Design Patterns of Parallel Control Systems

—

Entwurfsmuster paralleler Steuerungssysteme

Der Technischen Fakultät
der Universität Erlangen-Nürnberg
zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Philippe Antoine Stellwag

Erlangen – Januar 2011
Abstract

Decreasing feature sizes and the desire to continue with large-scale integration of semiconductor components let processor manufacturers adjoin to physical restrictions of single processor architectures. Hence, they started for a while to reduce the clock frequencies of their CPUs and put more than one execution core on silicon die. These multi-core processors have, for some time, become more interesting to the embedded and real-time software sector. This is due to the price for (ultra-)low voltage multi-core processors is on the decrease while the clock frequencies for uni-processors tends to stagnate. Continuing innovation in the sector of data and computationally intensive industrial real-time systems, however, are driven by, and need, raw processing power.

To accommodate the requirements of processing power intensive systems, developers have to parallelize their software. This requires concurrency control mechanisms that satisfy the semantic shift from pseudo to real parallelism. These mechanisms, however, induce a whole host of side effects, in particular concerning logical correctness, time awareness, and timing quality (e.g., low jitter), to such real-time systems. Additionally, state of the art approaches to tackle concurrency within shared-memory control systems jeopardize their labor efficiency, and resource contention might easily offset the computing benefits of multiple execution cores. To overcome these side effects, researchers have started to investigate coordination and synchronization mechanisms that avoid the drawbacks of traditional blocking mechanisms and resolve concurrent situations by appropriate nonblocking algorithms. Existing nonblocking algorithms, however, are highly use-case specific, rather complex, and thus, exhibit a lack of changeability and expandability. This is a serious problem within a changing operational environment, which is typical within innovation-driven software development.

This thesis considers an industrial control system similar to a robotic control, and investigates shared-memory communication within such data and computationally intensive systems. The results of this dissertation have two core contributions. First, it presents novel design patterns that allow every moderate-skilled developer to build nonblocking communication mechanisms that widely avoid the drawbacks of their blocking counterparts. And second, it investigates the impacts of a changing operational environment to nonblocking shared objects, and presents very promising results to by-pass their typical use-case specific embedding and complexity, for which they are usually blamed for.
Parts of this dissertation have been published in the following peer-reviewed papers:

**Poster Abstracts**


**Workshop Papers**


**Conference Papers**


Acknowledgments

This dissertation is based on research that I have carried out as research assistant in the Department of Computer Science 4 for Distributed Systems and Operating Systems at the Friedrich-Alexander University Erlangen-Nuremberg in Germany. The research activities have been made within the scope of an industrial project in close collaboration with Siemens. It would not have been written without the support, help, and cooperation of many people that have supported me in various ways. It is not possible to mention all these people here, but I would like to express my gratitude to some in particular.

First of all, I want to thank my thesis supervisor and first referee, WOLFGANG SCHRÖDER-PREISCHAT, the head of the I4 department, for the chance to carry out interesting research in the fields of nonblocking coordination and synchronization for real-time purposes as well as his support, advice, and encouragement on me.

Many thanks go to my third referee as well as industrial advisor, WOLFGANG PAPIERNIK, who provided the industrial project and supported me in discussing ideas, gave me an understanding of the industrial field of control systems, and brought me together with the respective developers and contact persons within the Siemens environment.

I want to thank GUIDO SEEGER who continually supported me in doing research at Siemens in organizational and technical concerns, in particular, to meet the right people, to attend on meetings that relate to my topic, and in helping me to reap an in-depth understanding in architectural aspects of industrial control systems.

I want to express my appreciation for JÜRGEN KLEINÖDER and MICHAEL KÄVER, who provided the contractual basis for my work at the I4 department as well as by Siemens.

Many thanks go to my I4 colleagues, in particular, DANIEL LOHMAN and FABIAN SCHELER, who helped me multiple times by writing and reviewing some of my papers and in discussing the "crazy parts" of my work. I want furthermore thank GUIDO-ARNOLDT SÖDLER, who invited me to join Woch’s group in Erlangen after we met us by Netlogix in Nuremberg during my study period.

Many thanks go to all my colleagues at Siemens, who supported my work, in
which they, for instance, spent time to answer my questions, to review my work, and/or to support me in any other way.

Thanks go to students, in particular to Alexander Ditter and Jakob Krainz, who wrote study and/or diploma theses under my supervision, in which they implemented and evaluated some of my ideas.

And last but not least, I want to express my appreciation for my parents, grandparents, and, in particular, for my girlfriend, Rose-Marie Hartmann, for their continuous advice, care, support, and patience.

Erlangen, Januar 2011
# Contents

1 Introduction 1

2 Basics and State of the Art 5
   2.1 Shared Resources in Real-Time Systems . . . . . . . . . . . . . . 6
   2.2 Side Effects of Real-Time Synchronization . . . . . . . . . . . . . 6
   2.3 Nonblocking Synchronization . . . . . . . . . . . . . . . . . . . . 9
      2.3.1 Atomic Memory Operations . . . . . . . . . . . . . . . . . 9
         2.3.1.1 Processor Support . . . . . . . . . . . . . . . . 9
         2.3.1.2 Memory Instructions in Nonblocking Protocols . 11
      2.3.2 Progress Properties . . . . . . . . . . . . . . . . . . . . . 12
      2.3.3 Correctness Properties . . . . . . . . . . . . . . . . . . . 13
      2.3.4 Blocking vs. Retrying vs. Helping Time . . . . . . . . . . 14
      2.3.5 Benefits and Drawbacks . . . . . . . . . . . . . . . . . . . 15
   2.4 Multi-Core Processors vs. Multiprocessors . . . . . . . . . . . . . 16

3 Industrial System of Investigation 19
   3.1 The Past of Robotics . . . . . . . . . . . . . . . . . . . . . . . . 19
   3.2 The Present of Robotic Controls . . . . . . . . . . . . . . . . . 20
   3.3 Future Robotic Controls . . . . . . . . . . . . . . . . . . . . . . 22
      3.3.1 Transition to Symmetric Multi-Processing . . . . . . . . . 22
      3.3.2 Requirements for Shared Memory Communication . . . . . 25
   3.4 Goals and Questions to be Answered . . . . . . . . . . . . . . . 27
   3.5 Division into Levels of Recycling (LoR) . . . . . . . . . . . . . 28
   3.6 Problem Summary and our Approach . . . . . . . . . . . . . . . 29

4 LoR-0: A Wait-Free Queue with Local Preferences 31
   4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
      4.1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . 32
      4.1.2 Contribution . . . . . . . . . . . . . . . . . . . . . . . . . 32
      4.1.3 Chapter Outline . . . . . . . . . . . . . . . . . . . . . . . 33
   4.2 Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
## CONTENTS

4.2.1 Status Quo ............................................. 33
4.2.2 Specific Requirements for Symmetric Multi-Processing ... 34
4.3 Wait-Free Queue for Multiple Enqueuers and Dequeuers .... 34
  4.3.1 Idea .................................................. 34
  4.3.2 Protocol .............................................. 36
    4.3.2.1 Enqueue .................................... 36
    4.3.2.2 Read ........................................... 37
    4.3.2.3 Dequeue ...................................... 38
  4.3.3 Verification .......................................... 40
  4.3.4 Evaluation ........................................... 41
    4.3.4.1 Testing Method ................................ 41
    4.3.4.2 Test Scenarios, Results, and Discussion .... 42
  4.3.5 Conclusion ............................................ 44
4.4 Helping Queue ........................................... 44
  4.4.1 Motivation .......................................... 44
  4.4.2 Idea ................................................. 45
  4.4.3 Protocol ............................................. 46
  4.4.4 Validation .......................................... 47
  4.4.5 Evaluation .......................................... 48
  4.4.6 Further Partitioning ................................ 49
  4.4.7 Optimization of the Search Path ...................... 49
  4.4.8 Conclusion .......................................... 50
4.5 Improved Traversal Algorithm ............................ 51
  4.5.1 Motivation .......................................... 51
  4.5.2 Idea and Protocol .................................. 51
  4.5.3 Evaluation .......................................... 52
  4.5.4 Conclusion .......................................... 53
4.6 Further Related Work ................................... 54
4.7 Chapter Summary ........................................ 54
  4.7.1 Based on the Presented Approach .................... 54
  4.7.2 Based on the Initially Positioned Questions ......... 55

5 LoR-1: A Wait-Free Dynamic Storage Allocator ............... 57
  5.1 Introduction ........................................... 57
  5.1.1 Motivation ......................................... 58
  5.1.2 Chapter Outline .................................... 59
5.2 Specific Requirements ................................... 59
  5.3 A Wait-Free DSA ........................................ 60
    5.3.1 Idea .............................................. 60
    5.3.2 Algorithm ......................................... 62
      5.3.2.1 Allocation .................................. 62
      5.3.2.2 Deallocation ................................ 63
      5.3.2.3 Transformation of the Tree to an Array .... 63
  5.3.3 Worst-Case Execution Time .......................... 64
  5.3.4 Verification ........................................ 65
CONTENTS

5.3.4.1 Correctness .................................. 65
5.3.4.2 Linearity ........................................ 67
5.3.5 Evaluation ........................................ 68
  5.3.5.1 Test Method and Scenarios .................... 68
  5.3.5.2 Results and Discussion ....................... 69
5.3.6 Conclusion .................................... 71
5.4 Further Related Work ............................ 73
5.5 Chapter Summary .................................. 74
  5.5.1 Based on the Presented Approach .............. 74
  5.5.2 Based on the Initially Positioned Questions ... 75

6 LoR-2: Using Lock-Free Operations in Real-Time Applications 77
  6.1 Introduction ...................................... 77
     6.1.1 Additional Requirements ..................... 78
     6.1.2 Chapter Outline ............................... 79
  6.2 Design Pattern of Linearizable Lock-Free Protocols .... 79
  6.3 Contention-Based Serializing ..................... 82
     6.3.1 Helping Mechanism ......................... 83
        6.3.1.1 Request Table ......................... 83
        6.3.1.2 Execution of Requests ............... 85
     6.3.2 Assumptions ................................ 87
  6.4 Mappable Protocols ................................ 87
  6.5 Wait-Free Reference Counting Idiom ............... 88
  6.6 Verification Aspects ............................ 90
     6.6.1 Progress Guarantee ......................... 90
     6.6.2 Memory Consumption ........................ 91
     6.6.3 Avoidance of the Pointer Recycling Problem ... 91
  6.7 Evaluation ....................................... 91
     6.7.1 Test Method and Benchmarks ................ 91
     6.7.2 Results and Discussion ..................... 92
  6.8 Further Related Work ............................ 95
  6.9 Chapter Summary .................................. 96
     6.9.1 Based on the Presented Approach ............ 96
     6.9.2 Based on the Initially Positioned Questions ... 97

7 LoR-3: A Wait-Free NCAS Library 99
  7.1 Introduction ...................................... 99
     7.1.1 Contribution ................................ 100
     7.1.2 Chapter Outline ............................. 101
  7.2 System Model .................................... 101
  7.3 rrrNCAS ........................................... 101
     7.3.1 Terminology .................................. 102
     7.3.2 Properties .................................... 102
CONTENTS

7.3.3 Design ................................................. 104
7.3.3.1 Lock-Free Operations .......................... 104
7.3.3.2 Wait-Free Operations ......................... 106
7.3.3.3 Disjoint-Access-Parallel Operations ......... 107

7.4 NCAS ................................................. 108
7.4.1 NCAS Structure ..................................... 108
7.4.2 NCAS Words and Value Function ................. 108
7.4.3 Read Method ....................................... 109
7.4.4 NCAS Method ...................................... 109
7.4.5 Cooperative NCAS ................................. 111
7.4.6 Priorities of NCAS Requests .................... 111

7.5 Operation Queue ................................... 112

7.6 Reference-Aware Memory Management ............. 115

7.7 Evaluation .......................................... 117
7.7.1 Building Data Structure Operations using rtNCAS 117
7.7.2 Test Methods and Scenarios ...................... 118
7.7.3 Results and Discussion ........................... 119
7.7.3.1 Results ........................................ 119
7.7.3.2 Discussion .................................... 122

7.8 Further Related Work ............................... 126

7.9 Chapter Summary .................................. 127
7.9.1 Based on the Presented Approach ............... 127
7.9.2 Based on the Initially Positioned Questions .... 128

8 Conclusions ........................................... 131
8.1 Overall Summary and Conclusions .................. 131
8.2 Summary of Contributions .......................... 134
8.3 Limitations and Suggestions for Further Work .... 135

A Appendix .............................................. 137
A.1 Octa-Core Test Environment ....................... 137
A.2 Sixteen-Core Test Environment .................... 137
A.3 Real-Time Application Interface for Linux ........ 138

B Translated Parts of this Dissertation (German) ... 139
B.1 Inhaltsverzeichnis ................................... 139
B.2 Einleitung ........................................... 143
B.3 Zusammenfassung .................................... 147
B.3.1 Aggregation dieser Arbeit und Schlussfolgerungen 148
B.3.2 Wissenschaftliche Beiträge ....................... 151
B.3.3 Limitierungen dieser, und Ausblick auf zukünftige Arbeiten153

Bibliography ............................................. 155
List of Abbreviations

API .................. Application Programming Interface
APU .................. Arithmetic Logical Unit
BT ..................... Bit-Test
CAS .................... Compare-and-Swap
ccNUMA ............. Cache Coherent Non-Uniform Memory Architecture
CoBS .................. Contention-Based Serializing
CPU .................... Central Processing Unit
DCAS .................. Double-Word Compare-and-Swap
DSA ..................... Dynamic Storage Allocator
EUVL .................. Extreme Ultraviolet Lithography
FAA ..................... Fetch-and-Add
FCFS .................. First-Come First-Serve
FCLS .................. First-Come Last-Serve
FPU ..................... Floating Point Unit
FSB ..................... Front Side Bus
GPL ..................... General Public License
HAL ..................... Hardware Abstraction Layer
HMI ..................... Human Machine Interface
I/O ..................... Input/Output
IRQ ..................... Interrupt Request
JARA .................. Japan Robot Association
LGPL .................. Lesser General Public License
LL/SC .................. Load-Linked/Store-Conditional
LLFP .................. Linearizable Lock-Free Protocol
LoR ..................... Level of Recycling
LP ..................... Linearization Point
MIMD .................. Multiple Instruction, Multiple Data
NUMA .................. Non-Uniform Memory Architecture
PLC ..................... Programmable Logic Controller
PROMELA ............. Process/Protocol Meta Language
R&D ..................... Research & Development
RAM ..................... Random Access Memory
RC ..................... Robotic Control
RIA ..................... Robot Institute of America
RISC .................. Reduced Instruction Set Computer
RTAI ................. Real-Time Application Interface
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi-Processing</td>
</tr>
<tr>
<td>TAR</td>
<td>Bit-Test-and-Reset</td>
</tr>
<tr>
<td>TAS</td>
<td>Bit-Test-and-Set</td>
</tr>
<tr>
<td>VDI</td>
<td>Verein Deutscher Ingenieure</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example of an unbounded priority inversion scenario.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>An example of a memory hierarchy with two quad-core processors.</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview of the investigated RC system.</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Circular structure of our wait-free queue.</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Flow chart of the <em>enqueue</em> operation.</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Flow chart of the <em>read</em> operation.</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Flow chart of the <em>dequeue</em> operation.</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Results of our two test scenarios.</td>
<td>43</td>
</tr>
<tr>
<td>4.6</td>
<td>Test case with six actuators (80% enqueue, 20% dequeue).</td>
<td>45</td>
</tr>
<tr>
<td>4.7</td>
<td>Structure with our wait-free helping queue.</td>
<td>46</td>
</tr>
<tr>
<td>4.8</td>
<td>A possible state of the alarm and helping queue.</td>
<td>47</td>
</tr>
<tr>
<td>4.9</td>
<td>Test case with helping queue.</td>
<td>48</td>
</tr>
<tr>
<td>4.10</td>
<td>Improved traversing algorithm.</td>
<td>51</td>
</tr>
<tr>
<td>4.11</td>
<td>Test case with helping queue and improved traversal algorithm.</td>
<td>52</td>
</tr>
<tr>
<td>4.12</td>
<td>Test case with helping queue, improved traversing and 4 partitions.</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>The initial state of the counters inside the tree of depth 5.</td>
<td>61</td>
</tr>
<tr>
<td>5.2</td>
<td>The tree of Fig. 5.1 with six allocations of memory.</td>
<td>61</td>
</tr>
<tr>
<td>5.3</td>
<td>Translation of the tree from Fig. 5.1 to an array.</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>Structure of three consecutively allocated nodes.</td>
<td>64</td>
</tr>
<tr>
<td>5.5</td>
<td>Example tree to show the impacts of our non-linearizable DSA.</td>
<td>68</td>
</tr>
<tr>
<td>5.6</td>
<td>Scalability benchmark with increasing levels of concurrency.</td>
<td>72</td>
</tr>
<tr>
<td>6.1</td>
<td>Typical control flow graph of LLFP.</td>
<td>80</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.2</td>
<td>The helping concept of CobS.</td>
<td>82</td>
</tr>
<tr>
<td>6.3</td>
<td>Scalability benchmarks with increasing levels of concurrency.</td>
<td>93</td>
</tr>
<tr>
<td>7.1</td>
<td>Uses hierarchy of rtNCAS.</td>
<td>103</td>
</tr>
<tr>
<td>7.2</td>
<td>Involved data structures in rtNCAS.</td>
<td>105</td>
</tr>
<tr>
<td>7.3</td>
<td>Overflow problem when building chronological relations.</td>
<td>112</td>
</tr>
<tr>
<td>7.4</td>
<td>The influence of concurrency on the ticket distribution.</td>
<td>114</td>
</tr>
<tr>
<td>7.5</td>
<td>Scalability benchmarks for the queue implementations with increasing levels of concurrency.</td>
<td>120</td>
</tr>
<tr>
<td>7.6</td>
<td>Scalability benchmarks for the stack implementations with increasing levels of concurrency.</td>
<td>121</td>
</tr>
<tr>
<td>7.7</td>
<td>Results for the third benchmark (without speculative executions) with increasing levels of concurrency and number of NCAS words.</td>
<td>123</td>
</tr>
<tr>
<td>7.8</td>
<td>Results for the third benchmark (with five speculative executions) with increasing levels of concurrency and number of NCAS words.</td>
<td>124</td>
</tr>
</tbody>
</table>
List of Tables

3.1 The four levels of recycling (LoR) ........................................ 29

4.1 Semantic of the status bits .................................................. 36

4.2 Possible states of an alarm element ................................. 41

5.1 Results in CPU clock cycles of our test cases with 16 threads. . 70

6.1 Processing states for requests .............................................. 84

6.2 Signals of the callback functions ....................................... 86
List of Program Listings

2.1 Pseudocode of a lock-free increment function on a shared counter. 11

5.1 Pseudocode of the allocation algorithm. . . . . . . . . . . . . . . 62
5.2 Pseudocode of the deallocation algorithm. . . . . . . . . . . . . . 62
5.3 Representation of a node in memory. . . . . . . . . . . . . . . . 63

6.1 The enqueue operation of the MSC queue [MS96]. It uses a DCAS
    instruction to avoid the ABA problem. . . . . . . . . . . . . . . . 81
6.2 Cyclic comparison of tickets. . . . . . . . . . . . . . . . . . . . . 84
6.3 Pseudocode of the two functions for executing requests. . . . . 85
6.4 Linearizable wait-free reference counting. . . . . . . . . . . . . . 89

7.1 Pseudocode of an NCAS operation. . . . . . . . . . . . . . . . . . 102
7.2 Pseudocode of our NCAS implementation. . . . . . . . . . . . . . 109
7.3 Pseudocode of enqueue, dequeue, and get_ticket. . . . . . . . 113
7.4 Memory allocation and reference counting functions. . . . . . . 116
7.5 Callback λ implements the enqueue operation of a FIFO queue. . 118
In the preceding years, developers always benefit from the increasing clock frequencies of processors to make their software applicable in terms of performance without the need to add further efforts. From now on, however, the free lunch is over [Sut05], especially for the embedded and real-time sector that lies around two years behind the desktop industry.

Modern lithography technologies (e.g., extreme ultraviolet lithography (EUVL)) allow technological nodes with a feature size as small as 11 nm [KK08] to be created, which gives rise to higher density of technological nodes on semiconductor components. Large-scale integration in semiconductor components, however, introduces higher (internal) clock frequencies and, as a side effect, disproportionately more power consumption and heat generation. To overcome the physical restriction of heat dissipation and to continue to increase the complexity of integrated circuits (what is well-known as Moore’s law [Moo65]), Intel and other processor manufacturers started, in 2005, to reduce the clock frequencies of their processors and put more than one execution core on silicon die [Gee05]. Multi-core processors have, for some time, become more interesting and important to the embedded and real-time software sector. This is due to the price for (ultra-)low voltage processors with multiple cores is on the decrease while the clock frequencies for uni-processors tends to stagnate. Continuing innovation in the sector of data and computationally intensive industrial real-time systems, however, are driven by, and need, raw processing power.

This dissertation focuses an event-triggered industrial real-time controller similar to a robotic control system, which has been developed for industrial PCs with uni-processors for almost two decades. Up to now, the continual increase in clock frequencies of uni-processors permitted to continually integrate new software technologies and make them applicable in terms of performance; per-
formance is – besides correctness in terms of time – of high magnitude for such industrial real-time systems, as it gives information about their operating efficiency, which, for instance, is a crucial factor for the customer’s purchase decisions.

Using the processing power of processors with an increasing density of execution cores, however, requires intensive refactoring and redesign of current code to accomplish enough independent units of execution to utilize the available processing elements without violating the time specifications. Real-time systems investigated in this thesis, however, automate technical processes by the use of producer/consumer patterns with central services, such as logging, and require inter-core communication that is realized through objects located in shared memory. The access methods of shared objects, however, will heavily affect the quality of the timing behavior of future parallel real-time applications. This has three core reasons: First, the closely connected processing elements of today’s multi-core processors usually do not have any time synchronization between their hardware clocks. This leads to a nondeterministic overlap in execution histories and hence, influences the level of resource contention. Second, memory accesses through cache hierarchies may induce significant latencies [BWCC+08], as coherent caches may be cold or suffer from false-sharing, if developers do not meticulously consider the underlying memory topology. And last, existing approaches of blocking mechanisms that ensure mutual exclusion while accessing shared objects induce convoy effects [Fra04] and other drawbacks in terms of performance as well as deterministic time behavior [Ort94]. Besides, blocking approaches induce sequential code; Amdahl [Amd67] and Gustafson [Gus88] agreed that sequential code does not scale with increasing parallelism. In the future, however, a continual increase in processing elements is due.

To overcome these side effects, developers and researchers started to investigate coordination and synchronization mechanisms that avoid blocking control-flows and resolve concurrent situations by appropriate nonblocking algorithms. For the most part, however, these approaches focus use cases that do not exhibit time requirements, induce extensive limitations that make it hard to employ in practice, or do not consider real parallelism [Ram97, Gre99, Hoh02, Zha03]. Existing approaches to synchronize concurrent accesses to shared objects in a nonblocking manner are, furthermore, being blamed to be hard to get right [Gro08], because of their algorithmic complexity. In the majority of cases, nonblocking access methods are, furthermore, highly optimized solutions. Thus, usually they cannot be adapted to use cases that differ from the original.

To negotiate all these drawbacks, this dissertation presents novel pragmatic methods to implement all access methods of shared objects in a nonblocking way that minimizes progress dependencies of units of execution, offers highly deterministic, bounded as well as adjustable runtimes, and do not pessimistically limit parallelism as known from previous approaches. In addition, we precede our investigations in a pattern-orientated way to attenuate the complexity of nonblocking approaches.

A design pattern usually "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Ale80]. In spite of this statement from Alexan-
der that originally focuses only on building, it is also convenient in the field of software design. To proceed our investigations in a pattern-orientated way, we start with a dedicated use case and present therefore a highly optimized nonblocking approach. Next, we consider this approach and its adaptability in other scenarios. We, furthermore, investigate possibilities to adapt nonblocking objects on a continuously changing operational environment. Usually, a continuously changing operational environment is a serious problem for nonblocking objects, because of their lack of changeability and expandability. In particular the latter causes developers headache, as nonblocking objects usually rely on powerful hardware instructions to modify memory atomically. These instructions, however, are limited with respect to the number of memory words to be modified simultaneously. As a result of this limitation, nonblocking objects are, in the majority of cases, only simple objects, such as singly-linked queues or stacks. Our results, furthermore, tackle all these issues by appropriate software approaches. Furthermore, they are widely operating system independent and can also be used in other real-time, non-real-time, and mixed applications.

The scope of our work are pragmatic nonblocking synchronization methodologies for real-time applications to avoid or rather mitigate the side effects of existing approaches while concurrently accessing shared resources in high-grade parallel environments. There are two related research topics that we do not address in this dissertation. First, we do not consider schedulability analyses for the presented synchronization mechanisms. Static worst-case execution time analysis, or WCET analysis in short, for multi-core processors provides the basis for a respective schedulability analysis. Research on that topic is one important aspect for the correctness in terms of time for the most hard real-time systems. Work on this area can be found, e.g., by Yan and Zhang [YZ08], which focuses WCET analysis for multi-core processors with shared level-2 instruction caches. However, work on static WCET analysis as well as schedulability analysis is completely orthogonal to the topic of this dissertation.

Second, we assume that the common schedule of real-time and non-real-time units of execution for the presented scenarios is feasible. For instance, the isolation of computationally intensive prep tasks, or tasks that suffer from a considerable amount of slack time, inside non-real-time units of execution is common practice of many control applications that we know. We do not digress into techniques of preventing (malicious) non-real-time tasks from monopolizing shared resources in parallel environments with shared memory. Work on that topic is also completely orthogonal to this thesis.

The contribution of this thesis is, on the one side, in pragmatic nonblocking synchronization methodologies for high-grade parallel real-time applications. And on the other, we investigate and highlight possibilities to re-use, change, and extend the semantic behavior of these approaches within a continuously changing operational environment. The problems and solutions presented in this thesis are originated within the investigation of an industrial robotic control. The approaches presented here are highly applicable in other real-time, non-real-time, and mixed scenarios, as they are widely independent of operating system details and easily portable to other machine architectures, which have to satisfy only a few requirements (mostly the availability of some machine instructions, such as compare-and-swap).
Outline of this Thesis

This dissertation is organized as follows:

- Chap. 2 discusses basics of real-time synchronization concerning concurrent accesses to shared objects and state of the art. This chapter sketches, furthermore, the overall motivation of this thesis. Specific related work, which discusses some work that is not particularly related to the topic of this thesis, but to specific topics of the following chapters, is shown inside the Chap. 4 to 7.

- Chap. 3 sketches the results of the investigation of the robotic control and motivates the industrial aspect of our work. It establishes, furthermore, the requirements for appropriate synchronization protocols.

- Chap. 4 presents a novel wait-free unsorted queue for an arbitrary amount of concurrent queue operations. The concept of local preferences is, furthermore, introduced with the help of a novel wait-free helping queue.

- Chap. 5 introduces the concept of how to re-use the helping queue pattern of the previous chapter to implement a wait-free dynamic storage allocator that is applicable for real-time scenarios, which either do not need allocation guarantees or allocate memory of only fixed size.

- Chap. 6 presents a novel auxiliary scheme for using linearizable lock-free objects in real-time applications. Lock-free objects are usually not applicable in real-time scenarios, as they have potentially unbounded execution times. With the presented methodology the WCET of such protocols can be bounded. Thus, existing work on the area of lock-free objects can be re-used in timely applications.

- Chap. 7 introduces a novel multi-word compare-and-swap library that allows developers of real-time applications to implement arbitrarily complex shared objects in an easy way, while ensuring linearizability and wait freedom amongst others. With this technique developers can, furthermore, re-use their sequential algorithms in parallel real-time applications without modification and without manually care about concurrency.

- And, finally, in Chap. 8 we conclude this dissertation with an overall summary of our work and findings as well as our contributions, and make suggestions for future work.
The content of this chapter is split into two parts. First, it discusses the basics (as mentioned in the following) provided by the chapters afterward:

**Shared resources in real-time systems.** Sec. 2.1 on pg. 6 discusses types of shared resources used in real-time systems and defines which of them are considered here. It depicts, furthermore, well-known techniques to handle shared resources within these systems.

**Side effects of real-time synchronization.** Sec. 2.2 on pg. 6ff sketches typical side effects of blocking real-time synchronization that is still going strong. It discusses, furthermore, with the help of the investigated robotic control application why blocking synchronization techniques should be avoided as far as possible.

**Nonblocking synchronization.** Sec. 2.3 (pg. 9ff) distinguishes different types of nonblocking synchronization and their applicability in timely applications. It, furthermore, discusses the costs in time introduced by nonblocking protocols compared to their blocking counterparts. Last, the pros and cons of this type of synchronization are being balanced.

**The emperor’s new clothes?** Sec. 2.4 on pg. 16ff compares multi-core processors to multi-processors. The impacts of their different architectures are furthermore discussed with respect to atomic memory operations.

And second, it motivates the work on synchronization protocols with stronger nonblocking progress guarantees within the scope of parallelizing an existing piece of software instead of real-time systems built up per greenfield strategy.
2.1 Shared Resources in Real-Time Systems

There are fundamental two types of shared resources. First, there are indivisible shared resources of the underlying machine, such as processors or buses. If needed, divisibility is achieved through multiplexing. Imagine a system with preemption, where multiple tasks using a processor by means of appropriate preemptive scheduling. And secondly, shared data structures in terms of a set of memory locations that might be accessible by multiple cores/tasks synchronously — i.e. divisible shared resources. Note that tasks that access shared data structures synchronously might also be multiplexed with the help of synchronization primitives [Dij68] or appropriate control flows [GC96]. This, however, is due to their logical correctness and not due to physical restrictions. Note furthermore, that indivisible hardware resources also rely on potentially shared data structures, such as ready lists for the system scheduler.

Within the context of this thesis the term *shared resources* is only used for shared data structures, where concurrent units of execution commonly use a set of memory addresses. Other multiplexing techniques, such as scheduling multiple units of execution on a processing element, are not scope of this thesis.

Blocking synchronization of shared data structures is usually undesired in real-time systems [FP88], as it induces side effects in many ways (as later discussed in Sec. 2.2 on pg. 6ff). Due to the lack of alternatives and experiences [Gre99], blocking synchronization, however, is often developer’s first choice.

If the chronological order of accesses to shared resources (which we call *synchronization events* in the following) is known in advance, then we can schedule these events so that there are no concurrent situations, i.e. where multiple units of execution access to a shared resource at the same time. Hence, no further synchronization might be needed. However, while this works in time-triggered systems due to a global time base for the most use cases, in event-triggered systems it will not due to the reaction principle on demand [EBK03]. Moreover, in parallel environments such an approach might heavily limit parallelism, particularly in data intensive systems. In uni-processor environments, the situation concerning lost performance often is not a matter; in parallel systems, however, it will.

The robotic control application investigated in this thesis shows an event-triggered architecture with priority-based scheduling, where the chronological order of synchronization events is not known in advance. Hence, within that context we require respective coordination of concurrent units of execution. However, synchronization heavily affects the timing quality, which we discuss within the next section, of the real-time application and, thus, is a serious problem by parallelizing the robotic control system, as it jeopardizes its labor efficiency.

2.2 Side Effects of Real-Time Synchronization

Coordination of concurrent tasks in real-time applications must be analyzable and predictable to satisfy the semantic integrity and the temporal correctness. However, "predictable synchronization" requires trade-offs in many ways, as
Ortega observed [Ort94]. In the following, we discuss the side effects of synchronization that requires great care in time-critical applications.

**Priority inversion.** Imagine three tasks $t_H$, $t_M$, and $t_L$. Each task has an initially assigned and fixed priority; thereby $t_H$ has a high, $t_M$ has a middle, and $t_L$ has a low priority. Tasks $t_H$ and $t_L$ using a shared object that is protected for concurrent accesses by a spinlock $S$. If $t_H$ tries to acquire $S$ that has been previously locked by $t_L$, $t_H$ is blocked on $S$. If now $t_L$ is preempted by $t_M$, the high priority task $t_H$ is blocked for an unbounded amount of time, i.e. the time till $t_M$ yields the execution core. This example of an unbounded priority inversion scenario is depicted in Fig. 2.1. Priority inversion scenarios, in particular if they are unbounded, are not tolerable in real-time circumstances, as they avoid high priority tasks to make progress.

Unbounded priority inversion is typically bounded through priority inheritance protocols. The famous multi-processor priority ceiling protocol from Rajkumar et. al [RSL88] defines a priority inheritance protocol that also prevents deadlocks for a set of periodic tasks with local resources. There are, however, a lot of drawbacks of priority inheritance protocols. For instance, they induce a significant amount of jitter, limit the schedulability of tasks sets [TZ99], and do not scale well with multiple execution cores. For further discussions on that topic, please refer to the papers from Yodaiken [Yod02] and Baskiyar et. al [BM05].

**Priority violation.** Imagine a real-time system with task priorities, where tasks waiting to enter a critical region are strictly served according to first-come first-serve (FCFS) principle, such as by ticket spinlocks [MCS91, Cor08]. Tasks are, furthermore, scheduled according to their priority. Hereby a scenario might happen, where the scheduler chronologically assign tasks waiting to enter a critical region without been able to consider their priorities. Priority violation affects the timing quality of the real-time application, as it introduces latencies, especially for tasks with high priorities. To avoid this problem, both waiting queues have to satisfy an equal chronological relation with a close collaboration of coordination and scheduling/dispatching of concurrent tasks. Also some cooperative nonblocking synchronization techniques, such as the work from Herlihy [Her91], suffer from some kind of priority violation, however, with
a significant weaker impact. Priority violation for the blocking case might avoid achieving global progress due to low priority tasks that monopolize critical sections. Obviously, such a scenario is not tolerable in real-time circumstances. For the cooperative nonblocking case, such as [Her91], however, this effect is often bounded as well as predictable due to their cooperative character.

**Starvation.** Imagine an amount of tasks using a common spinlock $S$ that is alternately used by high priority tasks and low priority ones cannot make progress and might therefore starve. Since the decisions about entering critical regions and the scheduling/dispatching of concurrent tasks closely have to collude, synchronization primitives are usually not starvation-resistant. The avoidance or rather the upper bound of starvation scenarios is typically ensured by system design [BM05]. Such an approach is straightforward for real-time systems build up per greenfield strategy, however, avoiding starvation through system design regarding an existing piece of software (concerning its parallelization) is much more tough, as it would imply to rewrite significant parts of the application.

**Execution-time jitter.** Overlaps of execution histories from parallel control flows, which executed on modern multi-core processors, are usually nondeterministic [RRRV08]. In other words, the execution of critical regions induces different costs in terms of time that is dependent on the level of resource contention. The different costs in terms of time concerning two core points: varying blocking time through different levels of convoysing [Fra04] and nondeterministic behaviors through cache effects. Techniques to minimize resource contention, such as (randomized) backoffs [ST97, SZ00, HSY04], also contribute some amount of varying execution times [Hoh02, 1ff.]. A significant high execution-time jitter is often not tolerable in hard real-time scenarios, as it might introduce timing anomalies [RWT+06, EPBM06] and qualitative drawbacks of the final outcome\(^1\). Furthermore, the execution-time jitter might impact the release-time as well as completion-time jitter of tasks to the same degree.

**No scalability.** Resource contention might easily offset the computing benefits of multiple execution cores. In the future a continuous increase in processing elements per die is assured and shared resources might suffer from an increasing level of contention. Critical regions are, per definition, not scalable, as they ensure mutual exclusion of concurrent tasks. With increasing resource contention it is conceivable and obvious that mutual exclusion becomes a bottleneck. Amdahl [Amd67] and Gustafson [Gus88] agreed in this observation that sequential code is the core reason for inadequate scalability of applications.

**Deadlocks, livelocks, and race conditions.** The incorrect use of appropriate synchronization primitives might lead to execution scenarios, where the whole system does not make progress anymore. This is a result of violating the sequences by acquiring and releasing locks. Additionally,

\(^1\)For example, jitter influences the temporal movement of the axes of the robot. Imagine a robot arm that doing laser cutting. Thereby varying execution times of the robotic control application have direct impacts on the condition of the cut of the work piece.
Sec. 2.3: Nonblocking Synchronization

neglected synchronization mechanism might lead to race conditions, such as lost updates. Such effects are highly nondeterministic and obviously, are not tolerable in real-time applications. Furthermore, the likelihood for such defects increases with accretive fine-grained locking. However, fine-grained locking is the only way to achieve scalability while accessing to shared resources protected by locks [McV99].

2.3 Nonblocking Synchronization

Nonblocking synchronization\(^2\) defines a technique to synchronize concurrent units of execution through their control flow by using powerful machine instructions, such as compare-and-swap or test-and-set, instead of blocking on dedicated synchronization primitives. Nonblocking synchronization has major benefits compared to traditional synchronization approaches that make it particularly interesting for the use in time-critical applications [Hoh02, Zha03, Ram97]. However, previous work on this area focuses on techniques that are (a) very use-case specific or of pure theoretical nature, (b) not applicable in parallel real-time scenarios, or (c) too inefficient in terms of space and/or time. As we investigate within the next chapters, whether nonblocking synchronization is useful in real-time circumstances as well as applicable in a pattern-orientated way, a thorough discussion on related work is made inside the corresponding chapters.

2.3.1 Atomic Memory Operations

While atomic memory operations (i.e. updates, reads, and writes) on shared data structures through blocking synchronization primitives are usually achieved by satisfying mutual exclusion of concurrent units of execution, nonblocking synchronization using atomic memory instructions of the underlying machine together with an appropriate control flow to coordinate concurrent units of execution accessing shared objects and therefore satisfy atomicity of these state transitions [MS96]. In contrast to their blocking counterparts, this ensures full interrupt transparency at every time, as state transitions of data structures usually take place by a single atomic instruction without any intermediate states.

