This can only be a condition for starvation if other threads succeed in enqueueing and dequeueing items.

In the BS and US k-FIFO queues a dequeue operation only loops in the following cases:

• The condition in line 45 fails. This is the case if head is changed concurrently by some other thread in line 18 (BS only) or line 55, after observing that the head segment is empty. A retry that happens because other threads observed the head segment as empty (and thus did advance head) does not cause starvation, because head would have to be fixed in the local thread anyways (causing a retry). This type of retry is bounded by one loop iteration. If head changes subsequently, other threads were able to dequeue items.

• The dequeueing CAS in line 50 fails. This is the case if either the item or the version number of the item in the slot changed. The item changes if some other thread was able to concurrently dequeue an item, indicating overall progress. Furthermore the CAS may also fail (due to a different item, or same item with different version number) if a dequeue and enqueue operation happen concurrently.

• The condition in line 53 fails. This is the case if some other thread observed the head segment as empty. The condition then only fails if head is not tail or
tail changed. This means that the pointer to the empty head segment can be
advanced, which is bounded by one round and would otherwise have to be done
in the local thread.

Hence, whenever an enqueue or dequeue operation has to retry, another enqueue or
dequeue operation was processed.

3.4 Related Work

We relate our BS and US $k$-FIFO queue algorithms to existing concurrent data structure
algorithms, which we also implemented and evaluated in a number of experiments in
Section 3.5.

The following queues implement regular unbounded-size FIFO queues: a standard
lock-based FIFO queue (LB), the lock-free Michael-Scott FIFO queue (MS) [55], and
the flat-combining FIFO queue (FC) [27] (FC). LB locks a mutex for each data structure
operation. With MS each thread uses at least two CAS operations to insert an element
into the queue and at least one CAS operation to remove an element from the queue.
FC is based on the idea that a single thread performs the queue operations of multiple
threads by locking the whole queue, collecting pending queue operations, and applying
them to the queue. The lock-free bounded-size FIFO queue (BS) [1] is based on an
array of fixed size where elements get inserted and removed circularly and enqueue
operations may fail when the queue is full, i.e. every array slot holds an element.

The Random Dequeue Queue (RD) [4] implements the $k$-FIFO sequential specifi-
cation where $k = r$ and $r$ defines the range $[0, r - 1]$ of a random number. RD is based
on MS where the dequeue operation was modified in a way that the random number
determines which element is returned starting from the oldest element. RD does not
scale better than MS in experiments reported on elsewhere [4].

The Segment Queue (SQ) [4] is similar to the US $k$-FIFO queue but without a lin-
earizable empty check. Hence SQ may return empty even if the queue is not empty. SQ
is implemented by a lock-free FIFO queue of segments but here the queue of segments
may become empty. A segment can hold $s$ queue elements. An enqueue operation in-
serts an element at an arbitrary position of the youngest segment. A dequeue operation
removes an arbitrary element from the oldest segment. When a segment becomes full
a new segment is added to the queue. When a segment becomes empty it is removed
from the queue. A thread performing a dequeue operation starts looking for an element
in the oldest segment. If the segment is empty it is removed and the thread checks
the next oldest segment and so on until it either finds an element and returns that, or
else may return null. SQ does not perform better than MS in experiments reported on
elsewhere [4]. A reason for that may be that SQ tries to atomically swap each value in a
segment using CAS until a valid item is removed or inserted, respectively. In comparison
the US k-FIFO queue algorithm first tries to find a corresponding index in the segment
and only performs a CAS on the location of the index.

The lock-free linearizable pool (BAG) [79] is based on thread-local lists of blocks
of elements. Each block is capable of storing up to a constant number of elements. A
thread performing an enqueue operation always inserts elements into the first block of
its thread-local list. Once the block is full, a new block is inserted at the head of the
list. A thread performing a dequeue operation always tries first to find an element in
the blocks of its thread-local list. If the thread-local list is empty, work stealing from
other threads’ lists is used to find an element. The work-stealing algorithm implements
a linearizable empty check by repeatedly scanning all threads’ lists for elements and
marking already scanned blocks which are unmarked when elements are inserted. The
implementation works only for a fixed number of threads.

The lock-free elimination-diffraction pool (ED) [8] uses FIFO queues to store ele-
ments. Access to these queues is balanced using elimination arrays and a diffraction
tree. While the diffraction tree acts as a distributed counter balancing access to the
queues, elimination arrays in each counting node increase disjoint-access parallelism.
Operations hitting the same index in an elimination array can either directly exchange
their data (enqueue meets dequeue), or avoid hitting the counter in the node that
contains the array (enqueue meets enqueue or dequeue meets dequeue). If based on
non-blocking FIFO queues, the presented algorithm lacks a linearizable empty check.
If based on blocking queues, there is no empty state at all. Parameters, i.e., elimination
waiting time, retries, array size, tree depth, number of queues, queue polling time, need
to be configured to adjust ED to different workloads.

The synchronous rendezvousing pool (RP) [2] implements a single elimination array
using a ring buffer. Both enqueue and dequeue operations are synchronous. A dequeue
operation marks a slot identified by its thread id and waits for an enqueue operation
to insert an element. An enqueue operation traverses the ring buffer to find a waiting
dequeue operation. As soon as it finds a dequeue operation they exchange values and
return. There exist adaptive and non-adaptive versions of the pool where the ring
buffer size is adapted to the workload.
3.5 Experiments

We evaluate the performance and scalability of the BS and US $k$-FIFO queue algorithms. All experiments ran on the MultiCore2 server machine. We implemented a framework to benchmark and analyze different queue and pool implementations under configurable scenarios. The framework emulates a multi-threaded producer-consumer workload where each thread is either a producer or a consumer. The framework can be configured for a different number of threads ($n$), number of enqueue or dequeue operations each thread performs ($o$), the computational load performed between each operation ($c$), the number of pre-filled items ($i$), and the queue implementation to use. The computational load $c$ between two consecutive operations is created by iteratively calculating $\pi$. A computation with $c = 1000$ takes a total of $2300\text{ns}$ on average. We fix the operations per thread to $o = 1000000$ and the number of producers and consumers to $n^2$ for all benchmarks. We evaluate the performance and scalability of the queues under low ($c = 10000$), medium ($c = 7000$), high ($c = 4000$) and very high ($c = 1000$) contention. The framework uses static preallocation for memory used in the data structures with touching each page before running the benchmark to avoid paging issues.

3.5.1 Understanding $k$

In order to provide a better understanding of the effect of $k$ on performance and scalability we first evaluate the performance of the BS version with increasing $k$. We omit the measurements for the US version as the results are similar. Performance is measured under very high ($c = 1000$) contention without ($i = 0$) and with ($i = 5000$) pre-filling the queue. Other contention scenarios lead to similar results. We relate the performance of our producer-consumer benchmark, measured in operations per millisecond, to the number of retries per operation and the number of so-called failed reads. Retries are an indicator of contention among CAS operations. They occur whenever an enqueue (line 2) or a dequeue (line 39) operation has to take another iteration of the while loop. Failed reads are attempts to find empty slots or items in find empty slot (line 5) and find item (line 42), respectively. Both retries and failed reads produce overhead and should thus be minimized in order to improve overall performance.

Figure 3.2 depicts this performance analysis with non-pre-filled results on the left and pre-filled results on the right side. Intuitively, one would expect that a larger $k$ results in fewer retries because of reduced contention among the inserting (line 9) and removing (line 48) CAS operations. Figure 3.2(b) shows that this is true for a
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

Figure 3.2: Very high and low contention producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine
setting where the queue is pre-filled with items, i.e., a queue with an initially dense population in the segment that is used for dequeueing. However, for a workload where the queue is initially empty there exists a turning point from which the number of retries starts to grow with increasing $k$. Figure 3.2(a) depicts this behaviour which appears when the segment used for dequeueing is only sparsely populated most of the time. In this case the dequeueing operations are likely to contend on the same, rare items in the head segment. Figure 3.2(c) and 3.2(d) illustrate the number of failed reads. As long as the number of retries is decreasing, failed reads are slowly increasing with larger $k$ since the segments to search for items or empty slots get bigger. As soon as the number of retries reaches the turning point in the pre-filled case failed reads are increasing exponentially. Figure 3.2(e) and 3.2(f) then visualize the impact of an increasing $k$ on the performance and show that there exists an optimal $k$ with respect to performance. The optimal $k$ is also robust in the sense that there exists only a single range of close-to-optimal $k$. Furthermore, the population density of a segment that is used for dequeueing has an impact on the range of $k$ where good performance can be observed. If $k$ gets too large, i.e., the population in the dequeueing segment is sparse, the performance significantly decreases. The depicted behaviour of $k$ suggests a controller that optimizes $k$ to dynamic workloads.

3.5.2 Performance and Scalability

We study the performance of LB, BS, MS, FC, RD, SQ, BS $k$-FIFO and US $k$-FIFO queues, and ED, BAG, and RP pools. For the RD, SQ, BS $k$-FIFO, and US $k$-FIFO queues we configure $r = s = k = 64$, which turned out be a good $k$ (on average) for a broad range of thread combinations and workloads. We use the non-adaptive version of the RP algorithm since the number of threads is constant in each run. The BAG implementation differs from the originally proposed one in a randomized work stealing selection. This modification was necessary to make BAG competitive in our benchmarks. Figure 3.3(a) illustrates the results for the very high contention workload where MS and RD perform best for up to 20 threads. With more than 20 threads scalability is negative for all data structures except RP, BS $k$-FIFO, and US $k$-FIFO. The BS $k$-FIFO queue algorithm is the only algorithm that scales near-linearly.

Similarly, the results with our high contention scenario, depicted in Figure 3.3(b), show that the scalability turnaround is at 30 threads and that both $k$-FIFO versions outperform and outscale all other algorithms. As the contention gets less in Figures 3.3(c) and 3.3(d) the turnaround gets shifted to a larger number of threads. The difference in performance and scalability of all algorithms is less significant with more computational
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

Figure 3.3: Producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine.

(a) Performance and scalability of very high contention producer-consumer benchmark ($c = 1000, i = 0$)

(b) Performance and scalability of high contention producer-consumer benchmark ($c = 4000, i = 0$)

(c) Performance and scalability of medium contention producer-consumer benchmark ($c = 7000, i = 0$)

(d) Performance and scalability of low contention producer-consumer benchmark ($c = 10000, i = 0$)

load. Note that SQ returns up to 2000 times falsely null due to the non-linearizable empty check.

3.5.3 Dynamic $k$

We implemented a prototypical PID controller which aims at identifying optimal $k$ automatically at runtime for best performance. Each thread $i$ stores performed enqueue operations $o_i$ and performed retries in enqueue operations $r_i$ in thread-local counters. The controller runs in an extra thread, reads the thread-local counters of all $n$ threads.
periodically (100ms), and resets them to 0 after reading. The goal of the controller is to minimize the ratio \( \sum_{i=1}^{n} r_i / \sum_{i=1}^{n} o_i \). The controller operates in the approximately linear part of this ratio. With the US \( k \)-FIFO algorithm the controller determines the segment size that enqueue operations use to create new segments which store their size for dequeue operations to look up. For the BS \( k \)-FIFO algorithm the maximum \( k \) needs to be bounded to provide a linearizable empty check.

![Figure 3.4: Variable-load producer-consumer microbenchmarks with an increasing number of static \( k \) versus a dynamically controlled \( k \) on a 40-core (2 hyperthreads per core) server machine](image)

We use a variable-load producer-consumer microbenchmark to evaluate the performance of the controller. Each thread performs \( o = 4000000 \) operations and starts with \( c = 1000 \). The workload changes for each thread whenever \( o/4 \) operations are performed by changing \( c \) to 500, 2000, and 1500. We compare the BS and US \( k \)-FIFO algorithms with dynamically controlled \( k \) to the unmodified baseline versions with statically configured \( k \) ranging from 1 to 120. Figure 3.4(b) shows the performance of the dynamic US \( k \)-FIFO queue. Here the controller improves the performance by 60% over the best statically configured configurations. The dynamic BS \( k \)-FIFO queue performs about 30% faster than the best statically configured configuration, see Figure 3.4(a).

### 3.6 Other quantitatively relaxed concurrent data structures

The concept of quantitatively relaxing concurrent data structures for better performance does not only work with concurrent FIFO queues. We obtained promising results
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

with a relaxed concurrent stack (k-stack) and a relaxed shared counter (k-counter). In the following we give a brief overview of our k-stack and k-counter algorithms and present experimental results.

3.6.1 k-Stack

The k-stack sequential specification is defined in Figure 3.5. Note that the only difference to the k-FIFO sequential specification presented in Figure 3.1 is in the enqueue operation. In the k-stack elements are added to the beginning of sequence q.

\[
push_k(e)(q,o) = (e \cdot q, o) \quad \text{(L1)}
\]

\[
\begin{cases}
(\varepsilon, o) & \text{if } e = \text{null, } q = \varepsilon \quad \text{(L2)} \\
(q', o') & \text{if } q = e_1 \ldots e_{i-1}e_i e_{i+1} \ldots e_n, \\
& q' = e_1 \ldots e_{i-1}e_{i+1} \ldots e_n, \\
& e = e_i, \\
& 1 \leq i \leq \min(k - o(e_1), n), \\
& o' = F(o, i) \quad \text{(L3.1)} \\
& \text{error otherwise} \quad \text{(L3.2)} \\
& e = e_i, \\
& o' = F(o, i) \quad \text{(L3.3)} \\
& \text{error otherwise} \quad \text{(L3.4)} \\
& F(o, i)(e_j) = \begin{cases}
o(e_j) & \text{if } j \geq i \quad \text{(L5)} \\
o(e_j) + 1 & \text{otherwise} \quad \text{(L6)}
\end{cases}
\end{cases}
\]

Figure 3.5: k-stack sequential specification

The top pointer of a concurrent stack may become a scalability bottleneck under high contention [80]. For the implementation of the k-stack we use the same high-level idea as already presented in the k-FIFO queue implementation, i.e., we reduce contention on the top pointer by maintaining a stack of segments. We implemented the stack that holds the segments similarly to the lock-free stack of [80] with the difference that there is always at least one segment, even if it is empty, on the stack. This avoids unnecessary removal and adding of a segment, e.g. in the empty state. As before a segment contains k atomic values which may either point to null indicating an empty slot or may hold an item, the implementation concept of an element pushed on the k-stack. Both push and pop operations are served by the top segment. Hence, up to k stack operations may be performed in parallel. A push operation tries to insert an item in the top segment. It adds a new segment to the stack if the top segment is full. A pop operation tries to remove an item from the top segment. It removes the top segment.
Listing 3.5: Segment stack algorithm

```c
1 global top;
2
3 void init():
4    new_ksegment = calloc(sizeof(ksegment));
5    top = atomic_value(new_ksegment, 0);
6
7 void try_add_new_ksegment(top_old):
8    if top_old == top:
9        new_ksegment = calloc(sizeof(ksegment));
10       new_ksegment->next = top_old;
11       top_new = atomic_value(new_ksegment, top_old.version + 1);
12       CAS(&top, top_old, top_new);
13
14 void try_remove_ksegment(top_old):
15    if top_old == top:
16       if top_old->next != null:
17          atomic_increment(&top_old->remove);
18       if empty(top_old):
19          top_new = atomic_value(top_old->next, top_old.version + 1);
20          if CAS(&top, top_old, top_new):
21             return;
22          atomic_decrement(&top_old->remove);
```

from the stack if it is empty and is not the only segment on the stack. Additionally, each segment contains an atomic counter `remove` that counts how many threads are trying to remove it from the stack. The counter is initially set to zero. Note that a bounded-size $k$-stack can be implemented similarly to the BS $k$-FIFO queue version.

The methods `init`, `try_add_new_ksegment`, and `try_remove_ksegment` implement the stack of segments, depicted in Figure 3.5. In the latter, the atomic counter `remove` is updated and the method `empty` that performs an empty check is called. We discuss the method `empty` within the `pop` method, as it is also called there.

The $k$-stack algorithm is depicted in Figure 3.6. The `push` method first tries to find an empty slot for the `item` using the `find_empty_slot` method (line 26). The `find_empty_slot` method randomly selects an index in the top segment and then linearly searches for an empty slot starting with the selected index and wrapping around at index $k$. Then the `push` method checks if the $k$-stack state has been consistently observed for an empty slot starting with the selected index and wrapping around at index $k$. Then the `push` method checks if the $k$-stack state has been consistently observed by testing whether `top` changed in the meantime (line 27) which would trigger a retry. If an empty slot is found (line 28) the method tries to insert the `item` at the location of the empty slot using `CAS` (line 30). If the insertion is successful the method verifies whether the insertion is also valid by calling the `committed` method (line 31), as discussed below. If any of these steps fails, a retry is performed. If no empty slot is
found in the current top segment, the push method tries to add a new segment to the
stack of segments (line 34) and then retries.

The committed method (line 36) validates an insertion, it ensures that the inserted
element is really inserted on the stack. It returns true when the insertion is valid and
false when it is not valid. An insertion is invalidated if a concurrent thread removes the
segment to which the element was inserted before the effect of the insertion took place.
Therefore, an insertion is valid if the inserted item already got popped at validation
time by a concurrent thread (line 37, 44, 50) or the segment where the item was inserted
was not removed by a concurrent thread (line 39). A remove counter larger than zero
indicates that the segment has been removed or concurrent threads are trying to remove
the segment from the stack (line 41). If the current top segment is not equal to the
segment where the item was inserted we have to conservatively assume that the segment
was removed from the stack (line 43) and undo the insertion (line 44). If the current top
segment is equal to the segment where the item was inserted, a race with concurrent
popping threads may occur which may not have observed the insertion of the item and
may try to remove the segment from the stack in the meantime. This would result in
loss of the inserted item. To prevent that, the method tries to increment the version
number in the top atomic value using CAS (line 48) forcing threads that concurrently
try to remove that segment to retry. If this fails, a concurrent pop operation may have
changed top which would make the insertion potentially invalid. Hence, in case of
loosing the race, the method tries to undo the insertion using CAS (line 50).

The pop method returns an item if the k-stack is not empty. Otherwise it returns
null. Similarly to the push method, the pop method first tries to find an item in the
top segment using the find_item method (line 57). The find_item method randomly
selects an index in the top segment and then linearly searches for an item starting with
the selected index and wrapping around at index k. Then, the pop method checks if
the k-stack state has been consistently observed by checking whether top changed in
the meantime (line 58) which would trigger a retry. If an item was found (line 59) the
method tries to remove it using CAS (line 61) and returns it if the removal was successful
(line 62). Otherwise a retry is performed. If no item is found and the current segment is
the only segment on the stack (line 64) an empty check is performed using the method
empty (line 65). This method stores the values of the k slots of the segment in a local
array (if they are empty) and subsequently checks in another pass over the segment
slots whether the values in the slots changed in the meantime. If a non-empty slot was
found, the empty method immediately returns false. If the empty check succeeded and
the top did not change in the meantime (line 66), null is returned (line 67). Otherwise,
3.6 Other quantitatively relaxed concurrent data structures

Listing 3.6: $k$-stack algorithm

```c
void push(item):
    while true:
        top_old = top;
        item_old, index = find_empty_slot(top_old);
        if top_old == top:
            if item_old.value == EMPTY:
                item_new = atomic_value(item, item_old.version + 1);
                if CAS(&top_old->s[index], item_old, item_new):
                    if committed(top_old, item_new, index):
                        return true;
                    else:
                        try_add_new_ksegment(top_old);
        else:
            //top_old->remove >= 1
            item_empty = atomic_value(EMPTY, item_new.version + 1);
            if !CAS(&top_old->s[index], item_new, item_empty):
                return true;
            else:
                top_new = atomic_value(top_old.value, top_old.version + 1);
                if CAS(&top, top_old, top_new):
                    return true;
                else:
                    if only_ksegment(top_old):
                        if empty(top_old):
                            if top_old == top:
                                return null;
                        else:
                            try_remove_ksegment(top_old);
```

```c
bool committed(top_old, item_new, index):
    if top_old->s[index] != item_new:
        return true;
    else if top_old->remove == 0:
        return true;
    else: //top_old->remove >= 1
        item_empty = atomic_value(EMPTY, item_new.version + 1);
        if !CAS(&top_old->s[index], item_new, item_empty):
            return true;
        else:
            top_new = atomic_value(top_old.value, top_old.version + 1);
            if CAS(&top, top_old, top_new):
                return true;
            else:
                if only_ksegment(top_old):
                    if empty(top_old):
                        if top_old == top:
                            return null;
                    else:
                        try_remove_ksegment(top_old);
```

```c
item pop():
    while true:
        top_old = top;
        item_old, index = find_item(top_old);
        if top_old == top:
            if item_old.value != EMPTY:
                item_empty = atomic_value(EMPTY, item_old.version + 1);
                if !CAS(&top_old->s[index], item_old, item_empty):
                    return true;
                else:
                    if only_ksegment(top_old):
                        if empty(top_old):
                            if top_old == top:
                                return null;
                        else:
                            try_remove_ksegment(top_old);
```
```
if no item is found in the current segment and there is more than one segment in the stack, the method tries to remove the segment (line 69) and performs a retry.

**Linearizability**

We now prove that the \(k\)-stack implementation is correct for the relaxed stack semantics depicted in Figure 3.5.

**Proposition 3.6.1.** The \(k\)-stack algorithm is linearizable with respect to the \(k\)-stack sequential specification.

**Proof.** Without loss of generality, we assume that each item pushed on the stack is unique. A segment \(s'\) is reachable from a segment \(s\) if either \(s'=s\) or \(s'\) is reachable from \(s->next\). An item \(i\) is on the stack, if \(push(i)\) has already committed and there exists a segment reachable from the top segment containing a slot whose value is \(i\).

Note that reachability is important, i.e., only having a slot containing the item is not enough to guarantee that the item is logically on the stack, because the slot could be in a segment (to be) removed by a concurrent pop operation.

We begin by identifying a linearization point of each method call. The goal is to show that the sequential history obtained from a concurrent history by ordering methods according to their linearization points is in the sequential specification of the \(k\)-stack. The linearization point of \texttt{push} is the reading of the empty slot (line 26) in the last iteration (successful insertion) of the main loop. The linearization point of \texttt{pop} that does not return \texttt{null} is the reading of a non-empty slot (line 57) in the last iteration (successful removal) of the main loop. The linearization point of a \texttt{null}-returning \texttt{pop} is the point after the first pass of the segment in the call to \texttt{empty} method (line 65) which returns \texttt{true}.

The correctness argument is based on the following facts.

1. **An item is pushed on the stack exactly once.** This is a consequence of our unique-items assumption and the control flow of \texttt{push(i)}, the only method that can modify a slot to contain \(i\).

2. **An item is popped at most once.** If an item \(i\) is on the stack, then it can only be removed once, because of 1. and the existence of a unique statement which replaces \(i\) with empty. If \(i\) is in some slot but not on the stack, then \texttt{push(i)} will erase \(i\) and retry insertion before committing. We have to show that while \(i\) is in some slot but not on the stack, no pop operation can return \(i\). Clearly, the call to method \texttt{committed} by \texttt{push(i)} must return \texttt{false}. This implies that until \texttt{committed} completes, the slot where \(i\) resides is not modified by any other thread. Otherwise, either after the first
3.6 Other quantitatively relaxed concurrent data structures

if statement (line 37) or following failed CAS attempts (lines 44 and 50) of replacing i with empty will lead to returning true. Furthermore, when control reaches the only exit point for returning false, it is guaranteed that there is no slot containing i. Thus, if i is not on the stack, no pop operation could have replaced it with empty.

3. If a pop operation returns null, then during its execution, there must be a state at which there are no items on the stack. Since returning null is without any side-effect, it suffices to prove the existence of a state which corresponds to a logically empty stack. The call to empty is only done when the top segment is the only segment in the stack. In the empty method, the value of top is checked at the beginning and after the first pass to ensure that the pointer is not updated by concurrent operations. Hence, the stack is indeed logically empty at the linearization point since the second pass succeeds.

4. An item j cannot be popped before an item i, if they are both on the stack, and i, j are in segments s, s’, respectively, with s’ reachable from s and s’ ≠ s. The segment s’ can become a top segment only after the segment s has been removed by some pop operation. Moreover, if a segment becomes unreachable from the top segment, it remains unreachable. These two observations imply that the linearization point of pop(i) must precede the linearization point of pop(j) which can only happen when s’ is a top segment.

5. An item i on the stack is popped only if it is one of the k−l youngest items on the stack, where l is the current lateness of the youngest item. By youngest we mean most recently pushed. Assume i is popped from the stack at the current moment in time and at that point the youngest item is t with current lateness l. This means that ever since t is the youngest item on the stack, no push operation was performed and there have been l pop operations performed none of which removed t. Let j be any of these l popped items. Since t is the last item pushed and it is still on the stack, t is in the top segment. Since j is removed before t, by 4. it must have also resided in the top segment. For the same reason, also i is in the top segment prior to its removal. Hence, at the moment in which pop(i) happens, there are at most k−l items in the top segment.

Thus, any concurrent execution generated by the k-stack algorithm is linearizable with respect to the k-FIFO sequential specification. Observe that already 1. and 2. show that the k-stack has pool semantics.

Without any particular difficulty, but with a somewhat lengthy argument, one can show that the k-stack algorithm is lock-free by showing that whenever a thread retries
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

an operation, another thread completes its operation ensuring progress of at least one thread. The proof is similar to the lock-freedom proof of the $k$-FIFO queue discussed in Section 3.3.3.

Experiments

We compare our $k$-stack implementation with a standard lock-based stack (LS), which acquires a global lock for each stack operation, and a non-blocking stack (NS) [80], which uses a CAS operation to manipulate the top pointer of the stack. Moreover, we compare our $k$-stack with the BAG, ED, and RD pool implementations.

Figure 3.6 depicts the performance analysis of our $k$-stack. We configure $k = 80$, which turned out to be a good $k$ (on average) for a broad range of thread combinations and workloads on our server machine. The analysis is done on the very high contention producer-consumer workload where half of the threads are producers and half are consumers. The $k$-stack outscales and outperforms all considered stack and pool implementations. Figure 3.6(b) shows the effect of $k$ on performance and scalability. There exists an optimal $k$ with respect to performance, which is also robust in the sense that there exists only a single range of close-to-optimal $k$. For large $k$ performance decreases due to higher sequential overhead, e.g. scanning for elements in almost empty segments. Note that an increase in performance above $k = 80$ is not to be expected on the given architecture.
3.6 OTHER QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

3.6.2 $k$-counter

Next, we present two versions of a $k$-relaxed shared counter ($k$-counter). The original counter is based on a global variable `counter` representing the shared counter and an increment method that atomically increments `counter`, depicted in Listing 3.7. The implemented relaxation is a so-called $k$-stuttering relaxation of the shared counter, i.e., up to $k - 1$ counter modifications may not have effect.

Listing 3.7: Shared counter

```c
1 global counter;
2
3 increment():
4   while (true):
5     x = counter;
6     if CAS(&counter, x, x+1):
7       return x+1;
```

In the first version we use an array `counters` of $k$ counters. The logical value of the shared counter is taken to be the maximum value among all the counters in the `counters` array, depicted in Listing 3.8. Each thread can write only to the slot with index `tid()` modulo $k$, where `tid()` returns the current thread id. Note that a random number could also be used instead of calling `tid()`. Each attempt of a thread incrementing the shared counter starts by determining the maximum value of the $k$ counters. After that it tries to update the counter at index `tid()` modulo $k$ to the maximum obtained value plus one provided that the value at index `tid()` modulo $k` did not change in the meantime.

Listing 3.8: $k$-counter (distributed counters)

```c
1 global counters[k];
2
3 increment():
4   while (true):
5     max = getMaxValue();
6     value = counters[tid() % k];
7     if CAS(&counters[tid() % k], value, max + 1):
8       return max + 1;
```

The second version of the $k$-counter is based on a $k$-relaxed version of the wait-free software CAS operation [50] ($k$-CAS for short), depicted in Listing 3.9. The original wait-free software CAS operation uses a structure `blk_original`, shown in Listing 3.10, to keep track of the state of concurrent CAS operations. The atomic value is located...
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

Listing 3.9: $k$-counter ($k$-relaxed CAS)

```c
1 global counter;
2
3 increment():
4 while(true):
5   x = counter;
6   if kCAS(&counter, x, x+1):
7       return x+1;
```

Listing 3.10: Wait-free CAS state variables

```c
1 struct blk_original {
2   pidtype X;
3   bool Y;
4   val_t V;
5   bool C;
6   val_t D;
7};
```

Listing 3.11: Wait-free $k$-CAS state variables

```c
1 struct blk_modified {
2   pidtype X[k];
3   bool Y[k];
4   val_t V;
5   bool C[k];
6   val_t D;
7};
```

in field $V$ and the CAS operation uses the decision fields $X$, $Y$, and $C$ to determine which thread gets permission to change $V$. We modified the `blk_original` structure so that the fields $X$, $Y$, and $C$ are arrays of size $k$ depicted in structure `blk_modified` in Listing 3.11. We keep the main CAS operation unmodified but use a balancing function that maps threads to array indices $i$ smaller than $k$ (the thread ID modulo $k$). A thread determines the state of its CAS operation by just accessing position $i$ in the arrays $X$, $Y$, and $C$. On success, a thread writes the new value into $V$. Hence, up to $k$ concurrent threads may perform the $k$-CAS operation in parallel and change $V$ resulting in a loss of at most $k-1$ state changes, which further results in at most $k-1$ lost shared counter updates.
Experiments

We compare our $k$-CAS-based shared counter and distributed shared counter implementations with a shared counter implementation based on a regular CAS operation depicted in Figure 3.7(a). The threads perform in total one million counter increment operations in each benchmark run. The CAS version performs best until 30 threads. After that the $k$-CAS-based shared counter and the distributed shared counter version outperform it. Figure 3.7(b) shows the effect of $k$ on performance and scalability. In the $k$-CAS version performance monotonically increases with larger $k$, whereas in the distributed shared counter version performances decreases until $k = 10$, but monotonically increases after that. Our educated guess is that this is caused by the trade-off between two possible sources of contention: (1) CAS on the same memory location, and (2) bad caching, i.e., accessing many different locations in memory. The distributed shared counter decreases (1) but increases (2). However, except for small values of $k$, we observe that the gain is larger than the loss.

Figure 3.7: Shared counter benchmarks on a 40-core (2 hyperthreads per core) server machine

3.7 Summary

We have introduced fast and scalable algorithms that implement bounded- and unbounded-size, lock-free, linearizable $k$-FIFO queues with empty (and full) check. We showed experimentally for both algorithms that there exist optimal and robust $k$ that result in best performance and scalability. Moreover, we demonstrated in experiments that our algorithms outperform and outscale many state-of-the-art concurrent queue
3. QUANTITATIVELY RELAXED CONCURRENT DATA STRUCTURES

and pool algorithms on different concurrent producer-consumer workloads. Finding the right $k$ for different workloads is key for best performance and scalability. We suggest to either set $k$ statically to around the number of available parallel processing units or use a controller which automatically adjusts $k$ at runtime as shown in our experiments. Finally, we showed that quantitative relaxations may be also applied to other concurrent data structures to gain better performance and scalability. We presented a $k$-stack algorithm and two shared $k$-counter algorithms and showed that they outperform and outscale all considered competitors.
CHAPTER 4

**Performance, Scalability, and Semantics of Concurrent FIFO Queues**

In the previous chapter we introduced the notion of a $k$-FIFO queue. In this chapter we present Scal queues \[42, 43\], which implement $k$-FIFO queues with generally larger, workload-dependent as well as unbounded $k$. Hence, Scal queues can be also seen as pools. We call $k$ the worst-case semantical deviation (WCSD) of the queue from a regular FIFO queue. The WCSD bounds the actual semantical deviation (ASD) of a $k$-FIFO queue from a regular FIFO queue when applied to a given workload. Intuitively, the ASD keeps track of the number of younger elements that overtook older elements and the age of dequeued elements. Since ASD cannot be obtained without prohibitive overhead we have developed a tool that computes lower bounds on ASD from time-stamped runs. Our micro- and macrobenchmarks on a state-of-the-art 40-core multiprocessor machine show that Scal queues, as an immediate consequence of their weaker WCSD, outperform and outscale regular FIFO queues and the $k$-FIFO queues with constant $k$ introduced in Chapter 3 at the expense of moderately increased lower bounds on ASD. Moreover, Scal queues outperform and outscale state-of-the-art concurrent pool algorithms on different micro- and macrobenchmarks.

### 4.1 Introduction

Based on the same principle of improving performance and scalability at the expense of strict FIFO semantics as discussed in Chapter 3 we propose Scal queues for implementing FIFO queues with relaxed semantics. The idea is to maintain (a distributed system of) $p$ instances of a regular FIFO queue (we chose Michael-Scott (MS) \[55\] for our experiments) and then select, upon each enqueue or dequeue operation, one of the
4.1 Introduction

$p$ instances before performing the operation on the selected instance without further coordination with the other instances. Thus up to $p$ queueing operations may be performed in parallel. Selection is done by a load balancer whose implementation has an immediate impact on performance, scalability, and semantics. In particular, the load balancer determines how close the semantics of the Scal queue is to the semantics of a regular FIFO queue.

\[
\text{enqueue}_n(e)(q) = (q \cdot e) \quad (L1)
\]

\[
\text{dequeue}_n(e)(q) = \begin{cases} 
(\varepsilon) & \text{if } e = \text{null}, q = \varepsilon \\
(q') & \text{if } q = e_1 \ldots e_{i-1}e_i e_{i+1} \ldots e_n, \\
\text{error} & \text{otherwise}
\end{cases} \quad (L2, L3.1, L3.2)
\]

Figure 4.1: Pool sequential specification

Scal queues implement the sequential specification of a pool, depicted in Figure 4.1. Note that the $k$-FIFO sequential specification depicted in Figure 3.1 represents a pool when $k = n$, where $n$ is the number of elements in the queue. Let $q$ denote the state of a pool where $q$ is the sequence of $n$ queue elements with $n \geq 0$. The initial, empty state of the pool is $(\varepsilon)$. The enqueue operation is a function from queue states and queue elements to queue states. It logically inserts an element at the end of $q$. The dequeue operation is a function from queue states and queue elements or the null return value, which indicates an empty queue, to queue states. It logically removes an element among the elements $e_1 \ldots e_n$ from the queue, where $n$ is the number of queue elements. The non-determinism in the choice of elements to be returned provides the potential for performance and scalability which, in our benchmarks, tend to increase with increasing $k$ (or $p$).

With the straightforward metric of operation throughput in place for measuring performance, the only remaining challenge is to quantify and to measure difference in semantics. For this purpose, we introduce the notion of semantical deviation as a metric for quantifying the difference in semantics between a queue with relaxed FIFO semantics and a regular FIFO queue. Intuitively, when running a given queue implementation on some workload, semantical deviation keeps track of the number of younger elements that overtook older elements and the age of dequeued elements. However, measuring actual semantical deviation on existing hardware is only possible indirectly and approximatively through time-stamping invocation and response events of operations online,
and then computing offline, using a tool that we developed, an approximation of the actual run that took place. The approximation is a sequence of linearization points that leads to a lower bound on the actual semantical deviation. We also define an upper bound on the actual semantical deviation of a given run.

Here a key observation is that there exist upper bounds on semantical deviation independent of at least all workloads in a given class (e.g. with a fixed number of threads) for most of the implementations we consider. It turns out that these implementations are instances of the notion of a $k$-FIFO queue for different $k \geq 1$ where $k - 1$ is their worst-case semantical deviation from a regular FIFO queue. A $k$-FIFO queue may dequeue elements out of FIFO order up to $k - 1$. Our bounded fairness metric measures how often an element was overtaken by younger elements and our age metric measures how many positions the element was away from the oldest element.

### 4.2 Scal Queues

Scal is a framework for implementing $k$-FIFO queues as well as potentially other concurrent data structures such as relaxed versions of stacks and priority queues that may provide bounded out-of-order behavior. We focus on $k$-FIFO queues and leave other concurrent data structures for future work. In the sequel we refer to $k$-FIFO queues implemented with Scal as Scal queues.

Scal is motivated by distributed systems where shared resources are distributed and access to them is coordinated globally or locally. For implementing $k$-FIFO queues Scal uses $p$ instances of a regular FIFO queue, so-called partial FIFO queues, and a load balancer that distributes queueing operations among the $p$ partial FIFO queues. Upon the invocation of a queueing operation the load balancer first selects one of the $p$ partial FIFO queues and then calls the actual queueing operation on the selected queue. The value of $p$ and the type of load balancer determine $k$, as discussed below, as well as the performance and scalability of Scal queues, i.e., how many queueing operations can potentially be performed concurrently and in parallel, and at which cost without causing contention. Moreover, in our Scal queue implementations selection and queueing are performed non-atomically for better performance and scalability. The semantics of Scal queues may nevertheless be significantly closer to FIFO semantics than what the value of $k$ may suggest because of the low probability of the worst case, as shown in Section 4.4. Note that $p$ and the load balancer may be configured at compile time or dynamically at runtime with the help of performance counters. For example, a load balancer may be chosen with $p = 1$ under low contention and with increasing $p$ as contention increases. Dynamic reconfiguration is future work.
Listing 4.1: Scal generic structure. Gray highlighted code is only used in the 2-random load balancer versions