2.3.1.1 Processor Support

Modern multi-core processors occupy some machine instructions to implement atomic memory updates. The compare-and-swap (CAS) instruction is one of them that atomically compares a usually word-sized memory location with a given value (called "old value") and conditionally swaps it to a "new value", if and only if the compare part succeeds. IBM introduced CAS as well as DCAS (for two consecutive machine words or a double-size word) for the mainframe

\(^2\)In the following, the term nonblocking synchronization is used to describe synchronization mechanism on the basis of an appropriate control flow without blocking semantic. If not described differently, we do not consider nonblocking synchronized shared objects with the help of locks. For a discussion on that topic please refer to Hohmuth [Hoh02].
architecture IBM System/370 [IBM75, 123 ff.] within the seventies. The latter instruction was introduced to avoid the appearance of the pointer-recycling problem – or also called ABA problem [DPS10, Mic04a].

A digression concerning the ABA phenomenon: Imagine a shared data structure $O$ that is used by multiple concurrent units of execution. The data structure is initialized to state A. All state transitions are, furthermore, made by means of CAS. If now a task $t_1$ tries to change $O$ from state A to B, and in the meantime another task $t_2$ successfully changes $O$ from A to B as well as from B back to A, $t_1$ can spuriously finish its state transition from A to B, though $t_2$ has changed $O$ in the meantime. This phenomenon might violate the semantic integrity of $O$. To avoid this effect, all states of $O$ have to be unique [Mic04a].

Since 1984 and the arrival of the Motorola MC68020, these instructions have been widely adopted also for microprocessors. Intel offers such instructions – the cmpxchg mnemonics – since the Intel 80486 [Int10a, 3-186]. The series of Motorola MC68020, MC68030, and MC68040 are the only ones that have a double-word compare-and-swap instruction (CAS2) that is able to conditionally swap two unrelated machine words [ADF+00]. Also RISC processors (such as MIPS 6000) have CAS-like instructions with a slightly different syntax and semantic, called load-linked (LL) and store-conditional (SC). LL fetches the current value of a given memory location and "locks" the memory location. On the basis of the current value a new value is determined. The SC instruction then conditionally swaps the memory location to the new value, if and only if the memory location is already locked (strong LL/SC). Also special-purpose processors (such as ARMv6 and ARMv7 series) for embedded systems support LL/SC instructions. With the help of LL/SC it is possible to implement a CAS operation in software, however, with a slightly weaker progress guarantee, as there is currently no hardware implementation of a strong LL/SC pair available [Mo97]. Available implementations of LL/SC suffer from an effect, where the "lock" regarding the memory location is spuriously released, e.g., through context switches or accesses to the same cache line, which causes the SC instruction to fail misleadingly.

A slightly different machine instruction is SWAP that is equal to CAS, but without the conditional part. We are not aware of a general-purpose architecture that does not support a SWAP instruction.

Other machine instructions for atomic memory operations are the well-known bit operations test-and-set, test-and-reset as well as bit-test, which are also used to implement conventional semaphores and spinlocks [IBM75, 125 ff.]:

**Bit-Test-and-Set (TAS):** This instruction conditionally sets a dedicated bit, if and only if it is not already set.

**Bit-Test-and-Reset (TAR):** This instruction conditionally clears a dedicated bit, if and only if it is already set.

**Bit-Test (BT):** This instruction atomically checks whether a dedicated bit is set (read-only).

To the best of our knowledge, all general-purpose microprocessor architectures support such bit operations.
One machine instruction for atomic memory updates is furthermore worth to be discussed here, as the nonblocking protocols, presented in this dissertation, use this type of instruction, i.e. fetch-and-add (FAA). This instruction atomically fetches a memory location and increments it by a given value. With that instruction we can easily implement increment and decrement operations of usually word-sized counters. The first processor with an add instruction was Intel 8086 [Int10a, 3-33]. This add instruction, however, cannot return the old or new value of the modified memory location, and therefore was not as useful for nonblocking synchronization. With the Intel 80486 the processor manufacturer released the first FAA instruction (the xadd mnemonics) [Int10b, 4-537]. Currently, only the x86, x86_64 and Itanium architectures support the xadd mnemonics. There are, however, nonblocking counter implementations in software with different progress guarantees available that can be used to substitute FAA and make our nonblocking protocols also applicable for other CPU architectures [AH90, MT97, MTY92].

2.3.1.2 Memory Instructions in Nonblocking Protocols

There are typical differences between the control flow of nonblocking lock-free (e.g., [MS96, HSY04, FR04]) and wait-free (e.g., [SSPL09, Sun04, HPS02]) protocols. The state transitions of nonblocking objects, however, are taken place in the same manner, i.e. by means of appropriate atomic machine instructions. An example of a lock-free protocol is depicted in List. 2.1, where the function lf_incr() increments a global counter behind the pointer global_cnt up to a maximum value defined in MAX_CNT_VALUE. In Line 10 the CAS poses as memory barrier and ensures the coordination of concurrent units of execution. If the global counter has changed since the first snapshot in Line 8, the CAS will return an indication of failure and then induces a retry. This is potentially dangerous in real-time scenarios, since we cannot predict a worst-case execution time (WCET) of lock-free operations, which is discussed in the next section.

```c
#define MAX_CNT_VALUE 42

int lf_incr(int *global_cnt) {
    int old_state;
    int new_state;

    do {
        old_state = *global_cnt;
        new_state = old_state < MAX_CNT_VALUE ? old_state + 1 : 0;
    } while (!CAS(global_cnt, old_state, new_state));

    return old_state;
}
```

Listing 2.1: Pseudocode of a lock-free increment function on a shared counter.

However, since lock-free protocols are usually very efficient in terms of time and space for lower contention levels [LaM94], we present in Chap. 6 on pg. 77ff a novel methodology to use lock-free protocols also in real-time circumstances. Retry loops as depicted in List. 2.1 are, furthermore, a typical design pattern of lock-free protocols, which is explained in Sec. 6.2 on pg. 79ff in detail.
2.3.2 Progress Properties

Correctness in real-time systems is of prime importance. Note that correctness in real-time scenarios is both, logical correctness as well as timeliness. In this section the latter is discussed with respect to highly concurrent objects.

The scientific community distinguish between three progress properties of non-blocking objects. In the following, we shall discuss these properties with respect to the temporal correctness of real-time systems:

Obstruction freedom. Herlihy and colleagues [HLM03] have recently introduced this weakest progress property that focuses only use cases with a minimal amount of contention. Obstruction-free protocols guarantee progress of concurrent units of execution executed "in isolation", i.e. if a task performs an obstruction-free operation and others do not, progress for this task is guaranteed. Obviously, obstruction-free operations suffer from starvation through mutual impacts of contention. Hence, in real-time scenarios obstruction-free implementations are usually inapplicable, since all units of execution might starve.

Lock freedom. Lock freedom guarantees progress for at least one unit of execution for concurrently executed operations. This implies that there are some tasks that might starve while others make progress. In real-time scenarios this is dangerous, since this leads to potentially unbounded amount of time until completion. Chap. 6 on pg. 77ff presents a methodology to solve this starvation problem and let developers re-use lock-free protocols also in real-time applications. Lock-free operations are also obstruction-free, but not vice versa [HS08].

Wait freedom. Lamport introduced wait-free computing in 1977 [Lam77]. About one and a half decade later, Herlihy [Her91] introduced the theory of wait freedom that defines a strong nonblocking progress property. It guarantees progress for all concurrent units of execution – independently of the execution speed or, in general, the progress of other units of execution – within a bounded number of steps. For real-time applications this progress property is profoundly interesting, since it does not suffer from starvation. The steps of wait-free operations are often dependent on the number of tasks and/or the number of concurrent operations that happen in the system simultaneously. Hence, for an unbounded number of units of execution, not all wait-free protocols can be performed in a bounded number of steps. However, for real-time applications, which we focus in this dissertation, the number of threads is limited and known a priori. Thus, "unbounded wait freedom" is not an issue to the work presented here. It should be mentioned that there is at least one paper that discusses the topic "wait freedom vs. bounded wait freedom". Please refer to Brit et. al [BM96] for further discussions on that topic. Wait-free protocols are implemented through helping schemes\(^3\), where a unit of execution helps other ones to complete a operation in contention. Wait-free operations are also lock-free, but not vice versa.

\(^3\)Helping schemes of wait-free protocols are also called auxiliary schemes.
2.3.3 Correctness Properties

This section presents well-known work on correctness conditions for highly concurrent objects that allow to verify their logical correctness. Linearizability is used in later chapters for plausibility checks of the presented approaches.

Quiescent consistency. Concurrent objects that are quiescent consistent have to fulfill two requirements:

1. Method calls have to take place sequentially.
2. Method calls "in isolation" of other units of execution have to make their object’s state transition atomically.

The first requirement implies that there is at most one pending method call. Hence, for parallel environments the first requirement is too weak. The second requirement postulates that state transitions are taken place in an atomic manner that avoids logical incorrectness through intermediate states. Due to the second requirement, state transitions of quiescent consistent objects through concurrent as well as non-concurrent method calls can take place in an arbitrary order. However, to verify the correctness on a quiescent consistent object, it is sufficient to observe the execution history of one method call, i.e. quiescent consistency is compositional [HS08, 49 ff.].

Sequential consistency. Method calls are sequentially consistent, if the state transitions of the respective object are taken place in program order. Usually load and store instructions of modern multi-core processors are not inherently sequentially consistent because of instruction re-ordering. However, through memory barriers developers can establish that some instructions are sequentially consistent, explicitly. This correctness condition is not compositional, i.e. to verify method calls to nonblocking objects, all execution histories have to be investigated. Hence, for larger systems, a verification of sequential consistency is hard to employ [SG05].

Linearizability. This strongest correctness condition consolidates the compositional property of the quiescent consistency together with the property that sequential method calls take place in program order, satisfied by the sequential consistency. Linearizability postulates the following two requirements:

1. Every execution history can be transformed to an equivalent sequential one.
2. State transitions of non-concurrent method calls take place in program order.

The first requirement implies that state transitions take place atomically. The point, where a state transition occurs can be anywhere between the method call and its response is called linearization point. The second requirement allows that state transitions of concurrent method calls can take place in an arbitrary order [HW90].
Linearizability gives rise to be able to verify concurrent objects separately and offers an intuitive notion of correctness. This thesis focuses systems with a high degree of re-use (e.g., due to inheritance or polymorphism). Hence, quiescent consistency is not applicable in such systems, as state transitions cannot guarantee to take place in program order, and sequential consistency needs static code analysis to be employable. The scope of this thesis, however, is not static code analysis to check whole applications in terms of correctness. We do not present complete linearizability correctness proofs of our work presented in the next chapters. Indeed, we present plausibility checks in terms of relevant aspects with the help of linearizability. The scientific community might study, re-implement, and experiment with our results. The results presented here are based on fundamental research to satisfy changed requirements of multi-core and many-core processors in terms of synchronize concurrent accesses to shared objects in a nonblocking and pattern-orientated manner, and might be a pivotal point for the community for further verification.

2.3.4 Blocking vs. Retrying vs. Helping Time

This section discusses the differences in time between nonblocking synchronized objects with different progress guarantees, which are mentioned previously in Sec. 2.3.2 on pg. 12, and their blocking counterparts with respect to their applicability in real-time circumstances.

The blocking time of traditional synchronization primitives is the time spent by a task waiting for another task\(^4\) to complete a critical region. In real-time systems, the blocking time has to be bounded. This is usually done by avoiding starvation scenarios through system design [Gre99] with synchronization primitives that do not care about progress of tasks, such as all locks with priority inheritance protocols (see [RSL88, LSR90, Yod02]). Other possibilities are synchronization primitives that inherently avoid starvation, such as ticket spinlocks [MCS91, Cor08]. Such primitives, though, heavily limit possible scheduling strategies, as mentioned in Sec. 2.2 starting on pg. 6, and hence, might be not applicable.

In contrast to blocking approaches, nonblocking synchronized lock-free protocols exhibit a retrying time that is the time for a task to take a snapshot of the old state, to calculate the new state, and to try to perform the state transition on the global object. Since starvation scenarios might lead to an unsuccessful state transition, these steps have to be repeated until success. The elimination of starvation for lock-free protocols for real-time systems built up per greenfield strategy is possible [Ram97, ARJ97, Hol02, Zha03]. However, for large existing systems, which we focus in this thesis, this approach actually does not work, since it would require the re-implementation of almost the whole application. We present a methodology in Chap. 6 to use lock-free protocols also in such legacy systems.

For this reason, nonblocking synchronized protocols that are satisfying wait freedom are profoundly interesting for parallelizing legacy systems, whereas

\(^4\)Note that in priority-driven uni-processor real-time systems a task usually has to wait only for a higher priority task. This statement, however, is not valid for real-time systems running on parallel environments, e.g., multi-core processor boards, where tasks with different priorities might run in parallel.
"legacy" means a code base developed exclusively for uni-processor platforms. However, wait-free protocols induce an additional helping time. This is the time, where a task helps at least another task to complete its method calls on a shared object. The helping time introduced by recursive helping scenarios [AT93] may be unacceptable with increasing level of contention. Though, the costs introduced by recursive helping scenarios are heavily affected by the design of the respective protocol; the benchmarks in Chap. 4 to 7 show that these costs are significant smaller as well as more deterministic than the costs introduced by blocking counterparts.

2.3.5 Benefits and Drawbacks

In this section the benefits and drawbacks of nonblocking synchronized shared objects are compared. Research on novel synchronization techniques has been neglected in the last years [BCB+08]. Nonblocking synchronization is often blamed for being quite complex and hard to get right [Gro08] compared to their blocking counterparts. This is true with scope to general-purpose scenarios, where logical correctness is usually sufficient. Synchronization in time-critical scenarios, however, is inherently a quite complex task, as correctness in terms of time is additionally a crucial requirement [Yod02]. Furthermore, work on novel nonblocking synchronization methodologies are absolutely justified because of qualitative benefits on the subject of timeliness.

Interrupt transparency. Due to their design, nonblocking synchronized protocols are fully interrupt-transparent. Hence, they do not suffer from priority inversion. In addition, priority violation scenarios of these protocols can be bounded. The cooperative character of these protocols is profoundly useful in time-critical applications, as high priority tasks cannot be spuriously influenced by lower priority tasks.

Executable everywhere. Due to their interrupt transparency, nonblocking operations can be performed everywhere, such as in asynchronous procedures (e.g., signal handlers or interrupt service routines). Shared resources that are protected against concurrency by mutual exclusion techniques usually cannot be accessed within asynchronous procedures, as the regular control flow of the program might be interrupted inside critical sections and hence, intermediate states would be visible otherwise.

Avoidance of serializing. Nonblocking synchronized protocols reduce the serialization of critical sections up to a set of a few atomic instructions. Serialization takes place on the machine level through memory barriers and/or bus locks, e.g., the Intel *fence or lock mnemonic [Int10a, Int10b]. The benefits of small critical sections are well known [LAP04] and mostly implemented through coding guidelines (e.g., [MP09]):

1. Deterministic execution times.
2. Minimal completion-time and release-time jitter.
3. Minimal latencies regarding to interrupts or task release times.
Disjoint-access parallelism. Since nonblocking synchronized protocols are interrupt-transparent, they are best suited for disjoint-access parallelism [IR94]. Thereby, if units of execution access disjoint memory locations, we are able to perform non-concurrent tasks in parallel. While for fine-grained locking detailed information on contented locks are needed [RHH09] to be able to perform accesses to disjoint memory locations in parallel, the situation for nonblocking protocols is more straightforward: Lock-free objects, and also concurrent objects that occupy weaker progress guarantees [HLM03], are always disjoint-access-parallel [DPS06]. However, the degree of parallelism for wait-free objects is not inherently described by the level of contention [Her91]. A discussion on disjoint-access parallelism for wait-free protocols in real-time circumstances is given in Chap. 4 to 7 in detail.

To come to the point, disjoint-access-parallel protocols show benefits in terms of scalability with increasing number of execution cores and hence, performance. They have, though, drawbacks with respect to determinism, i.e. a higher degree of execution-time jitter. To the best of our knowledge, investigations regarding disjoint-access parallelism in time-critical environments have been not previously made.

Autonomous from operating system aspects. Nonblocking synchronized objects are widely independent from operating system details. They use powerful machine instructions to achieve atomicity of object operations.

2.4 Multi-Core Processors vs. Multiprocessors

In data and computationally intensive real-time systems – such as the investigated robotic control system – performance issues are additionally important (beside logical and temporal correctness that have to be satisfied all the time), as performance issues give information about the efficiency of the system. We explain the effects of performance issues relating to the robotic control in the next chapter.

The system model focused in this thesis is based on symmetric/homogeneous and cache coherent multi-core general-purpose microprocessors without hyper-threading technologies\(^5\). Usually, such processors offer some kind of memory hierarchies, where the random access memory (RAM) defines the slowest volatile memory regarding to the access times. Above the RAM, general-purpose multicore processors usually have up to three levels of shared and local caches to reduce the memory access times at average. Caches also might heavily affect the execution time of the real-time application; however, nondeterministic effects induced by cache hierarchies can be influenced in a fine-grained manner by software developers, while influencing nondeterministic side effects of hyper-threading through software is – if at all – significantly harder to achieve.

\(^5\)Hyper-threading is a hardware optimization to utilize the processor pipeline by using multiple hardware threads that share, for instance, registers, floating point unit (FPU) or arithmetic logical unit (ALU). Imagine a hardware thread waiting due to a cache miss, which leads to gaps in the processor’s pipeline. These gaps can be used to schedule another hardware
Sec. 2.4: Multi-Core Processors vs. Multiprocessors

Figure 2.2: An example of a memory hierarchy with two quad-core (C1, ..., C4) processors with local level-1 (L1), shared level-2 (L2) and level-3 (L3) caches.

An example of a memory hierarchy with two quad-core processors is depicted in Fig. 2.2. Imagine a global cache coherent platform, where the memory views of all execution cores are valid through a hardware cache coherence protocol [Mar08, 20 ff.]. It is straightforward to see that this represents a cache coherent non-uniform memory architecture (ccNUMA), as the access times of a processor’s core is heavily affected where the accessed memory is located on the globally coherent shared address space [DP01]. Note that this statement is also valid for, e.g., only CPU #1. For instance, accesses to exclusive data on the shared L3 cache of the second processor from one of the cores of the first are significantly slower than accesses to the local L3 cache. Furthermore, accesses to exclusive data that lies in the local L1 cache are significantly faster than accesses to the local L3 cache.

Usually, the interconnection network is significantly slower than the connection from the memory controller to the RAM. For example the Intel’s QuickPath Interconnect (QPI) currently reaches only up to 25.6 GB/s (gross) [Int09], whereas some DDR3 modules reach up to 64.0 GB/s (gross) [IBM10] that is two and a half times faster\(^6\). Hence, inter-processor accesses usually induce higher latencies than inter-core accesses of one CPU to the RAM.

Despite of the different timing behavior of memory accesses on multi-core and multiprocessor platforms with a globally coherent shared address space, the programming models are quite identical, i.e. message-orientated models and models that rely on a global address space.\(^7\) This thesis focuses only programming models with a global address space that promises to control memory accesses in a thread. However, simultaneous multi-threading heavily suffers from nondeterministic effects, such as execution-time jitter and latencies, which are not useful in real-time systems.\(^{[Int02]}\)

\(^6\)Both cited technical reports do not focus embedded real-time platforms. However, (a) with a certain slack time it is expected that these technical capacities will move also the embedded and real-time sector and (b) the mentioned technologies are currently the upper limit of buyable general-purpose components.

\(^7\)Stream-based models, such as regular or named pipes, are not relevant here.
very fine-grained manner. However, also message-passing interfaces on shared-memory systems usually rely on objects located in shared memory [Mes09, 22], hence our nonblocking methodologies presented in the next chapters can also be used to build up a message-passing programming model. Inter-task communication through message passing, though, might induce high overheads in terms of time and lead to very pessimistic real-time schedulability bounds [Gam04]. Dadji and colleagues presented in [DMM08] a task and communication structure for a robotic control with a high degree of parallel kinematics that runs on a dual-core processor. They found out that the use of message-passing induces about 55% overhead for a dual-core CPU, and introduces a significant amount of complexity to their scheduling strategy.

Regarding to Michael J. Flynn’s taxonomy [Fly72], both multi-core and multi-processors as well as mixed multi-core multiprocessor boards are classified under multiple instruction/multiple data (MIMD) architectures. Thereby multiple data streams are simultaneously processed through multiple instruction streams. The MIMD architecture goes back to an early parallel computer architecture, known under the name of transputer, with an additional communication hardware on each silicon node for message-passing systems, which was introduced from the British semiconductor manufacturer Inmos in the nineties [Sch06, 4].
The system of investigation is widely similar to a robotic control (RC) that controls industrial production robots, e.g., in the automotive industry. This chapter sketches – after some explanatory notes on early robotic controls in Sec. 3.1 – today’s structure of the system with focus on shared-memory communication (Sec. 3.2 on pg. 20ff). Based on possible decomposition strategies, our work on time-efficient and time-dependent nonblocking synchronization technologies is furthermore motivated, and requirements are derived in Sec. 3.3 on pg. 22ff. Sec. 3.4 on pg. 27ff identifies the goals and questions to be answered in this dissertation. And last, Sec. 3.6 on pg. 29ff concludes this chapter with the problem summary and our approach for solution.

3.1 The Past of Robotics

There are reams of definitions for the term industrial robot. For example, definitions have been published from the Robot Institute of America (RIA) in 1979 and from the Japan Robot Association (JARA). [Zel05, Sec. 1.1]

An accurate and famous definition comes from a German engineering association, called VDI. The VDI guideline number 2860 [VDI90] specifies instructions for assembly and handling, terminology, definitions, and symbols for industrial robots. Therein, it is defined that such types of robots are

*(...) equipped with several axes, whose movements are freely programmable (can be modified without mechanical intervention) with
respect to sequence, paths, and angles of movement. The hydraulic or electronic axial drives are fitted with path sensors whose position is interrogated cyclically by the robot control system and coordinated with preset values" [Har86].

George Devol was the inventor of the first industrial robot, called Unimate (a compound coinage from "universal" and "automation"). This one-armed robot based on Devol’s US-patent 2,988,237 that was submitted in 1954 and issued in 1961. Later in 1961, Unimate was firstly used by General Motors, e.g., for spot welding. [Car07]

3.2 The Present of Robotic Controls

Today, industrial robots are used in variety of application areas [Zel05, Sec. 1.7.1], e.g., for packing, welding, assembly, cutting, beveling, or painting.

In the following, we sketch a typical communication structure of an industrial RC. The investigated RC communicates through interfaces with other components of the RC system, such as human machine interfaces (HMI), the debugging interface (DI), the programmable logic controller (PLC) for cyclic or acyclic I/O, and the drives for controlling the robotic axes. The structure of the RC software is illustrated in Fig. 3.1 on pg. 21.

Internally, the RC consists of three groups of tasks that realize (a) the linear dataflow from the motion program to the physical drives and (b) the information flow (which is discussed later on). For simplicity and without loss of generality we initially assume that there exists one task per group. The tasks running on an event-triggered real-time operating system (RTOS) with preemption and static assigned priorities on the basis of an industrial PC.

Prep. The Prep task decodes, interprets and prepares the motion program. The motion program defines a sequence of commands that induces the robot to do something. The acyclic Prep task is not subject to time constraints, but has to adduce a certain flow rate to asynchronously feed the Ipo task. Therefore, the Prep task has the lowest priority in the system. The prepared data, the so-called blocks, are enqueued to a FIFO queue.

Ipo. The cyclic Ipo task is mainly responsible for motion control and interpolation of the blocks that come from the Prep over the mentioned communication object. The results of the Ipo task are stored inside a double buffer\(^\text{10}\). Usually, the Ipo task is dispatched every 1.0 ms that is a multiple of the servo cycle (also called base period, hyper period, or major cycle [SP07]).

Servo. The cyclic Servo task does fine-interpolation as well as feedforward and position control. This task has the highest priority in the system. It works for one ipo cycle on the blocks that the Ipo task has produced in the previous ipo cycle. Usually, the Servo task is dispatched every 0.5 ms.

\(^{10}\text{For an in-depth discussion on this double buffer please refer to [SSPL09].}\)
Communication among tasks takes place through global objects located in shared memory. There is no memory protection used. Hence, tasks are able to directly access the code and data sections of the other tasks. Communication between tasks can be classified into two categories:

**Regular block communication.** The data of the motion program is modified block by block inside dedicated objects, so-called *processing stations* (i.e. decoder, interpreter, preprocessor, motion control, and interpolator as illustrated in Fig. 3.1). This class of communication realizes the block-based data flow inside the RC through the communication objects exemplary mentioned above inside the task descriptions.

**Non-block communication.** This type realizes all other communication (information flow) of the RC. This type of communication is split into external and internal communication. External means communication to components outside the RC, such as HMI (see Fig. 3.1). And internal means communication to coordinate and synchronize RC-internal information flows apart from the motion program data, such as objects for logging system-global messages or the management of machine data (e.g., correction factors or the status of the tool wear).

For robots with more than one independent arm controlled by a single RC, there is a separate input stream per arm, i.e. an own motion program. This has an influence on the count of instantiation of tasks. Moreover, multiple input streams mostly exhibit data dependencies, which are called *cross-references* in the following; however, this depends on the use cases implemented in the motion programs. Imagine a two-armed robot that realizes the handing over of
workpieces from one arm to the other. The coordination and synchronization, i.e. the velocity and motion control, of both robot arms requires partial access to blocks of both arms. Another example with massive data dependencies is the two-armed robot for interactive assembly from Zhang and colleagues [ZvCK99]. Cross-references can be slit into the following two categories:

**Logical dependencies.** Logical cross-references between different robot arms arise on predetermined sequence restrictions. Again, imagine a two-armed robot that realizes the handing over of workpieces from one arm to another. The computation of the velocity and motion profiles of the second arm must happen after the handing over of the workpiece. Such logical dependencies are realized through semaphores [Dij68], if only one robot arm waits for the other, or through rendezvous points\(^{11}\), if always the arm waits that arrives firstly.

**Arithmetic dependencies.** Here, computation results of other robot arms are used to compute the own motion and velocity profiles. For example, if robot arms are coupled; imagine an arm that follows another one regarding to its motion and velocity profiles.

For the uni-processor case, logical and arithmetic cross-references are currently handled through semaphores and barriers to coordinate cooperative tasks, and hard synchronization techniques (disabling/enabling IRQs) to synchronize concurrent tasks. The latter becomes obsolete in a parallel environment, as disable IRQs on one execution core while accessing shared data structures does not prohibit other cores to access these data structures simultaneously.

Many control systems that use shared memory communication offer a similar communication structure, such as numerical controls or drive systems in general [Pap96].

### 3.3 Future Robotic Controls

This section sketches feasible decomposition strategies of the industrial system presented in the previous section and, furthermore, gives reasons why shared data structures have, in the future, to deal with an increasing level of contention. On this basis we motivate our work on bounded time, nonblocking and time-efficient synchronization mechanisms and deduce the general requirements for shared-memory communication.

#### 3.3.1 Transition to Symmetric Multi-Processing

Parallelizing the today’s sequential RC is an evolutionary process that will be driven by the available processing elements. In a first step, executing the mentioned tasks – i.e. Prep, Ipo, and Servo – in parallel can be used to exploit the power of a few execution cores. While in the next parallelization steps, for

\(^{11}\)Rendezvous are a specialized version of the barrier synchronization pattern in which only two units of execution are involved [KCS+09].
example, processing stations can be outsourced inside own tasks to exploit the power of more than a few execution cores. However, cyclic tasks are highly prone to jitter and latencies while communicating with other – especially with lower priority – tasks. This is also relevant for the uni-processor case [Gam04], but is of major magnitude for the multi-core case, as significant more units of execution might simultaneously access to shared objects. For an in-depth discussion on sharing an object between two cyclic real-time tasks of the RC system, please refer to [SSPL09]; thereby communication rises about 50 times for a dual-core system in comparison to the sequential case by using spinlocks. For the presented nonblocking approach the communication overhead induces 24 times less overhead and about half as much less jitter compared to spinlocks. Furthermore, sharing objects between hard real-time and acyclic non-real-time tasks is also susceptible to such drawbacks, which are a serious problem for high priority tasks with time constraints. Note that the Prep task, mentioned in Sec. 3.2 on pg. 20ff, does not have any time constraints.

To prevent progress dependencies of high priority cyclic tasks from acyclic/cyclic low priority ones, a synchronization strategy that offers full interrupt-transparency is profoundly desirable. Note that accesses to shared data structures that are protected against concurrency by the use of priority inheritance locks, such as [LSR90], are indeed interrupt transparent as long as the dispatched task does not use a resource "simultaneously"; however, this has major drawbacks in terms of deterministic timing behavior, as mentioned in Sec. 2.2 on pg. 6ff, and maximal execution times as mentioned previously. Furthermore, for tasks executed on different processing elements, parallelism is wasted.

The industrial objective is to establish a basis with a sophisticated decomposition strategy that is based on symmetric multi-processing (SMP) and ensures to avoid or rather mitigate typical risks of a parallel version of the RC system. Evaluating possible decomposition strategies, however, is behind the scope of this thesis. In order to motivate our work on scalable nonblocking synchronization mechanisms, the following only roughly sketches cogitable decomposition strategies of the investigated industrial system. Thereby a system model based on symmetric multi-processing is supposed. This has two core reasons, as partly mentioned before:

1. Large-scale changes on the system structure, mentioned in Sec. 3.2 on pg. 20ff, should be avoided to minimize the parts of the RC that have to be rewritten. Prohibit tasks to communicate through shared memory would force to rewrite significant parts of the RC system.

2. SMP allows a coarse-granular parallelization (e.g., perform the Prep, Ipo, and Servo tasks in parallel) as well as a parallelization in a rather fine-grained manner. Since the tasks are dependent on the code sections of the other tasks, their execution inside lightweight threads is more adequate, then inside multiple heavyweight processes; this complies with the first point. Moreover, a parallelization strategy that is flexible regarding to its communication granularity gives rise to optimize time- and/or performance-critical parts evolutionary.

The parallelization of the RC system promises to be able to keep today’s cycle times constant or even shorten them, and continue to integrate new software
technologies. This, however, requires communication methodologies that are bounded in time and space, time-efficient, scalable as well as highly deterministic concerning its synchronization times. In the following, we discuss the importance for our work presented in the next chapters against the decomposition strategy.

Firstly, the target is not to decompose the RC algorithms due to safety reasons. So, there are three applicable strategies to split the RC software into multiple parts: per vertical, horizontal, or hybrid decomposition. Each part has to fulfill the following requirements and is called a task, if the system representation does not matter, or a thread respectively.

1. Each task is dependent on input data that might be produced by other tasks and otherwise are widely independent, i.e. a minimal amount of communication.
2. Each task relates a static priority to an overall technical process.
3. Each task is either cyclic and has a predefined period, or acyclic and has to fulfill a minimum throughput.

Vertical decomposition defines an approach where tasks are defined with respect to independent data. Mattson and colleagues presented in [MSM05, 34 ff.] a design pattern that based on data composition. Such a decomposition strategy for the RC system can be implemented at two granularity steps:

1. A vertical decomposition on the basis of motion programs,
2. and/or a vertical decomposition on the basis of axes.

The scalability for the first granularity step is rather limited, as it concerns only use cases with multiple robot arms. The scalability, furthermore, also depends on the motion program itself and the configuration of arms that might, for instance, induce cross-references, as mentioned in the previous section. Additionally, various new communication mechanisms are needed, e.g., for accesses to motion/velocity profiles of other arms. This is because the sequential RC system calculates these profiles of multiple arms strict sequentially, i.e. one after another within the context of one task. Hence, concurrency does not matter in the sequential case. A vertical decomposition strategy on the basis of axes is similar, but offers more scalability for processors that occupy an increasing number of execution cores.

Another decomposition approach might base on functional separation, which is called horizontal decomposition in the following. Mattson et. al describe a coarse-grained Task Decomposition Pattern in [MSM05, 29 ff.]. Parallelization based on horizontal decomposition is more deterministic for systems, where the input data heavily influences the count of independent tasks that can be performed in parallel, i.e. a horizontal decomposition strategy offers benefits also for use cases with only one robot arm. For instance, the separation of the Prep, Ipo, and Servo task inside own threads is a feasible decomposition for every single motion program, in contrast to a vertical decomposition, where
the number of motion programs and/or axes is a crucial parallelization factor. However, communication efforts increase by a functional separation, too.

Imagine central services, such as logging system global messages or central machine data (e.g., correction factors, or the status of the tool wear). With an increasing number of units of execution, it is obvious that contention will rise at least to the same degree. In the next chapter, a central service, as a case study, is discussed in-depth, and a novel approach to handle a central data representation is presented.

Last, a hybrid decomposition of the RC system is also cogitable. Thereby the goal is to minimize data dependencies of functional components on the one side, and data cross-references (see previous section) on the other. Though, it is obvious that also this type of decomposition requires efficient and timely communication mechanisms that have to deal with a varying and, in the course of time, increasing resource contention.

### 3.3.2 Requirements for Shared Memory Communication

Within the following, the general requirements for shared-memory communication on the basis of the previous section are derived. They are the basis of the techniques presented in the next chapters.

**Logical correctness.** It seems to be a self-evident point that shared-memory communication should work as specified and, furthermore, care about the typical drawbacks of real-time synchronization (see Sec. 2.2 on pg. 6ff). However, there is no silver bullet to ensure correctness for highly concurrent objects. This has two core reasons: First, run-time testing is not complete due to nondeterministic interleavings, and hence, is too weak for logical correctness proofs. And second, code representations and formal correctness proofs (e.g., on the basis of the methodologies discussed in Sec. 2.3.3 on pg. 13ff, or other model-based approaches, such as [HB07]) are not inherently synchronized among each other. Famous examples for the impacts of these non-synchronized models are the priority inheritance protocol from Lehoczky and colleagues [LSR90] with their invalid proof of correctness [Yod02], at which their code might not be compulsorily wrong, or the critique from Groves [Gro08] that the implementation of nonblocking protocols are hard to get right, despite of valid correctness proofs.

One reason for this observation is that developers have to be familiar with memory barriers\(^\text{12}\), the semantic impact of their code, and the execution model. For instance, in correctness models and/or the implementation, memory barriers are often implicitly assumed, which often causes such serious problems.

However, logical correctness is essential not only for real-time systems, but also for all other deterministic general-purpose computer systems. Further research on that topic is necessary to reduce or maybe purge the gap between testing models and code implementations; though, this is not scope of the work presented here, so we do not digress into that

---

\(^{12}\)Please refer to McKenney [McK10] for a detailed discussion on this topic.
research topic and content ourselves with existent techniques to ensure logical correctness.

**Bounded response time.** For hard real-time tasks it is essential to fulfill the timing specifications, which, as a consequence, makes the WCET of any operation to an important parameter. This requires mechanisms for communication that do not suffer from starvation. Starvation by reason of resource contention is typically avoided by system design. As mentioned in the previous chapter, for legacy systems this, however, induces much more efforts as for ones built up per greenfield strategy. Thus, starvation-resistant synchronization approaches are promising for the scope of this thesis.

Furthermore, in parallel environments with a common schedule of non-real-time and real-time units of execution, a stronger progress guarantee that ensures progress of all units of executions becomes important. This allows us to determine the WCET for real-time tasks and avoid starvation of non-real-time tasks. Typical blocking approaches cannot guarantee such a strong progress [RSL88] or induce major scheduling limitations [MCS91].

**Minimal jitter.** Increasing resource contention induces a significant amount of execution jitter that becomes noticeable when the axis moves and affects the quality of the final outcome. Thus jitter is a critical point for systems focused here. Hence, we have to minimize and monitor the jitter introduced by our approaches.

**Scalability.** In the future a continuous increase in the number execution cores is due. In order to take advantage of this development, the amount of units of execution has to grow as well to be able to exploit the offered parallel performance. This in turn requires communication mechanisms that scale well with increasing count of units of execution and care about resource contention that easily offsets the computing benefits of multiple execution cores.

Formally, scalability is defined as speedup \( S \) with the portion of code \( P \) that can be performed in parallel, and a number of processing elements \( N \) as input parameters [Amd67]:

\[
S = \frac{1}{(1 - P) + \frac{P}{N}}
\]  

(3.1)

For real-time systems, scalability in terms of parallelism is somehow a new aspect that did not play any role over the previous decades. However, beside temporal correctness, the aspect of "scalable performance" becoming increasingly important for many industry branches which work on/with data and computationally intensive real-time systems. This is due to the efficiency of these systems in terms of its operating speed is of utmost importance, which was prior implicitly given for years.
3.4 Goals and Questions to be Answered

The industrial overall goal is to parallelize the RC application to be able to integrate new technologies without extending the current cycle times. Cycle times give information about the labor efficiency of the RC system. Using traditional methods for shared-memory communication, however, would induce significant overhead in terms of synchronization times as well as jitter that jeopardizes the labor efficiency of a parallelized variant of the RC system. So, work on nonblocking synchronization, especially linearizable and wait-free methods, promises to reduce

- execution-time jitter,
- and the maximal execution times of shared object accesses,
- progress dependencies of high priority tasks from lower priority tasks,
- as well as wasting of parallel performance.

Thus the goal of the dissertation is to give answers to the following questions:

1. Is there a nonblocking way to handle central services? Thereby a use-case specific central service of the RC system is used to investigate this question and the following sub-items. This is done in Chap. 4 on pg. 31ff.
   
   (a) Is there a solution to combine global data with core-local preferences in a nonblocking way? Data that is used by a dedicated execution core per design, promises to avoid or even reduce inter-core communication.

   (b) Is there a solution to make accesses to global data nearly constant in time, independent from its actual "resource contention" without wasting space and time? This promises to induce a minimal amount of execution jitter while accessing to shared data structures.