```java
enqueue (element):
    index = load_balancer();
    fifo[index].enqueue(element);
    atomic_increment(&fifo[index].size);

dequeue():
    tail_old[p];
    start = load_balancer();
    while true:
        for i in 0 to p-1:
            index = (start + i) % p;
            element, tail_old[index] = fifo[index].dequeue()
            if element != null:
                atomic_decrement(&fifo[index].size);
                return element;
        for i in 0 to p-1:
            if fifo[i].tail != tail_old[i]:
                start = i;
                break;
        if i == p-1:
            return null;
```

The generic structure of Scal is shown in Listing 4.1. The occurrence of the ABA problem is made unlikely through standard version numbers. We refer to values enhanced with version numbers as atomic values (atomic_value). The gray highlighted code is only used in the 2-random load balancer versions, discussed in Section 4.2.2.

This code increments or decrements an atomic counter that represents the size of a given partial FIFO queue. Note that the adjustment of the size counters and the actual queue operation is performed non-atomic. Hence, the size counters just provide an estimate of the actual queue size.

The `enqueue` method (line 1) calls the load balancing method `load_balancer` (line 2) which determines in which partial FIFO queue the element is enqueued (line 3). Similarly, the `dequeue` method (line 6) also calls the load balancing method `load_balancer` in the beginning which selects a partial FIFO queue. The `dequeue` method then starts searching linearly for an element starting at the selected partial FIFO queue and wraps around at index \( p - 1 \). If an element is found in a partial FIFO queue, it is removed from that partial FIFO queue (line 12) and returned (line 15). Moreover the current tail value of the given partial FIFO queue is stored in a local array (line 12). If no element was found after checking all \( p \) partial FIFO queues the empty check is performed by checking whether the tail pointers of the partial FIFO
4. PERFORMANCE, SCALABILITY, AND SEMANTICS OF CONCURRENT FIFO QUEUES

queues changed in the meantime (line 17). If this is not the case null is returned (line 21). Otherwise dequeue has to retry (line 19).

In the following we discuss different load balancing strategies.

4.2.1 Round-Robin Load Balancing

We have implemented a round-robin load balancer (RR) for Scal that selects partial FIFO queues for enqueue and dequeue operations in round-robin fashion. Two global counters keep track on which of the \( p \) partial FIFO queues the last enqueue and the last dequeue operation was performed. The counters are accessed and modified using atomic operations, which can cause contention. However, scalability may still be achieved under low contention since the load balancer itself is simple. A Scal queue using RR implements a \( k \)-FIFO queue with \( k = t \cdot (p - 1) \) where \( t \) is an upper bound on the number of threads in the system. Note that \( k \) comes down to \( p - 1 \) if selection and queueing are performed atomically.

Note that a thread-local round-robin load balancer (TL-RR), where each thread holds two thread-local counters results in the same \( k = t \cdot (p - 1) \) as the RR load balancer, where \( t \) is an upper bound on the number of threads in the system. However, the balancing quality is on average worse in comparison to RR but it provides better performance, since the thread-local round-robin counters are not a scalability bottleneck.

4.2.2 Randomized Load Balancing

A randomized load balancer randomly distributes operations over partial FIFO queues. Randomized load balancing \[7, 66, 13\] has been shown to provide good distribution quality if the random numbers are distributed independently and uniformly. However, generating such random numbers may be computationally expensive. Therefore, it is essential to find the right trade-off between quality and overhead of random number generation. We use an efficient random number generator that produces evenly distributed random numbers \[62\]. The value of \( k \) for RA Scal queues is unbounded but may be determined probabilistically as part of future work. A first step is to determine the maximum imbalance of the partial FIFO queues. Suppose that \( t \) threads have performed \( m \) operations each on \( p \) partial FIFO queues using RA. Then, with a probability of at least \( 1 - O \left( \frac{1}{p} \right) \), the maximum difference (imbalance) between the number of elements in any partial FIFO queue and the average number of elements in all partial FIFO queues is \( \Theta \left( \sqrt{\frac{t \cdot m \cdot \log p}{p}} \right) \) \[66\] if selection and queueing are performed atomically. However, as previously mentioned, selection and queueing are performed
non-atomically in our implementation. The presented maximum imbalance is anyway relevant for a comparison with a refined version of RA discussed next.

In order to improve the load balancing quality of RA, $d$ partial FIFO queues with $1 < d \leq p$ may be chosen randomly. Out of the $d$ partial FIFO queues the queue that contributes most to a better load balance is then selected. More precisely, enqueue and dequeue operations are performed on the partial FIFO queues that contain among the $d$ partial FIFO queues the fewest and the most elements, respectively. We refer to such a load balancer as $d$-randomized load balancer ($d$RA). The runtime overhead of $d$RA increases linearly in $d$ since the random number generator is called $d$ times. Thus $d$ allows us to trade off balancing quality and global coordination overhead. Here, again the value of $k$ for $d$RA is unbounded. However, again with a probability of at least $1 - O\left(\frac{1}{p}\right)$, the maximum difference (imbalance) between the number of elements in any partial FIFO queue and the average number of elements in all partial FIFO queues is now $\Theta\left(\frac{\log \log p}{d}\right)$ [13] if selection and queueing are performed atomically. Again, determining the maximum imbalance for the case when selection and queueing are performed non-atomically, as in our implementation, is future work. However, the presented maximum imbalance shows an important difference to RA Scal queues. It is independent of the state of the Scal queue, i.e., the history of enqueue and dequeue operations. In particular, $d = 2$ leads to an exponential improvement in the balancing quality in comparison to RA. Note that $d > 2$ further improves the balancing quality only by a constant factor [13] at the cost of higher computational overhead.

We conducted several experiments to evaluate the balancing quality of four different random load balancers: random, 2-random, and 3-random use a simple but efficient random number generator as discussed in [62]; and hw-random is a random load balancer that takes the 58 most significant bits of time stamp counter register of the CPU (RDTSC) modulo $p$. During each experiment the load balancer is executed a million times and we keep track how often each partial FIFO queue is selected. Figure 4.2 shows the standard deviation of the amount of partial FIFO queue selections. For each random load balancer the experiment is repeated using values between 2 and 4096 for $p$. For example, the standard deviation of how often a partial FIFO queue among 16 partial data structures ($p = 16$) is selected by the 2-random load balancer is 1 and 10000 by the hw-random load balancer. hw-random produces the worst distribution among the four evaluated load balancers. The experiments also confirm that the $d$-random load balancers provide a significant improvement in balancing quality in comparison to the random load balancer. The standard deviation of the amount of performed operations on the $p$ partial data structures using the $d$-random load balancers is approximately 1.
However, in terms of execution time the $d$-random load balancers are $d$ times slower than the random load balancer since their random number generator is invoked $d$ times more often.

![Figure 4.2: Balancing quality of different random load balancers with increasing number of partial FIFO queues ($p$)](image)

4.2.3 Hierarchical Load Balancing

With hierarchical load balancing $p$ partial FIFO queues are partitioned into $0 < s \leq p$ non-overlapping subsets. We use a two-level hierarchy where the high-level load balancer chooses the subset and the low-level load balancer chooses one of the partial FIFO queues in the given subset. For partitioning we take the cache architecture of the system into account by making the subsets processor-local, i.e., $s$ is here the number of processors of the system. For the high-level load balancer we use a weighted randomized load balancer where the thread running on processor $i$ chooses the processor-local subset $i$ with a given probability $w$ while one of the remaining subsets is chosen with probability $1 - w$. This allows us to increase cache utilization and reduce the number of cache misses. On the lower level we use a randomized (H-RA) or 2-randomized (H-2RA) load balancer to choose the actual partial FIFO queue. Note that in principle multiple hierarchies of load balancers could be used and in each hierarchy a different load balancer could run. The value of $k$ for H-RA and H-2RA Scal queues is again unbounded but may be determined probabilistically similar to the value of $k$ for RA and 2RA Scal queues, respectively.

4.2.4 Linearizability

We now prove that the Scal queue implementations are correct pool algorithms.
Proposition 4.2.1. The Scal queue algorithm is linearizable with respect to the pool sequential specification.

Proof. We begin by identifying a linearization point of each method call. The goal is to show that the sequential history obtained from a concurrent history by ordering methods according to their linearization points is in the specification of a pool. Note that the load balancers do not have an effect on the linearizability proof. The linearization point of \texttt{enqueue} corresponds to the linearization point of the \texttt{enqueue} method of the MS queue (line 3) \[55\]. The linearization point of \texttt{dequeue} that returns an element corresponds to the linearization point of the \texttt{dequeue} method of the MS queue (line 12) \[55\]. The linearization point of a \texttt{null}-returning \texttt{dequeue} is the point after the first pass over the \(p\) partial FIFO queues, when no element was found (line 16). Since returning \texttt{null} is without any side-effect, it suffices to prove the existence of a state which corresponds to a logically empty pool. The second pass (line 16) over the \(p\) partial FIFO queues is only performed, if in the first pass no element was found. If in the second pass the tail pointers of the \(p\) partial FIFO queues did not change, the point after the first pass over the \(p\) partial FIFO queues represents the logically empty state.

\[ \Box \]

4.2.5 Lock-freedom

Proposition 4.2.2. The Scal queue algorithm is lock-free.

Proof. The load balancers do not have an effect on lock-freedom since they just consist of bounded size sequential code. The \texttt{enqueue} method is lock-free if the enqueue method of the used partial FIFO queue is lock-free, which is the case with MS \[55\]. The \texttt{dequeue} method that returns an element is lock-free if the dequeue method of the used partial FIFO queue is lock-free, which is the case with MS \[55\]. The \texttt{dequeue} method loops if in the empty check in the second pass over the tail pointers a change of the tail pointers is encountered (line 17). The tail pointer of a partial FIFO queue can only change when a new element was enqueued. Hence, another thread made progress.

\[ \Box \]

4.3 Semantical Deviation

We are interested in what we call the semantical deviation of a \(k\)-FIFO queue from a regular FIFO queue when applied to a given workload. Semantical deviation captures how many dequeue operations overtook older elements (lateness) and what the
4. PERFORMANCE, SCALABILITY, AND SEMANTICS OF CONCURRENT FIFO QUEUES

Since the age of dequeued elements was. Since semantical deviation cannot be measured efficiently without introducing prohibitive measurement overhead we propose lower and upper bounds of which the lower bounds can be computed efficiently from time-stamped runs of k-FIFO queue implementations. Our experimental results show that the lower bounds at least enable a relative, approximative comparison of different implementations in terms of their actual semantical deviation. Computing the upper bounds remains future work.

We represent a workload applied to a queue by a so-called (concurrent) history \( H \), which is a finite sequence of invocation and response events of enqueue and dequeue operations \([31]\). We work with complete histories, i.e., histories in which each operation has a corresponding invocation and response event and the invocation is before the response event. By \( \langle op \rangle \) and by \( \langle op \rangle \) we denote the invocation and response events of the operation \( op \), respectively. Two operations \( op_1 \) and \( op_2 \) in a history \( H \) are overlapping if the response event \( \langle op_1 \rangle \) is after the invocation event \( \langle op_2 \rangle \) and before the response event \( \langle op_2 \rangle \), or vice versa. An operation \( op_1 \) precedes another operation \( op_2 \) in a history \( H \), if the response event \( \langle op_1 \rangle \) is before the invocation event \( \langle op_2 \rangle \). Two histories are equivalent if the one is a permutation of the other in which precedences are preserved (only events of overlapping operations may commute). A history \( H \) is sequential if the first event of \( H \) is an invocation event and each invocation event is immediately followed by a matching response event \([31]\). Equivalently, a sequential history is a sequence of enqueue and dequeue operations.

Given a sequential specification \( C \) (here FIFO, k-FIFO, or POOL), an execution sequence corresponding to a sequential history \( H_S = op_1 \ldots op_m \) is a sequence of states \( C(H_S) = s_0s_1 \ldots s_m \) starting from the initial state \( s_0 \) with \( s_{j+1} = op_{j+1}(s_j) \) for \( j = 0, \ldots, m - 1 \). The sequential history \( H_S \) is valid with respect to the sequential specification \( C \) if no \( s_i \) in \( C(H_S) \) is the error state \( error \).

In particular, for a sequential history \( H_S = op_1 \ldots op_m \), FIFO(\( H_S \)) is the sequence of FIFO queue states obtained from the sequential specification of Figure \([3.1]\) with \( k = 1 \), where \( s_0 = (\varepsilon, o_0) \) and \( s_j = (q_j, o_j) \) with \( o_j(e_1) = 0 \) if \( |q_j| > 0 \) for \( j = 0, \ldots, m \); k-FIFO(\( H_S \)) is the sequence of k-FIFO queue states obtained from the sequential specification of Figure \([3.1]\) where \( s_0 = (\varepsilon, o) \). If \( H_S \) is valid with respect to FIFO, FIFO-valid for short, i.e., if no queue state in FIFO(\( H_S \)) is the error state \( error \), then each dequeue operation in \( H_S \) returns the head of the queue or null if the queue is empty. Similarly, if \( H_S \) is valid with respect to k-FIFO, k-FIFO-valid for short, then each dequeue operation in \( H_S \) returns one of the \( k \) oldest elements in the queue if the queue is not empty and each element is at most overtaken by \( k - 1 \) younger elements. Every FIFO-valid
sequential history is $k$-FIFO-valid.

We next define the notion of semantical deviation of a sequential history and characterize validity in terms of it. In order to do that we need the sequential specification of a POOL. Given a sequential history $H_S = op_1 \ldots op_m$,

$$\text{POOL}(H_S) = (q_0, o_0)(q_1, o_1) \ldots (q_m, o_m)$$

is the sequence of POOL states obtained from the sequential specification of Figure 3.1 with $k = n$ where $n = 1$ if $|q_i| = 0$ or $n = |q_i|$ if $|q_i| > 0$ for $i = 0 \ldots m$.

From $\text{POOL}(H_S)$ we define the sequence of lateness counters $l_0 \ldots l_m$ of $H_S$ by

$$l_j = \begin{cases} 
  o(e_i) & \text{if } q_j = e_1 \ldots e_n, q_{j+1} = e_1 \ldots e_i e_{i+1} \ldots e_n \\
  0 & \text{otherwise}
\end{cases}$$

We quantify bounded fairness through (maximum) lateness of $H_S$, denoted $L(H_S)$, which is the maximum number of younger elements that overtook an older element in $H_S$, i.e.,

$$L(H_S) = \max_{1 \leq j \leq m}(l_j).$$

The average lateness $ML(H_S)$ is the mean of the lateness counters of all elements returned in $H_S$, i.e.,

$$ML(H_S) = \text{mean}_{1 \leq j \leq m}(l_j).$$

From $\text{POOL}(H_S)$ we also define the sequence of (inverse) ages $a_0 a_1 \ldots a_m$ of $H_S$ by

$$a_j = \begin{cases} 
  i - 1 & \text{if } q_j = e_1 \ldots e_n, q_{j+1} = e_1 \ldots e_i e_{i+1} \ldots e_n \\
  0 & \text{otherwise}
\end{cases}$$

The (minimum inverse) age of $H_S$, denoted $A(H_S)$, is the (inverse) age of the youngest element ever returned in $H_S$, i.e.,

$$A(H_S) = \max_{1 \leq j \leq m}(a_j).$$

The average (inverse) age $MA(H_S)$ is the mean of the (inverse) ages of all elements returned in $H_S$, i.e.,

$$MA(H_S) = \text{mean}_{1 \leq j \leq m}(a_j).$$

We are now ready to present the characterization of $k$-FIFO validity in terms of semantical deviation, i.e., lateness and age.

**Proposition 1.** A sequential history $H_S$ is $k$-FIFO-valid if and only if $L(H_S) < k$ and $A(H_S) < k$. 
Finally, we recall the notion of linearizability [31] before introducing the remaining concepts. Given a history $H$ and a sequential specification $C$, $\text{lin}(H, C)$ denotes the set of all sequential histories that are equivalent to $H$ and valid with respect to $C$. If $\text{lin}(H, C)$ is not empty, $H$ is said to be linearizable with respect to $C$ [31]. Hence, $H$ is linearizable with respect to FIFO if there is a sequential history $H_S$ equivalent to $H$ that is FIFO-valid; it is linearizable with respect to $k$-FIFO if there is a sequential history $H_S$ equivalent to $H$ that is $k$-FIFO-valid. Note that every history linearizable with respect to FIFO is linearizable with respect to $k$-FIFO as well. A concurrent implementation of a sequential specification is said to be linearizable if all histories that can be obtained with the implementation are linearizable [31]. Linearizability is thus a consistency condition for specifying the semantics of objects in the presence of concurrency. The implementations of all discussed ($k$-FIFO) queues are linearizable.

In general, $\text{lin}(H, C)$ may contain more than one sequential history if $H$ is linearizable with respect to $C$. However, we are only interested in the sequential history $H_A$ in $\text{lin}(H, C)$ that represents the run that was actually performed. In particular, we are interested in the actual semantical deviation, i.e., $L(H_A)$ and $A(H_A)$. Unfortunately, $H_A$ cannot be determined on existing hardware without introducing prohibitive overhead. In practice, only $H$ can be obtained efficiently by time-stamping the invocation and response events of all operations. We therefore propose to approximate $H_A$ by computing two sequential histories $H_L$ and $H_H$ in $\text{lin}(H, C)$ such that

$$
L(H_L) = \min(\{L(H_S) | H_S \in \text{lin}(H, C)\})
$$

$$
A(H_L) = \min(\{A(H_S) | H_S \in \text{lin}(H, C)\})
$$

and, similarly for $H_H$ with $\min$ replaced by $\max$, holds.

The following proposition is a consequence of Proposition 1 and the definition of a $k$-FIFO queue.

**Proposition 2.** For all histories $H$ of a linearizable implementation of a $k$-FIFO queue we have that

$$
L(H_L) \leq L(H_A) \leq L(H_H) \leq k - 1
$$

$$
A(H_L) \leq A(H_A) \leq A(H_H) \leq k - 1
$$

We therefore call $k - 1$ the worst-case lateness (WCL) and worst-case age (WCA) of a $k$-FIFO queue. WCL and WCA represent the worst-case semantical deviation (WCSD) of a $k$-FIFO queue.
4.3.1 Computing $H_L$

We have designed and implemented a tool that computes $H_L$ from a given history $H$ without enumerating $\text{lin}(H,C)$ explicitly (assuming that the sequential specification $C$ is POOL not knowing any $k$ in particular). The tool scans $H$ for invocation events of dequeue operations in the order of their appearance in $H$ to construct $H_L$ in a single pass (and POOL($H_L$) to keep track of the queue states and lateness). For each invocation event $\langle op \rangle$ of a dequeue operation $op$ the following computation is performed until a linearization point for $op$ has been created: (1) if $op$ returns $null$ remember the (inverse) age for $op$ as zero, otherwise compute and remember the (inverse) age for $op$ assuming that the linearization point of the enqueue operation that matches $op$ is as far in the past as possible under the precedence constraints in $H$, (2) repeat (1) for all dequeue operations that overlap with $op$ and are not preceded by any other dequeue operations that also overlap with $op$, (3) among the dequeue operations considered in (1) and (2) find the dequeue operation $op'$ that returns an element other than $null$ and has the minimum remembered (inverse) age (any such $op'$ will do if multiple exist), or else if only dequeue operations that return $null$ have been considered in (1) and (2) then take any of those as $op'$, and finally (4) create a linearization point in $H_L$ for the enqueue operation that matches $op'$ and move that point under the precedence constraints in $H$ as far into the past as possible and create a linearization point for $op'$ in $H_L$ right before the invocation event $\langle op \rangle$. Note that after creating a linearization point for an operation its invocation and response events are not considered anymore in subsequent computations. The key insight for correctness is that bringing operations forward with minimum (inverse) age also minimizes lateness and thus produces $H_L$.

4.4 Experiments

We evaluate performance, scalability, and semantics of the Scal queue implementations described in Section 4.2. We compare our Scal queue implementations with the LB and MS FIFO queues described in Section 3.4, the BS and US $k$-FIFO queues introduced in Chapter 3, the RD and SQ quasi-linearizable FIFO implementations described in Section 3.4, and the BAG, ED, and RP pool implementations described in Section 3.4. The partial FIFO queues of the Scal queues are implemented with MS.

We ran the experiments on the MultiCore2 server machine. All benchmarked algo-
4. PERFORMANCE, SCALABILITY, AND SEMANTICS OF CONCURRENT FIFO QUEUES

Algorithms are implemented in C and compiled using gcc 4.3.3 with -O3 optimizations. In all experiments the benchmark threads are executed with real-time priorities to minimize system jitter. Each thread pre-allocates and touches a large block of memory to avoid subsequent demand paging, and then allocates and deallocates thread-locally all queue elements from this block to minimize cache misses and to avoid potential scalability issues introduced by the underlying memory allocator.

4.4.1 Microbenchmarks

We use the same benchmarking framework as introduced in Section 3.5. The framework emulates a multi-threaded producer-consumer workload where each thread is either a producer or a consumer. The framework can be configured for a different number of threads \(n\), number of enqueue or dequeue operations each thread performs \(o\), the computational load performed between each operation \(c\), the number of pre-filled items \(i\), and the queue implementation to use. The computational load \(c\) between two consecutive operations is created by iteratively calculating \(\pi\). A computation with \(c = 1000\) takes a total of 2300\(\)ns on average. We fix the operations per thread to \(o = 1000000\) and the number of producers and consumers to \(n/2\) for all benchmarks. We evaluate the performance and scalability of the queues under low \((c = 2000)\), medium \((c = 500)\), high \((c = 500)\) and very high \((c = 250)\) contention.

We evaluate each queue implementation with an increasing number of threads and determine its performance, scalability, and semantics. Performance is shown in number of operations performed per millisecond. Scalability is performance with an increasing number of threads. Semantics is average lateness \(ML(H_L)\) and average age \(MA(H_L)\) of \(H_L\) as computed by our tool described in Section 4.3.1.

For the RD, SQ, BS and US \(k\)-FIFO queues, and Scal queues we use \(r = s = k = p = 80\).

Figure 4.3(a) illustrates the results for the very high contention workload. TL-RR Scal performs best for up to 50 threads, due to the low sequential overhead of its load balancer. H-2RA Scal is the only algorithm that scales near-linearly and performs best for more than 50 threads. The other Scal algorithms also show good performance and scalability except for RR Scal, where the global round-robin counters become a scalability bottleneck. The best non-Scal algorithm is the BS \(k\)-FIFO queue. The remaining algorithms do not scale and perform well on the very high contention workload. The high contention workload depicted in Figure 4.3(b) results in a similar result. TL-RR Scal shows the best performance and scalability in the medium and low contention benchmarks, depicted in Figure 4.3(c) and Figure 4.3(d) respectively.
Figure 4.3: Performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine.

The average lateness and age of $H_L$ of the very high contention producer-consumer microbenchmarks are depicted in Figure 4.4. In Figure 4.4(a) and Figure 4.4(b) the queue is initially empty. In this scenario there are on average less than 80 elements in the queue. In Figure 4.4(c) and Figure 4.4(d) the queue is prefilled with 2000 elements. The results on average lateness and age are similar but not equivalent, see H-2RA Scal, for example. RR Scal has both lowest average lateness and age. The global round robin counters of the RR Scal load balancer serialize the threads resulting in a close to FIFO queue access pattern. The BS and US $k$-FIFO queues also show low average lateness and age. The random Scal queues result in high average lateness and age. However, the 2-random versions have lower average lateness and age than the regular...
4. PERFORMANCE, SCALABILITY, AND SEMANTICS OF CONCURRENT FIFO QUEUES

![Graphs showing performance metrics](image)

(a) Average lateness $ML(H_L)$ with $(c = 250, i = 0)$  
(b) Average age $MA(H_L)$ with $(c = 250, i = 0)$  
(c) Average lateness $ML(H_L)$ with $(c = 250, i = 2000)$  
(d) Average age $MA(H_L)$ with $(c = 250, i = 2000)$

Figure 4.4: Semantical deviation of very high contention producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine

random versions. TL-RR Scal has low average lateness and age in the workload where the queue is initially empty, but in the workload where the queue is prefilled its average lateness and age is similar to the result of the random Scal queues.

The average lateness and age of $H_L$ of the low contention producer-consumer microbenchmarks are depicted in Figure 4.5. In Figure 4.5(a) and Figure 4.5(b) the queue is initially empty. There the results can be grouped into three groups. In the first group are the RR and TL-RR Scal queues which both have low average lateness and age. In the second group are the BS and US $k$-FIFO queues with average lateness and age smaller than eight. In the third group are the random Scal queues. For them average
4.4 Experiments

Figure 4.5: Semantical deviation of low contention producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) server machine

Lateness and age increases with the number of threads. Figure 4.5(c) and Figure 4.5(d) depict the workload where the queue is initially prefilled. There the result is similar to the very high contention workload.

The TL-RR and H-2RA Scal queues appear to offer the best trade-off between performance, scalability, and semantics on the workload and hardware considered here. The BS and US $k$-FIFO queues with constant $k$ do not provide an advantage in terms of semantical deviation in a workload where the queue never contains more than $k$ elements, but bound semantical deviation independent of the queue size.
4. PERFORMANCE, SCALABILITY, AND SEMANTICS OF CONCURRENT FIFO QUEUES

4.4.2 Macrobenchmarks

We ran macrobenchmarks on the MultiCore2 server machine to analyze the performance impact and the applicability of our Scal queues and $k$-FIFO queues.

Transitive Closure and Spanning Tree Graph Algorithms

We ran two macrobenchmarks with parallel versions of transitive closure and spanning tree graph algorithms [11] using random graphs consisting of 100000 vertices where 100000000 unique edges got randomly added to the vertices. The used shared data structure is prefilled with 1000 randomly determined vertices. From then on each thread iterates over the neighbors of a given vertex and tries to process them (transitive closure or spanning tree operation). If a neighboring vertex already got processed by a different thread then the vertex is ignored. Otherwise, the vertex is processed and then added to a global queue. When a thread processed all neighbors of a given vertex it gets a new vertex from the global queue. The graph algorithm terminates when the global queue is empty.

(a) Spanning tree performance and scalability  (b) Transitive closure performance and scalability

Figure 4.6: Macrobenchmarks on a random graph with 100000 vertices and 100000000 edges with an increasing number of threads on the Intel-based 40-core server machine

The spanning tree and transitive closure macrobenchmark results are presented in Figure 4.6. Each run was repeated 10 times. We present the average execution time of the 10 runs as our metric of performance, less execution time is better. The RP pool is not used in this benchmark since it cannot handle a workload where producers are also consumers. In both, the spanning tree and the transitive closure macrobenchmarks the Scal queues based on hierarchical load balancing provide the best performance and
scalability. The BAG algorithm performs best for up to 10 threads. After that it scales negatively.

Mandelbrot Algorithm

We computed and rendered two images of the Mandelbrot set \cite{51} using producer and consumer threads which exchange data using a shared data structure. The producer threads divide the image into smaller blocks (in our experiments $4 \times 4$ pixels), write block coordinates in descriptor blocks, and enqueue the descriptor blocks in the shared data structure. The consumer threads dequeue the descriptor blocks from the shared data structure, perform the Mandelbrot calculation on the blocks, and store the results in the corresponding blocks of the final Mandelbrot image. Hence, the workload between the consumer threads is balanced. We use a producers-consumers ratio of 1 : 4 in our experiments, i.e. for each producer thread we add four consumer threads.

![Mandelbrot algorithm graphs](a) Low computational load (b) High computational load

Figure 4.7: Mandelbrot macrobenchmarks with block size $4 \times 4$ pixels and an increasing number of threads on the Intel-based 40-core server machine

The Mandelbrot macrobenchmark results are presented in Figure 4.7. Each run was repeated 10 times. We present the average execution time of the 10 runs as our metric of performance, less execution time is better. Figure 4.7(a) shows the performance of the low computational load Mandelbrot benchmark. Low computational load means that the Mandelbrot computation can be performed fast for most of the blocks. TL-RR Scal and RA Scal show the best performance, followed by the remaining Scal algorithms (except RR Scal) and the BS and US $k$-FIFO queue algorithms. The result of the high computational load Mandelbrot benchmark is depicted in Figure 4.7(b). High computational load means that the Mandelbrot computations are computationally
intensive for most of the blocks. In that benchmark all Scal queues (except RR Scal) show identical performance and scalability. The BS and US $k$-FIFO queues also provide performance and scalability.

4.5 Summary

We have introduced Scal queues, which aim at improving performance and scalability of FIFO queue implementations through load balancing by distributing queueing operations across multiple, independent queue instances. Load balancing directly determines performance, scalability, and semantics of Scal queues, in particular how close the queueing behavior is to FIFO. In order to quantify the difference between actual and ideal FIFO semantics, we have introduced the notion of semantical deviation, which captures how many dequeue operations overtook older elements (lateness) and what the age of dequeued elements was. We showed in micro- and macrobenchmarks that the Scal queue implementations outperform and out-scale many state-of-the-art concurrent queue and pool algorithms. The TL-RR Scal queue offers the best trade-off between performance, scalability, and semantics in the microbenchmarks. The TL-RR Scal queue and the Scal queues based on randomized load balancing show the best performance in the macrobenchmarks.
Part II

Concurrent Heap Management Systems
In this chapter we study, formally and experimentally, the trade-off in temporal and spatial overhead when managing contiguous blocks of memory using the explicit, dynamic and real-time heap management system Compact-fit (CF) [63, 19, 20]. The key property of CF is that temporal and spatial overhead can be bounded, related, and predicted in constant time through the notion of partial and incremental compaction. Partial compaction [63, 19] determines the maximally tolerated degree of memory fragmentation. Incremental compaction [20] of objects, introduced here, determines the maximal amount of memory involved in any, logically atomic portion of a compaction operation. We explore CF’s potential application space on (1) multiprocessor and multicore systems as well as on (2) memory-constrained uniprocessor systems. For (1), we argue that little or no compaction is likely to avoid the worst case in temporal as well as spatial overhead but also observe that scalability only improves by a constant factor. Scalability can be further improved significantly by reducing overall data sharing through separate instances of Compact-fit. For (2), we observe that incremental compaction can effectively trade-off throughput and memory fragmentation for lower latency.

5.1 Introduction

Compact-fit (CF) [63, 19] is an explicit, dynamic and real-time heap management system also known as a memory allocator. Heap management solves the problem of allocating and deallocating memory objects of possibly different size where the order
5.1 Introduction

in which the objects are allocated and deallocated may be arbitrary. It is dynamic if allocation and deallocation happens at runtime, as opposed to static, so-called pre-allocation, which may only be done when the amount of memory needed for program execution can be bounded at compile time. It is real-time if the time to allocate, deallocate, and access a memory object is either constant or at most proportional to the size of the object, independent of the overall state of memory and in particular the order in which objects are allocated and deallocated.

Heap management is explicit if deallocation must be invoked by the program using the system. The use of explicit heap management may therefore suffer from the well-known phenomena of memory leaks, where memory objects are continuously allocated but never deallocated, and so-called dangling pointers to memory objects that have been deallocated prematurely and may lead to undefined program behavior when accessed. Compact-fit is not an exception. It does not address the problems of memory leaks and dangling pointers.

Implicit heap management solves the problem of dangling pointers by first determining when to deallocate memory objects safely and then using an underlying explicit heap management to actually deallocate these objects. Garbage collectors are implicit heap management systems which logically operate in two phases that are performed repeatedly. In the first phase allocated but unreachable memory objects are determined, either directly through reference counting and cycle detection, or indirectly through tracing, or some combination of both [8]. A memory object is unreachable if the program has no means of accessing the object, neither directly through some reference nor transitively through other, reachable memory objects. An unreachable memory object may thus be deallocated safely without introducing a dangling pointer.

In the second phase unreachable memory objects are deallocated which is done by an underlying memory allocator. In other words, a garbage collector implicitly uses a memory allocator which is, however, often tightly integrated with the garbage collector for performance reasons. Compact-fit is closely related to the integrated memory allocator of the real-time garbage collector Metronome [10].

Garbage collectors solve the problem of dangling pointers but not the problem of memory leaks: a program may continuously allocate memory objects to which it maintains references, e.g. by inserting the objects into an ever-growing hashtable and never removing them again. The result is a so-called reachable memory leak which garbage collectors cannot avoid. In other words, using implicit heap management does not free a program from deallocating memory objects. It only makes deallocation implicit (remove references and return from procedure calls rather than deallocate
explicitly) and is therefore safe, i.e., the program still needs to go through the otherwise ever-growing hashtable and remove obsolete data from time to time. Fundamentally, reachable memory leaks are not detected by garbage collectors because reachability is only an overapproximation of memory liveness which itself is undecidable: after some time a memory object may never be accessed again but still remain reachable.

The true power of garbage collection is that it makes the use of the heap compositional. Programs may allocate memory objects, even in imported library code, and pass references to them around without keeping track of when to deallocate the objects as long as the references are eventually removed when the objects are not needed anymore. Compositionality of the heap is key to large-scale program design and particularly useful in concurrent programs where keeping track of when shared memory may be deallocated is especially difficult.

The price to pay for garbage collection is temporal and spatial overhead: computing unreachability (directly or indirectly) is proportional to the size of live, i.e., reachable memory (in the presence of cyclic references which is the case in most non-trivial applications), resulting in lagged deallocation of unreachable memory and thus increased memory consumption. Temporal overhead, when created in so-called stop-the-world fashion, precludes real-time applications. Spatial overhead precludes embedded applications, in particular if deallocation not only lags unreachability but also results in uncontrolled memory fragmentation which may also occur in explicit heap management without any garbage collection.

Temporal and spatial overhead of dynamic heap management, explicit or implicit, cannot be avoided but it can be bounded! The key to enabling dynamic heap management in real-time and embedded applications is to make it incremental and to bound memory fragmentation. Heap management is incremental if it may be done in phases whose duration are constant and which may be interleaved with program execution. The maximum duration of a heap management phase determines the latency introduced by heap management and thus directly defines the compatible class of real-time applications. The drawback of incremental heap management is lower throughput since the sum of the phases of a heap management operation is generally larger than the duration of the operation when not interrupted. Note that the focus of this paper is on making explicit heap management incremental while bounding memory fragmentation, which is one of the two fundamental prerequisites for incremental garbage collection. The other prerequisite is incrementally computing unreachability which is addressed elsewhere, e.g. in Metronome [10].

Memory fragmentation is the phenomenon of unoccupied memory blocks being dis-
persed in memory (external fragmentation) and/or designated through partitioning (internal fragmentation). If contiguous memory blocks of different size may be allocated and deallocated in arbitrary order, uncontrolled memory fragmentation may lead to unbounded gross memory consumption even if net memory consumption is bounded.

Compact-fit avoids external fragmentation and bounds internal fragmentation through partitioning and so-called partial compaction. Upon deallocating a memory object partial compaction may move another same-size object into its place but only if a given threshold on fragmentation is exceeded. In this case, deallocation takes time linear in the size of the deallocated object. Otherwise, deallocation is constant-time. Memory allocation as well as access are always constant-time. The principle topic of this paper is to make partial compaction incremental such that objects are moved incrementally in phases of constant duration and yet may still be accessed in constant time in between compaction phases. The result is what we call incremental Compact-fit, the first memory allocator that bounds the full spectrum of temporal and spatial overhead of memory allocation, deallocation, and access in terms of configurable constants. With incremental Compact-fit the duration of any heap management activity as well as the degree of memory fragmentation are bounded by constants, which makes this allocator the principle choice for any applications in which constant bounds on both temporal and spatial overhead are required.

Note that there are memory allocators that either bound temporal overhead in terms of constants such as Half-fit [58] and TLSF [53] or else spatial overhead such as the allocator of the Jamaica VM [74] but not both. Half-fit and TLSF provide constant-time memory allocation, deallocation, and access, but only control and not bound memory fragmentation through coalescing neighboring, unoccupied memory blocks. The Jamaica allocator avoids external fragmentation and bounds internal fragmentation but at the expense of constant-time memory allocation, deallocation, and access as well as memory locality by allocating small, same-size but generally dispersed memory blocks and assembling them into larger memory objects through trees (logarithmic-time access) or lists (linear-time access) that fit the requested size.

Next, we discuss the design principles and features of Compact-fit before providing an overview of the rest of the paper.

5.1.1 Compact-fit

Compact-fit partitions memory into virtual pages of equal size by maintaining a list of free pages and a segregated list of finitely many so-called size-classes where each size-class is a doubly-linked list of used pages that are further partitioned into virtual, so-
5. CONCURRENCY AND SCALABILITY VERSUS FRAGMENTATION AND COMPACTION WITH COMPACT-FIT

called page-blocks of equal and unique size. A memory object is allocated as contiguous block of memory in a free page-block of the size-class with the smallest page-block size that still fits the object. Memory allocation, deallocation, and access takes constant time (unless compaction is necessary when deallocating, which takes linear time in the size of the deallocated object). Allocation of memory objects larger than the page size is not part of CF itself but may be done on top of CF by array, tree-, or list-based data structures that combine sufficiently many pages to accommodate large objects resulting in allocation and deallocation times that are linear and memory access times that are constant, logarithmic, or linear, respectively, in the size of the objects. However, we do not consider large-object management here.

The size-class concept is generally subject to fragmentation through partitioning, that is, to bounded page-block-internal, page-internal, and size-external fragmentation [9], but enables CF to keep memory size-class-compact at all times [63, 19]. Memory is size-class-compact if each of its size-classes is compact. A size-class is compact with respect to a so-called partial compaction bound $\kappa$ if the size-class contains only non-empty pages of which at most $\kappa$ are not-full. A size-class is said to be totally compact, fully compact, or partially compact if it is compact with respect to $\kappa = 0$, $\kappa = 1$, or $\kappa > 1$, respectively. Note that, as opposed to the leftover space caused by fragmentation through partitioning, which is wasted for any request, the free space in not-full pages of a size-class, called size-class fragmentation, is wasted for any request but the requests that actually match the size-class. Partial compaction can only control the degree of size-class fragmentation.

CF always keeps all size-classes compact with respect to individual, per-size-class partial compaction bounds $\kappa > 0$. Overall memory fragmentation is therefore bounded and predictable in constant time. Note that $\kappa = \infty$ is also permissible and means that any number of not-full pages in a size-class is tolerated. A memory object is allocated, in constant time, in a free page-block either of a not-full page of the adequate size-class (implicitly compacting allocation), or else, if there is no not-full page in the size-class, of a free page that is then removed from the list of free pages and assigned to the size-class (non-compacting allocation). A memory object is deallocated, either in constant time, by marking the page-block used by the object as free, if the size-class remains partially compact (non-compacting deallocation), or else in linear time in the size of the object, by marking a used page-block of a not-full, so-called source page as free after copying the content of that (source) page-block to the (target) page-block used by the object, which, in this case, must be located in a full, so-called target page (compacting deallocation). If the page in which a page-block was marked as free becomes empty,
5.1 Introduction

In order to facilitate compacting memory that may contain references in time linear in the size of the moved objects, CF maintains a map (A2C) from abstract object addresses that do not change when moving objects, also referred to as handles, to the concrete object addresses in memory. Objects may only refer to other objects using their abstract addresses, which implies that memory access requires one level of indirection, unless compaction is turned off with $\kappa = \infty$. As a result, whenever an object is moved in memory, only its concrete address in the A2C map needs to be updated. CF stores the abstract address of each object in the object itself so that the object’s entry in the A2C map can be determined in constant time. Otherwise, determining the abstract addresses of objects selected for compaction, for which only the concrete addresses are known, would require searching the A2C map.

There is also a non-moving version of CF [63, 19], which virtualizes the concrete address space using an additional level of indirection that merely requires reprogramming a map (V2P) from virtual to physical addresses upon compaction instead of moving the actual content of the objects. Since objects do not move, their physical addresses can be used to generate unique abstract addresses, which avoids storing abstract addresses in objects. Nevertheless, in the worst case, the V2P map requires just as much memory as the object storage for abstract addresses. Moreover, experiments have shown that the non-moving version of CF may only pay off when used for larger objects [63, 19]. In the rest of the article, we only consider the moving version of CF.

5.1.2 Overview

After discussing related work (Section 5.2) and discussing the previously described, moving (and non-incremental) version of CF in detail (Section 5.3), we first argue probabilistically that, for any mutator behavior, both compaction and worst-case size-class fragmentation are less likely to happen with increasing partial compaction bounds $\kappa$. For systems whose memory resources are less constrained and applications that do not require tight guarantees, partial compaction may therefore be set to large $\kappa$, or even turned off entirely. This observation has lead us to develop an optimized, non-compacting version of CF without abstract addressing that does not maintain the A2C map and can therefore be used in any application without modifications. Macrobenchmarks show that the optimized version performs almost as fast as other constant-time state-of-the-art memory allocators. Moreover, less than 5% of the fragmentation can be attributed to size-class fragmentation and the rest to fragmentation through partitioning (Section 5.8). We argue that partitioning memory as in CF still has the benefit
of being subject to a probabilistic and not just an experimental fragmentation analysis (Section 5.4), at the expense of increased memory consumption.

We then introduce incremental CF for slow systems, at the other end of the spectrum, whose memory resources are constrained and that run applications requiring tight guarantees, in particular on system latency and memory consumption (Section 5.5). Incremental CF uses a global compaction increment $\lambda > 0$, which breaks up compaction into logically atomic operations that do not move more than $\lambda$ bytes at a time. If $n$ is the degree of concurrency, then there may be at most $n$ pending incremental compaction operations moving objects stored in $n$ source page-blocks from $n$ source pages to $n$ target pages. The memory occupied by the $n$ source page-blocks causes so-called transient size-class fragmentation in the $n$ source pages. The key result is that the time complexity of memory allocation, deallocation, and access remains asymptotically the same as with non-incremental CF while overall memory fragmentation is still bounded and predictable in constant time (Section 5.6). Incremental CF may improve system latency at the expense of allocation and deallocation throughput and transient size-class fragmentation (Section 5.8).

Figure 5.1: Deallocation throughput, system latency, and memory fragmentation with different versions and configurations of Compact-fit

Figure 5.1 gives an intuitive overview of the effect of different versions and configurations of CF on allocation and deallocation throughput, system latency, and memory fragmentation. A configuration $1\text{-CF}(\kappa, \lambda)$ denotes a single instance of a CF system with a per-size-class partial compaction bound $\kappa > 0$ and a global compaction incre-
ment $\iota > 0$. The instance may be shared by concurrently running threads using a number of different, standard synchronization techniques (Section 3.3). Incremental compaction is off if $\iota = \infty$. Partial compaction is off if $\kappa = \infty$, which implies that incremental compaction is also off. Full compaction is on if $\kappa = 1$. The fully compacting, non-incremental $1$-CF$(1, \infty)$ configuration minimizes memory fragmentation at the expense of throughput and latency. In comparison, the fully compacting, incremental $1$-CF$(1, \iota)$ configuration may require more memory because of transient size-class fragmentation and provide less throughput but may reduce latency. With $\kappa > 1$, memory fragmentation may go up proportionally to $\kappa$ with both configurations while throughput may be higher and latency may be lower as there may be fewer compaction operations. The non-compacting $1$-CF$(\infty, \infty)$ configuration may provide even higher throughput and lower latency but may also consume even more memory. The key advantage of this configuration is that it may be optimized as mentioned above.

A configuration $n$-CF$(\kappa, \iota)$ denotes $n$ instances of a CF system, one for each of $n$ threads, which is meant to improve scalability on multiprocessor and multicore systems (Section 5.7). Compared to the single-instance configurations, throughput may be higher but memory fragmentation may also go up with the compacting configurations since partial compaction bounds are enforced per instance and therefore per thread. Our experiments show that partial compaction on fast systems may only have an effect on scalability by a constant factor since the time required to perform a single compaction operation on such systems is close to the time required to perform any other CF operation, independently of the size of the involved object. More relevant to scalability is the degree of data sharing, in particular, through the A2C map (Section 5.8).

The contributions of this article are the design, implementation, and comprehensive, formal and experimental evaluation of concurrent versions of (1) an optimized, non-compacting CF system, (2) the previously described, compacting, non-incremental CF system [63, 19], and (3) a new, compacting, incremental CF system.

## 5.2 Related Work

We relate our work to dynamic heap management systems of different kinds: explicit sequential allocators, explicit concurrent allocators, and concurrent garbage-collection-based systems with compaction (cf. [36] for an extensive online bibliography).

Most of the established explicit sequential dynamic heap management systems [52, 65] are optimized to offer excellent best-case and average-case response times, but in the worst-case are unbounded in the size of the memory allocation or deallocation request,
5. CONCURRENCY AND SCALABILITY VERSUS FRAGMENTATION AND COMPACTION WITH COMPACT-FIT

i.e., depend on the global state of memory. The best known are First-fit, Best-fit \[44\] and DL \[46\] with allocation times depending on the global state of memory. Half-fit \[58\] and TLSF \[53\] are exceptions offering constant response-time bounds for allocation and deallocation, but even they may suffer from unbounded and unpredictable memory fragmentation.

Several concurrent dynamic memory allocators have been designed for scalable performance on multiprocessor systems. Hoard \[14\] provides fast and scalable memory allocation by avoiding false sharing of cache lines. A lock-free memory allocator with lower latency based on the principles of Hoard is given in \[54\]. A partly lock-free non-portable memory allocator, which requires special operating system support, is discussed in \[22\]. McRT-Malloc \[32\] is a non-blocking scalable heap management algorithm, which avoids atomic operations on typical code paths by accessing only thread-local data and uses the same memory layout (pages and size-classes) as CF. None of these systems provides temporal or spatial guarantees.

There are many concurrent compaction strategies implemented in garbage-collected systems, which do not provide temporal or spatial guarantees. In \[25\] a parallel stop-the-world memory compaction algorithm is given, where multiple threads compact the whole heap. Compressor \[38\] is a concurrent, parallel, and incremental compaction algorithm which compacts the whole heap during a single heap pass, achieving perfect compaction. A further parallel incremental compaction approach is presented in \[12\] where the heap is split into pieces which are compacted one at a time by moving objects to a new memory region. A fixup pass takes care of reference updates. An algorithm with improved compaction pause times via concurrent reference updates, using only half of the heap, is given in \[59\]. Each thread performs reference updates proportional to its allocation requests.

Garbage-collecting heap management systems that do provide response-time guarantees on allocation and deallocation operations are Jamaica \[74\] and Metronome \[10\]. With Jamaica allocation and deallocation take linear time in the size of the operation request. Compaction is not needed since memory objects do not occupy contiguous blocks of memory. Metronome is a time-triggered garbage collector, which uses the same memory layout as CF. Compaction in Metronome is part of the garbage collection cycles. The time used for compaction is estimated to at most 6% of the collection time \[9\], without precise guarantees. The performance of Metronome depends highly on the mutator behavior. MC² \[69\] is an incremental soft real-time garbage collector designed for memory constrained devices, which cannot provide hard guarantees on maximum pause time and CPU utilization, but comes with low space overhead and
tight space bounds. Stopless [64] is another garbage collector with soft guarantees on response times. It provides low latency while preserving lock-freedom, supporting atomic operations, controlling fragmentation by compaction, and supporting multiprocessor platforms. The main contribution of Stopless is a compaction algorithm which moves objects in the heap concurrently with program execution. Exact bounds for response times, as well as fragmentation, are missing in Stopless. In [37] the authors show experimentally that the cost of handles in a real-time garbage collector is negligible in comparison to implementations that do not use handles. CF will take advantage of handles.

We remark that CF, like many of the above mentioned systems, is based on segregated lists. Approaches that are not based on segregated lists, but rather on data structures which maintain locality of objects, are known to perform better when accessing objects by utilizing memory caches more effectively. However, the use of segregated lists enables providing and trading off temporal and spatial guarantees.

5.3 Non-incremental Compact-fit

CF is a dynamic heap management system that provides strict temporal and spatial (fragmentation) guarantees. Allocation as well as deallocation without compaction takes constant time, whereas deallocation with compaction takes linear time in the size of the object. To be precise, there are two CF implementations [63, 19], but we only focus on the more fundamental so-called moving implementation.

The set-up of CF is as follows: The memory is divided in pages of equal size. Each page (in use) contains a certain number of constant-sized page-blocks. In total there are finitely many available page-block sizes, which determine to which size-class a page belongs (namely all pages with a given page-block size belong to one size-class). The pages are assigned to a size-class only if they are used (non-empty). The number of page-blocks $\pi$ per page in a size-class is therefore determined by the size of a page and the block size. The state of a size-class depends on the state of the pages that belong to it and is described by the values of the variable tuple $\langle h, n, u_1, \ldots, u_n \rangle$ where $h$ is the total number of allocated page-blocks in the size-class (its portion of the heap), $n$ is the number of not-full pages, and for each not-full page $i$, $u_i$ is the number of used page-blocks in the page.