2. Is it possible to re-use this specific solution of the previous question by a similar use case, namely memory allocation? And, if positive, which qualitative as well as quantitative impacts induces this approach? Investigations concerning this point were been made in Chap. 5 on pg. 57ff.

3. In the course of time, components, such as central services of the real-time controller, are subject to permanently changing requirements (e.g., changing the sort sequence of the data representation). This is a serious problem for data structure operations implemented in a nonblocking manner, because nonblocking protocols are very limited in terms of expandability and changeability. Chap. 6 on pg. 77ff tackles this problem and gives, furthermore, answers to the following questions:

   (a) Is there a design pattern of lock-free protocols that allows to invent a methodology to re-use lock-free protocols in real-time circumstances? Lock-free protocols exist for a variety of simple data structures, wait-free counterparts, however, are rather rare.
3. Industrial System of Investigation

(b) Is there a way to continually benefit from disjoint-access parallelism—a property that lock-free protocols have inherently (see Sec. 2.3.5 on pg. 15ff)—and which timing influence does it induce in real-time circumstances?

c) Are there still benefits in terms of time compared to spinlocks, if lock-free protocols are performed with that methodology?

4. There are still two problems to consider: First, how to handle complex data structures in a nonblocking manner without wasting too much time or space, as known from previous approaches? And second, how to re-use sequential data structure algorithms without manually care about concurrency. The latter is promising, because manually care about fine-grained concurrency is a pretty error-prone process. An in-depth discussion on that topic, along with an evaluation to traditional spinlocks, is given in Chap. 7 on pg. 99ff.

We precede our investigations in pattern-oriented way. We start with a specific use case and design a nonblocking solution for this dedicated use case. On this basis, we probe the impacts of a changing operational environment to that solution, which partially results in more generic nonblocking communication approaches that allows higher levels of recycling. For our course of action we introduce a classification for the design of nonblocking objects in the next section that enables us to classify nonblocking approaches by its level of recycling.

3.5 Division into Levels of Recycling (LoR)

The approaches presented in the next chapters offer different levels of genericness, and hence, allow different levels of recycling (LoR). The LoR levels, which we introduce in this section, define the degree of re-use when constructing the solution space for a use case. In the following, we examine only nonblocking communication mechanisms as solution space. A solution space is developed on top of a problem analysis with or without the help of existing approaches (in form of design patterns); we argue that this has a major influence on the costs introduced by research and development (R&D), and timing drawbacks. The forestalled relation based on our experiences is summarized in Tab. 3.1.

Within the scope of this thesis, we evaluate two major aspects. First, we invent methodologies as proof of concepts for communication problems that arises if the investigated real-time controller is divided into multiple parts. And second, we consider the costs in terms of R&D and execution time, if we re-use our experiences to build more generic task synchronization mechanisms within the later chapters.

We start with a specific nonblocking solution on top of a problem analysis in Chap. 4 on pg. 31ff that matches LoR-0. Thereby, no recycling of other nonblocking communication approaches is used or, in other words, we design a nonblocking solution from scratch. Thus, the efforts for R&D are foreseeable high, whereas the fields of application are rather limited to the considered and (maybe) similar use cases. However, an LoR-0 solution can be highly optimized,
and – as a result – usually does not waste hardware resources, and has only minimal drawbacks in terms of timing.

Then, Chap. 5 on pg. 57ff presents a use case similar to the latter, which gives rise to re-use parts of the solution space presented in Chap. 4. This equates to fractional recycling and accordingly to LoR-1. The costs for R&D tends to be still high, and the adapted solution promises to be still optimized, if the new use case is very similar.

In Chap. 6, we investigate the construction of a solution space by recycling solutions of foreign use cases, i.e. we re-use lock-free data structure operations, which originally are used only in general-purpose applications, in timely circumstances, i.e. LoR-2. The efforts in terms of R&D, the timing drawbacks, and the wasted parallelism for using this methodology promise to be rather moderate.

Last, we abandon use-case specific requirements within the implementation of the solution space concerning methods to tackle concurrency and, instead, introduce in Chap. 7 a generic approach that is tailored for our field of application; this is equivalent to LoR-3. A famous example of a generic approach for real-time concurrency control is any kind of real-time spinlocks. The use of spinlocks induces minimal costs for R&D, but has significant timing drawbacks as discussed in Sec. 2.2 on pg. 6ff and evaluated later on.

3.6 Problem Summary and our Approach

Nonblocking approaches to tackle concurrent situations on shared data structures are promising in parallel real-time circumstances. However, they are often blamed to be quite complex and hard to get right. On the other side, they do not scale well with a changing operational environment, because of their lack of changeability and expandability. We argue that these both serious drawbacks are the main reasons, why developers of real-time systems only sparsely consider this type of concurrency control within their software.
In addition, parallelizing industrial real-time systems that have had a long-time development, have high demands in terms of timing behavior, progress guarantee, and scalability with an increasing number of execution cores on concurrency control mechanisms. Performance is, furthermore, an issue for such systems, in particular for data and computationally intensive systems, which we focus here.

To overcome these drawbacks, we investigate two major points. First, we examine use cases of an industrial robotic control and design nonblocking approaches that satisfy these industrial requirements. On this basis, we investigate the impacts of a changing operational environment for nonblocking synchronized objects, and present appropriate nonblocking mechanisms to solve that problem.
4

LoR-0: A Wait-Free Queue with Local Preferences

4.1 Introduction

Queues are one of the most commonly used data structures in applications and operating systems [JP05]. Up-and-coming multi-core processors force software developers to consider data structures in order to make them thread-safe. In real-time systems, however, parallelization is even more complicated as such systems must guarantee to meet their mostly hard deadlines. A considerable amount of research has been carried out on wait-free objects [Her91] to achieve this. Wait freedom guarantees that each potentially concurrent thread completes its operation within a bounded number of steps (see Sec. 2.3.2 on pg. 12). Applicable wait-free queues, which supports multiple concurrent enqueue, dequeue, and read operations, however, do not exist yet.

Therefore, we present a statically allocated and statically linked queue in this chapter, which supports arbitrary concurrent operations and guarantees wait freedom. The presented approach is also applicable in other scenarios, where unsorted queues with statically allocated elements are used; we investigate this statement in the next chapter. Moreover, in this chapter we introduce 'local preferences' to minimize contention.

However, as the response times of our enqueue operation directly depends on the fill level, the response times of a nearly filled queue still remain an issue. Moreover, our approach is jitter-prone with a varying fill level. In this chapter, we also address all of these issues with an approach using a novel helping queue. The results show that we are able to decrease the maximal execution times by approximately factor twenty. Additionally, we reduce the average response times of potentially concurrent enqueue operations in our queue.
To the best of our knowledge, our wait-free queue is the best known and practical solution for an unsorted thread-safe queue for multiple enqueuers, multiple dequeuers and multiple readers.

4.1.1 Motivation

Our ongoing attempts to parallelize a robotic control kernel forced us to make the alarm handler thread-safe. Therefore, we present a wait-free queue for multiple enqueuers, multiple dequeuers, and multiple readers. The alarm handler stores alarms (e.g., drive errors) and transfers the alarm information to units outside the robotic control (e.g., human machine interface (HMI) or debugging interface (DI)) as depicted in Fig. 3.1 on pg. 21, which do not have any real-time constraints.

The alarm information is stored in a shared unsorted queue that is concurrently accessed. The alarm handler defines a global object, which is stored in the shared memory. Inside the RC all units of execution are able to concurrently access the alarm handler object. In the following these units of execution are called actuators, which are potentially concurrent reading, enqueuing and dequeuing alarms. Clients outside the RC only read-out the alarm handler’s queue. Every thread consists of multiple processing station objects that realize the information flow of the motion program through the RC kernel to the physical drives as illustrated in Fig. 3.1. A respective alarm reaction (e.g., to stop the drives) is handled on the basis of the return values of each processing station. However, the alarm information is enqueued inside the processing stations, which means that the response time to synchronously triggering the alarm information will delay the respective alarm reaction. Dequeue operations are performed by an additional background task inside the RC kernel, which can delete dedicated or groups of alarm information. For example, a group of alarms can be such alarms with a specific axis or arm number.

The presented solution of an optimized wait-free queue for multiple enqueuers, dequeuers and readers is also applicable in other concurrent scenarios, where statically allocated arrays are used and where potentially high contention occurs. This claim is investigated in Chap. 5.

4.1.2 Contribution

Previous approaches to synchronize linked queues with locks can suffer from a whole host of problems (see Sec. 2.2 pg. 6ff) that are not tolerable in real-time systems. Furthermore, we have to take into account the worst-case execution time of synchronization methods leading us to design a wait-free [Her91] approach, which does not suffer from any of the mentioned problems. However, previous work in this area, as described later in Sec. 4.6, does not satisfy the real-time and/or scenario specific requirements or is not applicable in practice.

The subject of this chapter is a statically allocated and statically linked wait-free queue with local preferences. As actuators firstly use their statically allocated local elements to trigger alarms, which appear to be placed in a static array, we called this behavior ‘local preferences’. Furthermore, we must handle...
both multiple enqueuers and multiple dequeuers, respectively. We do not want any jitter-prone compare-and-swap based retry loops or helping schemes, such as [TZ01], nor (potentially expensive) kernel lock objects; our protocol needs atomic test-and-set (TAS), test-and-reset (TAR), fetch-and-add (FAA) and bit-test (BT) operations, which are available in the most multi-core processors, as explained in Sec. 2.3.1 pg. 9ff. Additionally, a helping queue mechanism is presented with an improved traversal algorithm to reduce the time for traverse the alarm queue to find a free element. This guarantees that enqueue operations have short response times and hence the delay of the alarm reaction is minimal. To the best of our knowledge, such an approach has not been previously studied.

4.1.3 Chapter Outline

This chapter is organized as follows: Sec. 4.2 analyzes the status quo solution and describes the use-case specific requirements that must be fulfilled by a thread-safe alarm queue. In Sec. 4.3 we present our approach for a data structure and its associated wait-free protocol for concurrent operations on it. Sec. 4.4 describes a helping queue mechanism to improve our wait-free queue in terms of determinism as well as performance. This is followed by the description, discussion and evaluation of an improved traversal algorithm in Sec. 4.5. Related work is shown in Sec. 4.6. And a summary of the results and experiences is given in Sec. 4.7.

4.2 Analysis

4.2.1 Status Quo

Currently the data structure used by the alarm handler is a linked queue. All queue elements are allocated statically when the RC boots. In order to increase determinism and avoid unnecessary overhead during the runtime, the system internally maintains two queues: one with free elements and one with the active alarms. When a new alarm is issued the alarm handler takes a free element, fills it with the alarm data and inserts it into the queue of active alarms. In order to ensure consistency, the enqueue and dequeue operations are performed with disabled interrupts. If no free element is available, the new alarm will be lost. This only means that the new alarm will not be displayed on the client-side (e.g., HMI or DI). As mentioned, the alarm reaction (e.g., to stop the drives) will nevertheless be handled based on the return value of the processing station. Currently, the alarm handler stores up to 600 alarms, which is more than enough for our robotic control.

Changes in the queue cause all active clients to re-read the queue of active alarms in order to update their displays. Dequeue operations, e.g., removing one specific alarm or deleting all alarms from a dedicated station, can be performed from any actuator in the RC kernel.
4.2.2 Specific Requirements for Symmetric Multi-Processing

The following two specific requirements have to be considered for possible solutions running on multi-/many-core systems. They are the specific basis of our solution presented in the next sections. The general requirements were already mentioned in Sec. 3.3.2 pg. 25ff.

Compatibility is split into two sub-items. First, the RC’s internal compatibility with today’s interfaces to guarantee that all existing alarm handler calls are valid for the SMP-compliant version. The second sub-item concerns the external compatibility, which is much more important. Our RC controls industrial production robots in the automotive industry. To understand the requirement of external compatibility, we have to keep in mind that this is what the customers will see on their HMI. Hence, future software releases must be totally backwards compatible to previous versions.

No assumptions about timing. The RC system can trigger alarms asynchronously at any time, which are then collected as well as recorded synchronously to the program code of the RC. This means that we cannot make any assumptions regarding the timing information of when operations will actually occur.

4.3 Wait-Free Queue for Multiple Enqueuers and Dequeuers

4.3.1 Idea

Fig. 4.1 gives an overview of the structure of our wait-free alarm queue. A set of local elements is assigned to each actuator; the static allocation is done during the boot process of the RC. These ‘local queues’ are connected with each other statically. The last element of an actuator’s queue points to the first element of the next actuator’s queue. The last element of the last actuator points to the first element of the first queue. As a result we have one large queue with a statical circular structure that can be traversed using the next pointers. Traversal of the queue, e.g., to find a free element, always starts at the first element of the corresponding actuator and ends at the last element of the previous actuator. We call this behavior ‘local preferences’ of actuators. If actuators do not exceed their local elements, there is no mutual impact of other actuators. Furthermore, we have relocated the contention from an ordinary dynamic queue, where enqueue and dequeue operations induce contention on the queue itself, to the queue elements. Our approach obviously induces no contention on the queue, because it is statically linked. However, actuators can dispute about the queue elements. As an example, if there are only a few free elements, a scenario can occur where actuators compete for one element; but if there are only a few alarm elements in use, the level of contention on the free
Figure 4.1: Circular structure of our wait-free queue.
Chap. 4: LoR-0: A Wait-Free Queue with Local Preferences

Status Bits, Counter | Semantics
--- | ---
init | This bit reserves an alarm element for an enqueue or dequeue operation.
use | This bit indicates whether the alarm element is 'in use' and contains valid alarm data.
del | This bit indicates if the element has been 'logical removed' and is free for further enqueue operations.
r_cnt | This read counter indicates if and how many readers are attached to this alarm element.

Table 4.1: Semantic of the status bits.

alarm elements is very low. Hence, both the behavior of local preferences and the statically linked structure of our queue ensures minimal contention.

Each element has its own set of status bits (init, use, del), which are used to coordinate disjoint-access [IR94] parallel, and synchronize concurrent enqueue, dequeue, and read operations on each element. Disjoint-access parallelization means that the enqueue and dequeue operations concurrently work on disjunctive elements. The read counter r_cnt handles multiple simultaneous read operations with potentially concurrent enqueue and/or dequeue operations on one element. The semantics of the status bits and the read counter are depicted in Tab. 4.1. The entire queue is statically linked, this means no pointers have to be modified at runtime.

4.3.2 Protocol

For coordination and synchronization of accesses to the queue, we use the three status bits as well as one counter at each local element, as described in the previous section. The following protocol guarantees consistent data during (potentially concurrent) enqueue, dequeue, and read operations.

4.3.2.1 Enqueue

The enqueue operation always starts traversing the queue at the first element in the local queue of the actuator issuing the alarm. It checks each element whether it is freely available or marked for deletion. In the second case, the element can be used, if no read action is performed on the element.

The protocol steps are illustrated in Fig. 4.2. At first, the enqueue operation tries to reset the del bit using TAR. If it succeeds, the element is marked for deletion. Then the enqueue function must decide whether there are still actuators reading this element or not. If there are readers still attached to this element (r_cnt > 0), the enqueue operation sets the del bit again using TAS and returns for further traversal of the queue. However, if r_cnt is equal to zero, the new alarm data is written to the element. After that, the alarm data is released by the actuator. Therefore, the use bit is set and the del bit as well as the init bit are reset using one assignment to the binary mask that stores
this three bits atomically. This step represents the linearization point \( L_{P_1} \) (see Sec. 2.3.3 pg. 13ff) of the enqueue protocol, because at this point the changes become visible to all other actuators. Now, the changes are active and become visible to the readers that are notified using the observer pattern [GHJ95].

If the reset of the \( \text{del} \) bit fails at the beginning, the actuator tries to set the \( \text{init} \) bit using TAS. If TAS fails, a concurrent enqueue or dequeue operation is presently using this element and the enqueue operation returns. If TAS succeeds, the actuator checks (BT) as to whether the element’s \( \text{use} \) bit is set. If it is not, the alarm data gets written without interleavings of other operations on this dedicated slot element. The actuator then releases the written data by setting the \( \text{use} \) bit and resetting the \( \text{del} \) bit as well as the \( \text{init} \) bit — using an assignment as mentioned above — and returns. If the \( \text{use} \) bit is set, the element is still 'in use' and the actuator resets the \( \text{init} \) bit via TAR and returns. Then further traversal of the queue is required. If the traversal of the queue ends up again at the first local element without finding a free element, then the alarm is lost.

4.3.2.2 Read

Reading an element’s data requires consistent alarm data. This is ensured by using the counter \( r\_\text{cnt} \) for each element, it is incremented using FAA at the beginning of each read operation and decremented using FAA after the oper-
Chap. 4: LoR-0: A Wait-Free Queue with Local Preferences

Figure 4.3: Flow chart of the read operation.

...
Sec. 4.3: Wait-Free Queue for Multiple Enqueuers and Dequeuers

Check whether the deletion criterion of the element is fulfilled.

\[
\text{atomic } \{ \text{use}=0; \text{del}=1; \}\]

Figure 4.4: Flow chart of the dequeue operation.
a matching deletion criterion. This is because concurrent operations can, in the meantime, potentially change the element’s data. If the parameter values do match again, the actuator atomically resets the use bit and sets the del bit using one assignment to the binary mask that stores these two bits. This represents the linearization point $LP_2$, because at this point the state changes from ‘in use’ to ‘removed’. Now, no further action is needed and the dequeue function returns. If the parameter values no longer match, the actuator resets the init bit via TAR and returns.

4.3.3 Verification

We implemented our protocol also in PROMELA and validated the state transitions using the SPIN model checker [HB07]. Therein, ten concurrent units of execution per queue operation are considered for safety properties, i.e. that invalid states are not, and all valid states are reachable. In our state model we assume, furthermore, that $r\_cnt$ can take two values, i.e. zero as well as greater than zero. The latter defines a class of values. Due to actuators do not block on dedicated synchronization primitives, we omitted model checks in terms of liveness. All possible states for each alarm element are depicted in Tab. 4.2.

In the following, we describe all the invalid/valid states (from state 1 to 16 as illustrated in Tab. 4.2) of an alarm element and discuss why invalid states cannot occur.

Firstly, there is an initial state 1., where all status bits and the read counter $r\_cnt$ are zero. If an element is in the initial state, the element is free or not used. Read operations on free elements were rejected by the first bit-test and thus cannot change the state of the element. Hence, also state 9 cannot occur. Dequeue operations on a free element cannot change the state of this element, because the first bit-test will fail, if the use bit is not set. Once, an alarm element has left the initial state, the initial state cannot occur twice.

The invalid state 2, where only the del bit is set, cannot occur because if a dequeue operation was successfully performed, the element is always free for overwriting — this is state 6. If the atomic step $LP_2$ (see Fig. 4.4 on pg. 39) of the dequeue operation is performed, the init bit is always set. Hence, this state cannot occur.

If an enqueue operation was successfully executed, the element’s use bit is always set — this is state 3.

The invalid state 4 cannot occur because the linearization points $LP_1$ and $LP_2$ (see Fig. 4.2 on pg. 37, and Fig. 4.4 on pg. 39) always set the use and del bit to different values. $LP_1$ and $LP_2$ cannot be interleaved as the init and del bits ensure that parallel enqueue and dequeue operations are mutually excluded (at the element level).

13The first optimistic check of the deletion criterion is equivalent to the optimization of the average case, i.e. to avoid the likelihood of the reservation of elements (by setting the init bit on an element) without a matching deletion criterion.

14Our state model can be easily extended to more than ten concurrent actuators. However, as been commonly known this leads to an explosion of the state space. So, we limit the state model to thirty concurrent actuators, where each actuator represents the execution of a dedicated queue operation.
Sec. 4.3: Wait-Free Queue for Multiple Enqueuers and Dequeuers

### Table 4.2: Possible states of an alarm element.

<table>
<thead>
<tr>
<th>r_cnt</th>
<th>init</th>
<th>use</th>
<th>del</th>
<th>State Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>initial state</td>
</tr>
<tr>
<td>2.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>invalid state</td>
</tr>
<tr>
<td>3.</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>element in use</td>
</tr>
<tr>
<td>4.</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>invalid state</td>
</tr>
<tr>
<td>5.</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>element in process by enqueue/dequeue</td>
</tr>
<tr>
<td>6.</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>element is free for overwriting</td>
</tr>
<tr>
<td>7.</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>before LP_2 in dequeue</td>
</tr>
<tr>
<td>8.</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>invalid state</td>
</tr>
<tr>
<td>9.</td>
<td>≥ 1</td>
<td>0</td>
<td>0</td>
<td>invalid state</td>
</tr>
<tr>
<td>10.</td>
<td>≥ 1</td>
<td>0</td>
<td>0</td>
<td>invalid state</td>
</tr>
<tr>
<td>11.</td>
<td>≥ 1</td>
<td>0</td>
<td>1</td>
<td>reading an element’s alarm data</td>
</tr>
<tr>
<td>12.</td>
<td>≥ 1</td>
<td>0</td>
<td>1</td>
<td>invalid state</td>
</tr>
<tr>
<td>13.</td>
<td>≥ 1</td>
<td>1</td>
<td>0</td>
<td>element in process by enqueue/dequeue</td>
</tr>
<tr>
<td>14.</td>
<td>≥ 1</td>
<td>1</td>
<td>0</td>
<td>readers still reading a removed element</td>
</tr>
<tr>
<td>15.</td>
<td>≥ 1</td>
<td>1</td>
<td>1</td>
<td>before LP_2 in dequeue</td>
</tr>
<tr>
<td>16.</td>
<td>≥ 1</td>
<td>1</td>
<td>1</td>
<td>invalid state</td>
</tr>
</tbody>
</table>

If an element has state 5, an enqueue or dequeue operation has reserved this element by setting the init bit. An element has state 6, if a dequeue operation was successfully performed. If a dequeue operation is located in front of the linearization point LP_2, the init and use bit are set, and the del bit must be zero — this is state 7.

State 8 is invalid. It cannot occur for the same reason that state 4 cannot occur.

If actuators are presently reading the alarm data (r_cnt ≥ 1), the element’s state is 11.

If an element is removed (del=1 & init=1 & use=0) and readers are still attached to this element (r_cnt ≥ 1), the element resides in state 14.

The descriptions for the remaining five states (10, 12, 13, 15, and 16) are identical to the explanations above.

#### 4.3.4 Evaluation

For a preliminary evaluation of our protocol, we measured the response time in processor cycles against the fill level of the alarm queue. Furthermore, we compared our wait-free solution to the status quo implementation in a sequential test scenario.

##### 4.3.4.1 Testing Method

We implemented a test environment using Windows XP on the hardware platform described within the appendix in Sec. A.1 on pg. 137. In this test environment we triggered highest priority threads on the basis of the multimedia timer with a minimum resolution of 1 ms.
To interpret the results shown in the next section, we have to keep in mind that the cores, which execute our periodic threads, do not have to deal with incoming interrupts. This is because we have set the `IntAffinity` boot option in `boot.ini` to force interrupts to the highest numbered execution core, which we do not use. The jitter shown in the next section arises from cache effects. Furthermore, it depends on the strategy as to how changes in the cores’ caches are written back (e.g., write-through, write-back). We analyzed the influence of Windows in our test scenarios, which is insignificant.

For collecting the measured values, we use the 64-bit model-specific register `RDTSC` [Mui97]. As the used processors$^{15}$ perform code in an out-of-order execution, we had to flush the processor pipeline before reading out `RDTSC` in order to get suitable measured values. The `CPUID` instruction was used to flush the pipeline, which introduces some jitter ($\text{min} = 220$ cycles, $\text{max} = 284$ cycles, $\sigma = 12$ cycles, $c_v \approx 0.05$ cycles; evaluated with 1,000 test runs) against what is remaining in the pipeline; where $\sigma$ stands for the standard deviation and $c_v$ quantifies the coefficient of variation.

### 4.3.4.2 Test Scenarios, Results, and Discussion

Fig. 4.5(a) and 4.5(b) show the results of our analysis. We performed two tests using different scenarios. For every test we use 600 global elements. In the following, we use the acronym $|E_l|$ for the number of local elements and $|E_g|$ for the number of elements for the entire queue.

The first scenario compares the sequential status quo implementation with our wait-free queue. We sequentially enqueued alarms until no free element was left in the queue and measured the response times with increasing fill level. As visualized in Fig. 4.5(a) the status quo solution shows some jitter but has quite a constant response time, which is independent of the fill level. In contrast the new SMP-aware alarm queue shows that the response time scales linearly according to the fill level. Since the queue is always traversed starting at the first local element, with each iteration one more step is needed to find a free element.

The second scenario shows six concurrent enqueue operations working on our wait-free queue. For this purpose we instantiated six highest priority threads on different execution cores and concurrently enqueued alarms until no free element was left in the queue. We measured the response times with increasing fill level, as shown in Fig. 4.5(b). It can be seen that the enqueue operations have no influence on each other, while they are using their local elements. All six threads operate at about the same speed and hence only trigger alarms in their local queue elements. When the fill level of the queue reaches about 98 percent, the contention on the last remaining free elements increases because a few actuators have exceeded their local queue and a few have not. This leads to the outliers, because now actuators with an exceeded local queue must traverse the entire queue in order to find a free alarm element. Furthermore, the interchange of data through the cache hierarchies and the main memory induces an additional overhead. All of these increase the response times.

$^{15}$see Sec. A.1 on pg. 137
Sec. 4.3: Wait-Free Queue for Multiple Enqueuers and Dequeuers

(a) Sequential test case to compare the status quo alarm queue to our wait-free solution using only one highest priority thread; $|E_l| = 600$, $|E_g| = 600$

(b) Test case of concurrent enqueue operations with six highest priority threads running on different execution cores; $|E_l| = 100$, $|E_g| = 100 \times 6 = 600$

Figure 4.5: Results of our two test scenarios.
There are also several scenarios for measuring the response times of dequeue operations started from a completely filled queue. Based on its parameter value, the dequeue function decides whether an element has to be removed or not, as mentioned in Sec. 4.3.2.3 pg. 38ff. Since the dequeue function does not affect the alarm reaction, we will not consider this use case any further.

4.3.5 Conclusion

The results in the previous section show that our wait-free solution scales linearly with the fill level. This of course depends on the fragmentation of the queue. As Jayanti et al. [JP05] have already argued for their implementation of a wait-free (FIFO) queue, the use of helping schemata also interchanges time and space complexity — their solution also scales linearly. Due to fact that we did not choose a complex algorithm to traverse the queue, there is still room for improvements in terms of response time and determinism in our solution.

Our algorithm mostly fulfill the requirements described in Sec. 3.3.2 on pg. 25ff, and Sec. 4.2.2 on pg. 34 respectively. We checked the state transitions of our protocol using the SPIN model checker [HB07]. Furthermore, our protocol does not suffer from priority inversion, deadlocks, livelocks and starvation since this is inhibited by the wait-free property [Her91]. Because we are dealing with a system operating under hard real-time constraints an upper limit for the execution time is essential. This is guaranteed by the fact that incoming alarms can be rejected without influencing the real-time system if the alarm queue is full. However, jitter is still an issue with our solution since it can only be determined in an experimental manner and also depends on the fill level of the queue. Moreover, the response times for enqueue operations on a (nearly) filled queue are still too expensive for our use case.

4.4 Helping Queue

4.4.1 Motivation

As the response times directly depend on the fill level of the alarm queue, there are some issues which we take into consideration in this section. If the fill level varies, the response times of enqueue operations also vary. Moreover, the fragmentation of the queue is also an issue and also leads to varying response times because of the linear traversal of the queue. If the alarm queue is nearly full, the response times increase exponentially because of contention regarding the last free elements. And the response times of enqueue operations on a full queue take up a lot of time, since an actuator must traverse the entire queue just to identify that it is full. In Fig. 4.6, a situation is illustrated, where six actuators performs 80 percent enqueue operations and 20 percent dequeue operations with groups of alarm elements. On the average, one performed dequeue operation removes 3.5 percent of all of the used elements. The response times on the y-axis depict the response times for enqueue operations only. The downward outliers come from performed dequeue operations, which remove elements in favor of further enqueue operations. This means that the costs for traversal
of the alarm queue by further enqueue operations are influenced by dequeue operations.

To a certain extent, the jitter and the response times for enqueue operations on a nearly full queue are a problem. This is because the enqueue operation delays the respective alarm reaction. Therefore we present in this section a wait-free helping queue mechanism on the basis of our solution described in Sec. 4.3 pg. 34ff. The helping queue mechanism minimizes the costs for traversing the queue in order to find a free element as well as the varying response times of enqueue operations.

4.4.2 Idea

Each local alarm queue is separated into two halves and for each half of the local alarm queue we save the number of free elements in free_at_top and free_at_bottom respectively. The size of both halves can differ by one alarm element, if the size of the local alarm elements is odd-numbered. We use a statically allocated array with alarm elements for each actuator and abstain from connecting the arrays of alarm elements among each other in a circular structure, as illustrated in Fig. 4.7 on pg. 46. Instead of this, one helping queue element is assigned to each actuator that stores

- a pointer local_queue* to its local alarm element array,
- a constant integer variable local_middle for the first element of the second half of the alarm array.
the integer variables `free_at_top` and `free_at_bottom` as described above,
• and a pointer `next*` to the helping queue element of the next actuator.

The next pointer of the last actuator’s helping queue element statically points to the first actuator’s helping queue element. Hence, we have a helping queue with a static circular structure that can be traversed using their next pointers.

### 4.4.3 Protocol

To enqueue an alarm, an actuator have to traverse the helping queue in order to find a free alarm element. Each actuator starts traversing the helping queue at its local helping queue element to ensure our local preferences and to avoid contention. For each helping queue element the actuator consecutively checks the two counters `free_at_top` and `free_at_bottom`. If both counters are less or equal to zero, the actuator continues to traverse the helping queue by using the next pointer. If the helping queue is traversed and ends up at the first element again without finding a free element, the alarm is lost. If the actuator found a counter which is greater or equal than one, then it has found a potentially free alarm element. Then the actuator atomically decrements this counter by using FAA. If the new value of the counter, which is returned by FAA, is greater or equal to zero, then the actuator has found a free element. The decremented counter ensures that the actuator has reserved one of the free elements on the respective half of the alarm element array. If the returned value of FAA is less
than zero, the actuator must increment the counter back via FAA and continues traversing the helping queue. If the actuator has decremented a counter and the return value of FAA is greater or equal one, then it must traverse the half of the respective alarm queue, as described in Sec. 4.3 pg. 34ff, to find the free alarm element.

To ensure that the number of free elements in the alarm queue is represented by the respective counters of the helping queue elements, the dequeue operation, as described in Sec. 4.3.2.3 on pg. 38ff, must increment the respective counter using FAA after an element is removed by atomically setting the use bit to zero and the del bit to one. This is \( LP_2 \) in Fig. 4.4 on pg. 39. This means that after a dequeue operation has removed an alarm element at the alarm queue level, the alarm element is released at the helping queue level (see Fig. 4.8).

### 4.4.4 Validation

Since we are using fetch-and-add to increment and decrement the two counters \texttt{free_at_top} and \texttt{free_at_bottom}, a race situation cannot occur at these state transitions. Each counter represents the number of free elements for a dedicated half of a local alarm array. For enqueue operations we use the counters to decide whether there is a free alarm element or not. For a successful dequeue operation we increment the respective counter after the linearization point \( LP_2 \) has been executed. This is because new enqueue operations can potentially enter a half of the list as they decremented the respective counter and FAA returns the new value with a number greater or equal zero. The alarm element must have already been removed so that the enqueue operation can use it. Therefore, we must remove the element by executing \( LP_2 \) before we increment a helping queue.
counter. Hence, the step of atomically incrementing a helping queue counter in the dequeue function is the linearization point for a helping queue element, because at this point we release the removed alarm element for further enqueue operations.

The helping queue procedure ensures that if an enqueue operation has decremented a helping queue counter and FAA returns a new value which is greater than zero, then it is guaranteed that there is at least one free element for the enqueue operation at the respective half of the alarm array. A possible scenario is depicted in Fig. 4.8 on pg. 47.

If an enqueue operation traverses the entire helping queue without finding a counter which is greater or equal to one, then the alarm is lost.

4.4.5 Evaluation

This section discusses whether we have tackled the issues mentioned in Sec. 4.4.1 on pg. 44f. We minimized the time involved for traversing the alarm queue, described in Sec. 4.3 pg. 34ff, as we integrate two counters for each helping queue element. These counters count the free alarm elements in the local alarm array. Additionally, traversing the helping queue elements decreases the time for traversing the entire alarm elements. As depicted in Fig. 4.9, we also reduce the contention on the last remaining free elements. On the other hand, traversing the helping queue also results in additional overhead.

Fig. 4.9 shows that actuators initially fill the first part of their local alarm
arrays until the fill level of the entire alarm elements reaches 50 percent. At this point the helping queue protocol realizes that the first halves of the alarm arrays are full (free at top == 0) and now starts to use the second halves of the alarm arrays as the free at bottom counters are still greater than zero. The significant reduction in the response times of 50 percent fill level of all alarm elements mainly comes from the time involved with traversing the alarm array as described in Sec. 4.3 pg. 34ff. This is because now every enqueue operation starts to traverse the alarm array starting from local middle. If the fill level of all alarm elements reaches about 95 percent, contention on the counters as well as the alarm elements occurs. After the fill level reaches about 100 percent, the average response times reach 226 processor cycles — as shown in Fig. 4.9, which are required to traverse the full helping queue once.

4.4.6 Further Partitioning

Our approach to partition each local alarm queue is not limited to two parts. It is straightforward to partition each local queue up to $|E|$ parts (in the most fine-grained case). However, it is a trade-off between reducing the relatively high time it takes to traverse the alarm queue and the additional (relatively low) costs for traversal the helping queue. In the worst case, if we partition each local alarm queue in $|E|$ parts, which is element-wise, the costs for traversing the helping queue are approximately identical to those when using the alarm queue as described in Sec. 4.3 on pg. 34ff without helping queue.

4.4.7 Optimization of the Search Path

In this section we show a mathematical approach to determine the optimal number of partitions. It is assumed that the number of actuators $A$ is known as well as the number of alarm elements $|E_g|$ that are needed. For our use case $A = 6$ and $|E_g| = 600$. Neglecting the hardware costs at this point, we further assume that the costs for traversing one element at alarm queue level are 1 and the costs for checking one partition for free alarm elements at helping queue level are also 1.

Additionally, we establish the following acronyms for the two unknowns:

- $p$: Number of partitions for each local alarm queue
- $|E_p|$: Number of alarm elements per partition

It is imperative:

$$A \times p \times |E_p| = |E_g| \quad (4.1)$$

As Equation 4.1 shows, $|E_g|$ is the worst-case search path to find a free alarm element without using a helping queue. The worst case is, if there is only one free alarm element left and we must check every alarm element to identify whether it is free.
Now, we must determine the unknowns $p$ and $|E_p|$ so that we minimize the search path for finding a free alarm element. The costs for this worst-case search path with helping queue are \( \Omega : A \ast p + |E_p| \), whereas \( A \ast p \) stands for the worst-case costs for checking every partition for free elements at the helping queue level and \( |E_p| \) defines the worst-case costs for traversing the alarm elements inside the partition.

The minimum of the worst-case search path is the equation of the costs for checking every partition for free alarm elements \( A \ast p \) and the number of alarm elements per partition \( |E_p| \) (see Equation 4.2).

\[
A \ast p = |E_p| \tag{4.2}
\]

Solving Equation 4.1 for \( |E_p| \) yields:

\[
|E_p| = \frac{|E_g|}{A \ast p} \tag{4.3}
\]

Inserting the result of Equation 4.3 into Equation 4.2, we get:

\[
A \ast p = \frac{|E_g|}{A \ast p} \tag{4.4}
\]

Now, we can also solve Equation 4.4 for $p$:

\[
p = \left\lfloor \sqrt{\frac{|E_g|}{A^2}} \right\rfloor \tag{4.5}
\]

With this result of Equation 4.5, we can determine $p$ and $|E_p|$. The following steps apply to our particular use case:

1. \( p = \left\lfloor \sqrt{\frac{600}{6^2}} \right\rfloor = 4 \)
2. Insert $p$ in Equation 4.3: \( |E_p| = \frac{600}{4^2} = 25 \)
3. Determine the worst-case costs for $p = 4$ and $|E_p| = 25$: \( \Omega = 6 \ast 4 + 25 = 49 \)

Please refer to Fig. 4.12 on pg. 53 for our measuring results with $p = 4$ and $|E_p| = 25$.

If we shift the values of $p$ and $|E_p|$, e.g., $p = \{5, 3\}$ and $|E_p| = \{20, 33\}$, the costs increase \( \Omega = \{50, 51\} > 49 \). Note that for $p = 3$ and $|E_p| = 33$ there are only 594 alarm elements available.

### 4.4.8 Conclusion

Our wait-free helping queue approach is an extension for our wait-free queue mentioned in Sec. 4.3 on pg. 34ff. It is an improvement relating to determinism, worst-case costs, and performance. Indeed, the response times still depend on the fill level of the alarm queue since we have to traverse a half local alarm.
Sec. 4.5: Improved Traversal Algorithm

4.5.1 Motivation

In Sec. 4.4 on pg. 44ff we have shown a mechanism to modify the WCET to find a free alarm element. This is done by reducing the search path. In this section we discuss a simple algorithm to traverse the alarm queue to find a free element. This does not affect the WCET, but it reduces the average overhead for traversing the alarm queue.

4.5.2 Idea and Protocol

The idea is straightforward. Each actuator performing an enqueue operation traverses the alarm queue in turn from the left and the right in order to find a free element. Note that the alarm queue is traversed after the actuator has found a free element by using the helping queue as described in the previous section. Assuming that only one actuator enqueues alarms, then each half of the local alarm queue is filled as illustrated in Fig. 4.10. We do not make assumptions about the preference of actuators, which half of the local alarm queue is used first — which means that we have numbered the filling steps from 1. to 4. On the average this traversal algorithm leads to a reduced traversal overhead for the alarm queue, since actuators firstly use their local preferences and now filling each half of the local alarm queues from the left and from the right in turn. However, in the worst case, an actuator must nevertheless traverse the complete half of the local alarm queue in order to find a free element. This represents the same effort as incurred when using a linear traversal strategy. This

queue to find a free element, but the costs therefor were reduced dramatically. Additionally, we significantly reduced the jitter induced by varying fill level. By reducing the contention by dividing each local alarm queue into two halves, we have also reduced the cache-based outliers. Consequently, we also decrease the costs for enqueue operations, if the alarm queue is full.

Moreover, the helping queue mechanism allows us to divide each local alarm queue into arbitrary parts. This means that we can raise the costs for traversing the helping queue as well as reduce the costs for traversing the alarm queue and hence minimize the accumulated costs for enqueue operations.

Figure 4.10: Improved traversing sequence in turn from left/top and right/bottom for each half of the local alarm queue.
improved traversal algorithm needs an additional bit per actuator to determine the traversal direction; this does not involve additional space or time overhead.

4.5.3 Evaluation

Fig. 4.11 shows the identical concurrent test scenario as described in Sec. 4.3.4 on pg. 41ff with our simple improved traversal algorithm. If the fill level of the entire alarm queue reaches 50 percent, which means that the first halves of the local alarm queues are full, further enqueue operations of actuators use the second half of each local alarm queue. The strategy as to how the elements of each half of the local alarm queue are alternately used occurs from the left/top and from the right/bottom. As we can see in Fig. 4.11, until the alarm queue reaches a fill level of approximately 81 percent, actuators only use their local alarm elements for triggering alarms. After the alarm queue has been filled to about 81 percent, contention on the last free alarm elements increases. Since we do not use the linear traversal mechanism as described in Sec. 4.4 on pg. 44ff the contention is obviously a little bit lower. Now, the accumulated response time of each actuator has been halved as illustrated in Fig. 4.9 on pg. 48. For this reason the possible different runtimes of each actuator are of more significance; hence we moved the cache-based outliers a little feed forward.\footnote{see also Fig. 4.12} However, the average response times are reduced by approximately 50 percent in the best case compared to Sec. 4.4 on pg. 44ff.

Actuator enqueue2 and enqueue6 are obviously slower than the others, because
their response times fall to the overhead of the helping queue protocol after their last outliers by approximately 91 percent.

4.5.4 Conclusion

In real-time systems developers must ensure that resources are not exhausted. In our RC kernel the alarm queue has approximately 20 percent above the capacity of the most common failure scenarios. New incoming alarms are only rejected under worst-case conditions. In this section we had to take into consideration the average response time for triggering alarms. This ensures that the respective alarm reaction — which is triggered on the basis of the return values of the processing stations — has the shortest possible delay. In comparison to a linear traversal mechanism as described in the previous section, this improved traversal strategy reduces the average response times by approximately 50 percent in the best case. Thereby the worst-case costs are not affected (apart from one bit to determine the traversal direction) and are equal to the linear traversal algorithm.

If we divide each local alarm queue into more than two parts (e.g., four parts as illustrated in Fig. 4.12), we are able to reduce the costs of our alarm queue approach so that it is — on the average — faster than today’s sequential solution as illustrated in Fig. 4.5(a) on pg. 43. The cache effects are caused by the hardware design (e.g., cache synchronization over the 2nd level cache, or the main memory) and cannot be completely eliminated if contention occurs.
4.6 Further Related Work

Lamport developed the first single writer and multiple reader algorithm without locks [Lam77]. His approach is, however, not wait-free [Her91] and hence does not satisfy our real-time requirements. Moreover, the use case presented here requires a queue that has to deal with multiple concurrent enqueue, dequeue, and read operations. Note, furthermore, that lock freedom only guarantees that always some process completes its operation within a bounded number of steps on a lock-free data structure; this can lead to starvation of other processes, as explained in Sec. 2.3.2 on pg. 12. Also other nonblocking (FIFO) queue algorithms, such as [Val95, MS96, FOL01, Har01, TZ01], only guarantee lock freedom.

Furthermore, the obstruction-free approach from Herlihy and colleagues [HLM03] also does not satisfy our real-time requirements. As discussed in Sec. 2.3.2 on pg. 12, obstruction-Freedom only guarantees progress in isolation of all other processes.

There are only a few wait-free queues, such as [TZ02, Dav04, JP05]. Tsigas and colleagues propose in [TZ02] some wait-free queue class implementations for an amount of real-time and non-real-time threads. They focus their work to use in real-time Java. Thereby real-time threads are only allowed to write/read to/from the queue and non-real-time threads have to take the contrary role. Our scenario has to handle multiple concurrent queue operations performed within the context of real-time and non-real-time threads that cannot be satisfied by the work of Tsigas et al. Their generic queue implementations use, furthermore, the priorities of the scheduler to guarantee progress. This might induce a significant loss in parallel performance, as discussed in Sec. 2.1 on pg. 6.

In [Dav04], David presented a single enqueuer, multiple dequeue approach of a wait-free queue implementation. Jayanti et al. showed in [JP05] a multiple enqueuer and single dequeue scenario for wait-free queues and stacks. However, the use case investigated here have to deal with multiple concurrent enqueuers, dequeuers, and readers, which is not satisfied by the work from David, and Jayanti and colleague respectively.

4.7 Chapter Summary

4.7.1 Based on the Presented Approach

We designed a wait-free solution of a queue to handle multiple concurrent enqueue and dequeue operations using local preferences to reduce contention. Additionally, we presented a helping queue mechanism to significantly reduce the costs to traverse the entire queue to find a free element. Moreover, an optimized traversal algorithm was presented that results, in the best case, in significantly shorter response times, and does not induce a perceivable decline in terms of the worst-case costs.

All of this allows us to extend the alarm handling in our RC to SMP systems without requiring much further effort. The property of wait freedom ensures
that developers do not have to worry about lock orders to avoid deadlocks and additional protocols to avoid unbounded priority inversion scenarios. The wait freedom property of the presented queue also do not prohibit actuators to make progress.

This makes our approach the best known and practical solution for an unsorted thread-safe queue for multiple enqueuers, multiple dequeuers, and multiple readers.

4.7.2 Based on the Initially Positioned Questions

At first, we consider the initially positioned questions, mentioned in Sec. 3.4 on pg. 27. With the help of this approach, we are able to combine global data with local preferences, i.e. threads primary use their local affinity when triggering alarms; however, if threads exceed their local preference, they are able to use the local pools of the other threads. The latter defines a scenario that should be avoided as far as possible, when dimension and partition the queue. This indeed requires information about worst-case failure scenarios that in turn depends on the kind of robot used as well as the motion program. Hence, a generic statement regarding the "best dimension and partition strategy" cannot be given within the context of this dissertation. However, we presented a mathematical approach that helps to take decisions about that. If such situations, where threads exceed their local affinity, can be avoided, then developers can easily set up the response times of enqueue operations to be nearly constant, independent of the count of simultaneously performed queue operations.

Second, the wait-free protocol for the presented queue does not re-use parts of former approaches; this is equivalent to LoR-0 as introduced in Sec. 3.5 on pg. 28ff. The protocol is built up per greenfield strategy. This gives rise to satisfy all requirements for the specific use case of the RC alarm handler. The results therefore are remarkable, i.e. if threads do not exceed their local preferences our enqueue operation outperforms the today's solution evaluated within a sequential test case.

However, there are also drawbacks to such an LoR-0 approach, which we shall not hold back. The time for research & development is relatively high, if considered to the initial problem, i.e. to develop an SMP-aware alarm handler. The presented solution is practical, its response times are bounded, deterministic, and it offers high-performance and scalability. However, developing nonblocking solutions from scratch is rather complex and error-prone, as already remarked by Groves [Gro08]. For the industrial point of view, this might be the crucial point for neglecting nonblocking synchronized protocols from scratch, inherently. To overcome this, we present in Chap. 6 and 7 two methodologies to implement nonblocking data structure operations in a more straightforward way.

This takes us to the next point. Specific approaches concerning nonblocking coordination and synchronization of shared data structure operations provide only minimal genericness. They are optimized to a dedicated use case and by implication are not adaptable to others. The reasons for that are very limited expandability and changeability of such nonblocking protocols. Imagine an additional requirement, where alarms should be sorted regarding first-come
first-serve (FCFS) or first-come last-serve (FCLS). A sortation of alarms cannot be easily satisfied by the presented approach. The drawback of minimal genericness is investigated within the next chapter, whereas the latter issues are tackled in Chap. 6 and 7.
5.1 Introduction

Dynamic storage allocator (DSA) algorithms are well studied in the field of general-purpose applications. Most of the DSA algorithms offer good average response times and a good overall performance [MRCR04]. However, in the real-time sector, DSA algorithms are rarely used because of several major drawbacks, such as wastage of memory (also known as fragmentation problem), nondeterministic and unbounded worst-case response times as well as performance issues [Pua02]. There are some DSA algorithms that try to satisfy the needs of real-time systems, such as the TLSF memory allocator from Masmane and colleagues [MRCR04]. And there are also some real-time aware kernels (e.g., RTAI\textsuperscript{17} or MaRTE OS\textsuperscript{18}) that offer the functionality of TLSF to their applications.

In parallel real-time systems, running on shared-memory multi-core processors, the use of lock-based DSA algorithms might be very dangerous and inefficient. For example, the TLSF memory allocator uses one lock to bring concurrent allocate, reallocate and free operations in a sequential order. Without care, this can lead to missing of deadlines of tasks [RWT\textsuperscript{+}06] and inefficient exploitation of parallel CPU performance. Furthermore, the use of locks induces jitter that is a critical factor in particular for hard real-time systems, as it might lead to timing anomalies [RWT\textsuperscript{+}06]. Jitter induces also qualitative drawbacks of the final outcome of some real-time systems (see Sec. 2.2 on pg. 6ff). Moreover, Lever et al. found out that most DSA algorithms only scale up to quad-core processors [LB00].

\textsuperscript{17}see Sec. A.3 on pg. 138
\textsuperscript{18}MaRTE OS homepage: http://marte.unican.es
Nonblocking DSA algorithms, such as the memory allocator from Dice and Garthwaite [DG02], can be a good remedy for these drawbacks. But nonblocking memory allocators with a stronger progress condition than lock freedom, namely wait freedom, do not yet exist. Wait freedom guarantees that each thread completes its operation within a bounded number of steps [Her91] and hence, satisfies the temporal needs of real-time systems. Therefore, we present a wait-free DSA algorithm by adapting the helping queue of the previous chapter.

The helping queue pattern was originally used to significantly improve the worst-case response time, the determinstic timing behavior, and the performance of the wait-free queue of the previous chapter. Thereby, the enqueue operation has to traverse over the queue elements to atomically allocate an element. The helping queue was introduced in Sec. 4.4 on pg. 44ff to improve the before mentioned points.

To briefly pass revue from another point of view, the helping queue consists of wait-free linearizable counter operations, as described later on. Each counter indicates the number of free elements in some part of the array; enqueue operations traverse the helping queue and try to decrement the counter. If the decrementing was successful, the corresponding array part is guaranteed to contain at least one free array element. The worst-case traversal costs decreases from $O(\text{array\_size})$ to $O(A \cdot p + |E_p|)$ (see Sec. 4.4.7 on pg. 49ff), at which $A \cdot p + |E_p|$ can be significant less - but not greater - than $\text{array\_size}$, and whereas $A$ defines the count of units of execution, $p$ represents the count of partitions per local queue, and $|E_p|$ defines the count of elements per local queue. We exploit one side effect of this approach: If one array part is completely free, atomically decrementing the counter by its total content, that is to zero, reserves the total array part. If the array part is contiguous, it could be used for data structures that require more memory than one element can deliver. We generalized this approach to binary trees and implemented a DSA that is able to satisfy arbitrary concurrent requests.

Our approach uses a statically allocated fixed-size array of 512-bit aligned memory blocks, which are organized in a balanced binary tree. Hence, sizes of memory blocks are proportional to some power of two. We use an approximate best-fit strategy [MRCR04]. The state of free memory blocks is managed on the basis of linearizable [HW90] wait-free counters, which are atomically decremented and incremented using the fetch-and-add instruction (FAA), as already explained in Sec. 2.3.1 on pg. 9ff.

The usable size of memory per allocated chunk of $k$ blocks is restricted to $k \cdot 512 - 96$ bits, since we use three integer variables of 32-bit length for internal state administration of memory blocks. The maximal size of memory, which can be allocated, is restricted to 4 GB minus the size of the mentioned state administration of 96 bits, because our DSA approach is currently optimized to 32-bit architecture only. Possible values for $k$ are $2^d - 1$ for some $d$.

5.1.1 Motivation

The use of DSA algorithms have significant benefits compared to the common static storage management [MRCR04] in real-time applications. A central class
implementation of a DSA – with maybe different behavior variants – can breed smaller code size compared to the common practice of object-specific, use-case individual implementations with a rather substantial portion of redundant code.

Additionally, using unique interfaces for memory allocation, reallocation and deallocation is rather straightforward compared to using different memory pools with different semantics and behavior for different use cases.

Moreover, the possibility of using DSA offers some kind of flexibility at runtime [HRW08], even if the heaps are allocated statically during the initialization of the application.

Up-and-coming multi-core processors force developers to rethink their memory management implementations in terms of concurrency and the issues involved. For example, the real-time controller sketched in Sec. 3.2 on pg. 20ff defines the information of the motion program in an RC-internal representation, called motion program sentences. These sentences are extensive shared data structures, whose size directly depend on the motion program. For example, a robot motion might control only one, or up to dozens of axes. The motion and velocity profiles of the involved axes are stored inside the sentence representation, so-called sentence elements. The uni-processor RC avoids any kind of concurrency by allocate, reallocate, and deallocate sentences and sentence elements within the context of the Prep task only (see Sec. 3.2). Decomposing the RC forces now to redesign this approach.

In general, the use of an appropriate DSA can satisfy not only these changed requirements, but also leads to a more well-defined program structure, as already observed, e.g., by Herter and colleagues [HRW08].

5.1.2 Chapter Outline

This chapter is organized as follows: Sec. 5.2 describes the requirements of a real-time applicable DSA algorithm. Sec. 5.3 presents the idea, algorithm, WCET estimation, verification, evaluation, and conclusion of our wait-free memory allocator approach. Related work is shown in Sec. 5.4, and Sec. 5.5 summarizes our results and experiences.

5.2 Specific Requirements

The following points have to be considered for an applicable wait-free DSA together with these discussed in Sec. 3.3.2 on pg. 25ff. They are the basis of our approach presented in the following.

**Correctness.** In addition to the discussion in Sec. 3.3.2, a correct DSA implementation has to satisfy the following conditions:

1. Allocated memory locations have to be valid,
2. and have to be at least as big as requested.
3. Allocation requests must not influence already allocated memory areas.
4. And allocated memory locations must not overlap.

**Minimal fragmentation.** Memory is a very limited resource, in particular in the cost-sensitive sector of real-time systems. Moreover, such systems run over a long period of time without reboot. The resulting requirement is to waste as little memory as possible.

**No assumptions about timing.** Allocation, reallocation, and deallocation of memory can occur at any time. This means that we cannot make any assumptions regarding when operations will actually occur.

## 5.3 A Wait-Free DSA

We succeeded in designing a wait-free DSA using a kind of buddy allocation strategy with approximate best-fit to the principle of atomic reservation of a memory area. Buddy allocation was invented by the Nobelist Markowitz and his colleagues in [MHK63], and firstly adapted by Knowlton in 1965 [Kno65].

### 5.3.1 Idea

The available memory is divided into predefined nodes or memory blocks that form a balanced binary tree. The tree has a predefined depth $d$ and is completely filled, i.e. it has $2^d - 1$ nodes. The structure of the tree does not change at runtime, as it is allocated statically.

The algorithm only allocates complete subtrees; that is a successful allocation request allocates a node and the complete subtree that starts at this node. This implies that possible allocation sizes are $(2^n - 1) \times \text{sizeof}(\text{node})$ for some $n$ smaller or equal to $d$. Allocation requests that do not fit into one of this sizes are rounded up to the next greater value, as depicted in Fig. 5.1.

Each node has a counter, which contains information on the number of free nodes in the node’s subtree. The counter is required to support the following two operations:

**atomic_incr(c, S)** This operation atomically increments the counter $c$ by a given amount $S$ and returns. There is no return value. We used the `xadd` instruction, which is already explained in Sec. 2.3.1 on pg. 9ff.

**atomic_decr(c, S)** This operation atomically decrements the counter $c$ by a given amount $S$, if and only if the counter is at least as great as $S$. It returns `true` on a successful decrement. Otherwise, the operations returns `false` and leaves the counter unchanged. To realize the conditional part of this operation, we use two consecutive and interruptible `xadd` instructions; the impact of this approach is discussed later in Sec. 5.3.4.

A subtree with depth $d$ has $2^d - 1$ nodes and $2^{d-1}$ leaves. The counter of the subtree’s root node is initialized to the number of leaves in the subtree. After the initialization the counter contains at most the number of unallocated leaves.
Figure 5.1: The initial state of the counters inside the tree of depth 5. The left column shows the possible allocation sizes and depths; each allocation request that do not fit into one of this sizes are rounded up to the next greater value.

Figure 5.2: The tree of Fig. 5.1 with six allocations of memory — three of size 1, one of size 3 and two of size 7. The character 'x' indicates undefined counters.

in this subtree. In Fig. 5.1 the initial state of an example tree of depth 5 is shown.

If a subtree is allocated, its counter is zero; the values of the potential other counters in this subtree are undefined. Note that a counter of zero does not inherently state that the subtree starting at the node is itself allocated. For instance, Fig. 5.2 shows the state of a tree after several successful allocations. Note that the right child node of the root node has a counter of zero, even if it has not been allocated itself. Instead of that all of its sub-subtrees have been allocated.

An allocation request that needs a subtree of depth $d$, i.e. with $2^d - 1$ nodes, has the number of leaves of this subtree, that is $2^{d-1}$ as size. Correspondingly, a deallocation request that frees a subtree of depth $d$ with $2^d - 1$ nodes also has size $2^{d-1}$. Note that we only allocate complete subtrees of depth $d$ that are equal to the allocation size $2^{d-1}$ to be able to guarantee that the allocated memory reclines in contiguous memory.
5.3.2 Algorithm

5.3.2.1 Allocation

The allocation algorithm starts at the root node and traverses the tree recursively in depth-first order. It tries to allocate memory by atomically decrementing the counter of a root node to zero. For instance, the root node of an allocation request of size 3 is in depth 3 (see Fig. 5.2 on pg. 61), as sizes that do not fit into a power of two are rounded up to the next greater value, i.e. 4.

A simplified pseudocode version of the algorithm is illustrated in List. 5.1. The recursive method is called with the root node and the number of leaves, i.e. the size, in the wanted subtree as initial arguments.

It tries to decrement the counter of the current node by the size of the allocation request\(^1\). If this fails, it returns an indication of failure; in our case NULL. The counter of the current node contains at most the number of unallocated leaves in the subtree\(^2\). Thus, the algorithm shall not continue, if the subtree does not contain enough memory.

It then checks whether the subtree starting at the current node already has the requested size, in which the node has been successfully allocated and its address is returned.

If the size of the subtree is greater than the requested size, the algorithm tries a recursion call for each of the two child nodes of the current node. If one of these was successful, the allocated node is returned. If neither of the two recursive calls was successful, the algorithm re-increments the counter of the current node and returns an indication of failure.

```plaintext
try_allocate (node N, size S) {
if( atomic_decr (N.counter, S)) {
  if(S == size of subtree starting at N)
    return N;
  for C in both childnodes {
    node tmp = try_allocate(C, S);
    if( tmp != NULL ) return tmp ;
  }
  atomic_incr (N.counter, S);
} return NULL;
}
```

Listing 5.1: Pseudocode of the allocation algorithm.

```plaintext
release (node N) {
  reinitialize_subtree(N);
  S := number of leaves of subtree starting at N;
  atomic_incr(N.counter, S);
  while(N != root node) {
    N := parent of N;
    atomic_incr(N.counter, S);
  }
}
```

Listing 5.2: Pseudocode of the deallocation algorithm.

---

\(^1\) i.e. by the number of leaves that a subtree of sufficient size has to have

\(^2\) See Sec. 5.3.4 on pg. 65ff for an explanation of the difference; at this point, it is sufficient to know that the counter is never greater than the number of unallocated leaves.
5.3.2.2 Deallocation

The deallocation algorithm is even simpler: Firstly, it resets all previously undefined counters of the subtree to be deallocated to their initial values. Then, it starts at the root node of the subtree that is to deallocate, walks up the tree towards the root node of the whole tree and increments every passing node by the number of leaves of the subtree that is to be deallocated. A simplified pseudocode version of the deallocation algorithm is shown in List. 5.2.

5.3.2.3 Transformation of the Tree to an Array

There is still an unanswered question left that characterizes how the nodes of a tree are organized in the memory. The tree structure can be easily transformed to an array. This transformation has to fulfill the condition that nodes in each subtree have to occupy a contiguous slice of the array.

This transformation can be defined recursively: A tree consisting of one leaf is equal to one array element. Every other tree consists of one root node and two subtrees of the same size. It is arranged as follows: Firstly, both subtrees are recursively converted to arrays. The array that represents the whole tree is then constructed by concatenating the array of the left subtree, then the array of the right subtree, and at last one array element for the root node. The result is exemplarily shown in Fig. 5.3, where a tree of depth 4 is translated to an array of size 15.

Each node is represented by a structure as illustrated in List. 5.3.

```c
struct node {
    uint32_t length;
    uint32_t data[13];
    uint32_t padding;
    uint32_t counter;
};
```

Listing 5.3: Representation of a node in memory.

For \( n \) consecutive allocated nodes, the allocation and deallocation algorithms use the member `length` of the first node and the members `padding` and `counter` of the last node. Everything between them is usable as user data by the respective application. This is depicted in Fig. 5.4 on pg. 64.
The address returned by our allocator is the start address of the data array in the first of the consecutive allocated nodes. The length member is required to find the root node of a subtree. The padding member is used for validity checks at runtime, i.e. to detect buffer overruns. It is not strictly necessary and can be omitted at the risk of not detecting heap corruption.

![Structure of three consecutively allocated nodes.](image)

**Figure 5.4:** Structure of three consecutively allocated nodes.

Every consecutive array of allocated nodes is equivalent to a subtree. The last node is the root node of the subtree. Its counter is accessible by the allocation and deallocation routines.

### 5.3.3 Worst-Case Execution Time

We determine the WCET for our memory allocator in this section. First, we examine the execution time of one single call of the allocation function, ignoring recursion calls. All actions apart from the recursion call have, by definition, upper execution bounds and the function does not contain loops, so the execution time required for one call itself is bounded.

As the function recursively traverses parts of the binary tree in depth-first search order, it is called only a finite number of times — namely at most once for each node.

Thus, an upper bound for the execution time of the allocation can be easily established. The recursive function is executed at most

\[
O\left(\frac{\text{available\_mem}}{\text{sizeof(node)}}\right) = O\left(\frac{\text{available\_mem}}{64\ \text{bytes}}\right)
\]  

(5.1)

Deallocated also has bounded response time. Firstly, the re-initialization of the nodes in the subtree is bounded. This re-initialization is done with a depth-first traversal of the subtree that visits each node of the subtree at most once. The WCET for this is

\[
O\left(\frac{\text{sizeof(mem\_chunk)}}{\text{sizeof(node)}}\right) = O(\text{sizeof(subtree)}).
\]  

(5.2)

Second, the loop which makes up the rest of the function, is executed a bounded

\(^{21}\)The address passed to "free()" belongs to the first node of the subtree (according to the layout of the tree in the array), while the root node is the last node.
number of times. This loop is executed once for each node on the path from the root node of the subtree that is freed to the root node of the total tree. The number of nodes on this path is bounded; an upper bound is $\log_2(\text{sizeof}(\text{tree}))$. Thus, deallocation completes in at most $O(\text{sizeof}(\text{subtree}) + \log_2(\text{sizeof}(\text{tree}))$ steps.

### 5.3.4 Verification

#### 5.3.4.1 Correctness

As mentioned in Sec. 5.2 on pg. 59ff, a correct implementation of a DSA needs to satisfy the following conditions to be correct:

1. Allocated memory locations have to be valid.
2. Allocated memory locations have to be at least as big as requested.
3. Allocation requests must not influence already allocated memory areas.
4. Allocated memory locations must not overlap.

The algorithm works on a predefined array, hence the system designer is responsible to dimension the array so that all allocation requests can be satisfied. Additionally, if the array is located in valid memory, it also guarantees that returned addresses are valid. Only if there is not enough free memory in the array available, the allocation operation returns an indication of failure, that is NULL.

The second condition is also satisfied: The algorithm returns a node only if the depending subtree is as big as needed. This is guaranteed by correctly designing the recursion of the allocation routine: If the size of the tree starting at the current node is known, the size of the two subtrees is also known. As we know the size of the total array and hence the total size of the tree, we are able to keep track of whether the current node and its subtrees have the correct size.

As the algorithm only modifies the counters of the nodes, the third condition is satisfied, if the algorithm only modifies the counters that do not lie inside allocated memory. As already explained in Sec. 5.3.2.3 on pg. 63ff, of all the counters in an allocated subtree, only the counter of the root node’s subtree is not inside the allocated memory, every other counter in the subtree may not be touched.

The subtree is allocated, so the counter of the subtree’s root node is zero. Thus, no allocation nor deallocation request will descend into the node’s subtree. Every allocation request tries at first to decrement the node’s counter and consequently will fail; it then will not descend into the subtree. Deallocation requests do not descend into subtrees anyway. The only way a deallocation request gets to change the subtree’s root node’s counter is, if it tries to deallocate the same subtree, which is valid. Thus, counters inside the array remain untouched by the algorithm.
The fourth condition is equivalent to the following condition, which is somewhat easier to prove: From the moment of allocation until the deallocation of an area of memory, every other concurrently allocated area will not overlap this area.

Our allocation algorithm deals with trees and only allocates complete subtrees. Hence, the only possible way two areas can overlap is, if one area is contained in the other, i.e. if one subtree is a sub-subtree of the other subtree. To show that our algorithm also satisfies the fourth condition, it is sufficient to show that if a node is allocated, there is no other node in its subtree that is already allocated, and there is no other pending allocation request inside the subtree.

To prove this, we introduce the following condition, which is always true, if the implementations of the two counter operations are atomic as mentioned in Sec. 5.3.1 on pg. 60ff:

Let \( n_i \) be any node in the tree, \( L_i \) be the set of leaves of the subtree that starts at \( n_i \), and \( c(n) \) be the value of the counter of the node \( n \). Let \( a(n) \) be true, if and only if the node \( n \) is currently allocated. Let, furthermore, \( R_i \) be the set of requests inside the tree starting at \( n_i \), meaning the set consisting of every pending allocation request that has decremented the counter of \( n_i \) and every deallocation request that has not yet incremented the counter of \( n_i \). For each request \( r \), the function \( s(r) \) shall return the size of the request.

For each node \( n_i \) and its counter \( c(n_i) \) the following relation holds:

1. The value of the counter
2. plus the number of allocated nodes in the corresponding subtree
3. plus the sum of the sizes of all pending allocation and deallocation requests that are currently in the subtree is equal to the number of nodes in the subtree.

Formally speaking, the Equation 5.3 holds for every node \( n_i \).

\[
c(n_i) + |\{ n_j \in L_i, a(n_j) \}| + \sum_{r \in R_i} s(r) = |L_i| \quad (5.3)
\]

Equation 5.3 is true for a freshly initialized tree. By examining the following five possible changes that can be caused by allocation as well as deallocation requests, it can be proven that Equation 5.3 always holds:

1. Any allocation request might enter the subtree. This is achieved by atomically decrementing the counter of the node; hence, the Equation 5.3 holds.
2. Any allocation request might leave the subtree unsuccessfully. It then atomically increments the counter by the size it tries to allocate, so the Equation 5.3 stays true.
3. Any allocation request might leave the subtree successfully. This means it succeeds in allocating a sub-subtree of the subtree. Thus the number of allocated nodes in the subtree atomically increases by the size of the allocation request, so the Equation 5.3 holds.
4. Any deallocation request might enter the subtree. Deallocation requests enter at the root nodes of allocated sub-subtrees when they deallocate the sub-subtree by atomically incrementing the counter of the sub-subtree’s root node. Therefor the number of allocated nodes atomically decreases corresponding to the increasing sum of sizes of pending requests, so the Equation 5.3 stays true.

5. Any deallocation request might leave the subtree. This happens when it atomically increments the counter of the subtree’s root node, before walking further up the tree. In this case, the counter atomically increases correspondingly to the decrease of the sum of sizes of pending requests, so the Equation 5.3 holds.

On this point, it is straightforward to prove that if an allocation request successfully allocates a node, then there are neither allocated sub-subtrees of the subtree of the node, nor are there any pending requests inside the subtree.

This is a consequence of the employed algorithm: If an allocation request tries to allocate a node’s subtree, it tries to atomically decrement its counter by the size of the subtree. If this succeeds, the number of allocated nodes in the corresponding subtree plus the sum of the sizes of all pending allocation and deallocation requests that are currently "in" the subtree must have been zero.

An allocation request of size \( n \) that has successfully decremented the counter of a node whose subtree consists of \( n \) nodes has allocated this subtree. There can be no other pending allocation requests in this subtree; they would have decremented the counter of the node, so it would have been less than \( n \) and the previously mentioned allocation request would have failed.

5.3.4.2 Linearizability

It might be a disadvantage of the employed algorithm that it is not linearizable [HW90], namely, there exists a parallel execution that cannot be formed into a respective sequential execution.

In the following, this is discussed by an example: Consider the tree in Fig. 5.5(a) on pg. 68. If two allocation requests, one with size 2 and another with size 1 were executed in parallel, then both requests will fail, if and only if the request with size 1 is issued within the time frame where the request with size 2 has already decremented the counter of the root node. This state is depicted in Fig. 5.5(b) on pg. 68. Note that there are no consecutive nodes to satisfy the allocation request of size 2. Hence, this request will re-increment the counter of the root node when it recognizes that there is not enough contiguous memory available. Within this time frame the allocation request of size 1 will be rejected, because the counter of the root node is zero.

This implies that our DSA algorithm is not linearizable, because the parallel execution history cannot be formed into a sequential one, as the sequential execution of both requests guarantees that the requests with size 1 will always succeed. This does not pose a problem, but within this time frame memory cannot be allocated even if there is enough to satisfy incoming requests.
Figure 5.5: This example tree of depth 3 shows the impacts of our non-linearizable DSA. State A represents the initial state of the tree. State B depicts its state after a first allocation request with size 2, which has decremented the root node of the tree; within this time frame of state B, a request with size 1 will be rejected, in spite of (1) the first one cannot be satisfied, and (2) there are enough free memory available to actually satisfy the second allocation request.

We can make a weaker guarantee though: For each concurrent history of allocations without allocation failures there is an equivalent sequential history without failures. Furthermore, for each concurrent history of allocations, where only the last allocation fails, there is either a equivalent sequential history where the last allocation also fails, or there is a sequential history which is equivalent except for the last allocation and whose last allocation does not fail.

Unfortunately, this does not extend to arbitrary concurrent histories. As some early failing allocation in the concurrent history might succeed in the sequential history, it is possible that later on there is not enough memory available in the sequential history, so later on allocations that did succeed in the concurrent history might fail in the sequential history. Note that this is only possible, if there was an allocation failure in the concurrent history. Basically, transformation of the concurrent history to a sequential history makes allocation failures happen later and possibly less often, but not more often.

5.3.5 Evaluation

This section defines the testbed and some micro-benchmarks to evaluate our wait-free DSA implementation to two famous DSA implementations.

5.3.5.1 Test Method and Scenarios

For experimental results we implemented four test cases and evaluated our approach to the TLSF memory allocator. Additionally, we also measured the times of the famous lock-free DSA from Michael found in the atomic_ops project\(^\text{22}\), even if it is not fully comparable, as it cannot guarantee an upper bound of its response times due to starvation (see Sec. 2.3.4 on pg. 14ff).

In each test case, we trigger 16 threads with the highest real-time priority and run-to-completion semantic, and assign each thread on a different idle core. This

\(^{22}\)atomic_ops homepage: [www.hpl.hp.com/research/linux/atomic_ops](http://www.hpl.hp.com/research/linux/atomic_ops)
allows us to measure the approaches with a high degree of concurrency without external impacts, such as interruption or rescheduling of the test threads. We call these test cases worst-case tests. We dimensioned, furthermore, the heap to 64 megabytes.

**Test case I.** This test case measures the time required to allocate 948 bytes of memory; the allocated memory is not freed. It terminates if no more free memory is available.

**Test case II.** This test case measures the time required to allocate 948 bytes of memory and immediately free them again. The test case terminates after a predefined number of 20,000 attempts.

**Test case III.** This test case measures the time it takes to allocate memory in exponentially increasing sizes, starting with 10 bytes and growing with a factor of approximately 1.25. Similar to the first test case, the allocated memory is not freed. The test case terminates, if no more free memory is available.

**Test case IV.** This test case measures only the time it takes to allocate memory in exponentially increasing sizes, as in the third test case. After allocation the memory is freed immediately. The test case terminates, if no thread can allocate as much as it wants to.

In the future a continuous increase in execution cores is expected. Hence, we shall identify the timing behavior of our approach with an increasing level of concurrency. Thus, test case II is used as scalability benchmark, where we vary the level of concurrency. Thereby, we perform the benchmark with 1, 2, 4, (…), 14, and 16 concurrent threads as mentioned above, and examine the impacts in terms of maximal and average response times as well as execution-time jitter. We call these test cases scalability tests in the following.

### 5.3.5.2 Results and Discussion

First, the results for the worst-case tests are shown in Tab. 5.1 on pg. 70. We use the following acronyms:

- *our* for our implementation,
- *TLSF* for the real-time memory allocator TLSF [MRCR04],
- *atops* for Michael’s lock-free DSA [Mic04b].

We use, furthermore, $c_v$ for the coefficient of variation that is the indicator for execution-time jitter.

As shown in Tab. 5.1, our DSA achieves the least dispersion about the mean, as the coefficient of variation $c_v$ is very small compared to the competitors. Thus, the times for allocation and deallocation of our memory allocator are highly deterministic. TLSF uses one global lock to bring concurrent allocate, reallocate, and free operations in a sequential order; obviously, lock contention
has major drawbacks in terms of performance, maximal response times, and
determinism. Moreover, Michael’s lock-free DSA suffers from starvation, as
can be seen on the high max times in all test cases, and the significant amount
of execution-time jitter, as shown on the high cv values. Our approach does not
use any blocking locks and guarantees progress that leads to significant smaller
maximal response times. This is, along with our minimal jitter, of paramount
importance for real-time applications.

Taking the outliers into account, the average response times (avg) of our ap-
proach are significant smaller compared to TLSF and atops. Note that the
average response times of atops highly depend on nondeterministic starvation
scenarios. Thus, for test case II atops shows significant smaller response times
compared to our approach (51,320 vs. 25,256 CPU cycles), even though it is
not applicable in real-time scenarios.

Without taking the outliers into account, all DSA implementations show a good
overall performance, as shown in the according median values (med). For test
case IV, however, TLSF induces about seven times higher response times com-
pared to our approach (18,958 vs. 131,076 CPU cycles). Instead, for the similar
test case II, the root node of our tree suffers from contention, which induces
slightly higher response times for the med times compared to TLSF. The rea-
son for this discrepancy is as follows: In test case II more concurrent threads
reside inside our memory tree, which lies on the small allocation sizes of 948
bytes, compared to test case IV (with exponentially increasing allocation sizes),

\[ \text{Table 5.1: Results in CPU clock cycles of our test cases with 16 threads.} \]

\[
\begin{array}{|c|c|c|c|}
\hline
 & I & II & III & IV \\
\hline
\text{min: our} & 900 & 1,449 & 720 & 396 \\
TLSF & 540 & 666 & 513 & 468 \\
atops & 405 & 450 & 414 & 405 \\
\hline
\text{med: our} & 1,134 & 54,315 & 24,534 & 18,958 \\
TLSF & 576 & 31,784 & 5,598 & 131,076 \\
atops & 2,304 & 3,456 & 2,480 & 2,097 \\
\hline
\text{max: our} & 17,102 & 488,736 & 55,611 & 67,140 \\
TLSF & 18,009 & 3.73e+07 & 890,802 & 1.72e+06 \\
atops & 8.00e+07 & 5.57e+08 & 6.30e+06 & 6.05e+06 \\
\hline
\text{avg: our} & 1,820 & 51,320 & 22,585 & 18,546 \\
TLSF & 2,010 & 281,432 & 88,132 & 181,691 \\
atops & 26,090 & 25,256 & 47,081 & 39,555 \\
\hline
\text{cv: our} & 1.08 & 0.32 & 0.66 & 0.72 \\
TLSF & 1.29 & 1.59 & 1.85 & 1.27 \\
atops & 34.41 & 102.47 & 8.06 & 8.59 \\
\hline
\text{util.: our} & 92.59\% & ./ & 67.82\% & ./ \\
TLSF & 97.90\% & ./ & 99.06\% & ./ \\
atops & 92.50\% & ./ & 17.41\% & ./ \\
\hline
\end{array}
\]

23Note that the RTAI configuration of our testbed uses priority ceiling to avoid priority
problems, as explained in Sec. 2.2 on pg. 6ff. However, the priority inheritance protocol
has only an insignificant influence in our test cases, as all the test threads occupy the same
priority.
where our *med* response times are significant smaller compared to TLSF (18,958 vs. 131,076 CPU cycles).

Note that the *min* times depicted in Tab. 5.1 partly still suffer from contention. For example, the 1,449 CPU cycles for test case II of our DSA, whereas the *avg* times are significant smaller, as can be seen on the left plot in Fig. 5.6(a) on pg. 72.