An allocation request for an object of size $l$ is served by a page of a best-fitting size-class. That is, for allocating an object a single page-block is used in a page whose
page-blocks are of the smallest size still big enough to fit \( l \). For example, if there are two size-classes, one with page-blocks of size 10 and one with page-blocks of size 20 units, then an allocation request for an object of size \( l \in \{11, 12, \ldots 20\} \) will be served by a page of the size-class 20. If all pages in the best-fitting size-class are full, then a new empty page is added to the size-class and the object is allocated in this new page.

We allow for a constant number \( \kappa > 0 \) of not-full pages per size-class. The aim in the design of CF is to control size-class fragmentation, which is the space occupied by free page-blocks in not-full pages (space not available for allocation in other size-classes). If deallocation happens, and the number of not-full pages becomes \( \kappa + 1 \) after this deallocation operation, then compaction is invoked. Compaction consists of moving a single object from a not-full page to the page-block of the deallocated object, which is the only empty page-block in that page. As a result, after compaction, the number of not-full pages in a size-class does not exceed \( \kappa \).

An object is assigned a unique abstract address (handle), which has to be dereferenced whenever accessing an object field. This introduces a constant object dereferencing overhead but facilitates predictability of reference updates during compaction, i.e., whenever an object is moved in memory it requires to update just its abstract address space entry.

Figure 5.2: Size-class automaton with \( \pi > 1 \)

We show the CF algorithm in full detail in Figure 5.2, using a deterministic automaton, one per size-class. For presentation purposes, we draw a quotient of the state space of the size-class: EMPTY stands for the single state \( (0, 0) \) representing an empty size-class; NOT-FULL represents all states with at least one not-full page where no compaction is needed, that is \( (h, n, u_1, \ldots u_n) \) with \( 0 < n \leq \kappa \); the state FULL represents
all states with no not-full pages and at least one full page, that is \( \langle h, 0 \rangle \) with \( h > 0 \); finally, \textsc{compaction} represents states \( \langle h, \kappa + 1, u_1, \ldots, u_{\kappa+1} \rangle \) in which compaction must be invoked.

The transitions in the automaton are labelled in the following way: \( A \) denotes allocation, \( D_i \) deallocation in page \( i \) (which may be full or not-full, the latter is recognized by \( i \leq n \)), and \( C \) denotes a compaction step. Moreover, a transition fires if its premise is satisfied, and results in a change of state described by its conclusion. For updating a state, we use the operators \( \leftarrow \) for assignment, \( \text{dec} \) for decrement, \( \text{inc} \) for increment, and \( \text{sl} \) for shift left. More precisely, \( \text{sl}(i) \) removes \( u_i \) from a state sequence, i.e., it changes a state \( \langle h, n, u_1, \ldots, u_{i-1}, u_i, u_{i+1}, \ldots, u_n \rangle \) to the sequence \( \langle h, n, u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n \rangle \).

We explain several instructive transitions in full detail, and refer the reader to Figure 5.2 for the full algorithm.

\[ A \left( h \leftarrow 1, n \leftarrow 1, u_1 \leftarrow 1 \right) \text{ from empty to not-full} \]

This transition fires whenever allocation is requested in the empty state. As a result the state changes to \( \langle 1, 1, 1 \rangle \).

\[ D_i \left( \frac{i \leq n, u_i \geq 1}{\text{dec}(h), \text{dec}(u_i)} \right) \text{ from not-full to not-full} \]

This transition is taken upon a deallocation step in a not-full page which remains non-empty after the deallocation. The change in the state is that the number of used page-blocks is decremented by 1, and, as in every deallocation step, the heap size decreases by 1.

\[ A \left( \frac{\text{inc}(h), n \leftarrow 1, u_1 \leftarrow 1}{\text{inc}(h), n \leftarrow 1, u_1 \leftarrow 1} \text{ from full to not-full} \right) \]

Whenever an object is allocated in a state of the class \textit{full} a new empty page has to be added to the size-class, and allocation happens in this page. As a result this new page becomes the only not-full page of the size-class with a single page-block used. The value of \( h \) increases by one, as with any allocation operation.

\[ D_i \left( \frac{n=\kappa, i \geq n}{\text{dec}(h), \text{inc}(n), u_i \leftarrow \pi - 1} \right) \text{ from not-full to compaction} \]

With this transition we are in a situation when after the required deallocation
operation, in the \( i \)-th page which was full, we have more than \( \kappa \) not-full pages. Therefore, compaction must be invoked in the next step.

\[
C\left(\frac{u_1=1, n>2}{\text{sl}(1), \text{dec}(\text{dec}(n))}\right) \text{ from COMPACTION to NOT-FULL}
\]

Being in state COMPACTION, the next transition has to be of type \( C \). Moreover, note that \( n = \kappa + 1 \geq 2 \). During the compaction step a page-block is moved from the first not-full page (represented by \( u_1 \)) to the last not-full page, namely the one in which deallocation just happened. This particular transition fires if the first not-full page has just one page-block. As a result it becomes empty after the transition, whereas the last not-full page becomes full. Since \( n > 2 \) the transition leads to the state NOT-FULL. The operation shift left is needed to remove the value \( u_1 \) for the now empty page.

\[
\text{EMPTY} \quad A\left(\frac{n=1}{\text{dec}(\text{dec}(n))}\right) \quad \text{FULL}
\]

\[
D\left(\frac{n=2}{\text{dec}(\text{dec}(n))}\right)
\]

Figure 5.3: Size-class automaton with \( \pi = 1 \)

We note that in case \( \pi = 1 \), i.e., in a size-class in which each page consists of exactly one page-block, there are no not-full pages. A page is either empty or full. In this case compaction can never happen. Therefore, the size-class automaton simplifies significantly as shown in Figure 5.3.

We have chosen the automaton presentation of CF in order to prepare the ground for the concurrent version. For the original presentation of CF, we refer the interested reader to [63, 19]. We extend the non-incremental CF with blocking and non-blocking synchronization mechanisms so that multiple threads can share a single (or multiple) instance(s). In particular, we make the size-class automaton transitions (including a combination of a deallocating transition followed by a compacting step) atomic. As a result, multiple threads can execute and use CF in parallel, interleaving between the atomic transitions. The details of the particular implementation and the various choices of synchronization mechanisms are discussed in Section 5.7. The results are encouraging for throughput oriented environments, see Section 5.8.
5.4 Probabilistic Analysis

We present a probabilistic analysis of CF which shows that, for any mutator behavior, both compaction and worst-case size-class fragmentation are less likely to happen with increasing partial compaction bounds $\kappa$. Compaction may therefore be set to large $\kappa$ or even turned off if guarantees on memory fragmentation are not required. A more detailed CF analysis involving different classes of mutator behavior may provide even more insight but is outside the scope of this paper and remains for future work. Interestingly, it is the partitioned memory layout of CF that allows for such an analysis, since the partitioning into pages and size-classes significantly reduces the state space of the model. Other memory allocators may not allow such an analysis.

1. What is the probability that compaction happens?
2. What is the probability of worst-case fragmentation?

We analyze the behavior of CF given a mutator, which is a sequence of allocations $A$ and deallocations $D$, hence a word in $\{A, D\}^*$. A mutator is not aware of the internal CF configuration, e.g. in which page deallocation happens. Therefore, we abstract away from the index $i$ in the deallocation label $D_i$ and the CF size-class automaton becomes a probabilistic I/O automaton (PIOA) [81], with input actions $A$ and $D$ provided by the mutator, and an output action $C$ provided by CF. Note that the states of this automaton are either input states in which $A$ and $D$ are enabled, or output states in which $C$ is enforced, which makes it simpler than general PIOA. For brevity we only discuss the behavior of a single state. In a state $\langle h, n, u_1, \ldots, u_n \rangle$ with $n \leq \kappa$, upon deallocation $D$, there are several possible next states that are reached with different probabilities: for all $i$ with $u_i > 1$, with probability $\frac{u_i}{h}$ deallocation happens in the not-full page $i$ which will remain not-full afterwards and the next state becomes $\langle h - 1, n, u_1, \ldots, u_{i-1}, u_i - 1, u_{i+1}, \ldots, u_n \rangle$; for all $i$ such that $u_i = 1$ with probability $\frac{1}{h}$ deallocation happens in page $i$ reducing the number of not-full pages and the next state is $\langle h - 1, n - 1, u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n \rangle$; and with probability $\frac{h - \sum u_i}{h}$ the next state is $\langle h - 1, n + 1, u_1, \ldots, u_n, \pi - 1 \rangle$ as deallocation happens in a full page. The allocation and compaction transitions remain the same as in the deterministic automaton, they happen with probability $1$ in states in which they are enabled. This way we get the full PIOA model, with initial state $\langle 0, 0 \rangle$.

The full PIOA model together with a mutator induces a discrete-time Markov chain, the full DTMC, by pruning out the allocation/deallocation possibilities that the mutator does not prescribe in each state and abstracting away from the transition labels.
5. CONCURRENCY AND SCALABILITY VERSUS FRAGMENTATION AND COMPACTION WITH COMPACT-FIT

The full DTMC model results in a large state space already for small values of \( h, \pi, \) and \( \kappa \).

To reduce the number of states, we consider only mutators of the shape \( A^hD^d \) which perform \( h \) allocations followed by \( d \) deallocations. We analyze portions of the full model by setting the state reached after performing \( h \) allocations as initial state. This is the state \( \langle h, 0 \rangle \) if \( h \mod \pi = 0 \), or \( \langle h, 1, h \mod \pi \rangle \) otherwise. Then we consider the portion of the full model reachable in \( d \) deallocations. We refer to \( d \) as the deallocation level. Even such versions of the full model are too big: for \( h = 80, \pi = 10, \) and \( \kappa = 5 \) the DTMC model in Prism has 1429506 states and 2818395 transitions, and for \( h = 80, \pi = 10, \) and \( \kappa = 6 \), Prism runs out of memory.

![Probability of compaction and worst-case fragmentation](image)

Figure 5.4: Probability of reaching compaction and worst-case fragmentation

The probability of compaction is the probability of reaching a compacting state, i.e., a state with \( n = \kappa + 1 \). The probability of reaching a specified state in a DTMC is the sum over all paths of the probability of reaching the state along a path, where the probability of reaching the state along a path is calculated as the product of the
5.5 Incremental Compact-fit

For applications which require low latency and run on memory-constrained systems, we provide an extension of CF that allows for incremental compaction, i.e., incremental moving of a single object.

The incremental extension of CF performs compaction, i.e., moving of a single object, by an incremental moving operation. The reason why compaction is made incremental is its dominating linear complexity. This incremental extension is the first step towards a design of latency-efficient concurrent CF. For a concurrent incremental version of CF, allocation, deallocation, and incremental compaction are made atomic, leaving space for other interleaving threads between the atomic steps. As a result the waiting times of concurrent threads, and therefore their response times, decrease, although the compaction throughput may also decrease.

There is a global fixed compaction increment $\iota > 0$ which determines the portion of a page-block being moved in an incremental step. The value of $\iota$ may even be larger
than some page-block sizes, in which case the whole compaction operation is done non-incrementally, in one step. We refer to a page-block under incremental moving as the source page-block, and the page-block to which the object is moved as the target page-block. The state of each size-class and its administration gain complexity in the incremental version. In a size-class, apart from the full and not-full pages, there may exist one source page. In a source page there are used page-blocks and source page-blocks. The latter are page-blocks that are in the process of being incrementally moved. One source page suffices, since compaction in CF requires moving a used page-block which is now always taken from the source page. Allocation never happens in a source page. A source page always contains at least one used page-block. If a source page looses all its used page-blocks (due to deallocation or compaction), it is removed from the size-class and placed into a global pool $E$ of emptying source pages. All pages in the pool contain page-blocks that are involved in ongoing incremental compaction operations. The space occupied by source page-blocks and free page-blocks in (emptying) source pages, which is (temporarily) not available for allocation in any size-class, is called transient size-class fragmentation. When all incremental compaction operations in an emptying source page finish, then the page is returned to the global list of free pages. On the other hand, if all incremental compaction operations within a source page finish, i.e., the source page has no more source page-blocks, and if there are still used page-blocks in the source page, then there are two possibilities: (1) the source page becomes a not-full page, if the number of not-full pages is smaller than the partial compaction bound, or (2) the source page is kept as a potential source page without source page-blocks, otherwise. The evolution of a page is shown in Figure 5.5.

The state of a size-class is described by a tuple

$\langle h, n, u_1, \ldots, u_n, s, m_1, \ldots, m_s \rangle$

where, as before, $h$ denotes the current heap size, $n$ is the number of not-full pages such
that \( n \leq \kappa + 1 \) with \( \kappa \) being the partial compaction bound, and the values of \( u_1, \ldots, u_n \) are the numbers of used page-blocks in the not-full pages, respectively. The value of \( u_s \) equals the number of used page-blocks in the source page, with \( u_s = 0 \) representing that there is no source page in the size-class. The variable \( s \) contains the number of source page-blocks in the source page and equals 0 if there is no source page. Note that \( s = 0 \) and \( u_s > 0 \) represents the existence of a potential source page, as discussed above. Finally, \( m_1, \ldots, m_s \) are the sizes of the portions of the \( s \) source page-blocks that have already been moved.

Figure 5.6 shows an abstraction of the size-class behavior. Similar to Figure 5.2, we use abstract states to describe the state changes: \textsc{empty} stands for the single state \( \langle 0,0,0,0 \rangle \) representing an empty size-class; the state \textsc{not-full, no source} represents all states with at least one not-full page where no compaction is needed and no source page is present, that is \( \langle h,n,u_1,\ldots,u_n,0,0 \rangle \) with \( 0 < n \leq \kappa \); the state \textsc{full, no source} represents all states with at least one full page, and no source page, that is \( \langle h,0,0,0 \rangle \) with \( h > 0 \); \textsc{not-full, source} represents all states with at least one not-full page where no compaction is needed and a source page, that is \( \langle h,n,u_1,\ldots,u_n,u_s,s,m_1,\ldots,m_s \rangle \) with \( 0 < n \leq \kappa, u_s > 0 \); \textsc{full, source} represents all states with no not-full pages, at least one full page, and a source page, that is \( \langle h,0,u_s,s,m_1,\ldots,m_s \rangle \) with \( h > 0 \) and \( u_s > 0 \); finally, \textsc{compaction} is used to represent states \( \langle h,\kappa+1,u_1,\ldots,u_{\kappa+1},u_s,s,m_1,\ldots,m_s \rangle \) in which compaction must be invoked.

We note that the automaton and the discussion in this section is under the assumption that the number of page-blocks in a page is larger than 1, \( \pi > 1 \). The degenerate case with \( \pi = 1 \) is of no interest.

A state change in a size-class happens upon allocation (A), deallocation (\( D_i, D^i_t \)), or incremental compaction (\( I, I_j, I_E \)) transitions. The distinction between \( D_i \) and \( D^i_t \) transitions will be clarified in the sequel and does not influence the global state changes. A transition \( I \) represents an initial incremental compaction step, \( I_j \) is any further incremental compaction step which involves a source page, and \( I_E \) is a further incremental compaction step which involves an emptying source page.

We next present the actual changes of states in a size-class in full detail upon allocation, deallocation, and incremental compaction.

**Allocation.** Allocation steps are the same as in the non-incremental automaton since the source page is not influenced by allocation. In detail, in a state \( \langle h,n,u_1,\ldots,u_n,u_s,s,m_1,\ldots,m_s \rangle \) there are three cases:

1. If \( n = 0 \), that is, there are no not-full pages, then after allocation \( h \) increases by 1,
5. CONCURRENCY AND SCALABILITY VERSUS FRAGMENTATION AND COMPACTION WITH COMPACT-FIT

Figure 5.6: Incremental size-class automaton with \( \pi > 1 \)

\( n \) becomes 1, and \( u_1 \) becomes 1.

2. If \( 0 < n \leq \kappa \) and \( u_n < \pi - 1 \), that is, there is a not-full page and after an allocation it will not get full, then both \( h \) and \( u_n \) increase by 1.

3. If \( 0 < n \leq \kappa \) and \( u_n = \pi - 1 \), that is, a not-full page will get full, then \( h \) increases by 1 and \( n \) decreases by 1. Note that this may change a state from "not-full" to "full" in case \( n = 1 \).

Allocation is not possible in a "compaction" state, i.e., a state with \( n = \kappa + 1 \).

Deallocation. We distinguish two types of deallocation steps denoted by \( D_i \) and \( D'_i \). A step \( D_i \) denotes deallocation in page \( i \) where the deallocated page-block is not a target of an ongoing incremental moving. In contrast, \( D'_i \) denotes deallocation in page \( i \) of a page-block which happens to be a target of an ongoing incremental moving. If \( i = 0 \), then deallocation happens in the source page; if \( 1 \leq i \leq n \), then deallocation happens in one of the not-full pages; and if \( i > n \) a page-block is deallocated in a full page.

Similar to the non-incremental CF, the change of state after \( D_i \) can be described by the following cases:

1. If \( 1 \leq i \leq n \leq \kappa \) and \( u_i > 1 \), or if \( i = 0 \), \( u_s > 1 \), and \( n \leq \kappa \), i.e., deallocation happens in a not-full or source page which will not get empty(ing), then \( h \) decreases by 1 and either \( u_i \) or \( u_s \) decreases by 1, respectively.
2. If \( 1 \leq i \leq n \leq \kappa \) and \( u_i = 1 \), i.e., deallocation happens in a not-full page which becomes empty afterwards, then both \( h \) and \( n \) decrease by 1, and the variable \( u_i \) is removed from the state.

3. If \( i = 0, u_s = 1, \) and \( n \leq \kappa \), i.e., deallocation happens in a source page which becomes emptying afterwards, then the source page is moved to the pool of emptying source pages \( E \) and both \( s \) and \( u_s \) are set to 0. As a result the size-class does not have a source page.

4. If \( i > n \leq \kappa \), which means that deallocation happens in a full page, then \( h \) decreases by 1, \( n \) increases by 1, and \( u_n \) gets the value \( \pi - 1 \). If originally \( n = \kappa \), then this step triggers a compaction operation.

In addition, there are four cases describing the change of state after \( D_t \) steps. They correspond to the cases for \( D_t \) except that at the end of such a step the ongoing incremental compaction operation to the deallocated target page-block is canceled and the source page-block is deallocated. Hence, the (canceled) ongoing compaction operation finishes earlier than it normally would. We refer to the situation when a thread performs a \( D_t \) step as a deallocation conflict.

Deallocation is also not possible in a “compaction” state with \( n = \kappa + 1 \).

**Incremental compaction.** Incremental compaction is triggered in case \( n = \kappa + 1 \), just like compaction is triggered in the non-incremental CF. In addition, there may be incremental compaction steps involving emptying source pages from any other state, and incremental compaction steps involving the source page from any state with a source page.

In a state \( \langle h, \kappa + 1, u_1 \ldots, u_n, u_s, s, m_1, \ldots, m_s \rangle \) an initial incremental compaction step is the only possible step. Note that in such a state \( u_n = \pi - 1 \) since the previous step was a deallocation in a full page. We refer to this unique free page-block in the last not-full page as \( t_b \). The initial incremental compaction step must be atomic together with the preceding deallocation step. We use \( \beta \) to denote the size of page-blocks in the size-class. We have the following cases:

1. If \( u_s = 0 \), meaning that there is no source page in the size-class, then since \( n = \kappa + 1 \geq 2 \) the first page becomes the new (potential) source page, i.e., \( u_s \) is assigned the value of \( u_1 \), \( s \) becomes 0, \( n \) decreases by 1, and \( u_1 \) is removed from the state. After this, the state is no longer a “compaction” state.
2. If $u_s > 0$, then a source page-block $pb$ is to be moved to $tb$. There are two possible cases:

- The page-block $pb$ is not a target page-block of an ongoing incremental moving operation. In this case there are two subcases representing an initial incremental compaction step: (1) if $\iota < \beta$, in which case the compaction operation needs more than just one step, then $u_s$ decreases by 1, $s$ increases by one, $m_s$ is assigned the value of $\iota$ and a portion of size $\iota$ is moved from $pb$ to $tb$; (2) if $\iota \geq \beta$, then the whole $pb$ is moved to $tb$ in one step and $u_s$ decreases by 1.

- The page-block $pb$ is a target of a (unique) ongoing incremental operation from a source page-block $sb$. In this case we are in a situation of a compaction conflict. Note that $sb$ must be in an emptying source page in $E$. Then the ongoing incremental moving operation from $sb$ to $pb$ is canceled, $pb$ is deallocated, and a new initial incremental moving operation starts from $sb$ to $tb$. Again $u_s$ decreases by 1.

In any case, $n$ decreases by 1. In case $u_s = 0$, the source page becomes emptying, it is moved to the pool of emptying source pages $E$, and $s$ becomes 0.

Note that the chosen way to resolve the compaction conflict is crucial for bounded compaction response times, since it avoids transitive compaction chains. Namely, a compaction conflict ends an existing compaction and starts a new one, so the duration of a particular compaction operation may only decrease due to a compaction conflict.

In addition, there are three more cases for a change of state due to an ongoing incremental compaction step $I_j$, where $j$ is an index of a source page-block in the source page that the incremental compaction step applies to. In a state $\langle h, n, u_1, \ldots, u_n, u_s, s, m_1, \ldots, m_s \rangle$ where $I_j$ is applicable, i.e., $u_s > 0$ and $s \geq j$, after an incremental compaction step $I_j$ we have:

3. If $m_j + \iota < \beta$, then $m_j$ is incremented by $\iota$, i.e., another portion of the source page-block gets copied to the target page-block.

4. If $m_j + \iota \geq \beta$ and $s > 1$, i.e., this is the last incremental step for the compaction operation which still keeps the source page, then the number of source page-blocks $s$ decreases by 1, the variable $m_j$ is removed from the state.

5. If $m_j + \iota \geq \beta$ and $s = 1$, i.e., the compaction operation finishes after this incremental step and the source page will no longer exist in the size-class, then $s$ gets the value 0. Furthermore, the source page either becomes a not-full page if $n < \kappa$ (in which
case $n$ increases by 1, $u_n$ is assigned the value of $u_s$, $u_s$ becomes 0) or it is kept as a potential source page.

Finally, there is a possibility for incremental operations $I_E$ which do not change the state, but only change the global pool $E$ of emptying source pages. We skip the details on the description and the update of $E$ due to $I_E$ operations.

We remark that the behavior of any thread can be expressed by a sequence of allocations and deallocations. If a deallocation triggers compaction, then before the thread can continue with any other allocation or deallocation operation all incremental steps needed for the compaction must be finished. The first of these steps is an initial incremental compaction step $I$ which may be an initial incremental moving step in case of compaction conflict. If it is the case, then all other incremental steps are of type $I_E$. Otherwise, if there is no compaction conflict, a sequence of $I_j$ incremental steps will be performed, and in case the source page becomes emptying a sequence of $I_E$ incremental steps, in order to complete the compaction operation.

5.6 Complexity vs. Fragmentation

Table 5.1 shows the time complexity of malloc and free as well as the worst-case system latency, memory size, and size-class fragmentation per CF configuration with $n$ threads and $m$ per-thread-allocated page-blocks in a size-class with $\pi$ page-blocks of size $\beta$ per page. The fragmentation caused by partitioning memory [9, 63, 19] is not considered here. Although the partial compaction bound $\kappa$ and the compaction increment $\iota$ are kept constant in our current implementations, both $\kappa$ and $\iota$ may be changed dynamically at runtime, which is an interesting topic for future work. System latency is here the portion of the delay a thread may experience, from invoking malloc or free until the operation actually begins executing, caused by currently executing, non-preemptive CF operations, not including the synchronization overhead.

Since all operations of the non-compacting 1-CF($\infty, \infty$) configuration take constant time, the complexity of malloc and free only depends linearly on the number of competing threads assuming fair scheduling. System latency is bounded by a constant. However, the worst case in memory consumption is one page for each allocated object due to potentially high size-class fragmentation, which has asymptotically the same bound as the overall memory consumption. The compacting 1-CF($\kappa, \infty$) configuration trades-off complexity of free and worst-case latency for better bounds on memory consumption by limiting size-class fragmentation through partial compaction. Note that in this case size-class fragmentation is independent from the number of threads and
5. CONCURRENCY AND SCALABILITY VERSUS FRAGMENTATION AND COMPACTION WITH COMPACT-FIT

<table>
<thead>
<tr>
<th>CF Configuration</th>
<th>malloc</th>
<th>free</th>
<th>latency</th>
<th>memory size</th>
<th>size-class fragmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-CF(∞, ∞)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n * m * π * β)</td>
<td>O(n * m * (π - 1) * β)</td>
</tr>
<tr>
<td>1-CF(κ, ∞)</td>
<td>O(n)</td>
<td>O(n + β)</td>
<td>O(β)</td>
<td>O((n * m + κ*(π - 1)) * β)</td>
<td>O(k * (π - 1) * β)</td>
</tr>
<tr>
<td>n-CF(∞, ∞)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n * m * π * β)</td>
<td>O(n * m * (π - 1) * β)</td>
</tr>
<tr>
<td>n-CF(κ, ∞)</td>
<td>O(1)</td>
<td>O(β)</td>
<td>O(β)</td>
<td>O(n * (m + κ*(π - 1)) * β)</td>
<td>O(n * κ * (π - 1) * β)</td>
</tr>
<tr>
<td>1-CF(κ, ι)</td>
<td>O(n)</td>
<td>O(n + β + ⌊θ/7⌋)</td>
<td>O(.min(β, ι))</td>
<td>O((n * m + n * π + κ*(π - 1)) * β)</td>
<td>O((n * π + κ*(π - 1)) * ι)</td>
</tr>
</tbody>
</table>

Table 5.1: Time complexity of malloc and free as well as worst-case system latency, memory size, and size-class fragmentation per CF configuration and size-class allocated objects.

The results for the n-CF configurations, in particular the worst cases in memory size and size-class fragmentation, as shown here, are obtained under the assumption that there is no sharing among the n CF instances. The time complexity of malloc and free of both multiple-instance configurations goes up to the respective single-instance cases if there is sharing among the n CF instances. While the non-compacting n-CF(∞, ∞) configuration requires in the worst case no more memory than the non-compacting single-instance configuration, the compacting n-CF(κ, ∞) configuration actually does require in the worst case more memory than the compacting single-instance configuration since partial compaction is performed per instance. However, allocation and deallocation throughput may increase with both multiple-instance configurations with a decreasing degree of sharing among the n CF instances (without an increase in worst-case system latency).