Our tests show, furthermore, that memory consumption is typical for buddy allocation. If the requested sizes fit well, the memory wastage is acceptable. If the sizes do not match the possible tree sizes, internal fragmentation increases that is typical for buddy allocation strategies.

And second, the results for the scalability tests are depicted in Fig. 5.6 on pg. 72 that widely confirm our results for the worst-case tests also for lower concurrency levels. At first, we discuss the shown results for the lock-free atops allocator. The significant discrepancy between atops and TLSF as well as our approach are starvation scenarios that arise on concurrency levels higher than one. This can be seen on the high *max* times depicted within the left plot of Fig. 5.6(a) as well as the small *avg* response times of the plot to the right. Obviously, the most test threads make progress while a few starve. Note that starvation is a highly nondeterministic effect, which can be seen on negative pitch between concurrency level 8 and 10, in spite of the 8 test threads are scheduled on two CPUs and the 10 threads are scheduled on three CPUs (see Sec. A.2 on pg. 137). Moreover, starvation is the reason for the already inherent nondeterministic execution-time jitter, as illustrated in Fig. 5.6(b). The reason for the difference between the *max* and *avg* times for concurrency level one is the memory fragmentation, not only for atops.

Next, we examine the graphs of the plots for the TLSF memory allocator. The *max* graph is strictly increasing for all concurrency levels. For concurrency level 2, however, memory allocation becomes rather expensive in terms of the *max* times. This can also be seen on the increase from concurrency level 2 for the *avg* times. The *max* and *avg* response times for the lock-based TLSF allocator only scales up to two concurrent test threads. The jitter induced by TLSF is up to 4.97 times higher as induced by our approach (for the second test case).

Our approach shows minimal additional costs in terms of the maximal/average response times with increasing concurrency level. These costs are significant smaller compared to TLSF. Only up to concurrency level 2, TLSF is negligible faster. The execution-time jitter introduced by our approach is, furthermore, not only minimal, but also shows a decreasing trend with increasing concurrency level. This is due to the increasing contention. Thereby accesses to shared memory by means of atomic primitives (i.e. FAA) are increasingly synchronized over the main memory. For lower contention levels, these accesses are partially processed through the significant faster caches.

### 5.3.6 Conclusion

Referring to the requirements mentioned in Sec. 5.2 on pg. 59ff, we conclude that most of them are satisfied.
Figure 5.6: Scalability benchmark with increasing levels of concurrency.
We evaluated our DSA in terms of correctness in Sec. 5.3.4.1 on pg. 65ff. Indeed our protocol is not linearizable as shown in Sec. 5.3.4.2 on pg. 67ff, i.e. an allocation request will sometimes fail to allocate memory even there is enough memory available, but from a practical perspective this situation happens only if we are running very close to an out-of-memory scenario with high fragmentation. If we enlarge the memory in Fig. 5.5 on pg. 68 to size 8, there is no possible execution history, where one of the four allocation requests of size 1, 1, 2, and 1 will be rejected. Additionally, for scenarios where all allocation requests have the same size, there is not concurrent execution where a request will spuriously fail, even if there is enough contiguous memory available. Therefore our weaker correctness condition may not be a problem from a practical point of view.

Moreover, the presented algorithms have bounded response times as mentioned in Sec. 5.3.3 on pg. 64ff, which – ignoring the cache effects – is dependent on the external fragmentation of the available memory. The internal fragmentation is typical for buddy allocation strategies. It is up to the application design to prevent a high fragmentation.

Additionally, the presented algorithm is completely interrupt transparent, i.e. it can also be used in asynchronous routines, such as IRQ handler, and does not introduce bounded priority inversion scenarios like approaches with conventional locking strategies. Moreover, it is highly scalable, as the wait-free property ensures that DSA operations cannot influence other threads to make progress.

Compared to other DSA, the response times of our approach are highly predictable, as a result of minimal jitter. Predictability is – apart from the maximal response times (where we also outperform the competitors significantly) – of utmost significance in real-time applications, like the robotic controller sketched in Sec. 3.2 on pg. 20ff. Especially lock contention for lock-based memory allocators, e.g., TLSF, lead to highly unpredictable response times and to very pessimistic maximal execution times. This in turn leads to a pessimistic schedulability analysis, and hence to low scalability with an increasing number of execution cores.

5.4 Further Related Work

There is a lot of published work on dynamic storage allocation. We present the most important work for the scope of this work.

Dice and Garthwaite found out that typical DSA implementations did not scale well in high-order multi-processor systems. They present in [DG02] a mostly lock-free malloc by using multi-processor restart-able critical sections (MP-RCS). Their MP-RCS implementation needs a kernel driver to modify the decisions of the scheduler where a thread will run (either in a notification routine or at the original interrupted instruction). Without a priori knowledge about the application and the blocking system calls, it is not possible to determine an upper bound for their malloc operations.

Michael presents a scalable lock-free DSA in [Mic04b]. He claims that his DSA can be used even in real-time applications. Nonetheless, lock freedom only guarantees that at least one thread makes progress at each step and hence can
lead to starvation of some other threads. Without further a priori knowledge about the application, we cannot determine an upper bound for the response times of his DSA operations. We used his DSA for our evaluation though, as it offers an impressive performance for lock-free DSA. Within the next section, a methodology is presented to bound the response times of lock-free operations that gives rise to use his approach also in hard real-time systems.

As mentioned in Sec. 5.1 on pg. 57ff, Masmane et al. present in [MRCR04] a memory allocator for real-time systems, called two-level segregate fit (TLSF). It offers fast and bounded response times with an average internal fragmentation lower than 15 percent. However, we have shown that under stress situations TLSF did not scale well, as it uses one global lock to bring concurrent allocate, reallocate, and free operations in a sequential order. Furthermore, in preemption scenarios TLSF might suffer from bounded priority inversion that also introduces a significant amount of execution-time jitter.

Puaut presents in [Pua02] a case study of performance measurements of different DSA implementations (e.g., sequential fits, indexed fits, segregated fits, and buddy systems). She evaluated their results under real and synthetic workloads in terms of suitability in real-time systems. She found out that the best technique for DSA highly depends on the use case. As an example, in scenarios with low allocation and deallocation rates, buddy systems as well as quick-fit strategies are the best-qualified techniques in terms of predictability and performance.

Also Kriemann presents in [Kri04, Chap. 8] performance measurements of a variety of DSA implementations investigated under real and synthetic use cases. Unfortunately, the scope of his dissertation does not cover temporal requirements.

Herter et al. present in a work-in-progress paper [HRW08] an approach of a DSA that will make cache performance predictable and allocates/deallocates memory in constant time. Their DSA idea is a cache-conscious modification of TLSF. Their approach allocates/deallocates from segregated lists to achieve constant response times. Additionally, they added a new parameter to their malloc to specify a dedicated cache set. This induces the requirement that every application developer has to be familiar with the cache of the used processor. Investigations for multi-core processors were, unfortunately, not made.

There are also several approaches for multi-core aware DSA for general-purpose systems. A few of them are: Hoard [BMBW97], LKMalloc [LK98], Vee and Hus’s allocator [VH99] and Vmem [BA01]. All approaches use locks to protect the heap data for concurrent accesses and, of course, do not cover temporal requirements.

5.5 Chapter Summary

5.5.1 Based on the Presented Approach

To the best of our knowledge, we designed the first wait-free approach of a dynamic storage allocator that guarantees highly predictable as well as bounded
response times. Thus, we can satisfy the temporal requirements of real-time applications. We have, furthermore, shown that our DSA scales well with increasing number of execution cores, and the response times are significant smaller compared to the TLSF memory allocator.

There are several optimizations to the presented algorithm we did not investigate. These optimization proposals are highly dependent on the application. For the use case mentioned initially, the presented DSA satisfies our needs; so, we omitted the investigation of the following approaches: Firstly, it could be possible to improve the memory efficiency of the algorithm by changing the size of a node from 64 bytes to a different value. This might improve the memory utilization under some conditions.

Moreover, for every allocated subtree the space occupied by the nodes on the path from the subtree to the root node of the total subtree is unusable. One possible optimization would be that nodes can be defined as a separated structure, so that they need less space than a leaf.

Another possible optimization would be to change the behavior of the recursive traversal function to decrease the number of write operations, which are more expensive than respective read operations. It should be feasible to traverse the tree without changing the counters and, once we have found a 'suitable' subtree, walk down from the root node along the direct path. This change would increase performance under low concurrency and high fragmentation conditions. It would likely be detrimental to performance in a high concurrency case, even though execution time would still remain bounded.

Yet another possible optimization would be lazy re-initialization of nodes that have to be released. In our status quo DSA algorithm, deallocation takes time proportional to the size of the freed memory chunk, by re-initializing every node in the subtree. This is not necessary, if the next allocation request needs memory of the same size. One possible solution would be marking the root node of a freed piece of memory with a flag and omitting the re-initialization of the subtree. A subsequent allocation request that encounters the flag and needs to descend into the subtree can take over as much re-initialization as needed. This optimization would most likely increase the performance of deallocation, at a slight increase of the costs for the allocation performance.

5.5.2 Based on the Initially Positioned Questions

In this section, we are able to confirm the initially positioned second question from Sec. 3.4 on pg. 27. This work of a wait-free DSA with a high degree of predictability was inspired by the pattern of the alarm handler queue, presented in Chap. 4, where a wait-free linearizable helping queue mechanism for an unsorted queue is presented and "local preferences" are introduced to heavily minimize contention on the queue elements.

Locality is a desirable property in parallel real-time scenarios, as it ensures the avoidance of interferences with other units of execution. This in turn leads to high predictable response times. However, in this chapter locality has a different meaning compared to the previous chapter: In the latter, locality defines a preference of units of execution to a set of dedicated queue elements. The local
preference is only exceeded, if threads cannot find a free queue element within their preferred area. In this chapter, locality defines an essential property that returned memory has to be contiguous.

Additionally, within the previous chapter a helping queue mechanism was introduced that heavily reduces the time needed to traverse the queue in order to find a free queue element. In this chapter we generalized this approach to binary trees and implemented a DSA that is able to satisfy arbitrary concurrent requests. This corresponds to an LoR-1 approach regarding our terminology (see Sec. 3.5 on pg. 28ff).
6.1 Introduction

Over the time, software is exposed to continuous changes on the basis of a changing operational environment. This phenomenon is also known under one of the laws of software evolution, i.e. the increase of software entropy [Leh96]. For shared objects implemented by nonblocking coordination and synchronization protocols, a changing operational environment is a serious problem. This is due to their lack of expandability and changeability, because the concurrency control for nonblocking objects are inherently interwoven with their method algorithms. Thus, the need of changing and/or amplifying method algorithms of nonblocking synchronized objects mostly go along with a complete redesign of their concurrency control, too. Hence, nonblocking protocols do not scale well with a changing operational environment.

Imagine the alarm handler queue presented in Chap. 4 on pg. 31ff, where the operational environment now requires that RC alarms have to be stored chronologically in first-come first-serve (FCFS) or first-come last-serve (FCLS) order, respectively. The solution presented there cannot tackle such a requirement, and existing work on wait-free queues and stacks that support multiple concurrent and arbitrary operations does not exist yet [JP05]. Thus, from the industrial point of view, work on appropriate wait-free protocols that are able to tackle such a requirement seems often daunting, because of their algorithmic complexity and their error-prone process of getting such protocols right [Gro08].

On account of this, we introduce in this chapter a methodology that is able to re-use linearizable lock-free protocols (LLFP) in real-time circumstances without
the loss of predictability that is an essential point in such systems. This approach resides in LoR-2, as discussed in Sec. 3.5 on pg. 28ff.

We call this approach contenton-based serializing or CoBS in short. If there is no contention while executing LLFP operations, then there is no need to care about their boundary of execution time, since there is guaranteed progress; that is equivalent to disjoint-access parallelism (see Sec. 2.3.5 on pg. 15ff). There is also no need to serialize anything. Performing LLFP operations in contention, however, threads have to help each other to ensure progress. Then the executions of all failed LLFP operations are serialized. Thereby, we execute linearization points (LP) under mutual exclusion. LP are, per definition [HW90], critical regions realized by one machine instruction. These minimal critical regions are performed under mutual exclusion by a thread holding a lightweight nonblocking software lock24. Executing LP under mutual exclusion ensures that only one thread helps an operation to succeed.

Lock-free protocols are often blamed to suffer from contention [LaM94] as a reason of starvation scenarios. Executing LLFP with CoBS guarantees that all threads make progress in at least $O(N^3)$ steps, whereas $N$ is the number of concurrent threads. Additionally, we show that CoBS outperforms even simple data structures that protect their critical regions with ticket spinlocks [MCS91] by more then factor five in terms of operations per time unit. However, the maximal execution times for CoBS are about the same size compared to the ticket spinlock case. In addition, the degree of disjoint-access parallelism can be used as adjustable parameter to trade a higher average-case performance for a low increase in execution-time jitter as well as maximal response times.

6.1.1 Additional Requirements

In parallel event-triggered real-time applications (such as the real-time controller presented in Chap. 3 on pg. 19ff), where a subset of concurrent real-time threads $\vec{T}_{RT}$ sharing common data objects $\vec{O}$ with each other and with a subset of concurrent non-real-time threads $\vec{T}_{NR}$, conflicting requirements arises. Namely, $\vec{T}_{NR}$ have to satisfy a good average performance, while $\vec{T}_{RT}$ have to guarantee a definable WCET for all operations on $\vec{O}$.

Such application scenarios, where some $\vec{O}$ are accessed by a set of units of execution with different time constraints, require synchronization mechanisms that guarantee progress for all time classes to be able to guarantee a definable WCET for the real-time side and avoid starvation for the other. Moreover, such use cases are often prone to jitter to a significant high degree. This is due to bounded priority inversion scenarios of using blocking synchronization mechanisms [RSL88, Yod02] that are not interrupt transparent, and so – depending on the application scenario – might also introduce significant latencies, which might be not tolerable as discovered in [SSPL09].

Thus, we minimize progress dependencies of threads as far as possible by using linearizable wait-free protocol implementations. However, application dependent requirements let us often design own protocols instead of re-using existing

24 A lightweight nonblocking software lock defines a bit that concurrent threads conditionally try to set. Threads that fail to acquire this lock, do not block on it and immediately return.
work. From the practical point of view this approach is mostly not feasible in terms of the costs involved for research and development. Hence, this encouraged us to rethink on ‘how to use linearizable lock-free protocols’ in such scenarios. Furthermore, for $T_{NR}$ there is also the need to optimize the average case of $O$ operations, while for $T_{RT}$ we have to guarantee a bounded WCET. This led us to develop a compromise of both oppositional requirements.

### 6.1.2 Chapter Outline

This chapter is organized as follows: Sec. 6.2 describes the design pattern used by linearizable lock-free protocols. Sec. 6.3 presents the design of CobS. Sec. 6.4 discusses backoff algorithms of lock-free protocols. Sec. 6.5 presents a novel reference counting idiom used by CobS to concurrently use, re-use and free memory. Sec. 6.6 validates CobS in terms of memory consumption, progress, and pointer recycling. Sec. 6.7 discusses our results by means of synthetic worst-case benchmarks. Sec. 6.8 sketches related work, and Sec. 6.9 summarizes our experiences and outlines future work.

### 6.2 Design Pattern of Linearizable Lock-Free Protocols

This section explains the typical design pattern used by existing linearizable lock-free protocols. Later on, we use the parts of the presented design pattern to build a sophisticated auxiliary scheme to be able to guarantee progress of LLFP operations.

LLFP rely on retry loops that are responsible for their undefinable WCET [Ram97]. Fig. 6.1 on pg. 80 depicts the typical control flow graph of LLFP, such as the operations of the famous linearizable lock-free FIFO queue from Michael and Scott (MSC queue) [MS96], or the operations of the lock-free stack algorithm from Hendler et al. [HSY04]. The grey blocks in Fig. 6.1 symbolize a contiguous control flow that can be arranged into the following core blocks:

1. **Preprocessing**: Semantically, the preprocessing block is responsible for memory allocation, e.g., a node that should be added to a queue, and one-time initializations of thread local data. The WCET of the preprocessing block can be bounded.

2. **Prolog**: This block represents the first part of the retry loop without the respective LP that are responsible for the state transition of the global state. Here, thread local data is set to global state. Imagine compare-and-swap-based retry loops, where the old and new values are determined and locally set. The WCET of the prolog block can be bounded.

3. **Epilog**: This block contains the control flow to the LP of the respective protocol. The count of retries depends on the successful execution of LP\(^25\).

\(^{25}\)In cases where LP represent read-only instructions and do not modify global data, we assume that write-statements following that LP modifying global data are also integral part of the epilog.
If there are several LP then always one will be executed. Thereby, at least one LP represents a terminal condition of the LLFP retry loop. There is one special case for a different terminal condition: Imagine an empty FIFO queue, where a dequeue operation should be executed, i.e. inside the prolog block sometimes there is at least one possible 'spurious error' that leads to an unexpected terminal state of the retry loop. The WCET of the epilog block can be bounded.

4. **Post-processing**: This block is responsible for executing backlogs or freeing memory. As example, the enqueue operation of the MSC queue that tries to swing the tail pointer of the queue to the inserted node, or a dequeue operation that frees the dequeued node after reading out the node’s data. The WCET of this control block can be bounded.

We have found out that this structure is valid to all LLFP that we are aware of. For example, a FIFO queue [MS96], a stack [HSY04] and some own linearizable lock-free counter implementations. Furthermore, the retry loops from LLFP that we investigate, always follow a strict design pattern: There always exists one unbounded retry loop, whose termination depends on the successful execution of one of the LP. Nested loops inside or outside the unbounded retry loop are either bounded or also depend on the successful execution of LP. For example, the LesOp operation presented in [HSY04].

**Example: MSC Queue**

In the following, we exemplarily show a linearizable lock-free operation as well as the mapping of the before mentioned parts. List. 6.1 shows the enqueue operation presented in [MS96]. It gets a pointer to a global queue $Q$ and a value to be added to $Q$ in Line 1. It allocates a new node in Line 2 and sets the member value to the respective parameter in Line 3 and the node’s next
Sec. 6.2: Design Pattern of Linearizable Lock-Free Protocols

pointer to NULL in Line 4. According to the structure mentioned above, Lines 2–4 represent the preprocessing block.

```c
enqueue(queue_t *Q, data_t value)
{
  node := new node(); /* preprocessing */
  node->value = value;
  node->next.ptr = NULL;

  while(42) {
    tail := Q->tail; /* prolog */
    next := tail.ptr->next
    if(tail == Q->tail) { /* epilog */
      if(next.ptr == NULL) {
        if(DCAS(&tail.ptr->next, next, <node, next.count+1>)) { break;
      } else {
        DCAS(&Q->Tail, tail, <next.ptr, tail.count+1>);
      }
    }
    CAS(&Q->Tail, tail, <node, tail.count+1>); /* post-processing */
  }
```

Listing 6.1: The enqueue operation of the MSC queue [MS96]. It uses a DCAS instruction to avoid the ABA problem.

The retry loop begins in Line 5. Here the thread-local variables tail and next are set to the respective global state, i.e. Lines 6–7 represent the prolog block.

The remaining code fragment of the retry loop (ll. 8–16) constitutes the control flow to the LP in Line 10 or 14 respectively, representing the epilog block. The first LP in Line 10 is performed only if the tail pointer of the queue Q points to the last node. Respectively, the second LP in Line 14 is only executed if a thread needs help in terms of swinging the tail pointer to the last node (backlog). Note, that if we have to help another thread by executing the second LP, we have to do a retry afterwards, i.e. the success of performing the second LP is not a terminal condition of the retry loop.

The backlog in Line 18 is responsible for swinging the tail pointer of Q to the inserted node. Note that this CAS instruction may fail and then introduces a backlog for the next enqueue operation. Line 18 represents the post-processing block.

Without the inherently unbounded retry loop (ll. 5/17), the WCET of all code fragments as mentioned above can be bounded. Note that there are possible dependencies of function and method calls, e.g., the memory allocator in Line 2. We argue that there are appropriate wait-free implementations for such dependencies (in this case, we are able to re-use the wait-free DSA presented in Chap 5 on pg. 57ff), so that we are able to substitute this dependency with components that satisfy our real-time requirements.

The dequeue operation of [MS96] is similar, but has no preprocessing block. Partitioning other LLFP works similar.
6.3 Contention-Based Serializing

This section presents our novel auxiliary scheme that performs the blocks mentioned previously so that there is a guaranteed progress of concurrent LLFP operations.

To benefit from a sophisticated application design that suffers from minimal contention, we firstly try to execute the LLFP operations in a speculative manner. Therefore, CobS performs the code parts mentioned in Sec. 6.2 on pg. 79ff with respect to the original protocol. To decide whether the speculative execution succeeds or fails, the return value of the performed LP inside the epilog are considered; i.e. if the execution of LP fails and normally there is the need for a retry, then we say that the speculative execution fails.

Hence, if there is no contention, i.e. threads do not interleave each other, then there is no need to do anything. However, if there is contention and threads do interleave each other in performing LP so that some threads cannot make progress, then threads that win in making progress have to help those who do not. Therefore, we developed a sophisticated helping mechanism to bound the count of retries of LLFP operations.

Fig. 6.2 depicts our rough idea. Every white box with round corners constitutes linearizable wait-free finite state machines. In the following, we describe how to handle situations when the speculative execution does not succeed.

---

\[ r_{\text{new}} = \{k,w,CB,s\} \]

\[ r_{\text{next}} = \{k,w_{\text{copy}},CB,s\} \]

---

\[ r_{\text{new}} = \{k,w,CB,s\} \]

---

\[ r_{\text{next}} = \{k,w_{\text{copy}},CB,s\} \]

---

\[ r_{\text{new}} = \{k,w,CB,s\} \]
6.3.1 Helping Mechanism

If the speculative execution of the LLFP fails (i.e. more than one thread competes for performing LP at the same time), then a description of the triggered operation (the request \( r \)) is enqueued on a statically dimensioned array (the request table \( \vec{RT} \)), as shown in Fig. 6.2.

Every \( r \) has a unique ticket \( k \), so that requests are performed according to the FCFS principle. Furthermore, each \( r \) also has a reference working state \( w \) to allow other threads to execute their operations (that contains parameter values, working data, and return values) with a private copy \( w_{\text{copy}} \) of \( w \). Additionally, each \( r \) has pointers to the callback functions \( \vec{CB} \) (i.e. preprocessing, prolog, epilog, and post-processing) as well as the request status \( s \). Requests can adopt the states shown in Tab. 6.1 on pg. 84. Valid state transitions are discussed in Sec. 6.3.1.2 on pg. 85ff.

Threads \( \vec{T}_r \), which have enqueued their \( r_t \) with \( t \in \vec{T}_r \) try to perform requests from \( \vec{RT} \) until their \( r_t \) was successfully executed by itself or by another thread.

Those threads \( \vec{T}_r \), which succeeded in speculatively executing their operation (see Fig. 6.2), afterwards have to care that at least one request \( r_{next} \in \vec{RT} \) (if it is not empty) is successfully processed and hence the processing of \( \vec{RT} \) makes progress.

To be able to perform a code branch of LLFP operations with respect to a (not) successful helping, there are two required post-processing callback functions, one for a successful helping (post-processing on success), and one for a failed helping scenario (post-processing on failure). Imagine a thread \( t \) that helps another in performing a request \( r \) (e.g., a stalled enqueue operation). If \( t \) fails in helping, it has to free its allocated node (see List. 6.1 on pg. 81, l. 2); else, if \( t \) succeeded in performing \( r \), the post-processing callback function has to perform the backlog in Line 18. The post-processing on failure part is not available in the original LLFP. However, we argue that clearups to do in post-processing on failure, such as freeing a node, often only require one additional line of code and hence are trivial.

6.3.1.1 Request Table

As mentioned before, \( \vec{RT} \) is an array of a fixed size. \( \vec{RT} \) is statically allocated in advance. The length of \( \vec{RT} \) is equal to the maximal number of threads performing operations on it. This ensures that every thread triggering a request will find a free slot.

For an adequate memory management, we use the cache-friendly linearizable wait-free unsorted queue presented in Chap. 4 on pg. 31ff that guarantees – in this case – constant execution times for its operations. The reasons for that are as follows: We are using application scenarios, where each thread occupies a static affinity to a dedicated execution core that prohibits migration to others. Since every thread requires at most one slot to stall its own operation that is currently to perform, we have a \( 1 : 1 \) cardinality for entries of \( \vec{RT} \) and threads. Consequently, we are able to relate each slot of \( \vec{RT} \) to a dedicated thread,
Table 6.1: Processing states for requests.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_REQUEST</td>
<td>The array element contains no valid request.</td>
</tr>
<tr>
<td>IN_USE</td>
<td>The array element is reserved for a thread.</td>
</tr>
<tr>
<td>OPEN_REQUEST</td>
<td>The array element contains a valid request.</td>
</tr>
<tr>
<td>EXEC_REQUEST</td>
<td>At least one thread processing this request.</td>
</tr>
<tr>
<td>EXEC_EPILOG</td>
<td>A dedicated thread performing the epilog.</td>
</tr>
<tr>
<td>EXEC_EPILOG_DONE</td>
<td>(w_{\text{copy}}) is not written back to (w) yet.</td>
</tr>
<tr>
<td>REQUEST_DONE</td>
<td>(r) was done; request owner must reset element.</td>
</tr>
</tbody>
</table>

that is why the costs for triggering requests is constant, cache-friendly, and obviously minimal. Threads that ask for the oldest request inside \(RT\), however, have to traverse the array once, which is constant in time and scales linearly with the number of threads. Since there is currently no solution for a wait-free linearizable queue [JP05] that is able to tackle multiple concurrent operations on it, we realize our request table by means of an array. However, novel results on that area are directly capable of being integrated in favor of smaller maximal execution times for finding the oldest entry.

Requests contain, furthermore, a reference working state \(w\). This \(w\) contains parameter values (\(\text{param section}\)), working data (\(\text{data section}\)), and return values (\(\text{ret section}\)) of the stalled operation. For example, the enqueue operation of the MSC queue (List. 6.1 on pg. 81) takes as parameter values a pointer to the global queue, and the value (or the pointer to the data) that should be added to the queue. Additionally, thread local working variables (e.g., \(\text{tail}\) and \(\text{next}\) in List. 6.1) are part of \(w\). Last, imagine the dequeue operation of the MSC queue, where the data section would contain the return value (i.e. the dequeued node). Since \(w\) is read-only, all threads have to acquire \(w_{\text{copy}}\), a private copy of \(w\) (see Fig. 6.2 on pg. 82). The thread that has successfully executed \(r\) has to write back his state \(w_{\text{copy}}\) to \(w\) (implicitly under mutual exclusion) and then atomically changes the request status \(s\). We argue that large read-only parameter values are stored as reference in \(w\) and hence the space overhead here is sufficiently small.

```c
1 int cmp_ticket(uint32_t ticket, uint32_t ref_ticket) {
2     uint32_t d1, d2;
3     if(ref_ticket == ticket) return 0;
4     if(ticket < ref_ticket) {
5         d1 = ref_ticket - ticket;
6         d2 = UINT32_MAX - d1;
7     } else {
8         d2 = ticket - ref_ticket;
9         d1 = UINT32_MAX - d2;
10     }
11     if(d1 <= d2) return -1;
12     return 1;
13 }
```

Listing 6.2: Cyclic comparison of tickets.

For the memory management for (reference) working states we also use the queue introduced in Chap. 4 on pg. 31ff. However, since each thread that allocates a (reference) working state has to free it, there are no costs in terms of time incurred for traversing with respect to the deallocate function. Additionally,
since multiple threads potentially holding references to the reference working state $w$ of a stalled operation, memory management becomes a serious problem. Thus, we additionally developed a novel constant-time linearizable wait-free reference counting algorithm to guarantee that each thread that holds a reference to $w$ is able to reliably read data. This reference counting idiom is presented later in Sec. 6.5. Each thread needs at most two working state structures, one for its request $r$ and one for performing (other) requests. Thus, the worst-case memory consumption for working states is $O(2N)$.

As mentioned above, requests are sorted chronologically to assure FCFS. This is done on the basis of a unique ticket $k$ for every request; $k$ is determined using the FAA instruction. To satisfy FCFS, we have to care about the overflow of $k$; a detailed discussion on that topic is given within the next chapter in Sec. 7.5. On this point, it is sufficient to know that finding the oldest ticket number, we compare the previous performed ticket with each open request ticket. The cyclic compare function is depicted in List. 6.2.

### 6.3.1.2 Execution of Requests

The pseudocode for the execution of open requests is depicted in List. 6.3. The first function `exec_next_request()` is performed after the speculative execution of a LLFP operation succeeded. Then the chronological next request $r_{next}$ that is not already successfully performed yet (i.e. status OPEN_REQUEST, EXEC_REQUEST, or EXEC_EPILOG), is processed; $r_{next}$ is returned by the function `get_next_request()`. Afterwards, if $r_{next}$ contains a valid request, the operation tries to execute it until success. There is a progress guarantee for this operation, which is discussed later in Sec. 6.6.1.

```c
1 void exec_next_request (void) {
2    req := get_next_request ();
3    if (req == NULL) return ;
4    while (req -> status != REQUEST_DONE) {
5        try_exec_request (req);
6    }
7 }
8
9 void enqueue_request (description_t *d) {
10    own_req := allocate_request ();
11    enqueue (own_req , d);
12    do {
13        exec_next_request ();
14    } while (own_req -> status != REQUEST_DONE);
15    dequeue (own_req);
16 }
```

**Listing 6.3:** Pseudocode of the two functions for executing requests.

If the speculative execution of a LLFP operation fails, `enqueue_request()` is performed (List. 6.3, l. 9) that enqueues the operation description to the request table $\vec{RT}$, and performs requests until the own request resides in state REQUEST_DONE. More precisely, `enqueue_request()` allocates a new request $r_{new}$ by performing `allocate_request()`, where the memory allocator allocates one free request from $\vec{RT}$ as well as a working state $w$ (see Sec. 6.5 on pg. 88ff). Thereby, the state transition from NO_REQUEST $\sim$ IN_USE for $r_{new}$ is performed. After that, the operation description of the LLFP is written to $r_{new}$ and its status changes to OPEN_REQUEST. Now, the current thread $t_r$ that
triggered $r_{\text{new}}$ performs requests according to FCFS, as it executes the function \texttt{exec\_next\_request()} until a thread has successfully completed its own request. Afterwards, $t_r$ releases its own request (i.e., decrements the reference counter and conditionally frees the memory block, as explained later in Sec. 6.5) and returns. Note that in the case of a LLFP dequeue operation, the dequeued node has to be returned by accessing the respective data section inside $r_{\text{new}}$ before calling \texttt{dequeue()} (l. 15), so that \texttt{allocate\_request()} (l. 10) and \texttt{dequeue()} is part of the CobS API.

The function \texttt{try\_exec\_request()} represents a finite state machine that performs the callback functions mentioned in Sec. 6.2 on pg. 79ff. This state machine coordinates the control flow against the signals returned from the callback functions and depicted in Tab. 6.2.

All callback functions – apart from the epilog and post-processing on success/failure – return one of the first four signals. The epilog only returns 1 for success or 0 for failure. Note that the epilog is performed using mutual exclusion to avoid multiple performed operations per request. All post-processing callback functions always return \texttt{SIG\_NO\_RETURN}, a dummy signal to assure a unique interface for all callback functions.

Note that the status $s$ of requests is changed only in one direction during lifetime: \texttt{NO\_REQUEST} $\sim$ \texttt{IN\_USE} $\sim$ \texttt{OPEN\_REQUEST} $\sim$ \texttt{EXEC\_REQUEST} $\sim$ \texttt{EXEC\_EPILOG} $\sim$ \texttt{EXEC\_EPILOG\_DONE} $\sim$ \texttt{REQUEST\_DONE}. This ensures that threads holding an obsolete $s$ failed in changing $s$ to an invalid state. State transitions are made by means of a CAS instruction.

There are two special cases, where a transition of $s$ breaks this strict direction. First, if one thread $t_e$ wins in changing $s$ from \texttt{EXEC\_REQUEST} to \texttt{EXEC\_EPILOG} and then failed in successfully executing the epilog, then it needs to change it back from \texttt{EXEC\_EPILOG} to \texttt{EXEC\_REQUEST}. This is necessary because

1. the epilog is performed under mutual exclusion,
2. and there is no guarantee that the execution of the epilog succeeds (with respect to one execution).

Within the time frame of the first point, a thread holding a lock cannot be preempted, as stated on the fifth point in Sec. 6.3.2 (pg. 87). And second, if $r$ was
Some previous work on linearizable lock-free protocols tries to minimize contention as far as possible (such as elimination trees for pools and stacks from
Shavit and Touitou [ST97], combining funnels from Shavit and colleague [SZ00], or the lock-free stack algorithm from Hendler and colleagues [HSY04]). Such additional protocols are not directly mappable to CobS. Thus, we shall discuss how to deal with such LLFP.

One such approach, collision trees [HSY04], uses an elimination tree [ST97] (another technique to minimize contention) as a backoff scheme. More precisely, if there are opposed stack operations (i.e., push and pop operations) within a certain time frame, then only values are exchanged instead of making data structure changes.

The reason why such a protocol is not directly mappable to CobS is that – in the case of the stack presented in [HSY04] – the global collision array is modified within the prolog block, which conflicts with our definition in Sec. 6.2 on pg. 79ff. There are the following options to by-pass this conflict:

1. The collision array can be considered as a stand-alone protocol. Then we can apply this stand-alone protocol in a nested CobS usage. This leads to additional space requirements and timing overhead.

2. We can omit the "collision tree" approach from [HSY04] and only use the linearizable lock-free stack algorithm. In terms of WCET and jitter, this option is a better choice than the first option. Moreover, we observed that there is only minimal contention, executing LLFP operations with CobS.

3. Last, we can extend the "collision tree" protocol so that helping threads do not conflict on it. This may not be a trivial task and increases the memory consumption of the collision array at least up to \(O(N^2)\).

Considering that there is no additional benefit of the first and the third option, we argue to omit such additive protocols, i.e., to use the second option. This does not affect the correctness of such protocols, e.g., the stack from Hendler and colleagues. Besides, using backoff algorithms in real-time applications does not induce benefits in terms of a smaller WCET, and only optimize the average case. Thus, we argue, furthermore, that there is no additional benefit for using such generic protocols for speculative contention minimization in real-time circumstances.

### 6.5 Wait-Free Reference Counting Idiom

To guarantee that all threads holding references to \(w\) are able to read data reliably, we present a novel constant-time linearizable wait-free reference counting idiom in this section. The algorithm is depicted in List. 6.4. Since the (reference) working states are re-used in CobS it, furthermore, defines a solution to avoid ABA scenarios [Mic04a].

The structure \(\text{ref}_t\) consists of a reference counter \(\text{ref}_\text{cnt}\) and a status mask \(\text{in}_\text{use}\). In this simplified version\(^\text{27}\), we employ \(\text{in}_\text{use}\) as follows: If \(\text{in}_\text{use}\) is 1,
then the corresponding memory block $b$ is used. If it is 0, the corresponding $b$ is free. We use a 16-bit reference counter that can handle at most $2^{16} - 1$ references, which is sufficient for our use cases. Otherwise, if the machine supports a double-word CAS (see Sec. 2.3.1 on pg. 9ff), this idiom can be extended up to $2^{63} - 1$ references, if one bit is used ulterior.

To allocate some $b$, we argue that there is a memory allocator $M$ with a static pool of preallocated blocks. $M$ firstly increments $\text{ref\_cnt}$ of some $b$ and then tries to set $\text{in\_use}$ by means of TAS. If the TAS fails $b$ is already in use and $M$ has to perform $\text{release()}$ to decrement $\text{ref\_cnt}$. Else, $b$ was successfully allocated by $M$.

```c
1 struct ref_t {
2   volatile uint16_t ref_cnt;
3   volatile uint16_t in_use;
4};
5
6 struct transfer_t {
7 * param, data, and ret section defined here */
8   ref_t ref;
9};
10
11 bool acquire (transfer_t *t) {
12   if (!t-> ref. in_use ) return false;
13   FAA(&t->ref.ref_cnt , 1);
14   if (!t-> ref. in_use ) {
15     release(t);
16     return false;
17   }
18   return true;
19}
20
21 void release (transfer_t *t) {
22   struct ref_t a = { 0, 1 };
23   struct ref_t b = { 0, 0 };
24   FAA(&t->ref.ref_cnt , -1);
25   CAS(&t->ref , a, b);
26}
```

Listing 6.4: Linearizable wait-free reference counting.

The structure $\text{transfer\_t}$ represents the data structure for our (reference) working state $w$. One member of $\text{transfer\_t}$ defines the reference counter structure $\text{ref\_t}$ as depicted in List. 6.4, Line 8.

The function $\text{acquire()}$ tries to increment $\text{ref\_cnt}$, if and only if $\text{in\_use}$ is still 1. Else, it returns false. If this function recognizes that $\text{in\_use}$ is 0 before $\text{ref\_cnt}$ was incremented, then there is no state transition here at all (l. 12). If $\text{acquire()}$ notices that $\text{in\_use}$ is 0 after $\text{ref\_cnt}$ was incremented (l. 14), then $\text{release()}$ is called (l. 15) that tries to free the corresponding $b$, before $\text{acquire()}$ returns false (l. 16). There are no loops inside $\text{acquire()}$ and no blocking statements. The LP of $\text{acquire()}$ is Line 12/14, were the decision is taken whether $b$ is still in use or not.

The $\text{release()}$ function atomically decrements $\text{ref\_cnt}$ (l. 24) and tries to free the corresponding $b$ by performing the CAS statement (l. 25). The CAS operation succeeded, if and only if $\text{ref\_cnt}$ is 0, and $\text{in\_use}$ is 1. It also represents the LP of $\text{release()}$.