The incremental 1-CF(κ, ι) configuration actually improves the worst case in system latency whenever the compaction increment ι is less than the page-block size of the size-class with the largest page-blocks, at the expense of the complexity of free through more preemptions and at the expense of memory consumption through increased transient size-class fragmentation. In comparison to the non-incremental, compacting 1-CF(κ, ∞) configuration, there may be up to n additional (emptying) source pages in
the system where \( n \) is the number of threads. The worst case in non-transient size-class fragmentation does not increase.

### 5.7 Implementation

Sequential CF [63, 19] uses three data structures to manage its heap: abstract address, page, and size-class. Additionally, empty pages and available abstract addresses are organized in global LIFO lists.

An abstract address (handle) is a forwarding pointer word.

A page contains a page header holding the meta data of the page and the storage space into which objects are allocated. The size of each page is 16KB. All pages are kept aligned in memory. The page header consists of: two pointers used to insert the page into a doubly-linked list, a counter of allocated page-blocks in the page, a reference to the size-class of the page, and a bitmap where each set bit represents a used page-block in the storage space. The bitmap is used for fast location of free and used blocks.

A size-class contains two doubly-linked lists of pages which store the full and the not-full pages, respectively, and a counter of the number of not-full pages.

Global data structures are used to organize data structures which do not belong to a particular size-class. Such are a LIFO list of empty pages and a LIFO list of free abstract addresses. The implementation details that make these data structures concurrent and scalable will be discussed in the following subsections.

Figure 5.7 presents an overview of all implemented CF versions (leaves of the tree) and introduces terminology.
5. CONCURRENCY AND SCALABILITY VERSUS FRAGMENTATION AND COMPACTION WITH COMPACT-FIT

5.7.1 Concurrent Non-incremental CF

We use blocking and non-blocking mechanisms to allow for concurrent use of CF by multiple threads. In particular, locks are used to make the size-class automaton transitions atomic (allocation, deallocation that does not cause compaction, and deallocation with compaction) and non-blocking mechanisms are used to render access to the global LIFO lists atomic and scalable.

In all concurrent implementations size-classes are kept 128B aligned in memory, to avoid cache conflicts of concurrent threads.

We implement locks at two possible levels: size-class locks and page locks. The choice of lock level is evident in the different implementation versions in Figure 5.7. The page lock level is finer than the size-class lock level, which exists in all implementations. In the presence of page locks, during compaction the size-class lock is released and only the page locks of the source and target page are locked. As a result, other threads may perform memory operations within the size-class that do not affect the source and the target page.

Our managing of the global lists of empty pages and free abstract addresses is inspired by the free list implementation used in [32]. Each of the two lists is organized on two (public and private) levels. Each thread owns one private list (of free elements) which is only accessible to the owner thread. Therefore the access to free elements in the private list needs no synchronization mechanisms. The public list is a list of lists of free elements. Its head contains a version number (used for synchronization between threads) and a reference to the first element (sublist) in the list. Both fields in the list head are updated simultaneously using a double-word compare-and-swap operation, hence the update is atomic. Whenever the reference to the first element changes, the version number increases, which prevents the ABA problem [30]. If a thread needs a free element, then it first accesses its private list. If the private list is empty, then it accesses the public list, in order to fetch the head of the public list of lists. After this, the newly fetched list becomes the private list of the thread. There is also a mechanism that returns elements from a private list to the public list, which is invoked if the private list grows beyond a predefined bound.

There is a slight difference in the implementation of the public list for the list of empty pages and for the list of free abstract addresses. In order to represent the public list in memory, we need for each sublist a pointer to the next sublist. In case of the list of empty pages, we use the empty space of the first page of each sublist to store such a pointer. For the list of free abstract addresses, an additional two-word data structure for storing the pointers is needed.
5.7.2 Concurrent Incremental CF

For incremental compaction, each page-block stores an additional field called compaction-block field. The field has a size of 4B, which is relatively small compared to the size of the page-blocks in size-classes with large page-blocks (larger than 1KB), which are typically subject to incremental compaction. If a page-block becomes a source/target of an incremental compaction operation, then its compaction-block field stores a reference to its corresponding target/source page-block, respectively. Whether a page-block involved in incremental compaction is a source or a target page-block is determined by the status of its page and the status of the page of its compaction block.

In addition, each abstract address contains a flag bit which signals whether the object that the abstract address refers to is a target of a canceled incremental compaction operation. We have discussed deallocation and compaction conflicts in Section 5.5. In the implementation, a deallocation conflict is detected if the compaction-block field of the page-block under deallocation contains a memory reference. A compaction conflict is also recognized by a memory reference in the compaction-block field of the source page-block under compaction. In case of a deallocation or a compaction conflict, an ongoing compaction operation needs to be canceled. This is done by setting the flag bit in the abstract address of the object that was deallocated and triggered the compaction operation. When the thread in charge of the canceled compaction gets to execute again, it first checks the flag in the abstract address and if the flag is set the thread terminates its compaction operation and releases the abstract address.

5.7.3 Local vs. Global Size-classes

An orthogonal optimization for concurrent CF which improves scalability is using thread-local size-classes. Every thread has a private heap organized in private size-classes. Each thread allocates only in its private heap, but may deallocate shared objects in other thread’s heaps. If the percentage of shared objects in the system is low, this optimization leads to less conflicts, thus improving the overall performance.

5.8 Experiments

We report on micro- and macrobenchmarks with concurrent non-incremental CF, and microbenchmarks with concurrent incremental CF.
5. CONCURRENCY AND SCALABILITY VERSUS FRAGMENTATION AND COMPACTION WITH COMPACT-FIT

5.8.1 Hardware Setup

The experiments with non-concurrent non-incremental CF ran on the \texttt{Embedded1} platform. For this setup we measure the number of processor-instructions to eliminate interferences like cache effects. Therefore our mutators are instrumented using the \texttt{ptrace} \footnote{http://man7.org} system call.

The experiments with concurrent non-incremental CF ran on the \texttt{MultiCore1} server machine. The experiments with concurrent incremental CF were conducted on the \texttt{Embedded2} platform. In all experiments the benchmark threads were executed with real-time priorities.

5.8.2 Non-concurrent Non-incremental CF

In this section, we benchmark the moving (CFM) and non-moving (CFNM) implementations as well as the partial compaction strategy of CF in a number of experiments. Moreover, we compare both CF implementations with the dynamic heap management algorithms First-fit, Best-fit, DL, Half-fit, and TLSF. The implementations of First-fit, Best-fit, Half-fit, and TLSF we borrow from Masmano et al. \footnote{\texttt{http://www.greg.hr}}. We took the original implementation of DL from Doug Lea’s web page \footnote{http://DougLea.com}.

Our mutators provide synthetic workloads designed to create worst-case and average-case scenarios. We have not obtained standardized macrobenchmark results for lack of an automatic code translator or virtual machine implementation that incorporate the abstract pointer concept.

Increasing Size Allocations

In this benchmark, we run a mutator with incremental behavior: it allocates memory objects of increasing size starting from 8B increasing by 4B until the memory gets full at 7MB. Then, it deallocates each second object. We measure this process in the deallocation experiments. Finally, the mutator allocates the deallocated objects once more. We measure this process in the allocation experiments. In Figure 5.8, the \textit{x}-axes show the number of invoked memory operations, whereas the \textit{y}-axes represent the corresponding measured number of executed instructions.

Figure 5.8(a) shows the number of processor instructions for allocation. The behavior of First-fit and Best-fit is highly unpredictable. DL appears more bounded. Half-fit and TLSF perform allocation operations fast and in constant time. The behavior of the CF implementations is according to our theoretical results: constant for CFM and linear for CFNM. Note that the \textit{y}-axes of the graphs have logarithmic scale. Both CF
implementations are bounded but slower than Half-fit and TLSF due to the additional
administrative work for (potential) compaction. CFM is as fast as DL, and faster than
First-fit and Best-fit. The average number of instructions for allocation with CFM is
169.61, the standard deviation is 8.63.

The deallocation benchmark, with full compaction, is presented in Figure 5.8(b). All
algorithms except CF perform deallocation in constant time by adding the deal-
located memory range to a data structure that keeps track of the free memory slots. CF
performs compaction upon deallocation, and therefore takes linear time (in the
size of the memory object) for deallocation. The overhead of performing compaction
leads to longer execution time, but both CF implementations are bounded and create
predictable memory. For the given block-frame size of 32B, CFNM does not perform
better than CFM since returning blocks to the free-list of free block-frames takes ap-
approximately the same time as moving a whole memory object. Experiments showed that the minimum block-frame size for which deallocation in CFNM is faster than in CFM is 80B.

Using the partial compaction strategy results in constant deallocation times for CFM, as shown in Figure 5.8(c). Note that this graph shows the same picture as Figure 5.8(b) except for CFM and CFNM where partial compaction is applied. The compaction bounds for partial compaction are set sufficiently wide to avoid compaction. CFNM shows a step function with tight bounds per size-class. The average number of instructions for deallocation with CFM and partial compaction is 185.91, the standard deviation is 16.58.

Figure 5.9: Rate-monotonic microbenchmark
5.8 Experiments

Rate-Monotonic

In the rate-monotonic scheduling benchmarks, we use a set of five periodic tasks resembling a typical scenario found in many real-time applications. Each task allocates memory objects of a given size, and deallocates them when the task’s holding time expires. Three of the tasks allocate larger objects and have long holding times, the other two allocate small objects and have short holding times. The tasks have various periods and deadlines. They are scheduled using a rate-monotonic scheduling policy. Since the different tasks create a highly fragmented memory, this benchmark represents a memory fragmentation stress test.

For better readability, the $y$-axes of the graphs show the cumulative number of instructions, i.e., the sum of the number of executed instructions for all operations starting from the first invoked operation up to the currently invoked one. The $x$-axes show the number of invoked operations, as before. Note that a linear function represents memory operations that take constant time.

The allocation measurements are presented in Figure 5.9(a). Best-fit is highly unpredictable. Half-fit and TLSF are constant and fast. CFM is also constant, faster than DL, but slightly slower than Half-fit and TLSF. On average, a CFM allocation request takes 169.61 instructions with a standard deviation of 8.63.

Figure 5.9(b) shows the deallocation measurements. The differences in growth of the CFM curve correspond to compaction. During the first 550 deallocation operations CFM has to perform a lot of compaction operations but afterwards no compaction is necessary. The total runtime is shorter than the time needed for DL. CFNM takes linear time in the size of the memory object even if there is no compaction performed. The curve reflects this property.

Applying partial compaction leads to constant-time deallocation with CFM and makes it fast and more predictable, as shown in Figure 5.9(c). This graph shows the same picture as Figure 5.9(b) except for CFM and CFNM where partial compaction is applied. In order to apply partial compaction we have used the following compaction bounds on the 46 size-classes in the system: In the size-classes 15-18 and 28-29, two not-full pages are allowed. In the size-classes 19-27, we allow for three not-full pages. All other size-classes can have at most one not-full page. The mean number of instructions for CFM deallocation with partial compaction is 171.61, the standard deviation is 5.09.

Fragmentation

Our final experiments measure fragmentation. We compare CFM (with partial compaction) with TLSF, since the latter is considered the best existing real-time heap
management system in terms of fragmentation [52]. The results are shown in Figure 5.10. The numbers next to CFM, e.g. CFM 3, denote the maximal number of not-full pages allowed in each size-class.

![Fragmentation microbenchmark](image)

Figure 5.10: Fragmentation microbenchmark

For the experiments we have used a mutator that allocates the whole memory using memory objects of size 20B-100B. Before we run the fragmentation test around 20% of the number of allocated objects is freed. The memory holes are randomly distributed throughout the memory. The fragmentation tests count how many objects (y-axis) of size 20B-16000B (x-axis) are still allocatable by each heap management system. CFM obviously deals with fragmentation better than TLSF, even if we allow up to nine not-full pages in each size-class. Moreover, the fragmentation in CFM is fully controlled and predictable.

![Allocation throughput of a single thread with decreasing partial compaction](image)

Figure 5.11: Allocation throughput of a single thread with decreasing partial compaction
5.8 Experiments

5.8.3 Concurrent Non-incremental CF

The microbenchmarks run mutator threads that each allocate 2048 objects of random size, then deallocate the objects, and then start over again. The sizes of allocated objects correspond to the distribution of object sizes allocated in a popular optimizer for programmable logic arrays called Espresso used in several memory allocator performance evaluations, e.g. in [34]. Each microbenchmark runs for ten seconds performing more than one million allocation/deallocation operations.

Figure 5.11 shows the impact of partial compaction on the allocation throughput of a single thread. Larger partial compaction bounds $\kappa$ provide higher allocation throughput because of less compaction activity. Independently of $\kappa$, the size-class lock configuration performs better then the page-lock configuration since the latter needs locks for both the size-class and the source and target pages.

![Figure 5.11: Allocation throughput with an increasing number of threads](image)

(a) full compaction  
(b) optimized, non-compacting  
(c) opt., non-comp. with sharing

Figure 5.12: Allocation throughput with an increasing number of threads
Figure 5.12 depicts the allocation throughput with an increasing number of threads. Up to seven threads run in parallel on seven cores while the eighth core is used to minimize the influence of collecting data on the performance data. The performance of the fully compacting and the optimized, non-compacting version of CF without abstract addressing (in both cases with no sharing across the thread-local CF instances) are shown in Figures 5.12(a) and 5.12(b), respectively. The thread-local size-class versions show linear scalability in the number of threads whereas the global size-class versions neither scale in the fully compacting nor in the non-compacting configurations. Again, the size-class lock configurations result in better allocation throughput than the page lock configurations. Scalability only improves by a constant factor with increasing partial compaction (cf. Figures 5.12(a) versus 5.12(b)). Scalability of the thread-local size-class versions depends on the degree of sharing across the thread-local CF instances. Figure 5.12(c) shows allocation throughput at varying degrees of sharing: mutator threads allocate and deallocate 512 objects periodically according to the Espresso object size distribution. Each mutator frees its own just allocated objects and objects previously allocated by other threads in a ratio that determines the degree of sharing.

Figure 5.13: Allocation throughput for Hummingbird and Emacs

The macrobenchmarks are based on Emacs and Hummingbird allocation/deallocation traces [15]. In the Emacs trace about 51% of the allocated objects are of size 40B, 15% are of size 648B, and 11% are of size 104B. The remaining objects of the trace are also of small size. In the Hummingbird trace about 25% of the allocated objects are of size 8B and 23% are of size 32B. The remaining allocation requests vary from 16B
to around 38.1MB (object sizes greater than 16KB are ignored here). Hummingbird’s allocation behavior is very different from the behavior of a typical mutator where 99% of the objects are of small and similar sizes [34].

Figure 5.13 shows the allocation throughput of a single thread running the Hummingbird and Emacs benchmarks. Larger $\kappa$ values allow the Hummingbird benchmark to allocate more objects per second. In the Emacs benchmark the allocation throughput does not improve for larger $\kappa$.

Figure 5.14: Memory usage and size-class fragmentation for Hummingbird and Emacs

Figure 5.14 shows the required memory size (in number of used pages) and size-class fragmentation (in number of not-full pages) during the execution of the Hummingbird and Emacs traces with increasing $\kappa$. As expected, size-class fragmentation increases with increasing $\kappa$, whereas the required memory size remains constant for $\kappa \geq 5$ with the Hummingbird trace and $\kappa \geq 3$ with the Emacs trace since most not-full pages with smaller page-block sizes tend to remain relatively full (in line with our probabilistic claims of Section 5.4).

Finally, Table 5.2 shows the results of macrobenchmarking TLSF [53] and the optimized, non-compacting version of CF without abstract addressing (configured to 16B, and alternatively to 32B, for the smallest page-block size). The temporal performance of malloc and free operations (in clock ticks measured on the Opteron machine) for TLSF and non-compacting CF is similar with TLSF slightly outperforming CF (except for malloc in the worst case where CF is slightly better).
5. CONCURRENCY AND SCALABILITY VERSUS
FRAGMENTATION AND COMPACTION WITH COMPACT-FIT

<table>
<thead>
<tr>
<th></th>
<th>malloc (in clock ticks)</th>
<th>free (in clock ticks)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TLSF</td>
<td>CF</td>
</tr>
<tr>
<td>avg time</td>
<td>max time</td>
<td>avg time</td>
</tr>
<tr>
<td>Emacs</td>
<td>228</td>
<td>93359</td>
</tr>
<tr>
<td>Hummingbird</td>
<td>411</td>
<td>109079</td>
</tr>
</tbody>
</table>

Table 5.2: Performance: TLSF versus optimized, non-compacting CF (without abstract addressing)

Figure 5.15: Allocation throughput, system latency, and transient size-class fragmentation with decreasing compaction increments

5.8.4 Concurrent Incremental CF

The microbenchmark runs mutator threads allocating and deallocating objects from 16B to 16KB randomly. The threads operate on global size classes.
Figure 5.15(a) shows that the allocation throughput decreases with decreasing compaction increments $\iota$ since the incremental compaction overhead increases, due to an increasing number of lock acquire/release operations, administrative data updates, and memory copy interruptions. System latency, shown in Figure 5.15(b), tends to decrease, measurably if page-block sizes larger than around 512B are involved, with decreasing $\iota$. Here, we ran one mutator thread with higher priority than seven other mutator threads, periodically yielding to avoid starvation, and measured the maximum time the higher-priority thread spent in the atomic portion of any incremental compaction operation. True system latency that includes the wait time for locking was too noisy with the version of Linux we used. Transient size-class fragmentation, which is bounded by the number of threads, generally increases slightly with increasing $\iota$ as shown in Figure 5.15(c).

5.9 Summary

Compact-fit is an explicit, dynamic heap management system that allows, through the notion of partial and incremental compaction, formally relating fragmentation, compaction, throughput, and latency when managing contiguous pieces of memory. We have studied this relationship, formally and experimentally. All versions of CF can be made concurrent and scalable with partial compaction being only a constant factor. Scalability rather depends on the degree of sharing and synchronization mechanisms, similar to other heap management systems.
CHAPTER 6

Hierarchical Allocation Buffers

Thread-local allocation buffers (TLABs) are widely used in memory allocators of garbage-collected systems to speed up the fast-path (thread-local allocation) and reduce global heap contention yet at the expense of increased memory fragmentation. Larger TLABs generally improve performance and scalability but only up to the point where more frequent garbage collection triggered by increased memory fragmentation begins dominating the overall heap management overhead. Smaller TLABs decrease memory fragmentation but increase the frequency of executing the slow-path (global allocation) and thus may reduce performance and scalability. In the Hotspot JVM a complex, TLAB-growing strategy implemented in several thousand lines of code determines the TLAB size based on heuristics. We introduce hierarchical allocation buffers (HABs) and present a three-level HAB implementation with processor- and core-local allocation buffers (PLABs, CLABs) in between the global heap and TLABs. PLABs and CLABs require low-overhead OS-provided information on which processor or core a thread executes. HABs may speed up the slow-path of TLABs in many cases and thus allow using smaller TLABs decreasing memory fragmentation and garbage collection frequency while providing the performance and scalability of otherwise larger TLABs. Our implementation works with or without the TLAB-growing strategy and requires two orders of magnitude less code. We evaluate our implementation in the Hotspot JVM and show improved performance for a memory-allocation-intensive benchmark.

6.1 Introduction

Heap management in runtime systems like Java virtual machines (JVMs) may be a scalability bottleneck in applications with multiple threads accessing the global heap frequently. Thread-local allocation buffers (TLABs) reduce global heap contention by
preallocating large pieces of memory from the global heap. The preallocated memory is stored thread-locally to handle allocation requests of a given thread. This approach does not only reduce contention on the global heap but also allows a fast-path for memory allocation that does not require any synchronization or atomic operations since the TLAB of a thread is not shared with any other threads. However, larger TLABs introduce additional memory fragmentation that depends linearly on the number of threads since large blocks of memory are committed to thread-local use only. High memory fragmentation may result in more frequent garbage collection which may decrease application throughput. To trade-off scalability and memory fragmentation in modern JVMs complex TLAB-growing strategies incorporate different factors like allocation rate, number of threads, heap size and feedback from the garbage collector to determine TLAB sizes for all threads. The implementation of such a strategy in the garbage collector of the Hotspot JVM \cite{77} requires several thousand lines of code and thus significantly contributes to its complexity.

![Diagram of hierarchical allocation buffers (HABs)](image)

Figure 6.1: Hierarchical allocation buffers (HABs) with three levels: on the highest level is the global heap, in the middle are the PLABs or CLABs $P_1$ and $P_2$, and on the lowest level are the TLABs $T_1$ – $T_4$.

We introduce hierarchical allocation buffers (HABs), which consist of multiple levels of allocation buffers where an allocation buffer on a given level preallocates memory out of an allocation buffer on the next higher level. The traditional approach with TLABs is thus a two-level HAB system with the global heap on top and TLABs below. For recent multi-core architectures with several cores per CPU and several CPUs per machine we propose to use a three-level HAB system with one more level in be-
6. HIERARCHICAL ALLOCATION BUFFERS

tween as depicted in Figure 6.1. In our implementation this level uses processor- or core-local allocation buffers (PLABs, CLABs) which require low-overhead OS-provided information on which processor or core a thread executes. PLABs and CLABs speed up the slow-path of TLABs in many cases and thus allow using smaller TLABs decreasing memory fragmentation and garbage collection frequency while providing the performance and scalability of otherwise larger TLABs. We show in experiments that a statically configured HAB system may provide similar performance as a TLAB-only system using a TLAB-growing strategy.

Our three-level HAB implementation reflects the underlying processor architecture of the MultiCore2 server machine with four Intel Xeon E7 processors where each processor comes with ten cores and two hardware threads per core. The allocation buffers of the middle level can be configured to be PLABs or CLABs. The TLABs on the lowest level allocate from the PLABs or CLABs associated with the processor or core on which the allocating thread is currently running on. We evaluate the performance of our three-level HAB implementation integrated into the Hotspot JVM and show performance improvements due to better cache utilization and less contention on the global heap.

6.2 PLABs, CLABs, TLABs

TLABs are an architecture-independent concept for implementing allocation buffers. Each thread maintains its private allocation buffer for fast allocation of memory. However, allocation buffers may also be implemented in an architecture-dependent fashion. For example, allocation buffers can be assigned to processors, i.e., a thread running on a processor may use the allocation buffer of the processor for its allocation requests [26]. We study the use of processor-local allocation buffers (PLABs) as well as core-local allocation buffers (CLABs) situated in between TLABs and the global heap. A PLAB is assigned to a given processor which may comprise of multiple cores. Threads running on different cores but on the same processor share the same PLAB. A CLAB is assigned to a given core. Threads running on the same core share the same CLAB. Using PLABs may increase parallelism and cache utilization and thus reduce contention. On multicore machines using CLABs over PLABs may increase parallelism and cache utilization even further. Note that the size of PLABs and CLABs should be multiples of the TLAB size to avoid additional internal memory fragmentation.

Access to PLABs and CLABs is done in two steps. First, the processor or core on which a given thread currently runs is determined (selection). Then, the actual allocation in the selected PLAB or CLAB is performed atomically (allocation). Selection and
allocation is done non-atomically for better performance and scalability. Thus a thread may migrate to another processor or core in between selection and allocation resulting in what we call a foreign allocation. Note that the probability of foreign allocations is low and we show in experiments that foreign allocations indeed rarely happen.

### 6.3 Implementation

In this section, we discuss the implementation of HABs in the Hotspot JVM and present simplified pseudo-code of the core algorithm. Garbage-collection-specific details were removed for simplicity. We modified the heap implementation of Hotspot’s parallel garbage collector for the upcoming JDK7. The modifications are limited to the code path of allocating new and refilling existing TLABs. Additionally, we disallow direct inlined access to the global heap. In our benchmarks we did not observe any slow down when removing inlined access to the global heap.

Listing 6.1: TLAB allocation

```c
HeapWord* allocate_new_tlab(size_t size) {
    if (size <= PLAB_SIZE) {
        int hw_id = get_hw_id();
        for (int i = 0; i < PLABS; i++) {
            HeapWord* tlab = allocate_on_processor(size, (hw_id + i) % PLABS);
            if (tlab != NULL) {
                return tlab;
            }
        }
        return allocate_in_global(size);
    }
    return allocate_in_global(size);
}
```

Listing 6.1 shows the method for allocating a TLAB of a given size. If a thread’s TLAB is full the thread invokes this method to allocate a new memory region for its TLAB. TLABs larger than PLAB size are directly allocated from the global heap. For smaller TLABs, we try to allocate them in a PLAB with preference to the PLAB of the current processor. We iterate over all PLABs starting at the PLAB of the current processor and try to allocate a new TLAB. As soon as a TLAB is successfully allocated it is returned to the caller. If a TLAB could not be allocated in any PLAB, we fall back to allocate memory from the global heap. If this also fails the method returns NULL, and eventually a GC cycle will be triggered.

Listing 6.2 shows the method for allocating a TLAB of a given size on a dedicated processor (or core), and if the PLAB is full how it is refilled. If an allocation request
6. HIERARCHICAL ALLOCATION BUFFERS

Listing 6.2: TLAB allocation on a given processor (or core)