There is one concurrent execution history that we shall discuss in more detail: Namely, imagine a thread $t_1$ that successfully releases and frees $b$, while another
thread \( t_2 \) incremented the reference counter for \( b \), i.e. \( t_2 \) has executed Line 13. Then \( \text{ref\_cnt} \) is 1, while \( \text{in\_use} \) is 0, and \( t_2 \) calls \text{release()}\), decrements \( \text{ref\_cnt} \) and fails in performing the CAS operation in Line 25, as \( \text{in\_use} \) is 0. This execution history is linearizable, if and only if the corresponding memory allocator \( M \) re-uses \( b \), even though \( \text{ref\_cnt} \) is not equal to zero. Otherwise, an allocator that tries to acquire a free memory block \( b_f \) with \( \text{ref\_cnt} \) greater than zero that lets \( M \) fail in allocate \( b_f \), would not be linearizable; however, a non-linearizable \( M \) would lead to a not definable upper bound for memory consumption. Hence, we have to care that memory is re-used even if the reference counter is greater than zero.

This straightforward reference counting idiom let us concurrently use, re-use, and free (reference) working states in a transparent and safe manner.

### 6.6 Verification Aspects

In this section the progress guarantee of \text{CoS}S, its memory consumption, and the avoidance of the pointer recycling problem, while requests are re-used, are discussed.

#### 6.6.1 Progress Guarantee

To satisfy the temporal needs of real-time applications, we have to bound the WCET of LLFP operations performed with \text{CoS}S. We argue that \text{CoS}S guarantees such a bound.

There are \( N \) threads performing a LLFP operation \( op \). Thereby, the speculative execution of \( op \) and the execution of requests can interleave each other, i.e. if a thread \( t_s \in N \) wins in speculatively executing \( op \), then all other threads \( \vec{T}_o \setminus t_s \) (with \( \vec{T}_o \cup t_s = N \)) performing \( op \) at the same time may not make progress. Since threads that make progress by speculatively executing \( op \), have to care about that at least one request \( r_{next} \in \vec{R}T \) is successfully performed afterwards, there is a time where all \( N \) threads try to perform the oldest request \( r_{next} \). Exactly in that case, it is guaranteed that one thread can successfully perform \( r_{next} \), as this is assured by the progress guarantee of lock freedom (see Sec. 2.3.2 on pg. 12).

As every speculative execution may inhibit a request execution to make progress, the WCET to successfully execute \( r_{next} \) is \( O(N) \) steps. Furthermore, since the length of \( \vec{R}T \) is \( N \), the WCET to successfully execute every \( r \) is \( O(N^2) \) steps. This bound is imperative, if and only if there is a guarantee of processing requests recording to a strict FCFS principle. Actually, we cannot guarantee that \text{get\_next\_request}() (see List. 6.3 on pg. 85) always returns the request with the oldest ticket number. The reason for that is discussed within the next chapter. At this point it is sufficient to know that due to concurrency, \text{get\_next\_request}() might sometimes not return the chronological next request, and on that account, we have to call \text{get\_next\_request}() every time we try to perform the next request. This increases the WCET to successfully execute \( r_{next} \) to \( O(N^3) \).
6.6.2 Memory Consumption

Since every thread potentially helps other threads, the memory consumption for the LLFP operations scale linearly with the number \( N \) of threads, i.e. \( \mathcal{O}(N) \). Imagine the enqueue operation in List 6.1 on pg. 81, if it is performed once, the memory consumption is one node and two local stack variables (i.e. \( \text{tail} \) and \( \text{next} \)). Now, every thread that performs a stalled LLFP enqueue operation by means of ConS needs one node and the stack variables to perform this one operation. Note that after the stalled operation was successfully performed, \( N - 1 \) helping threads can free their nodes afterwards.

The static length of the request table \( \vec{RT} \) is \( N \).

The memory consumption for the (reference) working states is \( \mathcal{O}(2N) \), since there is one reference working state \( w \) per request table entry, and every helping thread has a private working state \( w_{\text{copy}} \).

6.6.3 Avoidance of the Pointer Recycling Problem

We run across the pointer recycling problem since threads working on a dedicated request that potentially is already performed by another thread. Since we change \( s \) by means of a CAS instruction only, each thread recognizes when the request status has changed in the meantime, if and only if all request states are unique.

Thus, \( s \) consists of 3 bits for the processing state (as depicted in Tab. 6.1 on pg. 84), and a 29-bit ABA counter that allows threads to recognize changes on \( s \), if threads are stalled within processing requests at most \( t \) time units. This \( t \) defines the time needed for sequentially processing \( 2^{29} - 1 \) requests. In other words, if the processing of one request takes 4.2\( \mu \)s (which is a realistic time for processing a dequeue operation of the MSC queue without cache misses), then all threads have to be scheduled within 37.58 minutes; then it is guaranteed that an ABA scenario cannot happen. If this time is not sufficient and the machine supports a FAA instruction for 64-bit counters and CAS instruction for 64 consecutive bits, then this time can be extended up to 307,094 years, when 61 bits are used as ABA counter.

This approach of avoiding ABA problems is well-known. Further discussions on that topic can be found by Michael [Mic04a].

6.7 Evaluation

This section shows and discusses the results of some benchmarks with increasing contention. We evaluate ConS to the original LLFP and an implementation that uses starvation-resistant ticket spinlocks [MCS91].

6.7.1 Test Method and Benchmarks

To evaluate ConS, we have implemented the following data structure types:
(a) the MSC queue presented in [MS96],
(b) an ordinary FIFO queue protecting their critical regions with one ticket spinlock\textsuperscript{28},
(c) and the MSC queue ported to CobS.

We implemented these data structures and operations on the basis of the test environment mentioned in Sec. A.2 on pg. 137. As memory management for nodes (see List. 6.1 on pg. 81, l. 2) we used a statically allocated array of nodes. Therein one bit is used as 'in use' indication that is set via TAS and cleared via TAR. Furthermore, the number of speculative executions is an adjustable parameter for our CobS library; so we performed the CobS-based benchmarks without and with one speculative try.

1. **Benchmark:** To show the timing behavior of all queue implementations with an increasing contention level, we instantiated up to sixteen RTAI one-shot threads with core affinity dispatched at the same time. We let RTAI schedule these threads according to \texttt{SCHED\_FIFO}. Each thread performed 10,000 test runs until termination, and triggered alternately enqueue and dequeue operations. However, we measured only the execution times of enqueue operations to avoid the dilution of runtime differences between enqueue and dequeue operations.

2. **Benchmark:** As mentioned in Sec. 6.1.1 on pg. 78ff, we are also interested in throughput. Thus, we additionally measured the count of performed enqueue and dequeue operations for one hundredth second with increasing contention level.

Unfortunately, since we are not aware of any wait-free FIFO queue with a dynamic data structure that supports multiple concurrent queue operations, we cannot make a corresponding evaluation. Please refer to Jayanti [JP05] for discussions on that topic.

### 6.7.2 Results and Discussion

The results of both benchmarks are shown in Fig. 6.3. The plots show the maximum and average response times in $\mu$s as well as the coefficient of variation ($c_v$) as the indication for execution-time jitter with an increasing level of contention. Moreover, the right plot of Fig. 6.3(b) depicts the number of object operations – that is enqueue as well as dequeue – for a one-hundredth second. All plots show results for the original MSC queue, the CobS-based queue without and with one speculative execution (spec.), and the queue using ticket spinlocks (TS queue).

First, we consider the maximal response times of enqueue operations, which are shown within the left plot of Fig. 6.3(a). The red graph represents the measured values of the original MSC queue. Note that there is no guarantee for an upper bound of these response times, and each upper response time

\textsuperscript{28}Note that ticket spinlocks [MCS91] avoid starvation inherently, and therefore are the most suitable blocking approach for our application scenarios (see Sec. 6.1.1 on pg. 78ff).
Sec. 6.7: Evaluation

(a) The left figure depicts the maximal response times, and the one to the right shows the average times of performed enqueue operations.

(b) The left figure shows changes in execution-time jitter, and the right one illustrates the count of enqueue/dequeue operations performed for a one-hundredth second.

Figure 6.3: Scalability benchmarks with increasing levels of concurrency.
may represent a missed deadline of a unit of execution that is not tolerable in real-time circumstances. Additionally, the maximal response times of these lock-free operations are nondeterministically to a high degree. The significant gap between the maximal and average response times (see also the right plot of Fig. 6.3(a)) is highly dependent on starvation scenarios, where the most threads make progress while performing lock-free operation of the MSC queue and a few starve.

The CobS-based queue, if no speculative executions are performed (blue graph), shows very similar maximal times compared to the TS queue (green graph); the differences are negligible. Note that without speculative executions, all operations performed with CobS are executed regarding to FCFS, which is analog to the ticket spinlock case. However, if CobS performs queue operations speculatively first, the maximal response times rise up to a factor ten (black graph), if the contention level increases. Note, furthermore, that the count of speculative executions can interfere with the execution of the stalled operations. Furthermore, using speculative executions lead to a higher WCET, but the likelihood of higher maximal times decreases, since more speculative tries were performed successfully; hence, fewer operations have been inserted into the request table. This can be seen on the low black graph between concurrency level 2 and 8.

Next, as stated initially at the beginning of this chapter, we are also interested in throughput. Therefore we consider therefore the right plot of Fig. 6.3(a) that shows the average response times of enqueue operations. Although, CobS is not on par in terms of avg times without speculative executions compared to the TS queue, the CobS-based queue achieves a higher throughput (also without speculative executions), as can be seen on the right plot of Fig. 6.3(b). This is due to threads helping each other to perform stalled operations. Hence, the correlation of one performed operation and the execution times for a TS queue operation does not apply to the CobS case. This is the crucial factor for our higher throughput without speculative tries. However, CobS introduces thereby a similar amount of execution-time jitter, and – as mentioned – comparable maximal response times. If CobS performs queue operations speculatively first, both the average times and the throughput in queue operations per time unit clearly outperform the values of the TS queue. Though, this goes along with an increase in the maximal response times if the contention level increases, even though the likelihood therefore decreases. Thus, developers can choose the count of speculative executions as adjustable parameter to trade off a higher throughput against a smaller worst-case response time; the setup highly depends on the use case.

Last, jitter is often a critical aspect in real-time systems, as already mentioned in Sec. 2.2 on pg. 6ff. As we expected, the coefficient of variation increases with an increasing number of speculative tries. Hence, we perform only benchmarks with one speculative try, represented by the black graph of the left plot of Fig. 6.3(b). Therein the CobS-based queue shows a comparable trend as the original MSC queue (red graph). If we do not use speculative execution, CobS induces a similar amount of jitter compared to the TS queue; the kink by 14 test threads arises out of cache effects of our benchmark that we cannot eliminate completely. This is due to the method of recording the response times of enqueue operation and the cache sizes of the used set of processors (see Sec. A.2 on pg. 137). During
the run of the benchmark, each thread stores its measured values to a global 4-byte integer array. Indeed, we guarantee that each thread always accesses disjoint parts of the array and each part lies on different cache lines (64 bytes for our testbed). However, for these benchmarks with 16 threads, the 256 KB L1 caches are too small to include the benchmark completely and hence, the number of cache misses rises enormously. If we reduce the number of test runs, this kink does not arise and the jitter for the TS queue and CobS-based queue show a decreasing trend. The decreasing trend with increasing contention is due to memory accesses to shared-memory locations increasingly going over the main memory.

CobS uses short critical regions by performing the epilog of a request under mutual exclusion. This ensures that a request will be successfully executed only once. However, threads do not block on the epilog to be performed, and are not prohibited to make progress. The higher throughput of CobS compared to the TS queue is the consequence of performing object operations that do not try to manipulate contended memory locations speculatively, or rather helping threads that have to change contended memory locations. However, the maximal response times of CobS are comparable to the queue protecting its concurrent operations by means of a ticket spinlock.

6.8 Further Related Work

In the following, we classify previous work on using nonblocking protocols in real-time applications. Herlihy introduced in [Her91] the first wait-free universal construct that has $O(\infty)$ memory consumption, serializes all operations on object $O$ and makes copies of potentially large objects. Even if he claims that the memory consumption can be reduced to $O(N^3)$, where $N$ is the number of threads accessing $O$, it suffers from two major issues: First, serializing all $O$ operations induces convoy effects [Fra04], which potentially leads to high latencies and jitter — independent of the priority of threads. Second, Jayanti showed in [Jay98] that Herlihy’s universal construct requires at least $N$ steps to complete an operation. In contrast, CobS guarantees progress in $O(N^3)$ — and we claim that this can be reduced to $O(N^2)$, if a solution for a FIFO is found that returns the oldest entry without traversing all queue elements. Furthermore, CobS successfully performs object operations in at least 1 step, as a result of speculative executions.

Ramamurthy presented in [Ram97, pp. 138 et seq.] a MWCAS implementation for real-time systems with priorities. Preemption of processes with a lower priority is allowed; preemption of processes with the equal or a higher priority is not allowed to guarantee progress. However, his MWCAS supports only uniprocessor systems. We have implemented a MWCAS\textsuperscript{29} operation based on his three-phases concept in order to make our CobS completely wait-free. Though, we observed that executing the epilog with such a MWCAS operation is very inefficient in terms of memory consumption and response times, as

\textsuperscript{29}This approach of a software CAS operation that is able to conditionally swap multiple independent words in an atomic manner is presented within the next chapter.
1. all variables must be encapsulated into MWCAS structures,

2. and accesses to such structures must be performed over a dedicated read interface due to the status of each MWCAS structure.

In short, this makes CobS indeed fully interrupt transparent, but also at least factor 100 slower and jitter rises enormously; thus, we do not pursue this aspect further.

Hohmuth presented in [Hoh02] a pragmatic methodology for using lock-free operations in real-time systems. However, his investigations are only focused on uni-processor systems.

Partitioning LLFP and execute these parts with respect to timing constraints without further scheduler assumptions is completely novel. It may be considered as a specialized universal construct. "Specialized" because it is applicable only for linearizable lock-free protocols, and "universal" as it is applicable for all such protocols. Previous work on the use of lock-free data-sharing in real-time computing was also made by Anderson and colleagues [ARJ97, ARMJ97], whose approaches bound the number of retries in lock-free operations on the basis of appropriate real-time scheduling decisions for uni-processor systems (similar to [Ram97]). Zhang extends this idea to multi-processor systems in [Zha03, Chap. 7]. However, scheduler-based approaches do not work if concurrent non-real-time and real-time units of execution are scheduled in common, as the execution of lock-free operations by non-real-time threads may lead to an obstruction in progress of lock-free operations executed by real-time threads. This procedure would define the weakest progress guarantee called obstruction freedom\(^{30}\) that is not suitable in real-time scenarios.

There is some work on nonblocking reference counting presented, e.g., by Detlefs et al. in [DMMS02] and by Sundell in [Sun04]. However, [DMMS02] only guarantees lock freedom. The idiom presented in [Sun04] guarantees wait freedom and combines the memory allocator and the reference counting. Their DeRefLink() function has \(O(N)\) time complexity, where \(N\) is the number of threads. Instead, the reference counting idiom presented in this chapter has constant time complexity.

6.9 Chapter Summary

6.9.1 Based on the Presented Approach

CobS defines a methodology to separate code blocks of linearizable lock-free protocols and execute these parts so that there is a definable WCET. The main benefits of CobS are:

**Scalability.** If there is no contention, i.e. threads do not compete for modifying global data, concurrency is not limited by CobS. Only if threads compete for global data, concurrent LLFP operations are serialized. Due to the

\(^{30}\)Obstruction freedom is explained in Sec. 2.3.2 on pg. 12
helping scheme of CobS, its throughput in object operations per time unit is clearly better than with locks, even if no speculative execution is used. However, the maximal response times of operations implemented by means of CobS are comparable to a lock-based implementation.

**Definable WCET.** Executing lock-free operations with CobS guarantees that there is no starvation of concurrent threads; starvation scenarios are highly nondeterministic, and the reason for a not definable WCET of lock-free operations. Executing LLFP with CobS leads, furthermore, to comparable maximal execution times, as implemented with ticket spinlocks.

**Bounded memory consumption.** There is an additional memory overhead to implement the auxiliary scheme for threads. This additional overhead is bounded. Moreover, for real-time systems, where the number of threads is known in advance\(^{31}\), the memory overhead is also definable a priori.

**Less jitter.** In CobS, the degree of disjoint-access parallelism can be used as adjustable parameter to trade a low execution jitter for a higher average-case performance. The use of speculative execution highly depends on the use case. However, also if no speculative execution is used, CobS clearly shows a better throughput that does not go along with a significant increase in maximal response times.

**Simplicity in using LLFP in real-time scenarios.** CobS provides a mechanism to use existing results of the area of LLFP in real-time scenarios. We argue that using CobS is quite simple by partitioning LLFP into the code parts mentioned in Sec. 6.2 on pg. 79ff. Our experience is that the complexity for using CobS is minimal compared to developing, testing, and verifying own novel nonblocking protocols.

Note that using CobS, all properties are inherited from the LLFP itself. If the LLFP suffers from issues like the ABA problem, then executing this LLFP with CobS suffers also from these issues.

The helping mechanism as described in Sec. 6.3.1 on pg. 83ff is not restricted to a specific LLFP operation, i.e. the overhead for the request table accumulates only once, independently of the count of different LLFP implementation used.

**6.9.2 Based on the Initially Positioned Questions**

Within this section, we consider the initially positioned questions regarding the third point (see Sec. 3.4 on pg. 27ff). In this chapter, we have tackled the impacts of a continuously changing operational environment to nonblocking coordination and synchronization protocols. Increasing software entropy is a serious problem for shared objects implemented in a nonblocking manner that additionally implement the concurrency control. This is due to the lack of expandability and changeability of such objects. Therefore, we have developed a methodology to re-use linearizable lock-free algorithms, which usually cannot guarantee an upper bound in their time complexity, of the general-purpose application

\(^{31}\)For real-time systems considered within the scope of this thesis, the number of threads is typical known in advance.
domain in real-time circumstances. Thus, our approach is not a solution to make dedicated nonblocking protocols expandable and adaptable to an altering operational environment, but it offers the possibility to re-use approaches that currently fit to our operational environment, but comes for the general-purpose domain; that is an LoR-2 approach, as explained in Sec. 3.5 on pg. 28ff.

On account of this, we have investigated some linearizable lock-free protocols and identified these code parts that are the reason for their undefinable WCET. In an empirical and experimental manner, we have found out that there is an abstract control-flow description of this class of protocols that can be used to construct an auxiliary scheme to be able to define a WCET for such protocols performed with that helping approach. Thereby, our approach is widely independent of complex and restrictive scheduling decisions. The only requirement that have to be satisfied outside the scope of CobS is that units of execution performing the epilog of LLFP must not be suppressed or rather interrupted. We argue that satisfying this requirement is straightforward for the use cases considered here.

In order to avoid that all object operations performed by means of CobS are serialized, we have introduced the approach of speculative execution. Thereby, CobS does not serialize operations that do not conflict. We have ascertained that speculative execution leads – in the worst-case – to higher execution times and more execution jitter, but also to better average case times. However, even if CobS does not perform operations speculatively, the throughput in object operations per time unit is significant better compared to the ticket spinlock case. It is the developers’ responsibility to define the number of speculative tries, which is an adjustable parameter of the CobS library to trade off a higher throughput against a smaller worst-case response time. For the application domain investigated within the context of this thesis, such an average-case optimization absolutely makes sense in some parts. For instance, imagine objects that are only shared between different Prep task instantiations and have logical and/or arithmetic dependencies (see Sec. 3.2 on pg. 20ff). However, CobS shows also a higher throughput even without speculative execution compared to ticket spinlocks. This all goes along with almost no additional costs in terms of time compared to ticket spinlocks, even though operations are not performed speculatively.
7

LoR-3: A Wait-Free NCAS Library

7.1 Introduction

Within the previous three chapters, we have examined only simple data structures (such as singly-linked queues, lists, or stacks), and how to handle operations on it in a nonblocking manner that satisfies the time constraints of real-time (control) systems. However, simple data structures constitute only a part of those used in such systems focused here (see Chap. 3 on pg. 19ff). The other part includes more complex data structures on the one side, and/or consequently, more complex data structure operations on the other.

In addition, the increase of software entropy [Leh96] also affects shared data structures and their operations so that some of these cannot be implemented anymore in a nonblocking way by means of atomic memory instructions (see Sec. 2.3.1 on pg. 9ff).

Moreover, fine-grained parallelization of real-world real-time systems is a rather error-prone, time-consuming, and an inherent complex process. Thus, a methodology that enables developers to re-use sequential algorithms of shared data structures without the need to manually care about concurrent execution scenarios is of high magnitude. This is due to the inherent deeply rooted operational environment of real-time systems, where software defects, errors, and bugs may result in significant consequences; re-using sequential algorithms in parallel environments without the need to manually care about concurrency highly thwarts this error-prone process of parallelizing real-time systems.

In this chapter, all these problems are tackled with an approach that we call irrNCAS, a library that is able to conditionally swap arbitrarily located and multiple machine words in an atomic manner (NCAS) and guarantees a bounded
time complexity for such operations. The NCAS method implemented in rt-NCAS has a weakly wait-free\footnote{With weak wait freedom we understand a progress property analog to Herlihy’s definition in [Her91], but without guarantee of success; i.e. in case of concurrent executions a weakly wait-free operation may abort without modification, but will finish within a bounded time interval. Machine instructions, such as Intel’s cmpxchg with lock prefix (see Sec. 2.3.1 on pg. 9ff), offer exactly this progress guarantee.} progress guarantee. rtNCAS also offers weakly wait-free read interfaces to safety read the NCAS words. This allows the implementation of arbitrary complex data structure operations that are wait-free and linearizable.

Our work was inspired by Ramamurthy’s NCAS presented in [Ram97, pp. 138 et seq.]. However, his NCAS only focuses on priority-based real-time systems running on uni-processor boards. rtNCAS mainly focuses on multi- and many-core processor systems, but also works on uni-processor systems. Furthermore, we are not restricted to a priority-based real-time scheduler as Ramamurthy’s approach. rtNCAS is able to conditionally swap up to 256 independent words in an atomic way and thus enables us to implement all desirable simple and complex data structure operations in an easy way. rtNCAS is currently able to handle up to 4,096 concurrent threads. On today’s point of view, the real-time controller sketched in Chap. 3 on pg. 19ff. likely will not exceed this limit within the next few years.

Thus, rtNCAS enables developers to implement arbitrarily complex data structures in highly concurrent real-time environments in an easy way, while ensuring linearizability, wait freedom as well as disjoint-access parallelism. Moreover, developers can re-use their sequential algorithms on top of rtNCAS without further modifications and care about concurrency. In addition, the degree of disjoint-access parallelism can be used to trade a low execution-time jitter for a higher average-case performance, similar to speculative execution in CobS presented in the previous chapter.

We show that wait-free data structure operations using rtNCAS induce more than three times less execution jitter than their blocking counterparts. Additionally, we exhibit about two times more throughput (measured in data structure operations per time unit) without inducing a significant increase in their maximal response times.

### 7.1.1 Contribution

The rtNCAS library comprises the following core contributions:

1. Weakly wait-free and linearizable NCAS and read operations for shared-memory multi-/many-processing that can handle multiple memory words in an atomic manner. These operations are primarily used to build shared data structures in a lock-free manner, but they can also be used for lock-free data transactions.

2. A methodology for a helping scheme, which we call operation queue, in order to perform data structure operations in a wait-free manner. This helping scheme in combination with our NCAS and read operations can satisfy the temporal needs of hard real-time systems.
3. We have recycled the constant time, linearizable, wait-free reference counting idiom – presented in the previous chapter – to concurrently use, re-use, and free memory blocks inside rtNCAS in a safe way. Due to a nontrivial integration and use pattern of this reference counting within the scope of rtNCAS, this chapter describes this idiom orthogonally to the previous chapter.

7.1.2 Chapter Outline

This chapter is organized as follows: Sec. 7.2 describes our system model and derives the assumptions and general conditions under which rtNCAS works correctly. In Sec. 7.3 we present the design of rtNCAS. Sec. 7.4 discusses details of the NCAS operation, Sec. 7.5 presents the operation queue and Sec. 7.6 describes our reference counting idiom. Sec. 7.7 evaluates rtNCAS with some micro-benchmarks and Sec. 7.8 sketches related work. Finally, Sec. 7.9 sums up the work presented in this chapter and answers the questions mentioned in Sec. 3.4 on pg. 27ff.

7.2 System Model

We consider an asynchronous shared-memory system with multiple closely connected processing elements. To show under which conditions rtNCAS works correctly, we make the following assumptions:

1. The number of threads is limited and known in advance. Most real-time applications (especially those with hard timing constraints) fulfill this precondition.

2. The target machine has to support some atomic instructions used by our rtNCAS. Most multi-core CPUs offer such instructions (see Sec. 2.3.1 on pg. 9ff):
   - Fetch-and-Add (FAA).
   - Compare-and-Swap (CAS).
   - Double-word Compare-and-Swap (DCAS).\(^{33}\)

3. The target machine has to guarantee a WCET for the atomic instructions stated above.

7.3 rtNCAS

In this section we sketch the overall structure of our rtNCAS library. The components contributing to the library are described in more detail in the subsequent sections.

---

\(^{33}\)Analog to the previous chapter, we use the syntax of Michael and Scott [MS96] to describe double-words inside the pseudocode listings: \(<word1,word2>\).
7.3.1 Terminology

An *NCAS method* is equivalent to the pseudocode depicted in List. 7.1. It is used to atomically change several unrelated variables, if and only if they contain the given old value.

```c
NCAS( tochange : array[n] of pointer to word;
old : array[n] of word;
new : array[n] of word;
n : integer ) {
  --begin atomic--
  for i = 0 to n-1:
    if * tochange[i] != old[i]
      return false;
  for i = 0 to n-1:
    * tochange[i] = new[i];
  return true;
--end atomic--
}
```

Listing 7.1: Pseudocode of an NCAS operation.

An *NCAS operation* corresponds to an invocation of an NCAS method; the NCAS method implements the NCAS operation.

A *data structure operation* is a user-defined function on a shared data structure (such as enqueue on a FIFO queue). The actual semantics of the data structure operation is implemented in a user-defined callback function \(\lambda\) activated through an upcall [Cla85] by our \(\text{rtNCAS}\) library.

7.3.2 Properties

The user-defined data structure operations implemented via \(\text{rtNCAS}\) have to be correct and provide strong progress guarantees that allow the definition of upper bounds for their WCET. These properties make such operations applicable in hard real-time applications (see Sec. 2.3.2 on pg. 12).

\(\text{rtNCAS}\) guarantees that such operations are always linearizable, which offers an intuitive model for understanding and verifying the behavior of highly concurrent implementations (see Sec. 2.3.3 on pg. 13ff). We have verified \(\text{rtNCAS}\) completely by means of linearizability to ensure correctness of our design. However, to understand the key idea behind \(\text{rtNCAS}\) and avoid tedious correctness proofs, this chapter only sketches relevant parts thereof; please refer to Krainz’s diploma thesis [Kra10] for the complete proofs.

Implementations of data structure operations using \(\text{rtNCAS}\) also support strong progress properties. To achieve those strong properties, we use the techniques illustrated in Fig. 7.1.

1. *Lock-free operations* offer interrupt transparency and neither depend on system libraries, nor induce restrictions to the scheduler (we do not consider backoff algorithms to minimize contention, such as [ST97], here).
### Figure 7.1

Uses hierarchy [Par76] of rtNCAS. The right grey big block depicts the progress properties ensured by the transformation steps on top of a sequential data structure operation. Thereby, lower boxes represent stronger properties. The left grey big block depicts the rtNCAS techniques used within the transformations to achieve these progress properties.
Under concurrent usage they also frequently yield better performance than blocking counterparts [Zha03]. As no progress guarantees can be given for such operations, it is also not possible to determine an upper bound for the WCET of such operations.

2. **Wait-free operations** [Her91] are lock-free and have a determinable and finite WCET. Hereby, the temporal progress requirements of real-time applications can be satisfied.

3. **Disjoint-access-parallel operations** [IR94] offer parallel executions of operations accessing disjointed memory locations. This yields additional benefits in terms of performance in multi- and many-core environments.

Starting with a sequential implementation of a data structure operation, we apply several transformations to it. Each of these transformations ensures the next stronger progress property and finally, a wait-free and disjoint-access-parallel data structure operation is reached.

This modular design, as illustrated in Fig. 7.1 on pg. 103, allows us to adapt rtNCAS to different use cases; for instance, some soft real-time tasks might be able to tolerate more jitter for a higher performance, whereas jitter is a critical factor for hard real-time units of execution (see Chap. 3 on pg. 19ff).

### 7.3.3 Design

Sequential operations on data structures usually rely on exclusive access to the parts of the structure by one unit of execution. Their methods are typically implemented using reads and writes to single words spread out over the whole critical region to manipulate the state of a data structure. Fig. 7.2 depicts an example where three words `tail`, `tail->next` and `len` — comprising the current state of a linked-list — are manipulated when adding an element. Such operations correspond to sequential operations in Fig. 7.1 on pg. 103.

#### 7.3.3.1 Lock-Free Operations

In a first step the formerly sequential operation has to be transformed into a lock-free operation. This is accomplished by reading the old state of the data structure via a **read method** (see Fig. 7.1), computing a new state, and, finally, atomically exchanging the old state with the new one by an NCAS operation. In case of concurrent interference the NCAS operation may fail and the complete operation including reading the old state and computing the new state has to be repeated.

An NCAS structure (see Fig. 7.2) is passed as argument to the **NCAS method**. This structure contains the old and the new state of the data structure that should be exchanged atomically and the memory locations containing the state wrapped into so called NCAS words.

The usage of the NCAS operation as described above is both lock-free and linearizable. As illustrated in Fig. 7.1, we need the following two requirements:
Sec. 7.3: rtNCAS

Figure 7.2: This figure shows all data structures involved in rtNCAS and illustrates an example of adding a node n3 to a FIFO queue by using rtNCAS. An enqueue operation implemented via locks would modify tail, tail->next, len inside a critical region to ensure atomicity. To add n3 to the queue using rtNCAS, all words of the data structure are encapsulated into NCAS words. The thread performing the enqueue operation has to create an opqueue structure containing a unique ticket to be able to deduce a chronological order and a user-defined callback \( \lambda \) to create an NCAS structure describing the atomic state transformation to enqueue n3. This opqueue structure is enqueued to the operation queue providing a wait-free helping scheme, where concurrent threads help each other to perform those stalled operations.
Requirement L1. We need a weakly wait-free NCAS method to support lock-free data structure operations.

Requirement L2. We need a read method to retrieve the actual value from an NCAS word.

The NCAS method and the read method is described in Sec. 7.4 on pg. 108ff in detail.

7.3.3.2 Wait-Free Operations

The usage of the NCAS operation within such a retry-loop prevents the estimation of a WCET for the data structure operation in the presence of concurrent executions. This is because the number of retries needed to successfully complete the data structure operation is unpredictable a priori. This is unacceptable under hard real-time conditions. To overcome this issue we transform the lock-free operation into a wait-free one by means of a helping scheme implemented via an operation queue as depicted in Fig. 7.2 on pg. 105. This operation queue guarantees progress of data structure operations built on top of it.

The operation queue is implemented by a wait-free FIFO and works as depicted in Fig. 7.2: Every thread encapsulates its data structure operation into a so called opqueue structure. This structure contains a pointer to the callback $\lambda$, a pointer to the NCAS structure generated by the callback $\lambda$ and a unique ticket to establish a chronological order among all elements within the operation queue. After enqueuing its own operation each thread performs the following steps until its own operation is completed: Get the oldest opqueue structure from the operation queue and try to perform the encapsulated data structure operation. The latter is done in the following three steps:

1. The opqueue structure is checked for an active NCAS structure; i.e. an NCAS structure whose associated NCAS operation might still be executed successfully. If one is found, the second step is skipped.

2. The callback $\lambda$ is used to create a new NCAS structure (see Fig. 7.2). A thread then tries to replace the reference to the NCAS structure in the opqueue structure with a reference to the newly created one. To ensure that only non-active NCAS structures are replaced, a CAS instruction is used here.

3. The reference to the current NCAS structure is reloaded and its NCAS operation is executed. Thus, in case of multiple threads concurrently working on the same opqueue structure, exactly the same NCAS structure is used to perform the actual NCAS operation in a cooperative manner. Finally, in case of successful execution of the NCAS operation, the opqueue structure is dequeued.

The operation queue imposes two additional requirements:

Requirement W1. We need a wait-free FIFO implementation that allows us to find and use the oldest entry without removing it.
**Requirement W2.** The NCAS implementation has to support cooperation:
If all active threads work on the same NCAS operation (using the same NCAS structure), the described NCAS operation must be completed successfully.

This helping scheme facilitates the implementation of wait-free data structure operations. Thereby, a progress guarantee and an upper bound for the WCET can be given for these data structure operations. The implementation of the operation queue is explained in Sec. 7.5 on pg. 112ff.

### 7.3.3.3 Disjoint-Access-Parallel Operations

The helping scheme implemented by the operation queue has one disadvantage: All data structure operations are performed sequentially. This is not disjoint-access-parallel [IR94], leads to convoy effects [Fra04], and may cause poor average performance. In order to avoid these side effects, we introduce the concept of speculative execution (see Fig. 7.1 on pg. 103), similar to the previous chapter.

Before enqueueing an opqueue structure into the operation queue each thread tries to execute the data structure operation speculatively by the corresponding lock-free operation (see Sec. 7.3.3.1 on pg. 104ff). If the speculative execution fails, an opqueue structure will be enqueued into the operation queue and the thread carries on as described in the previous section. If the speculative execution is successful, the thread still works on the oldest entry of the operation queue. This is required to guarantee progress for the oldest entry in the operation queue. Otherwise, the oldest entry might starve due to continuously interfering speculative executions. By forcing every thread to execute at least one opqueue structure from the operation queue, it can be guaranteed that the oldest entry in the operation queue is completed in finite time.\(^{34}\)

The speculative execution creates one new requirement for the NCAS implementation:

**Requirement D1.** It has to be able to deal with different priorities of NCAS operations.

An NCAS operation in terms of an opqueue structure stalled within the operation queue has a higher priority than NCAS operations that are executed speculatively.

Using speculative execution, finally, we support the implementation of wait-free and disjoint-access-parallel data structure operations (see Fig. 7.1 on pg. 103) based on top of the wait-free operations provided by the helping scheme introduced above. Furthermore, the degree of disjoint-parallel accesses can be chosen

\(^{34}\)An operation can only be stalled by a thread performing a speculative execution. Such a thread then executes at least one opqueue structure of the operation queue. Thus, it cannot stall another operation queue entry. There is only a bounded number of threads. Hence, a stalled operation inside the opqueue is successfully performed in at most \(O(N^2)\), whereas \(N\) is the number of threads.
with respect to the number of speculative executions that can take place concurrently. Hereby, a low jitter can be traded for a better average performance as mentioned before.

7.4 NCAS

In this section we explain the implementation details of our weakly wait-free NCAS method and the read method in detail. Before describing the NCAS method and the read method, we introduce two additional data structures required by our implementation: The *NCAS structure* and the *NCAS word*.

7.4.1 NCAS Structure

Every NCAS operation is wrapped into a structure, an *NCAS structure*, that, as depicted in Fig. 7.2 on pg. 105, contains

- the addresses of the variables that are to be changed,
- their expected old values,
- their respective new values,
- and a status variable.

The status variable is able to adopt four states, i.e. WORKING, SUCCESS, WRONG_VALUES and KILLED. It is initialized to WORKING on creation of the NCAS structure and during the lifetime of the structure, it can only be changed from WORKING to a state different from WORKING. SUCCESS indicates a successful execution of the NCAS operation, WRONG_VALUES means that the values of the variables to be changed differ from the expected old values, and KILLED denotes that this NCAS operation has been cancelled by another NCAS operation.

For the NCAS structures, we use a reference counting idiom explained in Sec. 7.6 on pg. 115ff to ensure safe concurrent usage. rrtNCAS provides a function to create an NCAS structure within the callback λ. The addresses of the words to be changed and their old and new values are passed as parameters to this function, which is called create_ncas_struct().

7.4.2 NCAS Words and Value Function

Every variable that is supposed to be changed by an NCAS operation is wrapped into another structure, called an *NCAS word* (see Fig. 7.2 on pg. 105). An NCAS word either directly contains the value of one machine word or it contains a reference to another NCAS structure. In our implementation NCAS words consist of two machine words; there is a status bit that differentiates between "contains value directly" and "contains reference". In either case the remaining bits of the two machine words are interpreted accordingly. In order to guarantee
consistent updates of NCAS words, changes on them are always made by means of DCAS instructions.

On an NCAS word, an abstract value function is defined as follows:

If the NCAS word contains a value directly, the result of the function is this value. If the NCAS word contains a reference, the result of the value function depends on the values and the status of the referenced NCAS structure. If the status is not equal to SUCCESS, the value of the NCAS word is the value of the corresponding "old value" field in the referenced NCAS structure. If the status is SUCCESS, the value of the NCAS word is the value of the corresponding "new value" field in the referenced NCAS structure.

### 7.4.3 Read Method

Our $\textit{rtNCAS}$ library contains a read method, reflecting requirement $\textbf{L2}$ in Sec. 7.3.3.1 on pg. 104ff that implements the value function, described in the previous section. If this method is performed successfully, it returns a value that has been the value of the NCAS word for some moment during the invocation of the method.