```c
HeapWord* allocate_on_processor(size_t size, int id) {
    HeapWord* top = plabs_[id].top();
    if (top == PLAB_REFILL_IN_PROGRESS) {
        return NULL;
    }
    HeapWord* tlab = plabs_[id].allocate(size);
    if (tlab == NULL) {
        HeapWord* result = CAS(plabs_[id].top_addr(), top, PLAB_REFILL_IN_PROGRESS);
        if (result == NULL) {
            return NULL;
        }
        tlab = allocate_in_global(PLAB_SIZE);
        if (tlab == NULL) {
            plabs_[id].reset();
            return NULL;
        }
        plabs_[id].init(tlab, size);
    }
    return tlab;
}
```

cannot be handled the method returns NULL. The implementation uses an optimistic non-blocking approach to avoid expensive locking. The access to a PLAB is synchronized using the PLAB’s top variable. The top variable indicates where the free memory in the PLAB starts or whether the PLAB is currently being refilled. If a thread detects that the PLAB is currently being refilled by another thread it returns NULL indicating to the caller that no allocations are currently possible in this PLAB. The top variable is always modified using a compare-and-swap operation.

If top is a valid pointer, the thread attempts to allocate the new TLAB in the PLAB. The allocation in the PLAB advances the top pointer by the size of the new TLAB using a compare-and-swap retry cycle. If the new TLAB does not fit into the PLAB it returns NULL, and the protocol to refill the PLAB with a new memory region is started. The refill protocol starts by setting top to PLAB_REFILL_IN_PROGRESS. The succeeding thread allocates a new memory region from the global heap and reinitializes the PLAB with the new memory region. If the allocation fails at any step, for example, when trying to set top to PLAB_REFILL_IN_PROGRESS or when trying to allocate a new PLAB, the method returns NULL. As mentioned in the previous section we tolerate context switches in between reading the processor ID and performing the actual allocation, which may result in foreign allocations. Moreover, a foreign allocation may also be performed by a thread that encounters that the PLAB_REFILL_IN_PROGRESS flag has been set. Note that this is a completely lock-free implementation for avoiding lock-
related complications with system invariants in the stop-the-world garbage collector.

6.3.1 Operating System Support

Our current implementation requires that the underlying operating system provides a mechanism for threads to look up the processor and core they are running on. In recent Linux kernels the `getcpu()` system call is optimized to provide a low-overhead mechanism for determining the CPU on which the invoking thread runs. However, it is not guaranteed that the thread is still executing on the CPU after the call returns. Therefore, allocation in PLABs and CLABs have to be synchronized and could even result in foreign allocations where a thread on processor A allocates memory assigned to processor B.

In order to reduce the additional overhead when accessing PLABs or CLABs or even allow unsynchronized access to CLABs we would require additional support of the OS. For instance, a notification when the thread is preempted would suffice to detect possible migrations and context switches. If a thread detects that it was preempted in between determining the current processor and the actual allocation in the PLAB it could restart the operation. In [22] the authors introduce multi-processor restartable critical sections (MB-RCS) for SPARC Solaris, which provide a mechanism for user-level threads to know on which processor they are executing and to safely manipulate CPU-specific data.

6.4 Related Work

Several JVMs already provide specific support for different processor architectures. For example, the latest version of the Hotspot JVM already supports NUMA architectures where the heap is divided into dedicated parts for each NUMA node.

PLABs were previously discussed in [26]. The implementation is based on a special mechanism called multi-processor restartable critical section which allows to manipulate processor-local data consistently and guarantees that a thread always uses the PLAB of the processor it is running on. In our implementation we do not provide that guarantee. If there is a context switch between determining processor and PLAB operation and the thread is scheduled after that on a different processor we tolerate that. Moreover, just using PLABs eliminates the fast-path provided by TLABs. In our work we combine the benefits of both PLABs and CLABs with TLABs.

Thread- and processor-local data is not only relevant in allocators of JVMs but also in explicit heap management systems. Multi-processor restartable critical sec-
6. HIERARCHICAL ALLOCATION BUFFERS

Figure 6.2: Speedup of using HABs with different TLAB and PLAB/CLAB configurations over the corresponding TLAB-only configurations of the unmodified Hotspot JVM.

tions are used in [22] to implement a memory allocator that holds allocator-specific metadata processor-locally to take advantage of the cache and to reduce contention. McRT-Malloc [32] is a non-blocking heap management algorithm, which avoids atomic operations on typical code paths by accessing thread-local data only. Hoard [14] is a memory allocator that combines, in more recent versions, thread-local with non-thread-local allocation buffers.

6.5 Experiments

For our experimental evaluation we use the MultiCore2 server machine. We ran the SPECjvm2008 memory-allocation-intensive javac benchmark [76] with 80 threads on
the Hotspot JVM in server mode \cite{78}. Each benchmark run was configured to last six minutes with two minutes of warm-up time in the beginning. We repeated each experiment seven times and measured the performed operations per minute. The default generational garbage collector of the JVM is configured with a maximum heap size of 30GB and a new generation size of 10GB. Parallel garbage collection is performed in the old and new generation.

![Figure 6.3: The data of Figure 6.2 but using the unmodified Hotspot JVM with the TLAB-growing strategy as common baseline.](image)

For the data in Figure 6.2 we removed the maximum and minimum from the seven runs and calculated the average of the remaining five runs. On the x-axis are TLAB sizes of increasing but fixed size, except where it says “growing” indicating that the TLAB-growing strategy of the unmodified Hotspot JVM is used. Note that with the TLAB-growing strategy the TLAB size settles at around 2MB. On the y-axis the speedup over the corresponding TLAB-only configurations of the unmodified Hotspot
6. HIERARCHICAL ALLOCATION BUFFERS

<table>
<thead>
<tr>
<th>TLAB</th>
<th>PLAB</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>4MB</td>
<td>0.32%</td>
<td>0.33%</td>
<td>0.51%</td>
<td>0.94%</td>
</tr>
<tr>
<td>1KB</td>
<td>0.11%</td>
<td>0.06%</td>
<td>0.09%</td>
<td>0.05%</td>
<td>0.15%</td>
</tr>
<tr>
<td>4KB</td>
<td>0.11%</td>
<td>0.06%</td>
<td>0.06%</td>
<td>0.06%</td>
<td>0.19%</td>
</tr>
<tr>
<td>8KB</td>
<td>0.13%</td>
<td>0.10%</td>
<td>0.08%</td>
<td>0.10%</td>
<td>0.29%</td>
</tr>
<tr>
<td>16KB</td>
<td>0.10%</td>
<td>0.08%</td>
<td>0.11%</td>
<td>0.08%</td>
<td>0.12%</td>
</tr>
<tr>
<td>32KB</td>
<td>0.11%</td>
<td>0.08%</td>
<td>0.08%</td>
<td>0.11%</td>
<td>0.30%</td>
</tr>
<tr>
<td>64KB</td>
<td>0.22%</td>
<td>0.15%</td>
<td>0.16%</td>
<td>0.29%</td>
<td>0.45%</td>
</tr>
<tr>
<td>128KB</td>
<td>0.52%</td>
<td>0.47%</td>
<td>0.63%</td>
<td>1.31%</td>
<td>2.60%</td>
</tr>
<tr>
<td>growing</td>
<td>0.55%</td>
<td>0.47%</td>
<td>0.64%</td>
<td>1.10%</td>
<td>2.23%</td>
</tr>
</tbody>
</table>

Table 6.1: Percentage of foreign allocations with different PLAB and TLAB configurations.

JVM (baselines) is depicted. In Figure 6.2(a) the results using HABs with PLABs and in Figure 6.2(b) the results using HABs with CLABs are depicted. For smaller TLABs a higher speedup can be achieved since the slow-path is triggered more often. However, performance also improves with the TLAB-growing strategy. In the presented results CLABs perform on average slightly better then PLABs.

Figure 6.3 is based on the same data as presented in Figure 6.2 but the y-axis depicts the speedup over the TLAB-only configuration of the unmodified Hotspot JVM using the TLAB-growing strategy, which is the default setting of the JVM. The results confirm that for smaller TLAB sizes than with the TLAB-growing strategy (around 2MB) similar or even better performance can be achieved. Figure 6.3(a) shows the results using HABs with PLABs and Figure 6.3(b) shows the results using HABs with CLABs. In particular, the results show that HABs with a small TLAB size provide similar performance as the original TLAB-growing strategy of the Hotspot JVM. In this case, a statically configured HAB implementation may thus replace a significantly more complex implementation of a TLAB-growing strategy.

In Table 6.1 and Table 6.2, the amount of foreign allocations using different PLAB and CLAB configurations with different TLAB sizes are presented. The amount of foreign allocations increases with increasing PLAB or CLAB size. Overall, however, the amount of foreign allocations is low, which shows that allowing allocations in PLABs or CLABs that do not match the current processor or core the thread is running on can be tolerated here. For the 512KB and 1MB CLAB sizes the TLAB-growing strategy
6.6 Summary

We introduced hierarchical allocation buffers (HABs) for improving performance and scalability of heap management on state-of-the-art multiprocessor and multicore server machines. We implemented and evaluated three-level HABs in the Hotspot JVM and showed performance improvements in a memory-allocation-intensive benchmark due to better cache utilization and less contention on the global heap. The results show that taking the underlying hardware architecture of general purpose machines into account even more than before may require significantly less complex code than architecture-oblivious solutions without a loss in performance.

Table 6.2: Percentage of foreign allocations with different CLAB and TLAB configurations.

<table>
<thead>
<tr>
<th>CLAB</th>
<th>512KB</th>
<th>1MB</th>
<th>8MB</th>
<th>16MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>0.00%</td>
<td>0.01%</td>
<td>0.05%</td>
<td>0.07%</td>
</tr>
<tr>
<td>4KB</td>
<td>0.01%</td>
<td>0.01%</td>
<td>0.06%</td>
<td>0.15%</td>
</tr>
<tr>
<td>16KB</td>
<td>0.01%</td>
<td>0.02%</td>
<td>0.07%</td>
<td>0.13%</td>
</tr>
<tr>
<td>64KB</td>
<td>0.03%</td>
<td>0.03%</td>
<td>0.10%</td>
<td>0.17%</td>
</tr>
<tr>
<td>128KB</td>
<td>0.06%</td>
<td>0.05%</td>
<td>0.15%</td>
<td>0.26%</td>
</tr>
<tr>
<td>growing</td>
<td>-</td>
<td>-</td>
<td>1.42%</td>
<td>2.81%</td>
</tr>
</tbody>
</table>

immediately determines to use TLABs larger than the given CLAB size, so CLABs are not used at all. Evaluating different synchronization strategies and allocations policies remains future work.

6.6 Summary

We introduced hierarchical allocation buffers (HABs) for improving performance and scalability of heap management on state-of-the-art multiprocessor and multicore server machines. We implemented and evaluated three-level HABs in the Hotspot JVM and showed performance improvements in a memory-allocation-intensive benchmark due to better cache utilization and less contention on the global heap. The results show that taking the underlying hardware architecture of general purpose machines into account even more than before may require significantly less complex code than architecture-oblivious solutions without a loss in performance.
This thesis summarizes some of my research work on concurrent object scalability on multicore systems. In the first part we analyzed the trade-off between adherence to concurrent data structure semantics and scalability. We quantitatively relaxed the sequential specification of a FIFO queue to a $k$-FIFO queue and presented different implementations with constant $k$, workload-dependent $k$, and unbounded $k$. We showed in various experiments that our $k$-FIFO queue implementations outperform and out-scale many state-of-the-art concurrent queue and pool algorithms on different micro- and macrobenchmarks. In order to quantify the difference between actual and regular FIFO semantics, we have introduced the notion of semantical deviation, which captures how many dequeue operations overtook older elements (lateness) and what the age of dequeued elements was. We showed for our $k$-FIFO queue implementations what their semantical deviation is in different experiments.

We see many interesting directions for future work based on the first part. Which applications tolerate semantical deviation to what extent? Is the parameter $k$ the right choice of information that should be exposed to application programmers for performance-oriented multicore programming (rather than, e.g. the memory hierarchy)? Can concurrent data structures other than FIFO queues be relaxed in a similar way? For stacks and shared counters we already obtained promising results but can we obtain similar results also for priority queues, hashtables, and even software-transactional memory [72].

In the second part of the thesis we discussed different concurrent heap management systems. We presented the design and implementation of a real-time memory allocator called Compact-fit (CF) which provides temporal and spatial guarantees. We analyzed in experiments the throughput, latency, and scalability versus memory fragmentation trade-off. CF may open up a path to dynamic heap management on memory-
constrained systems running high-performance applications that require tight temporal and spatial guarantees, although further studies involving specialized operating system infrastructure for embedded devices may be necessary there.

Moreover, we introduced hierarchical allocation buffers (HABs), a memory partitioning scheme for high-throughput heap management systems and presented an implementation of HABs in the Hotspot Java virtual machine. We studied in experiments the trade-off between throughput and scalability versus memory fragmentation. The HABs approach may be still improved by other orthogonal features like a cooperative thread scheduling mechanism for executing threads that share a significant amount of data on the same processor to further benefit from caching. Moreover, an integration of the HABs architecture into the TLAB-growing strategy of the Hotspot JVM for tuning not only TLAB sizes but also PLAB or CLAB sizes automatically may improve performance even further.
Team Contributions

This thesis is based on five original papers published in peer-reviewed conferences [19, 42, 40, 41, 43] and three technical reports [20, 29, 39] for which at the time of thesis submission a journal article and two conference papers were under review. The work was done under the guidance of Prof. Dr. Christoph M. Kirsch, at the Department of Computer Sciences, University of Salzburg, Austria. The other co-authors of the papers contributed to the ideas, implementation and evaluation of the presented algorithms, and the writing up of the papers as follows:

- Michael Lippautz helped implementing the pool algorithms introduced by others used in Chapter 3 and 4, helped implementing the Mandelbrot, transitive closure, and spanning tree macrobenchmarks used in Chapter 4, and helped evaluating experimentally the BS and US $k$-FIFO queue and Scal queue algorithms presented in Chapter 3 and 4.

- Together with Harald Röck, we designed the HAB architecture presented in Chapter 6, implemented HABs in the Hotspot JVM, and evaluated it experimentally. Moreover, Harald Röck helped implementing and evaluating the Scal queues presented in Chapter 4 and the incremental Compact-fit version presented in Chapter 5.

- Ali Sezgin helped developing the relaxed shared counters presented in Section 3.6.2 and proving linearizability of the $k$-stack implementation presented in Section 3.6.1.

- Ana Sokolova helped formalizing semantical deviation presented in Chapter 4, formalizing Compact-fit presented in Chapter 5, and proving linearizability of the $k$-stack implementation presented in Section 3.6.1.
BIBLIOGRAPHY


BIBLIOGRAPHY


129