In order to reliably read the value of an NCAS word with a reference to an NCAS structure, the read method tries to avoid the deallocation of the respective NCAS structure by means of our reference counting idiom described later in Sec. 7.6. The read method may fail to reliably read a value if the referred NCAS structure has been deallocated in the meantime. In this case the read method returns an error code.

```c
1 NCAS(NCS: pointer to NCAS structure) {
2     /* phase 1: inserting references into NCAS words */
3     for i = 0 to n-1:
4         if (! insertRef(NCS.toChange[i], NCS))
5             break;
6     /* phase 2: changing the values of all NCAS words */
7     CAS(NCS.status, WORKING, SUCCESS);
8     /* phase 3: replacing references with actual values */
9     if NCS.status == SUCCESS
10        for i = 0 to n-1:
11           replaceRef(NCS.toChange[i], NCS, NCS.new[i])
12     else
13        for i = 0 to n-1:
14           replaceRef(NCS.toChange[i], NCS, NCS.old[i])
15 }
```

**Listing 7.2:** Pseudocode of our NCAS implementation.

### 7.4.4 NCAS Method

Our NCAS method implements requirement $\textbf{L1}$, mentioned before in Sec. 7.3.3.1 on pg. 104ff. It consists of three phases, similar to Ramamurthy’s approach [Ram97], and its pseudocode is shown in List. 7.2. The behavior of these three phases can be described as follows:
Phase 1. A reference to the current NCAS structure \texttt{NCS} is inserted in every NCAS word that needs to be changed (ll. 3–5). Note that inserting these references does not affect the values of the NCAS words.

Phase 2. The NCAS method tries to change the status of the NCAS structure \texttt{NCS} from \texttt{WORKING} to \texttt{SUCCESS} by a single CAS instruction (l. 8). The CAS instruction fails, if the status of \texttt{NCS} has been set either to \texttt{WRONG\_VALUES} or \texttt{KILLED} before.

Phase 3. The references to the current NCAS structure \texttt{NCS} are replaced by the actual values (ll. 11–16). If the CAS instruction in phase 2 was successful, all references are replaced by the new values. Otherwise, the old values are restored; this works, since the replacing of the references in phase 1 does not affect the values of the NCAS words.

All manipulations of the NCAS words \texttt{NCS.toChange[i]} in \texttt{insertRef()} and \texttt{replaceRef()} are performed in a concurrency-safe way by means of DCAS instructions. Hereby, it is ensured that a reference is inserted only if the NCAS word contains the expected value in the first phase, and that the reference is only replaced by the actual value if \texttt{NCS.toChange[i]} still contains a reference to \texttt{NCS} in the last phase.

The NCAS method could fail in two cases. Both of them are detected in the first phase and are taken into account no later than in the second phase. Furthermore, both cases only arise in presence of speculative execution (see Sec. 7.3.3 on pg. 104ff), otherwise all NCAS operations are executed sequentially with the help of the operation queue, described later in Sec. 7.5.

Speculative execution might lead to the situation that two threads concurrently work on different NCAS structures \texttt{NCS\_A} and \texttt{NCS\_B} containing references to shared NCAS words. Assume that a reference to \texttt{NCS\_A} has already been inserted into an NCAS word and one of the threads tries to insert a reference to \texttt{NCS\_B} into the same NCAS word. While it is straightforward to replace the reference to \texttt{NCS\_A} with a reference to \texttt{NCS\_B}, it must be ensured that only one NCAS operation succeeds if both are in state \texttt{WORKING}. Otherwise, the data structure would be updated inconsistently in the third phase. This is prevented by setting the status of \texttt{NCS\_A} to \texttt{KILLED} by means of a CAS instruction before replacing the reference. This ensures that the values of the NCAS words are not changed by the NCAS operation referred by \texttt{NCS\_A}.

In the description above we assume that the NCAS operation referred by the NCAS structure \texttt{NCS\_B} \textit{wins}, i.e. the NCAS operation on \texttt{NCS\_B} can be successfully completed while that on \texttt{NCS\_A} is cancelled. The actual winner, however, will be determined by means of priorities that are explained later on in Sec. 7.4.6.

Additionally, the data structure could have been updated during the current NCAS operation. This can happen, if \texttt{NCS\_A} has already proceeded to the state \texttt{SUCCESS} in the example above. Here, a thread in the first phase would detect that an NCAS word does no longer have the expected old value. In this case, inserting a reference to the NCAS structure into the NCAS word is not possible without changing values.\footnote{After setting the reference, the NCAS word would contain the expected old value.} As a consequence, the whole NCAS operation cannot
be executed anymore. The status of the NCAS structure is then changed to \textsc{Wrong\_Values} by means of a CAS instruction. The value of the NCAS word is determined using the read method described in the foregoing section. If the read method fails, the status of the NCAS structure also is updated to \textsc{Wrong\_Values} and the NCAS operation fails. Again, this ensures that the values of the NCAS words are not erroneously changed by the current NCAS operation.

### 7.4.5 Cooperative NCAS

A problem occurs when two or more threads simultaneously work on the same NCAS structure. If one thread is still in the first phase despite the object’s status already being \textsc{Success}, setting a reference may change the value of the NCAS word. Thus, in the first phase a thread must ensure that a reference is inserted into an NCAS word only if the NCAS structure’s status is not \textsc{Success}. This is done by splitting the insertion of a reference into inserting a \textit{pre-reference}, then re-checking the status and only if it is not \textsc{Success} the pre-reference is replaced by a normal reference.

A pre-reference is similar to a reference, in that an NCAS word that contains a pre-reference has its value determined by the (pre-)referenced NCAS operation object. The difference is that the value is independent of the status of the object; it is always the expected old value. Thus, setting a pre-reference does not change the value of the NCAS word. By using a unique representation for the pre-reference, it is ensured that a thread does not mistake a pre-reference set by another thread for a pre-reference of its own; thus, ABA problems [Mic04a] are prevented. A pre-reference is distinguished from a reference by means of an additional bit next to the status bit in an NCAS word.

The second phase of the NCAS method requires that a reference is inserted into every NCAS word, otherwise the values of these NCAS words cannot be changed atomically. For this purpose, every pre-reference has to be replaced by a reference before the second phase can be completed. The proper use of a DCAS instruction ensures that a thread cannot replace its pre-reference by the actual reference if another thread successfully completed the NCAS operation in the meantime. Hence, setting a reference is impossible, after the status of the NCAS operation has already been changed. Thereby, we can guarantee that the values of the NCAS words are not altered in the first phase, although several threads may cooperatively work on the same NCAS structure, hereby reflecting requirement \textbf{W2} in Sec. 7.3.3.2 on pg. 106ff.

### 7.4.6 Priorities of NCAS Requests

According to requirement \textbf{D1} in Sec. 7.3.3.3 on pg. 107ff different priorities among NCAS operations have to be supported. NCAS operations with a higher priority shall not be disturbed by operations with lower priority. This is ensured by a priority comparison in the first phase of the NCAS method, if a reference to another NCAS structure is encountered when inserting references and its state is \textsc{Working}. If this NCAS operation has a higher priority, the
Figure 7.3: This figure illustrates the overflow problem with tickets of a 32-bit ticket source, if they are used to build a chronological relation. If the oldest and the newest used ticket face each other on the ticket cycle, then they are neither smaller than, nor greater than each other. Hence, such a scenario has to be avoided.

own NCAS operation is cancelled by setting its status to KILLED; if the other NCAS operation has lower priority, it is cancelled instead.

To find out which NCAS operation has a higher priority, the ticket associated with each entry of the operation queue is passed through to their associated NCAS structures. Speculatively executed NCAS operations do not have a ticket and thus, are assigned a lower priority. Hence, speculative executions of NCAS operations never cause the cancellation of other NCAS operations, they cancel themselves instead.

### 7.5 Operation Queue

We use a helping scheme based on an operation queue to support the implementation of wait-free data structure operations. The operation queue itself requires a wait-free FIFO implementation (see Sec. 7.3.3 on pg. 104ff, requirement W1).

Currently, we are not aware of an adequate pointer-based wait-free FIFO queue. Hence, our FIFO is based on a preallocated static array. For details on the problems that arise out of constructing a wait-free pointer-based FIFO queue that supports multiple concurrent queue operations, please refer to Jayanti [JP05].

Within that static array, our allocator and reference counting idiom (see Sec. 7.6 on pg. 115ff) give us wait-free methods to allocate elements, increment and decrement their reference counter and deallocate them safely once their reference counter reaches zero. Furthermore, there is a method to iterate over all allocated elements of the array.

The operation queue uses tickets to build a chronological order as illustrated in Fig. 7.2 on pg. 105. These tickets are generated using an FAA instruction on a 32-bit ticket source, so, there are $2^{32}$ tickets available. There is a cyclic comparison function to compare tickets. This function is cyclic, since for a ticket
value \( x \) all values \( x + t \mod 2^{32} \) (with \( t \in \{0, \ldots, 2^{31} - 1\} \)) are bigger than \( x \) and all values \( x - t \mod 2^{32} \) (with \( t \in \{0, \ldots, 2^{31} - 1\} \)) are smaller than \( x \). However, on \( a \) and \( b \), with \( a = b + 2^{31} \), the function is undefined (see Fig. 7.3). Hence, the appearance of two such values must be prevented. To preserve the FIFO property of the list, it is sufficient that the difference between the oldest and newest ticket in the queue does not exceed \( 2^{31} - 1 \). This also prevents cyclic comparison structures and non-comparable tickets.

By default, the entries of the operation queue do not have tickets. The enqueue and dequeue operations simply add/remove a ticket to/from the entry, using the enqueue and dequeue functions in List 7.3. Finding the oldest element in the queue then consists of iterating over all allocated elements and finding this one with the oldest ticket using the function \texttt{get_ticket()} on each element, similar to ConS presented within the previous chapter.

```c
/* Note: Every element has a ticket holder structure
 * "th" with 2 members: flag and ticket
 * flag==2 -> element has a ticket
 * flag==1 -> a pending enqueue() has not yet
 * assigned a ticket to the element
 * flag==0 -> element is not in use
 */
ticket_src_t t;

enqueue (elem_t x) {
    CAS(x.th.flag, 0, 1);
    myticket = FAA(t, 1);
    DCAS (x.th, <1,0>, <2, myticket >);
}

dequeue (elem_t x) {
    x.th = <0,0>;
}

get_ticket (elem_t x) {
    <flag,t> = x.th;
    if flag == 0; then return ELEM_NOT_IN_USE;
    if flag == 2; then return t;
    myticket = FAA(t, 1);
    DCAS(x.th, <1,0>, <2, myticket >);
    return x.th.ticket;
}
```

Listing 7.3: Pseudocode of enqueue, dequeue, and get_ticket.

The implementation of the function \texttt{enqueue()} and \texttt{get_ticket()} guarantees that for \( N \) threads, each of which enqueues one element and then continually finds the oldest element and dequeues it until its own element is dequeued, the distance between the oldest and the newest ticket is at most \( 3N^2 \). This upper bound is proved below in four steps. We use the term critical phase for the ticket assignment relating to an entry. This critical phase starts from Line 13 to either Line 15 or Line 27 within the context of a \texttt{get_ticket()} operation (see List 7.3), depending on what is executed successfully at first. Note that there are potentially \( N - 1 \) concurrent \texttt{get_ticket()} operations to an \texttt{enqueue()} on an entry, and tickets are assigned by a thread that successfully executed the DCAS in Line 15 or 27, respectively.

We consider the operation queue \( OP \) at time \( c \) and which impact the previous, current and later time frame of the \( OP \) has regarding to the ticket distribution, as depicted in Fig. 7.4. The first step proofs the maximal distance of two con-
Figure 7.4: This figure depicts the influence of concurrency while performing the critical phases of the ticket assignment regarding to the chronological order of operation queue entries. $OP_c$ is the operation queue at the time $c$. The tickets assigned for $OP_c$ depend on the execution histories of the critical phases performed (1) in the previous time frame, (2) within the current time frame $c$, and (3) for the next time frame of the operation queue. Note that due to the use of reference counting for the entries of the queue, there are always three time frames to consider.

secutively enqueued entries in $OP_c$. Then we proof that the maximal distance of tickets between the newest and oldest entry of the operation queue $OP_c$ is bounded, without considering tickets that are shifted from concurrent critical phases started within the prior time frame. The third step considers the impact of tickets that are shifted from concurrent critical phases started within the prior time frame. And the last one proofs the bound for tickets shifted from concurrent critical phases started after an $enqueue()$ of an entry for $OP_c$.

1. If all concurrent $get_ticket()$ operations concern the same entry, the distance between the tickets of two consecutively enqueued entries $E_1$ and $E_2$ is at most $2N$.

Proof. Due to interferences between a thread that performs $enqueue()$, and at most $N - 1$ remaining threads that help to assign a ticket to an entry by means of $get_ticket()$, the global 32-bit ticket source is incremented at most $N$ times for an entry. Hence, the distance between the tickets for two consecutively enqueued entries is at most $2N$, if the older entry, say $E_1$, gets the oldest ticket of the $N$ tickets drawn for $E_1$, and the newer entry, say $E_2$, gets the youngest of the $N$ tickets drawn for it.

2. Suppose there are $N$ entries within $OP_c$. If for each entry of the queue the $enqueue()$ operation was not concurrent to the critical phases of entries that are no longer or not yet in the queue, the maximum distance between the oldest and the newest ticket is at most $N^2$.

Proof. There is at most one ticket assigned to each entry. For $N$ entries in the queue at most $N^2$ tickets are drawn (see the previous proof and note that a distance between the tickets of two consecutively enqueued entries of $2N$ leads to a maximal distance of $N$ tickets for one of the next enqueued entries). There is no ticket drawn for entries currently not in the queue that ends up increasing the distance between the tickets of the oldest and the newest entry, because then the critical phase of the entry that draws it would be concurrent to at least one critical phase of a current entry, which is prohibited per definition. Thus the maximal distance here is $N^2$. 

3. The maximum distance between tickets drawn in the prior time frame for elements that are no longer in the queue and elements that are still in the queue is bounded by $N + (N - 1)^2$.

Proof. For entries, whose critical phase happened before (i.e., non-concurrently) the critical phases of these entries which are currently in the queue, tickets can be released that increase the maximal distance of tickets in $O_P_c$. There are $N$ threads, so there are only $N$ tickets that increase the maximal distance between the tickets of the oldest and newest entry. For an entry, whose critical phase happened concurrently to the critical phases of the entries that are currently in the queue, at most $N - 1$ tickets can increase the distance between the tickets of the oldest and newest entry. There are at most $N - 1$ of such entries; hence the upper bound is $(N - 1)^2$.

4. The maximum distance between tickets that are currently in the queue and drawn concurrently for entries that are not yet in the queue is bounded by $(N - 1)^2$.

Proof. The proof is equal to second paragraph of the previous one.

As we restrict the number of threads to $2^{12}$, the above condition for avoiding cycles or undefined behavior is met.

7.6 Reference-Aware Memory Management

As NCAS structures and entries in the operation queue are created dynamically and shared between threads, memory management becomes a serious problem. In this section we present our novel wait-free reference counting idiom and a wait-free special purpose memory allocator to enable multiple threads to work on shared objects that are guaranteed to be valid. Furthermore, as NCAS structures are potentially re-used, our reference counting provides a solution to avoid the ABA problem [Mic04a].

The memory management and the reference counting implementation is split into two layers. The bottom layer offers three functions

- to allocate a single object,
- to deallocate an object,
- and to iterate over all allocated objects.

The top layer uses and hides the bottom layer and offers the following functions

- to allocate an object and set its reference counter to 1,
- to increment and decrement the reference counter$^{36}$,

$^{36}$The decrement function deallocates the object if the reference counter reaches 0.
• and to iterate over all allocated elements.

The bottom layer is a variation of the simple design for a special purpose allocator. It uses an array of objects and a flag for each object to indicate whether it is allocated; allocation and deallocation simply manipulate these flags. The allocation and iteration functions traverse the array to find free or allocated elements, respectively. The iteration function uses an array, which split into two parts: The first part contains mainly used elements and fragmented free ones. The second part only contains free elements. So, the iteration function, then only has to traverse the first part; the border between the two parts is moved dynamically when allocating/deallocating elements. This optimization improves the average time for the iteration over all allocated elements from $O(\text{number of elements})$ to $O(\text{number of elements in the first part})$. Note that this optimization concerns situations, where only a few NCAS operations are stalled inside the operation queue, because it is quite unlikely in real applications that data structures permanently suffer from full contention.

The top layer is implemented using the functions offered by the bottom layer. The allocation function uses the allocation function from the bottom layer. In order to separate the dependencies of the allocation functions from each other or – in other words – avoid cyclic dependencies, both refc_alloc() and allocate() use one bit to indicate whether an object is allocated; thereby, the allocator is passed as argument to refc_alloc().\footnote{Parnas describes such a design decision to avoid cyclic dependencies as "sandwiching" [Par76]. His focus, however, is building hierarchical operating systems instead of non-blocking task synchronization mechanisms with strong progress guarantees.} The increment and decrement functions use the bottom layer’s deallocation function. The iteration function is the same as in the bottom layer. List. 7.4 depicts the allocation, increment and decrement functions of the top layer.

```
1 refc_alloc() {
2   x = allocate();
3   FAA(&x. refc_counter, 1);
4   x. refc_allocated = 1;
5   return x;
6 }
7
8 refc_incr(x) {
9   if (!x. refc_allocated) return false;
10  FAA(&x. refc_counter, 1);
11  if (!x. refc_allocated) {
12     refc_decr(x);
13     return false;
14  }
15  return true;
16 }
17
18 refc_decr(x) {
19  FAA(&x. refc_counter, -1);
20  r = DCAS(&x. refc_allocated, x. refc_counter>, <1,0>, <0,0>);
21  if (r) deallocate(x);
22 }
```

Listing 7.4: Memory allocation and reference counting functions.

In order to implement the wait-free FIFO described in the foregoing section, our memory management has to provide an allocation guarantee. An object cannot be allocated if it is already allocated or there is a pending refc_incr() invocation.
on this object. Thus, given \( N \) objects to manage, the memory management can satisfy an allocation request, as long as the number of allocated objects plus the number of non-allocated objects with a pending \texttt{refc_incr()} invocation is less than \( N \).

Since the amount of threads \( T \) that are using \texttt{rtNCAS} has to be known in advance, we are also able to determine the number of objects \( O \) that each thread will allocate (which also includes \texttt{refc_incr()} tries on \( O \)) at most. Hence, \texttt{rtNCAS} needs \( T \times O \) objects; if the memory management has at least that many objects, we can assume that every allocation attempt is successful.

### 7.7 Evaluation

The evaluation presented in this section is twofold: Firstly, we demonstrate how to build wait-free data structure operations using \texttt{rtNCAS}. And secondly, we provide some micro-benchmarks to examine the timing behavior of \texttt{rtNCAS} under stress situations. Furthermore, we quantify the scalability of data structure operations implemented via \texttt{rtNCAS} with increasing contention level and NCAS words to be changed.

We compare data structure operations using \texttt{rtNCAS} to such using ticket spinlocks [MCS91]. The reason for that is that we are not aware of any approach that fits into our environment (see Chap. 3 on pg. 19ff); we discuss related work later in Sec. 7.8 in detail. Moreover, we have chosen ticket spinlocks also as the basis for a comparison with Cobs (see Sec. 6.7 on pg. 91ff).

#### 7.7.1 Building Data Structure Operations using \texttt{rtNCAS}

Wait-free data structure operations using \texttt{rtNCAS} can be easily implemented by providing a callback \( \lambda \) specific for that operation (see Fig. 7.2 on pg. 105). List. 7.5 on pg. 118 depicts the method \texttt{lambda_enqueue()}, which defines the callback function \( \lambda \) for an enqueue method of a FIFO queue. Thereby, a pointer to the queue \( q \) and a new node \texttt{new_node} are passed as parameters to \texttt{lambda_enqueue()}.

The application code that has to perform an enqueue operation uses a wrapper function, say \texttt{enqueue()}, that

1. initialize the NCAS word structures to be needed,
2. wrap the memory locations that are to be changed into these NCAS words,
3. and pass this operation description (also including arguments and a pointer to the callback \( \lambda \), depicted in List. 7.5) to our library.

Firstly, \texttt{lambda_enqueue()} reads the current state of the FIFO queue by calling \texttt{read_enqueue()} using the read method introduced in Sec. 7.4.3 on pg. 109. These are the "old values" passed to the NCAS method. Secondly, the new state of the queue is computed using a sequential enqueue operation \texttt{calc_enqueue()}, this yields the "new values" passed to the NCAS method. Here, \texttt{calc_enqueue()}...
can be replaced by any other simple or complex data structure operation; the
algorithm, which calculates the new state, has not to care about concurrency.
Finally, an NCAS structure is created by create_ncas_struct() that is given a
state_table containing NCAS words, the "old values" and the "new values" as
parameter. So, building wait-free operations on top of our rtNCAS is straight-
forward, as the original sequential algorithm can be used directly and there is
no need to explicitly care about concurrency.

```c
1  lambda_enqueue ( q: pointer to queue;
2     new_node: pointer to new node ) {
3     node, tail, next, len: NCAS word;
4     if (!read_enqueue(q, new_node, &node, &tail, &next, &len))
5         return ERR_CONCURRENT_MOD;
6     if (!(state_table := calc_enqueue(&node, &tail, &next, &len)))
7         return ERR_INVALID_VAL;
8     return create_ncas_struct(
9         /* state_table :=
10         * { ADDR, OLD_STATE, NEW_STATE>:
11         * <&q->tail->next, &node, new_node>,
12         * <&q->tail, &tail, new_node>,
13         * <&new_node->next, &next, NULL>,
14         * <&q->len, &len, len+1> }
15         */
16
17     state_table );
18 }
```

Listing 7.5: Callback λ implements the enqueue operation of a FIFO queue.

### 7.7.2 Test Methods and Scenarios

We have implemented a wait-free queue as well as a stack on top of rtNCAS.
Furthermore, we built a reference implementation with ticket spinlocks, which
we have already used within the last chapter; this also allows a direct comparison
between CoS and rtNCAS.

The rtNCAS-based queue consists of a head pointer that points to the oldest
node inside the queue (see the user-defined FIFO queue in Fig. 7.2 on pg. 105),
a tail pointer to the newest node, a length variable that stores the number of
nodes currently in the queue, and an operation counter that is incremented each
time an operation on the queue is performed.

The rtNCAS-based stack has a head pointer to the newest node, and also a
length variable as well as an operation counter that occupy an identical semantic
as the rtNCAS-based queue.

For all implementations we performed the following benchmarks on the basis of
the test hardware described in Sec. A.2 on pg. 137.

1. **Benchmark:** This benchmark investigates the timing of rtNCAS-based
data structure operations with increasing contention. Therefore, it in-
stantiates up to sixteen concurrent one-shot RTAI threads. Each thread
alternately performs 10,000 contrary data structure operations until ter-
mination, i.e. enqueue and dequeue operations for the FIFO queue im-
implementations, and push and pop operations for the stacks respectively. However, in order to avoid the dilution of the differences between enqueue/dequeue operations on the one side, and push/pop operation on the other, the benchmark only measures enqueue or rather push operations. Furthermore, each thread occupies an affinity to a dedicated execution core to avoid migration of threads to other cores. For the implementations with rtNCAS, we performed this benchmark with 0, 1, and 5 speculative executions.

2. Benchmark: This benchmark identifies the scalability of the two data structures implemented via rtNCAS. Therefore, it triggers concurrent one-shot RTAI threads as before, and measures the performed operations per hundredth second with increasing contention level. For the implementations with rtNCAS, the benchmark uses 0, 1, and 5 speculative executions.

3. Benchmark: The third benchmark varies the number of concurrent test threads as well as the number of NCAS words to be changed. It instantiates up to 12 one-shot RTAI threads and uses up to 30 NCAS words (both in steps of two) to ensure that the code and data of the benchmark completely fit into the L1 caches of the used testbed. With each configuration, the benchmark performs 10,000 test runs in which each test thread reads the current configured number of shared NCAS words, calculates a trivial \(^{38}\) new state of these words, and performs a state transition using rtNCAS (with 0 and 5 speculative executions). In order to get a 3-dimensional contour of this scattered data, we put an equidistant grid of interpolated measuring points on top of the measured data. We, furthermore, weight those points, which are closer to the measuring points, up to twice as much.

7.7.3 Results and Discussion

7.7.3.1 Results

The merged results of the first two benchmarks are shown in Fig. 7.5 on pg. 120 for the queue implementations; for the stack implementations, the results of the first and second benchmark are furthermore shown in Fig. 7.6 on pg. 121. Each figure consists of four plots; in each case the maximal and the average response times, the coefficient of variation, as indication for execution jitter, and the throughput for a one-hundredth second are illustrated. Moreover, each plot shows results for the rtNCAS implementations with 0, 1, and 5 speculative executions as well as for the implementations with ticket spinlocks.

The results for the third benchmark are furthermore shown in Fig. 7.7 on pg. 123 (without speculative executions) and Fig. 7.8 on pg. 124 (with five speculative executions), where the maximal, average, and median response times, as well as the coefficient of variation of the measured values are plotted 3-dimensional against different numbers of threads and NCAS words to be changed.

\(^{38}\)The calculation of the new state of an NCAS word consists of an increment by one of the old state.
(a) The left figure depicts the maximal response times, and the one to the right shows the average times of performed enqueue operations.

(b) The left figure shows changes in execution-time jitter, and the right one illustrates the count of enqueue/dequeue operations performed for a one-hundredth second.

Figure 7.5: Scalability benchmarks for the queue implementations with increasing levels of concurrency.
(a) The left figure depicts the maximal response times, and the one to the right shows the average times of performed push operations.

(b) The left figure shows changes in execution-time jitter, and the right one illustrates the count of push/pop operations performed for a one-hundredth second.

Figure 7.6: Scalability benchmarks for the stack implementations with increasing levels of concurrency.
7.7.3.2 Discussion

The results for the stack and queue implementations (see Fig. 7.5 on pg. 120, and Fig. 7.6 on pg. 121) are pretty similar, despite their different data structure operations. For the enqueue operation at most ten NCAS words are used, whereas for the push operation at most five NCAS words are touched. For both data structure operations, the introduced execution jitter for the \texttt{rtNCAS} implementations is significantly smaller compared to the ticket spinlock case, if speculative executions are not performed (blue vs. green graphs of the left plot in Fig. 7.5(b) and Fig. 7.6(b) respectively). The speculative execution (black graphs depict our results for one, and red graphs depict our results for five speculative executions) can interfere with the execution of stalled operations inside the operation queue; this potentially leads to higher execution jitter and maximum response times. However, the likelihood of higher maximum times decreases, as more speculative executions were performed successfully; hence, fewer operations have been inserted into the operation queue. This is equivalent to the optimization for the average case; for instance, this can be seen on the local maximum on the left plot of Fig. 7.6(a) (red line) between concurrency level 6 and 10, where the maximum response times slightly decreases from concurrency level 8 to 10.

The exponential increases in jitter and maximal response times starting from concurrency level 14 (see left plots of Fig. 7.5(a), 7.5(b), 7.6(a), and 7.6(b)) has the same reason as mentioned in Sec. 6.7.2 on pg. 92ff. During the run of our benchmarks with sixteen threads, the 256 KB L1 caches of our testbed are exhausted, and its data and/or instructions are displaced. This is the reason for this significant increase in jitter and maximum times. Note that with an increasing number of speculative executions, this kink leaves the general impression to become smaller. However, the execution jitter is – for the \texttt{rtNCAS} benchmarks with speculative execution – inherently on a higher starting basis.

In real-time systems, an upper bound for execution times of operations is inherently a correctness requirement. Though, for those tasks that occupy, for instance, no hard real-time constraints, average response times are also a relevant aspect and of high magnitude. Furthermore, we show that for hard real-time tasks, the goals to optimize the average cases and to minimize jitter compulsorily go not along with significant increases in maximal response times. Therefore, we also comprise the second benchmark to the following discussions. The average response times for the data structure implemented by means of ticket spinlocks are about a factor ten smaller; this can be seen on the right plots of Fig. 7.5(a) and 7.6(a). However, this is the reason for our highly deterministic timing behavior, i.e. minimal execution jitter. Furthermore, due to the fact that \texttt{rtNCAS}-based data structure operations implement some kind of helping, our throughput in contrary operations is anyway slightly higher for scenarios with pretty low contention and for high contention scenarios. This can be seen on the right plots of Fig. 7.5(b) and 7.6(b) respectively. These both plots display the results for the second benchmark (see Sec. 7.7.2 on pg. 118ff). In other words, the correlation of one performed operation and the execution times for the spinlock case does not apply to the \texttt{rtNCAS} case. This is the crucial factor for our better throughput.
(a) The left figure depicts the maximal response times, and the one to the right shows the average times.

(b) The left figure shows the median of the measured response times, and the right one shows changes in execution-time jitter (coefficient of variation).

**Figure 7.7:** Results for the third benchmark (without speculative executions) with increasing levels of concurrency and number of NCAS words. Note that the z-axes of the maximal, average, and median plots show response times in microseconds.
(a) The left figure depicts the **maximal** response times, and the one to the right shows the **average** times.

(b) The left figure shows the **median** of the measured response times, and the right one shows changes in execution-time jitter (coefficient of variation).

**Figure 7.8:** Results for the third benchmark (with five speculative executions) with increasing levels of concurrency and number of NCAS words. The z-axes of the maximal, average, and median plots show the response times in microseconds.
Sec. 7.7: Evaluation

Up to now, we have neglected another dimension — the number of NCAS words to be changed within a data structure operation based on $\text{rtNCAS}$. Thus, we now additionally consider this aspect. The results therefore, i.e., the third benchmark, are shown in Fig. 7.7 on pg. 123 for the case without speculative executions and in Fig. 7.8 on pg. 124 for the case with five speculative executions. We have omitted the case with only one speculative try due to the fact that it barely has benefits in throughput, but occupies significant drawbacks in terms of execution jitter and worst-case response times.

The effect in maximal response times with an increasing level of concurrency for a fixed set of NCAS words is slightly higher than the effect induced by a fixed level of concurrency for an increasing number of NCAS words to be touched within a data structure operation. This can be seen on the left plots of Fig. 7.7(a) and 7.8(a). However, for high contention scenarios, the impacts in maximal response times increase disproportionately. In particular for the benchmark with five speculative executions this impacts become apparent. Our experience is that complex data structure operations based on stacks, lists, queues, hash tables, or tree structures typically occupy an amount of words to be changed simultaneously not more than ten.

The above drawn conclusion about the minimal execution jitter induced by $\text{rtNCAS}$ for a set of five or rather ten NCAS words to be changed and increasing levels of contention (without speculative executions) also applies, if we take the third dimension into account. Thereby, the variation of the number of words has only minimal impacts, as can be seen on the right plot of Fig. 7.7(b). Hence, $\text{rtNCAS}$ used without speculative executions and with different levels of contention as well as with different numbers of NCAS words to be changed introduces a minimal amount of non-determinism, which can also be seen on the minimal differences between the right plot of Fig. 7.7(a) (average case) and the left plot of Fig. 7.7(b) (median case). This effect is highly desirable in real-time circumstances. Only for low contention levels and an amount of words to be changed simultaneously less than 15, the differences between a thread perform an NCAS operation alone (due to no contention) and not more than three threads help each other to perform an NCAS operation becomes visible in jitter. With increasing levels of concurrency and number of words to be changed simultaneously, scenarios without contention become unlikely.

For the results of the third benchmark with five speculative tries, the peeks in average and median response times of the benchmark case without speculative tries can be smoothed (see right plot of Fig. 7.8(a) vs. Fig. 7.7(a), and left plot of Fig. 7.8(b) vs. Fig. 7.7(b)). However, this goes along with a clear increase in terms of maximal response times (see left plot of Fig. 7.8(a) vs. Fig. 7.7(a)), in particular for scenarios that suffer from high contention and with a high amount of words to be changed. Nevertheless, the amount of jitter is general on a moderate level.

These $\text{rtNCAS}$ results also come along with full interrupt transparency that allows us to isolate tasks with hard real-time constraints from tasks with weaker progress requirements (see Chap. 3 on pg. 19ff).
7.8 Further Related Work

In the following, we classify previous work on building generic nonblocking synchronization primitives into three categories, software transactional memory, universal constructions based on process consensus protocols, and NCAS operations, which show a lot of similarities, but are often investigated under different aspects. Thereby, universal constructions or rather NCAS mechanisms are often used to build transactional memory systems.

The first category defines software transactional memory that is largely out of scope for this chapter. Thus, we concentrate on approaches to build data structure operations. In [AM99], Anderson and colleagues extended their previous work [ARJ97, ARMJ97] on using lock-free objects in real-time systems to support real-time transactions on memory-resident data. However, their work only focuses preemptive priority-based uni-processor real-time systems. Moreover, using lock-free transactions on the basis of appropriate process scheduling is currently limited to deadline-monotonic [LW82] and earliest-deadline-first [LL73].

Pizlo et. al investigated the impacts of lock-free transactions in terms of responsiveness of high priority threads and throughput with real-time Java [PPJV04]. Their work also focuses only preemptive priority-based uni-processor real-time systems, and they also bound the number of retries of a lock-free transaction on the basis of appropriate scheduling decisions, like Anderson and colleagues.

The next category concerns universal constructions that are templates of auxiliary schemata to build concurrency-safe object operations with a certain progress guarantee from their sequential specification. In [Her91], Herlihy introduced the first theoretical wait-free universal construct that consumes an unbounded amount of memory, serializes all operations on object \( O \) and makes copies of potentially large objects. He claims that the memory consumption can be reduced to \( O(N^3) \), whereas \( N \) is the number of threads accessing \( O \). Herlihy’s universal construct suffers from two major issues: Firstly, it serializes all operations on \( O \), which induces convoy effects [Fra04] analog to locks. This potentially leads to high latencies and jitter as well as poor average performance, independent of the priority of threads. Secondly, Jayanti showed in [Jay98] that Herlihy’s universal construction requires at least \( N \) local computation steps to complete a single operation.

Chuong and colleagues presented in [CER10] a theoretical universal construction for large objects based on a linearizable wait-free process consensus protocol. Thereby, processes can abort their operations in a transaction-like manner, if their linearization has not been already started to perform. Their work focuses also multi-core systems, where processes potentially crash in critical sections. However, there are multiple problems with their approach. First, their used memory management system is not addressed further that let arise a lot of open questions and implied assumptions, which are not explained within this paper. Second, their approach is not applicable for arbitrary data structure operations. Thus, for instance, they present in their paper a simple as well as an optimized auxiliary mechanism; the latter is optimized especially for FIFO queues and implies restrictions on how many concurrent data structure operations can be handled correctly. Thereby, only one enqueue operation is allowed to be performed.
Sec. 7.9: Chapter Summary

127

concurrently to a dequeue operation. Furthermore, their optimized auxiliary mechanism allows multiple simultaneous dequeue operations only on an empty queue. Restrictions of their approach with respect to other data structure types were not been discussed. Last, data structure operations are always performed consecutively that may waste parallel performance in almost the same manner as locks; also performance investigations of their approach were not been made.

In [AM95], Anderson et. al presented a universal construction to implement multi-object operations in a wait-free and lock-free manner, e.g., to dequeue and re-enqueue a node from a priority queue. Their worst-case time complexity is $O(N)$, while their best-case time complexity is constant.

The last category defines NCAS implementations used to derive arbitrary concurrency-safe objects from a sequential implementation by means of a software NCAS operation. In [Sun09], Sundell presented a wait-free NCAS operation with a helping scheme, which he called "greedy helping and grabbing". To resolve conflicts where multiple threads compete for locking memory words, his approach changes the ownership of the word locks. Both the read and the NCAS interface guarantee wait freedom and linearizability (see Sec. 2.3.2f on pg. 12ff). However, his NCAS cannot handle ABA situations [Mic04a] internally\(^{39}\), and hence only truly unique new values are allowed or at least a limited series of updates to the same values. In terms of a general usage of his approach, it may be not a trivial task to check whether an application can satisfy this assumption or not. Though, his read operation is very efficient, since it can read the current values without helping by using a reference counting idiom. However, the scope of his work is in transactions of sets of values instead of building data structure operations. Building custom data structure operations is also supported; this, however, only guarantees lock freedom.

In [HFP02], Harris et. al presented a practical NCAS operation built from a single-word CAS instruction, which also guarantees only lock freedom. Their implementation benefits from systems with memory alignment, which allows the use of two bits (for 4 byte aligned pointers) ulterior.

Sundell presented in [Sun04] a wait-free memory allocator with reference counting. However, the time complexity of his DeRefLink() function is $O(N)$ steps, where $N$ is the number of threads. In contrast, our reference counting idiom in Sec. 7.6 on pg. 115ff has a constant time complexity.

7.9 Chapter Summary

7.9.1 Based on the Presented Approach

Within this chapter the approach of rtNCAS was presented, a library that offers linearizable, lock-free, wait-free, and disjoint-access-parallel interfaces for reading and conditionally swapping multiple words in an atomic manner. Devel-

\(^{39}\)Note that all CAS operations suffer from the ABA problem, if the usage does not ensure that unique values are conditionally swapped all the time. However, for NCAS operations built via software, the implementation has to ensure internally that values to be swapped by means of CAS instructions are truly unique, which is up to the implementation and not to its usage.
opers of shared data structures now have an easy way to achieve full interrupt-transparency with a strong progress guarantee. Furthermore, with rtNCAS developers can re-use their sequential algorithms on top of it without modifications.

rtNCAS is not limited to real-time applications and can also be used in other environments, as the implementation of rtNCAS is fully operating system independent.

To the best of our knowledge, rtNCAS is the first approach that performs operations on disjoint memory words in parallel by speculatively executing operations and guarantees wait freedom even for complex data structure operations, without inducing dependencies to the system scheduler or limit the number of concurrent operations built on top of rtNCAS.

In addition, the wait-free property of data structure operations using rtNCAS may further reduce release-time jitter of threads that is often critical in real-time applications.

7.9.2 Based on the Initially Positioned Questions

This chapter has tackled the issue that complex data structures cannot be implemented in a nonblocking way that satisfy the requirements of a wide range of real-time controllers and applications, similar to locks. This is equivalent to LoR-3 as mentioned in Sec. 3.5 on pg. 28ff. Previous approaches imply many restrictions, such as limitations on scheduling algorithms or the cardinality of concurrent data structure operations. rtNCAS only limits the number of threads accessing to a shared data structure simultaneously to 4,096. For the today’s point of view, the real-time controller sketched in Chap. 3 on pg. 19ff. will not exceed this limit within the next few years. Moreover, rtNCAS is able to conditionally swap up to 256 words simultaneously. We are not aware of any typical data structure operation that has to touch more than 15 words at the same time; thus, we argue that this limitation is sufficient to the most real-time application scenarios in practice.

rtNCAS offers full interrupt-transparency without dependencies to the real-time scheduler. We have shown that this goes along with almost comparable maximal execution times, if we evaluate shared objects implemented by means of rtNCAS to such implemented by ticket spinlocks. Furthermore, the average performance of rtNCAS significantly outperforms ticket spinlocks, and the execution jitter introduced by rtNCAS is considerable small. Both values, i.e. the average performance as well as the varying execution times, can be adjusted in rtNCAS on the basis of the number of speculative executions. We have chosen ticket spinlocks [MCS91] as competitor in our evaluation for two reasons. First, we are not aware of any approach that fit into our use case in terms of timing constraints, in terms of use patterns of today’s data structure operations (concurrent caller cardinalities), and in terms of limited space capacities of such embedded systems. And second, locks are well-known and still going strong in real-time systems. Thus, we suppose that future approaches that might refine parts of our rtNCAS approach can easily evaluate their work to rtNCAS on the basis of ticket spinlocks without the need to re-implement rtNCAS.
completely. We argue that the implementation of ticket spinlocks is a piece of cake compared to re-implement \texttt{rtNCAS}. Additionally, ticket spinlocks are well suited for an evaluation basis for one reason: operations built on top of it do not suffer from starvation, if concurrent threads are scheduled appropriately.

The stress tests of our evaluation ensure that the maximal execution times are close to the worst-case execution times of the considered approaches. We have shown that the runtime differences of \texttt{rtNCAS} and spinlocks are reasonable small.

There is an overhead in space introduced by \texttt{rtNCAS} that depends on the number of concurrent threads $T$. However, this space overhead is bounded to $O(T \times O)$, whereas $O$ is constant, and can be determined in advance.
In this chapter, we give a brief summary of the results of this dissertation and sketch lessons learned out of our research activities. In addition, we summarize our contributions, sketch the limitations of the presented approaches, and make suggestions for future work.

8.1 Overall Summary and Conclusions

Upcoming multi-core processors force developers of – especially data and computationally intensive – real-time systems to concern with novel concurrency control mechanisms. Their goal will be to keep the timing quality and to continuously increase the labor efficiency of their systems. This, however, requires synchronization mechanisms that

• are adapted to modern processors that occupy a high and increasing level of parallelism,

• exhibit strong progress properties to satisfy the temporal requirements of timely applications that primarily have been not developed for multi-core CPUs,

• isolate the logical and temporal properties of high priority real-time tasks from tasks with weaker priorities,

• allow a separate and modular verification without the need to consider the entire execution model of the considered system, i.e. occupy compositional correctness properties,
and that exhibit a higher level of genericness to solve more complex requirements that do not fit to special purpose concurrency control objects.

We have shown that appropriate mechanisms for time-dependent systems do not exist up to now or rather induce a plenty of restrictions. To overcome this, we have presented novel synchronization mechanisms that satisfy the above stated requirements. We have, furthermore, preceded our investigations in a pattern-orientated way by introducing the level of recycling (LoR) terminology. Our LoR terminology describes the level of re-use when designing a nonblocking communication object for a dedicated use case.

An isolated coordination of concurrent threads that does not depend on the real-time scheduler has thereby various advantages. Progress of nonblocking operations with strong progress guarantees, i.e. wait-free ones, does not depend to appropriate scheduling conditions. This also relieves the construction of the anyway complex scheduling decisions of real-world timely applications. Moreover, the outcome of this isolation is interrupt-transparency, which in turn enables the cooperation of concurrent threads and, thus, avoid the waste of parallel performance. This has, as shown within the evaluations of the previous four chapters, significant advantages in terms of a low level of jitter and a high performance without inducing significant drawbacks in maximal execution time compared to traditional approaches without progress isolation.

This dissertation has shown that nonblocking communication in parallel shared-memory real-time systems is feasible in a pattern-based way that scales well with an increasing amount of parallelism. Thereby, our requirements for shared-memory communication based on the investigation of an industrial real-time controller (see Chap. 3 on pg. 19ff).

In Chap. 4 on pg. 31ff we have investigated the alarm handling of this real-world piece of RC software. On this basis, we have stated the requirements that result from its operational environment and designed a solution that occupies a strong nonblocking progress guarantee, i.e. wait freedom. To the best of our knowledge, this results in the first wait-free unsorted queue that supports multiple concurrent operations. The queue operations are based on atomic bit instructions and the entire solution is highly tailored for the described use case. This approach was, furthermore, entirely designed from scratch and, thus, ensures the abidance by the stated requirements of the operational environment, i.e. a highly optimized LoR-0 approach.

Thereby, our queue additionally has a helping queue structure that enables the high-grade optimization in terms of time for the time-critical enqueue operations. This is of high magnitude, as enqueue operations are synchronously executed when alarms occur and, thus, delay the respective alarm reaction. In addition, we introduce "local preferences" to highly increase the determinism in time of enqueue operations. We have, besides, presented a mathematical approach to optimize the search path when traversing the queue, which is based on an array, to find a free element. This all enable us to highly adjust the timing behavior to the initially stated requirements. However, due to this one-way optimization of only enqueue operations, the genericness of this unsorted queue approach is rather limited with respect to other use cases.
This solution construction has induced a plenty of efforts for R&D concerning the design, test, and verification of the control-flow synchronized queue methods. These results are equal to the initially expected ones in Sec. 3.5 on pg. 28ff. This confirms our initial stated assumption that the complexity of nonblocking objects, if built from scratch, is the reason not to consider such approaches in practice.

Memory management for parallel real-time controller is of prime importance, in particular if simultaneously used by units of execution that occupy different priorities and cycle times. Thus, a nonblocking approach for memory management is promising with respect to the timing quality of a real-time system. In Chap. 5 on pg. 57ff we have tackled this aspect. Thereby, we have re-used the approach of the wait-free alarm handler within the context of a wait-free dynamic storage allocator; this is equal to our LoR-1 terminology. We have although re-used local preferences of units of execution from the "alarm handler" queue with a different meaning, i.e. memory areas to be allocated have to be contiguous. Furthermore, we have generalized the two layers of the prior helping queue approach to binary trees. To the best of our knowledge, this results in the first completely wait-free memory allocator algorithm that occupies highly deterministic response times and is highly scalable with an increasing number of execution cores.

While re-using the idea of the previously described queue was straightforward, the efforts for implementation, test, and verification of the adapted approach, however, were comparable high. Moreover, the genericness of the novel approach is still limited to memory allocation concerns. The timing behavior of the presented DSA – in contrast to the previous use case of the unsorted queue – are not as highly optimized as within the prior scope. The reason for that is due to the interpretation of locality. While locality for the alarm handler defines a preference of threads, locality within the scope of DSA means that allocated memory has to be contiguously. This exhibits slightly more timing drawbacks in terms of contention by concurrently traversing the tree and manipulating the counter of the nodes.

Over the time, software is exposed to continuous changes on the basis of a changing operational environment. For instance, imagine the sortation of the alarm information. A sortation on basis of the presented queue cannot be solved easily. This is due to a general lack of expandability and changeability for non-blocking operations. To map a continuous changing operational environment to operations synchronized in a nonblocking way, we have introduced CoBS. This approach allows to re-use arbitrary lock-free operations in systems with temporal constraints. CoBS recycles nonblocking solutions from use cases without timing constraints and, thus, is an LoR-2 approach. Re-using lock-free algorithms in real-time circumstances is promising, because there is a lot of work on lock-free communication objects, but work on wait-free protocols is rather rare and use case specific.

Thus, Chap. 6 on pg. 77ff describes a methodology to partition linearizable lock-free operations and execute these parts cooperatively by means of a sophisticated auxiliary scheme. The WCET for operations built on top of CoBS can be bounded. In addition, CoBS enables to execute lock-free protocols speculatively.
(i.e. disjoint-access parallelism), which has advantages in environments with up to a moderate level of contention.

COBS occupies a high level of genericness, as it re-uses any kind of linearizable lock-free operation in timely applications. We have shown that the efforts in R&D for using lock-free protocols in real-time circumstances are moderate, in particular compared to design own nonblocking operations. In addition, in use cases that suffers only from low contention, COBS offers high-performance. In use cases with up to a high level of concurrency, the timing behavior for COBS-based operations is similar to traditional spinlocks. However, operations that based on COBS scale considerably better with increasing parallelism than their blocking counterparts. Furthermore, operations built on top of COBS induce only a low level of jitter and occupy a high parallel performance.

In Chap. 7 on pg. 99ff we have tackled shared objects that have to perform complex operations. Complex operations have to simultaneously modify larger parts of the shared memory that cannot be solved by current hardware instructions. For instance, the CAS instructions of today’s multi-core processors are currently limited to two machine words. This prohibits us to directly design more complex nonblocking operations. Thus, in this chapter we have presented rtNCAS, a library that is able to conditionally swap almost arbitrarily many words in a linearizable and wait-free manner. rtNCAS can be used as generic approach to control concurrency, similar to traditional spinlocks. It additionally enables developers to re-use their sequential data structure operations within parallel environments without the need to manually care about concurrency; this is an LoR-3 approach according to our terminology. In comparison to spinlocks, this approach induces only a low level of jitter, offers scalable high-performance, and comparable maximal response times. This goes along with full interrupt-transparency, i.e. wait freedom and linearizability as a strong nonblocking correctness condition.

We have shown that implementing wait-free operations on top of rtNCAS is a piece of cake compared to design own wait-free algorithms. The timing behavior of our approach is, furthermore, adjustable according to the use case. In addition to the number of words to be swapped simultaneously and the number of concurrent threads, the number of speculative executions can be used to trade a low execution jitter for a higher average-case performance.

8.2 Summary of Contributions

This thesis has the following main contributions:

- The investigation of design patterns when building time-critical non-blocking shared objects in real-time systems. The investigations were been made on the basis of a case study based on an industrial real-time controller. Therefore we have introduced the LoR terminology that offers an intuitive model for describing the level of re-use when building such data structure operations.

- The wait-free queue as a highly optimized special purpose approach. It
is the first queue that allows arbitrarily concurrent enqueue and dequeue operations. This queue focuses use cases, where enqueue operations are highly time-critical.

• The first completely wait-free dynamic storage allocator that re-uses the design pattern of the wait-free queue within another scope. It focuses use cases, where memory is allocated only in a fixed size or where allocation guarantees are not required.

• The methodology to construct time-aware lock-free operations. This enables developers to re-use lock-free operations on shared data structures in real-time circumstances. This approach allows, furthermore, to keep the deterministic timing of real-time systems while accessing to shared objects and exploit the parallel performance of upcoming multi-core processors. Developers are now able to cope better with a changing operational environment on the basis of non-complex objects.

• The methodology to enable transaction-based mutual inclusion of concurrent threads that access to shared objects. For this purpose we have introduced a novel wait-free, linearizable, and disjoint-access parallel NCAS. This approach contributes a sophisticated auxiliary schema to conditionally swap arbitrarily located and almost arbitrarily many words in an atomic manner, while ensuring wait freedom and linearizability among others. In addition, this generic concept enables developers to re-use their sequential data structure operations in parallel environments without the need to manually care about concurrency.

Additionally, the thesis describes some auxiliary contributions that advance the state of the art:

• The constant-time wait-free reference counting approach. It enables developers to build a reference-aware memory management in time-critical systems.

• The approach of building chronological relations on the basis of tickets in systems with strong progress requirements, which enables us to build wait-free FIFOs on the basis of predefined static arrays. Wait-free FIFOs are of high magnitude not only for the most nonblocking auxiliary schemes.

• The evaluation of disjoint-access parallelism in real-time circumstances. This topic will be of major relevance for future parallel real-time systems to continuously benefit for the parallel performance of future multi-core and many-core platforms.

8.3 Limitations and Suggestions for Further Work

The answers of our initial positioned questions show great promise for the coordination of concurrent accesses to shared objects in real-time systems software. We are, furthermore, thoroughly convinced that the use of nonblocking objects
in real-time circumstances induces, in the future, much less worries in terms of the complexity of their construction. This is due to the power of our more generic approaches that allow developers to easily construct any kind of non-blocking objects with strong progress guarantees. Thereby, the genericness of CobS and rtNCAS can be used to trade a low increase in hardware costs (compared to nonblocking approaches implemented from scratch) for a minimum of complexity and efforts for R&D.

Some of our work exhibits limitations. In the following, we shall sketch two of the most interesting ones to give suggestions for further work.

Nonblocking approaches for real-time cases typically based on auxiliary schemes, which partly suffer from priority violations. This effect, however, is bounded and partly has significant fewer dimensions compared to the blocking counterparts and, thus, might be tolerable in hard real-time circumstances. The minimization of this side effect promises to further reduce jitter and WCET. The reason for this effect is due to the request table in CobS; for rtNCAS the operation queue causes this nondeterministic effect. This leaves room for further research activities. An idea would be to introduce more waiting queues, one for each priority. The priority-based execution of the requests or rather operations included within these waiting queues requires novel mechanisms for a cooperative eradication.

The use of atomic memory instructions, such as CAS or TAS, mostly causes cache effects, which are highly nondeterministic. This is due to cache coherence techniques and the parallel execution of code. The avoidance of such instructions, if possible, and the minimization of read operations on shared memory, hence, promises to further reduce jitter, find a lower bound for WCET, and increase the (parallel) performance. An idea would be to investigate the observation and modification of the global status of "stalled operations" within CobS and rtNCAS. An indirection for the observation, where threads are able to observe the status of stalled operations locally as long as there are no status changes promises to further reduce communication over main memory. Maybe a design pattern can be found, where and under which circumstances such memory accesses can be partly substituted by local ones.
A.1 Octa-Core Test Environment

As hardware environment an octa-core PC with two Xeon E5440 quad-core processors was used, running at a 2.83 GHz clock frequency, 256 KB L1 cache per core for instructions and data, 6 MB L2 cache per core pair (that is 12 MB per CPU) and 1,333 MHz FSB.

Today, the investigated industrial system runs on different powerful Intel processors. Hence, we used the x86 architecture as a basis for appropriate test environments (see also Sec. A.2), even though a Xeon processor creates too much heat to integrate it into the industrial target system.

A.2 Sixteen-Core Test Environment

This hardware environment defines a machine with sixteen physical execution cores consisting of four Intel Xeon E7340 quad-core processors without hyper-threading. Each core runs at 2.40 GHz clock frequency. Furthermore, all CPUs have 256 KB L1 cache for data and instructions as well as 4 MB L2 cache per core pair (i.e. 8 MB per CPU) and 1,066 MHz FSB.
Additionally, this machine has 32 GB main memory and runs Debian *lenny*, 2.6.29.4 vanilla kernel patched with RTAI 3.7.1 (see also Sec. A.3).

### A.3 Real-Time Application Interface for Linux

The Real-Time Application Interface\(^{40}\), or RTAI in short, is a project initiated for about one decade by Paolo Mantegazza, the head of the "Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano" [DM03]. RTAI provides a hard real-time extension to Linux\(^{41}\) for different architectures, such as standard PC hardware. It is mostly released under LGPL version 2; the RTAI core is released under GPL version 2.

RTAI implements a real-time kernel as well as a HAL for the Linux kernel; thereby, the Linux kernel runs in RTAI as low priority real-time task. Moreover, the RTAI core implements interrupt handling, several real-time scheduler that apply to our multi-core purposes, timer, inter-task communication mechanisms (such as mailboxes and semaphores), and mechanisms to asynchronously communicate with non-privileged Linux processes.

With RTAI it is feasible to perform timely applications in kernel-space as well as in user-space. The latter comes almost without timing penalties in latency and overhead. Within the development stage, we have executed our benchmarks in user-space to use the protection mechanisms of Linux, as our hardware platforms are no single-user environments, where we could work under mutual exclusion.

\(^{40}\)RTAI homepage: [http://www.rtai.org](http://www.rtai.org)

\(^{41}\)Linux kernel archives: [http://www.kernel.org](http://www.kernel.org)
B.1 Inhaltsverzeichnis 

1 Einleitung ................................. 1
2 Grundlagen und Stand der Kunst .......... 5
  2.1 Gemeinsam genutzte Ressourcen in Echtzeit-Systemen .......... 6
  2.2 Nebenwirkungen von Echtzeit-Synchronisation ................. 6
  2.3 Nicht-blockierende Synchronisation ......................... 9
    2.3.1 Atomare Speicherooperationen .................. 9

2.3.1.1 Prozessorunterstützung
2.3.1.2 Speicherinstruktionen in nicht-blockierenden Protokollen
2.3.2 Fortschrittseigenschaften
2.3.3 Korrektheitseigenschaften
2.3.4 Blockierungs- vs. Wiederholungs- vs. Hilfslaufzeiten
2.3.5 Vor- und Nachteile
2.4 Multicore-Prozessoren vs. Multi-Prozessoren

3 Untersuchung eines industriellen Systems
3.1 Robotersysteme in der Vergangenheit
3.2 Gegenwärtige Robotersysteme
3.3 Zukünftige Robotersysteme
3.3.1 Transition zur symmetrischen Simultanverarbeitung
3.3.2 Anforderungen an die Kommunikation über gemeinsamen Speicher
3.4 Betrachtete Ziele und Fragen dieser Arbeit
3.5 Einteilung in Ebenen der Wiederverwendung (LoR)
3.6 Zusammenfassung des Problems und unser Lösungsansatz

4 LoR-0: Eine wartefreie Liste mit lokalen Präferenzen
4.1 Einführung
4.1.1 Motivation
4.1.2 Wissenschaftlicher Beitrag
4.1.3 Gliederung des Kapitels
4.2 Analyse
4.2.1 Status quo
4.2.2 Spezifische Anforderungen für symmetrische Simultanverarbeitung
4.3 Wartefreie Liste für beliebig viele Eintrager und Entnehmer
4.3.1 Idee
4.3.2 Protokoll
4.3.2.1 Eintragen in die Liste
4.3.2.2 Lesen von Listeneinträgen
4.3.2.3 Austragen aus der Liste
4.3.3 Verifikation
4.3.4 Evaluation
4.3.4.1 Testmethode
4.3.4.2 Testszenarien und Ergebnisdiskussion
4.3.5 Schlussfolgerung
4.4 Hilfsliste
4.4.1 Motivation
4.4.2 Idee
4.4.3 Protokoll
4.4.4 Validation
4.4.5 Evaluation ........................................... 48
4.4.6 Weiterführende Partitionierungsstrategie .......... 49
4.4.7 Optimierung des Suchpfads .......................... 49
4.4.8 Schlussfolgerung .................................... 50
4.5 Verbesserter Traversierungsalgorithmus ............... 51
4.5.1 Motivation ........................................... 51
4.5.2 Idee und Protokoll ................................... 51
4.5.3 Evaluation ........................................... 52
4.5.4 Schlussfolgerung .................................... 53
4.6 Weitere angrenzende Arbeiten ......................... 54
4.7 Zusammenfassung des Kapitels ......................... 54
4.7.1 auf Basis des präsentierten Verfahrens .......... 54
4.7.2 auf Basis der initial gestellten Fragen .......... 55
5 LoR-1: Ein wartefreier Allokator für dynamischen Speicher 57
5.1 Einführung ............................................. 57
5.1.1 Motivation ........................................... 58
5.1.2 Gliederung des Kapitels ............................. 59
5.2 Spezifische Anforderungen ............................. 59
5.3 Ein wartefreier DSA ..................................... 60
5.3.1 Idee .................................................. 60
5.3.2 Algorithmus ......................................... 62
5.3.2.1 Allokation ...................................... 62
5.3.2.2 Deallokation ................................... 63
5.3.2.3 Abbildung der Baumstruktur auf ein Array .... 63
5.3.3 Längstmögliche Laufzeit ............................. 64
5.3.4 Verifikation .......................................... 65
5.3.4.1 Korrektheit ..................................... 65
5.3.4.2 Linearisierbarkeit ............................... 67
5.3.5 Evaluation ........................................... 68
5.3.5.1 Testmethode und -szenarien .................... 68
5.3.5.2 Ergebnisdiskussion .............................. 69
5.3.6 Schlussfolgerung .................................... 71
5.4 Weitere angrenzende Arbeiten ......................... 73
5.5 Zusammenfassung des Kapitels ......................... 74
5.5.1 auf Basis des präsentierten Verfahrens .......... 74
5.5.2 auf Basis der initial gestellten Fragen .......... 75
6 LoR-2: Verwendung sperrfreier Operationen in Echtzeit- 
   Applikationen ........................................... 77
6.1 Einführung ............................................. 77
6.1.1 Zusätzliche Anforderungen .......................... 78
6.1.2 Gliederung des Kapitels ............................. 79
6.2 Entwurfsmuster linearisierbarer sperrfreier Protokolle . 79
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Konkurrenzsituationsbewusste Serialisierung</td>
<td>82</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Hilfsmechanismus</td>
<td>83</td>
</tr>
<tr>
<td>6.3.1.1</td>
<td>Auftragstabelle</td>
<td>83</td>
</tr>
<tr>
<td>6.3.1.2</td>
<td>Ausführung von Aufträgen</td>
<td>85</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Annahmen</td>
<td>87</td>
</tr>
<tr>
<td>6.4</td>
<td>Zuordnbare Protokolle</td>
<td>87</td>
</tr>
<tr>
<td>6.5</td>
<td>Wartefreies Idiom zum Zählen von konkurrenten Referenzen</td>
<td>88</td>
</tr>
<tr>
<td>6.6</td>
<td>Aspekte der Verifikation</td>
<td>90</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Fortschrittsgarantie</td>
<td>90</td>
</tr>
<tr>
<td>6.6.2</td>
<td>Speicherverbrauch</td>
<td>91</td>
</tr>
<tr>
<td>6.6.3</td>
<td>Verhinderung von Problemen durch die Wiederverwendung von Zeigern</td>
<td>91</td>
</tr>
<tr>
<td>6.7</td>
<td>Evaluation</td>
<td>91</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Testmethode und -szenarien</td>
<td>91</td>
</tr>
<tr>
<td>6.7.2</td>
<td>Ergebnisdiskussion</td>
<td>92</td>
</tr>
<tr>
<td>6.8</td>
<td>Weitere angrenzende Arbeiten</td>
<td>95</td>
</tr>
<tr>
<td>6.9</td>
<td>Zusammenfassung des Kapitels</td>
<td>96</td>
</tr>
<tr>
<td>6.9.1</td>
<td>auf Basis des präsentierten Verfahren</td>
<td>96</td>
</tr>
<tr>
<td>6.9.2</td>
<td>auf Basis der initial gestellten Fragen</td>
<td>97</td>
</tr>
<tr>
<td>7</td>
<td><strong>LoR-3: Eine wartefreie NCAS-Bibliothek</strong></td>
<td>99</td>
</tr>
<tr>
<td>7.1</td>
<td>Einführung</td>
<td>99</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Wissenschaftlicher Beitrag</td>
<td>100</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Gliederung des Kapitels</td>
<td>101</td>
</tr>
<tr>
<td>7.2</td>
<td>Systemmodell</td>
<td>101</td>
</tr>
<tr>
<td>7.3</td>
<td><strong>rtNCAS: Eine NCAS-Bibliothek für Echtzeit-Systeme</strong></td>
<td>101</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Terminologie</td>
<td>102</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Eigenschaften</td>
<td>102</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Aufbau</td>
<td>104</td>
</tr>
<tr>
<td>7.3.3.1</td>
<td>Sperrfreie Operationen</td>
<td>104</td>
</tr>
<tr>
<td>7.3.3.2</td>
<td>Wartefreie Operationen</td>
<td>106</td>
</tr>
<tr>
<td>7.3.3.3</td>
<td>Parallelität von nicht-konkurrenten Operationen</td>
<td>107</td>
</tr>
<tr>
<td>7.4</td>
<td>NCAS</td>
<td>108</td>
</tr>
<tr>
<td>7.4.1</td>
<td>NCAS-Struktur</td>
<td>108</td>
</tr>
<tr>
<td>7.4.2</td>
<td>NCAS-Wörter und Wertfunktion</td>
<td>108</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Lese-Methode</td>
<td>109</td>
</tr>
<tr>
<td>7.4.4</td>
<td>NCAS-Methode</td>
<td>109</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Kooperatives NCAS</td>
<td>111</td>
</tr>
<tr>
<td>7.4.6</td>
<td>Prioritäten von NCAS-Aufträgen</td>
<td>111</td>
</tr>
<tr>
<td>7.5</td>
<td>Liste der Operationsbeschreibungen</td>
<td>112</td>
</tr>
<tr>
<td>7.6</td>
<td>Referenzbewusste Speicherverwaltung</td>
<td>115</td>
</tr>
<tr>
<td>7.7</td>
<td>Evaluation</td>
<td>117</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Entwerfen von Datenstruktur-Operationen mit Hilfe von rtNCAS</td>
<td>117</td>
</tr>
</tbody>
</table>
B.2 Einleitung


Moderne Lithografie-Verfahren (wie z. B. extrem ultraviolette elektromagnetische Strahlung (EUVL)) ermöglichen eine kontinuierliche Strukturverkleinerung bis zu Strukturgrößen von 11 nm [KK08]. Dies erlaubt es der Halbleiterindustrie eine zunehmend höhere Dichte integrierter Schaltkreise zu erreichen. Kleinere Strukturgrößen induzieren allerdings auch höhere (interne) Frequenzströme und überproportional mehr Stromverbrauch und Verlustleistung, was sich wiederum in mehr Hitzeentwicklung auswirkt. Um diesen physikalischen Restriktionen von Stromverbrauch und Verlustleistung entgegenzuwirken und weiterhin eine Grundlage zu schaffen, die Komplexität integrierter Schaltkreise zu steigern (was unter dem Moore'schen Gesetz wohlbekannt ist [Moo65]), haben Intel und andere Prozessorhersteller seit 2005 angefangen die Taktfrequenzen ihrer Prozessoren zu reduzieren und mehr als eine Ausführungseinheit auf einem Stück Silizium zu integrieren [Gee05].

Seit einiger Zeit werden diese Multicore-Prozessoren auch zunehmend immer interessanter und wichtiger für den Bereich eingebetteter Echtzeit-Software, da die Preise für (sehr) stromsparende Prozessoren mit mehreren Ausführungseinheiten sinken und die Taktfrequenzen für Einzelprozessoren stagnieren. Fortwäh-
rende Innovationen im Bereich von daten- und rechenintensiven industriellen Echtzeitsystemen sind allerdings abhängig von einer stetig steigenden Rechenleistung.


Um diese gravierenden Nebenwirkungen zu bewältigen, haben Forscher und Entwickler damit begonnen sich mit Möglichkeiten der Koordination und Kooperati-
on von Tasks zu befassen, um einen blockierenden Kontrollfluss beim Zugriff auf gemeinsame Objekte zu vermeiden und konkurrenle Situationen über entsprechende nicht-blockierende Verfahren aufzulösen. Die meisten dieser Verfahren fokussieren jedoch Anwendungsszenarien ohne zeitliche Anforderungen, induzieren weitreichende Limitierungen, die es schwer machen solche Verfahren in der Praxis einzusetzen bzw. betrachten nebeneinläufige Ausführungen, allerdings keine echte Parallelität [Ram97, Gre99, Hob02, Zha03]. Zudem sind nicht-blockierende Verfahren aufgrund ihrer algorithmischen Komplexität bekannt dafür, dass sie sehr schwer, korrekt zu implementieren sind [Gro08]. In der Mehrzahl aller Fälle sind nicht-blockierende Zugriffsmethoden auf gemeinsame Objekte zudem hochgradig optimierte Speziallösungen, die sich nicht ohne Weiteres auf andere Anwendungsszenarien adaptieren lassen.

Um all diese Nachteile zu bewältigen, präsentiert diese Dissertation pragmatischen Verfahren, um alle Zugriffsmethoden auf gemeinsam genutzte Objekte nicht-blockierend zu koordinieren, um damit Fortschrittsabhängigkeiten von Aktivitätsträgern auf ein Minimum zu reduzieren, ein hochgradig deterministisches und nach oben begrenztes Zeitverhalten zu erzielen sowie auf den spezifischen Anwendungsfällen justierbare Laufzeiten zu ermöglichen, ohne damit signifikant die skalierbare Parallelität der Anwendung zu limitieren, wie es von bestehenden Verfahren bekannt ist. Zusätzlich dazu gehen wir in einem Entwurfsmusterorientierten Weg an unsere Untersuchungen heran, um die aus der Komplexität nicht-blockierender Verfahren induzierten Nachteile zu untersuchen und, soweit wie möglich, zu bewältigen.

Wiederverwendung in anderen bzw. sich ändernden Echtzeit-, Nicht-Echtzeit- und gemischten Anwendungsszenarien.


### Gliederung dieser Arbeit

Diese Dissertation gliedert sich wie folgt:

- Kap. 2 diskutiert Grundlagen zu Echtzeit-Synchronisation in Bezug auf die
Koordination von konkurrenten Zugriffen auf gemeinsam genutzte Objekte sowie den Stand der Kunst. Ähnliche Arbeiten, die zwar nicht direkt in Bezug auf das Thema dieser Dissertation stehen, aber ähnlich im Sinne der Thematik der entsprechenden Kapitel sowie Unterkapitel sind, werden in den einzelnen Kapiteln (Kap. 4 bis 7) behandelt.


- Kap. 5 stellt ein Konzept vor, dass es ermöglicht das Muster der Hilfsliste des vorherigen Kapitels wiederzuverwenden, um damit einen wartefreien dynamischen Speicherallokator zu implementieren, welcher für Anwendungsszenarien im Echtzeit-Bereich konzipiert ist, die entweder keine Allokationsgarantien benötigen oder Speicher nur in fester Größe allozieren.

- Kap. 6 präsentiert ein neues Hilfsschema, um linearisierbare sperrfreie Objekte in Echtzeit-Umgebungen wiederzuverwenden, welche normalerweise eine unbegrenzte bzw. unbestimmbare maximale Laufzeit induzieren und deswegen für den Einsatz in Echtzeitsystemen ungeeignet sind. Mit der in diesem Kapitel präsentierten Methodik lässt sich die WCET solcher Protokolle nach oben abschätzen und ermöglicht deshalb die Wiederverwendung solcher Protokolle in zeitabhängigen Anwendungsszenarien.


- In Kap. 8 fassen wir unserer Arbeiten und Erkenntnisse sowie unserer wissenschaftlichen Beiträge zusammen, und geben Vorschläge für zukünftige, daran angrenzende Arbeiten.

### B.3 Zusammenfassung

B.3.1 Aggregation dieser Arbeit und Schlussfolgerungen

Die Umstellung der Prozessorhersteller von Einzel- zu Multicore-Prozessoren zwingt Entwickler, insbesondere von daten- und rechenintensiven Echtzeitsystemen sich mit neuartigen Mechanismen zur Nebenläufigkeitsteuerung zu beschäftigen. Das Ziel dabei wird sein, die heutige Qualität ihrer Systeme in Bezug auf das deterministische Zeitverhalten weiterhin zu bewahren und deren Produktivität weiterhin zu steigern. Dies jedoch fordert Synchronisationsmechanismen, welche

- auf die Gegebenheiten moderner Prozessoren angepasst sind, die über einen hohen und zunehmenden Grad an Parallelität verfügen,
- starke Fortschrittsgarantien offerieren, um den zeitlichen Anforderungen von Echtzeit-Applikationen nachzukommen, die ursprünglich nicht für die Ausführung auf Multicore-Prozessoren entwickelt wurden,
- die logischen und zeitlichen Eigenschaften von Echtzeit-Tasks mit einer hohen Priorität von solchen mit einer niedrigeren Dringlichkeit isolieren,
- eine modulare Verifikation deren logischen Korrektheit erlauben, ohne die gesamte Ausführung des kompletten Systems betrachten zu müssen, d. h. über eine kompositorische Korrektheitseigenschaft verfügen,
- und eine höhere Generizität aufweisen, um damit auch komplexere Anforderungen bewältigen zu können, die sich nicht über entsprechende Spezialkonstrukte von nebenaufgünstigen Objekten realisieren lassen.


Speicherverwaltung in parallelen Echtzeitsteuerungen ist von größter Bedeutung, insbesondere wenn sie von mehreren Aktivitätsträgern gleichzeitig genutzt wird, welche über verschiedene Prioritäten und Zykluszeiten verfügen. Daher ist ein nicht-blockierender Ansatz für die Speicherverwaltung in Bezug auf die Zeitqualität von Echtzeitsystemen vielversprechend. In Kap. 5 ab S. 57 sind wir diesen Punkt angegangen. Dabei haben wir die Lösung einer wartefreien Alarmbehandlung im Kontext von wartefreier dynamischer Speicherallokation wiederverwendet; dies ist nach unserer Terminologie ein LoR-1-Ansatz. Wir ha-


In Kap. 7 ab S. 99 sind wir gemeinsam genutzte Objekte angegangen, die komplexe Operationen auszuführen haben. Komplexe Operationen müssen größere Veränderungen an gemeinsamen Speicher atomar vollziehen, was mit Hilfe gängiger Hardware-Instruktionen nicht direkt gelöst werden kann. Beispielsweise sind die CAS-Instruktionen heutiger Multicore-Prozessoren auf zwei Maschinenwörter beschränkt. Dies ist der Grund dafür, dass wir ad hoc nicht in der Lage sind komplexe nicht-blockierende Operationen zu entwerfen. In diesem Kapitel haben wir deshalb rTNCAS vorgestellt, eine Bibliothek, welche konditional eine fast beliebige Anzahl von Wörtern verändern kann sowie für Operationen, die mit Hilfe dieser Bibliothek entworfen wurden, u. a. Linearisierbarkeit und Wartefreiheit garantieren kann. rTNCAS kann analog zu traditionellen Spinlocks generisch verwendet werden, um konkurrierende Operationen entsprechend zu koordinieren. Zudem erlaubt es Entwicklern ihre sequentiellen Datenstruktur-Operationen in parallelen, zeitgebundenen Umgebungen zu verwenden, ohne sich dabei manuell um nebenläufige Ablaufszenarien kümmern zu müssen. Gemäß unserer Terminologie entspricht dies einem LoR-3-Ansatz. Im Vergleich zu Spinlocks offeriert rTNCAS jedoch hochgradig deterministische sowie performante Ausführungszeiten mit vergleichbaren maximalen Antwortzeiten. Zudem garantiert rTNCAS völlige Unterbrechungstransparenz mit einer starken Fortschrittsgarantie, nämlich Wartefreiheit, sowie Linearisierbarkeit als starke Richtigkeitsgarantie.


### B.3.2 Wissenschaftliche Beiträge

Die wichtigsten wissenschaftlichen Beiträge dieser Arbeit sind wie folgt:


Der erste komplett wartefreie dynamische Speicherallokator, welcher das Entwurfsmuster der wartefreien Liste der Alarmbehandlung in einem anderen Anwendungskontext wiederverwendet. Dieses Verfahren fokussiert Anwendungsfälle, in denen Speicher ausschließlich in fixer Größe zugeteilt wird oder in denen keine Zuteilungsgarantien notwendig sind.


Zu den oben beschriebenen Beiträgen, weist diese Arbeit zusätzlich die folgenden wissenschaftlichen „Hilfsbeiträge“ auf, welche den aktuellen Stand der Kunst erweitern:

Sec. B.3: Zusammenfassung

153


B.3.3 Limitierungen dieser, und Ausblick auf zukünftige Arbeiten


Einige der hier präsentierten Arbeiten weisen Limitierungen auf, von denen wir im Folgenden zwei der interessantesten skizzieren sowie Vorschläge für weitere Arbeiten geben werden.


Die Benutzung atomarer Speicherinstruktionen, wie z. B. CAS oder TAS, führt meistens zu nicht-deterministischen zeitlichen Nebeneffekten bedingt durch Zugriffe auf kohärente Zwischenspeicher-Hierarchien sowie der parallelen Ausfüh-
Bibliography


(Cited on page 102.)

(Cited on pages 7 and 14.)

(Cited on page 54.)

(Cited on pages 58 and 73.)

(Cited on pages 6 and 22.)

(Cited on page 138.)

(Cited on page 18.)

(Cited on page 96.)

(Cited on page 17.)

(Cited on page 16.)

(Cited on page 10.)


BIBLIOGRAPHY

(Cited on page 74.)

(Cited on page 126.)

(Cited on pages 14, 23, and 25.)

(Cited on page 126.)

(Cited on page 17.)

(Cited on page 25.)

(Cited on pages 7, 14, 26, 78, 91, 92, 117, and 128.)

(Cited on page 9.)

(Cited on page 18.)

(Cited on page 60.)

(Cited on pages 10, 87, 88, 91, 111, 115, and 127.)

(Cited on pages 69 and 73.)
(Cited on pages 10 and 87.)

(Cited on pages 1 and 143.)

(Cited on page 15.)

(Cited on pages 57, 58, 69, and 74.)

(Cited on pages xxiii, 9, 11, 54, 79, 80, 81, 92, and 101.)

(Cited on page 24.)

(Cited on page 11.)

(Cited on page 11.)

(Cited on page 42.)

(Cited on pages 2, 7, and 144.)

(Cited on page 22.)


[Sut05] H. Sutter. The free lunch is over. in Dr. Dobb’s Journal, 30(3), March 2005. (Cited on pages 1 and 143.)


(Cited on page 19.)

(Cited on page 74.)

(Cited on pages 7, 14, 15, 25, and 78.)

(Cited on pages 3 and 146.)

(Cited on pages 19 and 20.)

(Cited on pages 2, 9, 14, 96, 104, and 145.)

(Cited on page 22.)